19'th Annual Tcl Association
Tcl/Tk Conference
Proceedings
Chicago, IL
November 14-16, 2012

TCL Association Publications

Proceedings of the 18'th Annual Tcl/Tk Conference
ISBN: TBA

Special thanks to Dawson Cowals for designing the Tcl Association logo.
For graphic design or web development consulting please visit him on the web at
http://www.dawsoncowals.com/

Table Of Contents

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 1
November 14, 10:45-12:15

Bringing Context to the Internet of Things

Dr. Emmanuel Frécon

This paper is dedicated to my newly become wife

***Abstract*—The context manager is aimed at being the hub of the house, a place where all sensors report (directly or indirectly) their data, sometimes in aggregated form, but also where all applications will search for information relevant to them, i.e. sensor values, location or information about their surroundings. The context is instantiated from a dynamic model to fit the needs of a variety of scenarios and settings. The manager provides an easy-to-use Web API and integrates external cloud services relevant for applications running in the house.**

***Index Terms*—IoT, Tcl, REST, JSON, Web, Integration, Web Services, Sensors, Actuators, Middleware, Energy, Smart Homes**

I. Introduction

THE Internet of Things promises a near future where domestic and work environments, but also cities and factories, are augmented with sensors and actuators that all are Internet entities. The deployment of IPv6 is key to this evolution by enabling each sensor or actuator to be accessed from any Internet enabled application or user, thus from almost anywhere. The Internet of Things is often seen as the catalyst of more intelligent environments, where applications will use Things to perform actions and sense on our behalf, with little human intervention.

As the number of connected Things will grow, making sense of what is accessible and can be done and how they relate to one another will be harder and harder. For taking good and qualified decisions, applications will not only need to know how to access the sensors and actuators, but also where these are located, the people in their vicinity, their immediate neighbouring Things, etc. The context manager is a modular Tcl[1] web service that attempts to provide this contextual information to applications, i.e. the extra logical layer empowering applications with the overlaying of dynamic sensor data on top of locational data of more static nature. While the context manager primarily targets home environments, it can also be used in other environments. The context manager relies on simple PubSub[2] and pull mechanisms to account for the low resources available on sensors. It also offers a streaming interface based on WebSockets for push and pull of sensor and actuator data.

Emmanuel Frécon is with the Interactive Collaborative Environments Laboratory, Swedish Institute of Computer Science, Box 1263, SE-16429 Kista, Sweden, e-mail: emmanuel@sics.se.

II. Related Work

There starts to exist a number of cloud-based services that target the IoT (Internet of Things) and provide APIs to store and later retrieve data that has been sent for storage into the cloud. Probably the most well-known of these services is COSM[1] (formerly known as pachube). But there are a number of other services such as sen.se[2], nimbits[3] or thingspeak[4] to mention a few. Common to all those services is a web-based API that is easily integrated directly from sensor platforms, providing tiny (connected) sensors or gateways off-site storage. The same API can be used to retrieve data from

[1]Cosm is available at https://www.cosm.com/ and is open for new account registration.

[2]Sen.se is available at http://open.sen.se/ and is open for beta testing by the way of invitations only.

[3]Nimbits is available at http://www.nimbits.com/ and is open for new account registration. Nimbits also touts private cloud solutions by allowing the integration of the nimbits solution within existing architectures.

[4]Thingspeak is available at https://www.thingspeak.com/ and is open for new account registration

the web services, thus opening up for data-mining activities if ever necessary. Common to those APIs is the use of REST[3] and JSON[4] for retrieving and posting data from and to the cloud. This is to ease integration from low-power consumer-oriented hardware platforms such as arduino[5] or gadgeteer[6].

In addition to providing an ”infinite” data storage for sensors and actuators, the power of these services lies in the community of people around them and the ability to integrate values from several sources to reason in improved and more sensible ways. In short, these services provide a whole ecosystem of devices, applications and people, by being able to pinpoint sensors using location services and providing ways to get notified whenever their value change. From the point of view of a house and household, sen.se seeks to take a step further by allowing users to build user interfaces to control their houses. Sen.se provides ways for its users to visually create applications by connecting sensors to web-side logic boxes and finally present the resulting “computation” through its dashboard, a web-based UI combining output of values with input of commands to be utterly received by actuators.

However, these services fail to provide more information about the context within which all these sensors and actuators are being placed, especially when it comes to smaller scale installations such as a house or a building. In order to be able to make energy-smart decisions, leading to smart actuation of the devices that are accessible to them, applications need to know about inhabitants, relative locations of sensors, external conditions, etc. So far, the merging of the Semantic Web[7] with sensor networks, also known as the Sensor Web or the Sensor Internet [8][9][10][11] has focused on the creation of specifications for different functionalities related to the management of sensor-based data (observations, measurements, sensor network descriptions, transducers, data streaming, etc.), and for the different types of services that may handle these data sources (planning, alert, observation and measurement collection and management, etc.). The cost of providing network abstraction and ontologies often comes with increased complexity. So while these middlewares effectively provides ways to reason about devices and actuators at a high level, they seldom solve the problem of providing the general context while still lowering the threshold for regular users.

III. Goals

The context manager is aimed at being the nav of the house, i.e. the place where all relevant sensors report (directly or indirectly) their data, sometimes in aggregated form, but also where all applications will dig for information that is relevant to them, i.e. both values from some sensors, but also their location or information about their surroundings. Being such a nav, the context manager is designed to be placed and hosted in a home gateway, i.e. a “number crunching” appliance that provides computing power and intelligence at a lower price in a central place[5]. In this context, houses are taken in their larger forms and can be entire buildings if necessary, and the design should open up for federations of context managers to adapt to the needs and privacy concerns of both building owners and flat owners (or inhabitants).

The main goal of the context manager is to provide dynamic ways to model the context, e.g. a house and all its online devices, be them sensors or actuators. The dynamism of the context is essential at different levels: first it is important to be able to host new devices as they are installed in the house, secondly it is important to be able to model the context in various ways because all houses are far from being the same and because there might be cultural differences between location that have an impact on the context itself. Consequently, the context manager takes an object-oriented approach, where the possible content of the context, i.e. the objects themselves is driven and controlled by a simple schema, i.e. a model of what objects can be made available in the context, but also a model

[5]The current implementation of the context manager has been verified to be fully functional on the open source BeagleBoard xM, an ARM A8 development board.

of their relations. The use of schemas could introduce complexity to the conceptual approach, so the context manager features a simplified schema with few rules and in a human-readable format. Several schemas can be aggregated, allowing for experts to provide base schemas, perhaps somewhat more complex in their form while still providing power to the end users and the inhabitants, so as to adapt to the specific needs of a household, a building or a custom-made online sensor.

The context manager seeks to provide an open API that follows the current trends within Web-based services and development. Web asynchronous communication is slowly moving from SOAP[12] and XML[13] standards into REST and JSON for a number of reasons. One of the advantages of these new standards are their ease of reading, i.e. core communication can be tested directly in the web browser, and the results from a query are easily read in a textual format that is much more compact than XML. It is out of the scope of this document to advocate for either one or the other standard, but since the context manager aims at providing an easy interface to application programmers, REST/JSON are more suitable to the task. Apart from allowing programmers to test queries against an existing instance, both the format of the queries and of the result are in general less cumbersome to parse, thus easily integrated into existing code and onto low-power platforms such as mobile phones or even sensor platforms.

IV. Design

A. Schema and Model

In order to cope with different sorts of environments and to account for the cultural differences between housings in various regions of the world, the context manager is based on a dynamic schema that directs the content of the objects that will be instantiated to describe a house or any other environment. The schema can include remote (web) schemas, thus providing ways for experts and/or interested users to collaborate, but also providing for the inclusion of new classes of objects that will support newly created sensors. In order to easily be accessible to technically inclined users, the schema supports few paradigms: single inheritance, a few base types (`Boolean`, `Integer`, `Float`, `String`, `Timestamp`[14] and arrays) and constraints. Constraints describe rules to which field values should comply to, providing minimum and maximum bounds or constraining only a few possible values. Constraints offer a way to model physical units and laws: for example, temperature can be expressed in Celsius and is always greater than -274.15. The syntax for the schema minimises idioms and is designed to be human-readable with lesser effort.

Objects modeling the context are instantiated from the schema, and sensors and/or external services will update the fields of these objects as new values are measured, made available or acquired. Initial instantiation will provide decent default values for all fields and subsequent updates will always be checked against the possible constraints that direct the content of one or several fields. The context manager provides a number of techniques for remote services to be notified when objects and their contained fields are modified as time passes.

B. Data Flow and Storage

The context manager reads its schema (and included schemas) during initialisation, subsequently reading a file describing the initial context. Typically, the initial context will be composed of more or less transient information such as the rooms composing a house together with their interconnections, but also an initial instantiation of the objects that will be involved in the dynamic representation of devices, sensors, actuators and inhabitants, together with their spatial relations. Modification of fields in objects, and queries for the inter-relationships is supported by a REST/JSON inspired API.

The API supports (basic) authentication and HTTPS[15] encryption if necessary, because of their widespread use and their ability to scale down to the few resources available on sensors. As times goes by, all values set are automatically mirrored to a noSQL database (cluster) implemented on top of REDIS[16]. The API supports access to historical

data. It also enables setting values in the future, thus supporting prediction. Whenever the scheduled time is reached, a value will automatically be set to the one that had been set in the future. The API also permits setting values in the past for the automated storage of historical data. So-called triggers implement a PubSub mechanism, allowing remote Web services, applications and sensors to be notified whenever value(s) in objects change upon given conditions. Finally, WebSockets[17] can be kept opened against particular objects, offering both a way to stream updates from the object as time passes, but also to update its fields whenever needed.

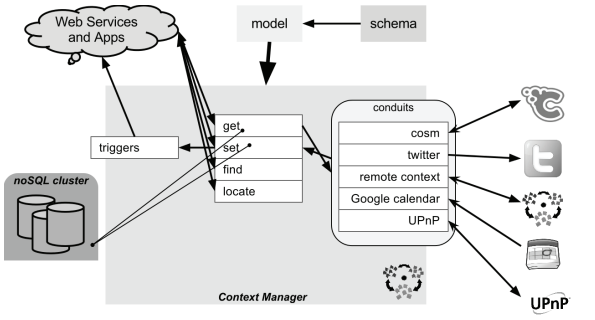


Figure 1. The API of the context manager provides REST/JSON entry points both to query the state of the context, but also to modify it. In addition, it supports external known and generic Web Services, while being able to predict and automatically store historical data through a connected noSQL database.

C. Extensibility through Conduits

The context manager is extensible through the concept of "conduits". Conduits are logical entities connected to external web services that will direct data into or from the context depending on a number of conditions. Typically, conduits will perform some transformation on the data to or from the external web service, while also retaining data that is specific to the remote service, e.g. login credentials, authorisation details, session information, etc. At present, there are conduits for Twitter, for import and export of data to the COSM cloud service, to remote context managers, to nearby UPnP objects and services and for the control of objects' values according to Google calendar bookings. Conduits are loaded as a set of plugins during the initialisation phase, they access the context manager through its modular internal API.

V. Implementation

A. Functionality

The context manager roughly provides the following set of functionality:

- It takes a schema and a model to provide a logical context of a building. This context can be accessed and modified using REST/JSON calls for maximised flexibility and integration. This means that most operations to and from the context can actually be made (tested?) from the comfort of any Web browser[6].
- The context manager provides a number of ground operations to:
  - Get the content of whole or part of the context, including the values of the fields of the instantiated objects and including values from the past, whenever they are accessible.
  - Modify values of objects that already are instantiated, which will be an operation that is often used when the value of a sensor changes.
  - Provides means to search for objects by the content of their field, the name of their class, etc.
  - Provides means to understand arrays as a technique to organise (part of) the model in a hierarchy and to find specific objects within such a hierarchy.
  - Provides means to trigger external web services whenever (part of) an object has changed, i.e. to mediate the content of the object to remote Web Services. Triggers offer enough flexibility so as to be able to:

[6]There are several JSON formatting extensions for most of the Web browsers. Such an extension will be necessary since the context manager minimises output by removing all unnecessary indentation or line breaks.

    - Restrict which field of the object are under watch and how to mediate their value to the remote service, i.e. as part of the URL, in the body of the posted data, etc.
    - Specify in details the headers, the MIME type and the method of the HTTP request (GET, POST, PUT, DELETE).
    - Control the maximum frequency of this mediation to avoid flooding the network.
    - Mediate only under certain conditions, expressed as a mathematical expression involving any of the fields of the object (based on the Tcl `expr` command).
    - Control if mediation should happen every time the value is updated, or only if it has changed since last time (the default).
  - Provides means to stream the flow of changes to remote clients via WebSockets, expressed as JSON representations of the object. This interface provides approximately the same level of control as the triggers described above[7]. WebSockets have two advantages:
    - They easily pass through firewalls and multi-level NAT hierarchies, thus making sure that even clients at the edges of the network can be notified of changes.
    - Once the connection has been established, packets containing object data are kept to a minimal (with almost no additional overhead or verbose header), which makes them suitable for transmission across WSN (wireless sensor networks).
  - Automatically saves versions of objects to a database for later retrieval. Given the unstructured nature of the context, noSQL databases are a perfect match, especially since they form the base of a number of data-mining techniques.

[7] All control that is only relevant to how the HTTP request to the external Web service should be made are left aside since this is not relevant in the case of WebSockets, i.e. in a framework that keeps the connection opened at all time.

- The context manager offers a pluggable architecture through the concept of "conduits", i.e. logical entities connected to external web services that will direct data to or from the model depending on a number of conditions. Typically, conduits will perform some transformation on the data to or from the external web service, while also retaining data that is specific to the remote service, e.g. login credentials, authorisation details, session information, etc. There are a number of conduits already available:
  - The COSM conduit is able to pull and push data from the COSM Cloud service. The conduit supports both the access of non-public feeds via an API key and to public feeds, polling their content at the necessary frequency whenever needed. Data can be transformed on the way to and from the feeds, matching *feed* names against *fields* names in the context manager, but also performing any mathematical operation supported by the `expr` command at copy time.
  - The remote context conduit is able to pull and push data from remote context managers, using triggers at the remote managers to get notified of changes. The conduit can be forced to poll for data instead to ease firewall and NAT traversal. As for the COSM conduit, data can be transformed on the way to and from the remote context. Being able to incorporate (parts of) remote context into a local context opens up for the creation of federations of context, and the ability to (re)use the sensors of your neighbours when taking decisions.
  - The local context conduit is a simplification of the remote conduit that only acts between local objects. It allows for the transfer (and transformation) of fields

values between different objects of the context, whenever some conditions are met.

- The Google calendar conduit binds the events of a given calendar to a `Boolean` that will turn `on` when there is a booking in the calendar, and `off` when there is no booking. Combined to local conduits and actuation (see section VII-B2), this can be used to specify when some devices should be turned on or off.
- The UPnP[18] conduit is able to pull and push data from remote UPnP services. The conduit is built on top of an SSDP discovery mechanism, thus being able to bind objects of the context manager to a service that has a given (discovered) name, or to a service that is at a given known location. While the conduit has been designed to bridge the context manager to objects within the LinkSmart middleware[11], it makes a number of assumptions to be able to be used in more generic cases. Similarly to the other conduits, the UPnP conduit is able to push and pull data to and from the known state variable of a UPnP server. For this to work in a generic way, the conduit assumes that the service has methods which name contains the name of the state variable and that contain the keywords "get" or "set" to get or set the content of the variable.

B. Security Mechanisms

There are two intertwined security mechanisms that will control the access to the context manager. First of all, the context manager is able to run on top of HTTPS[15] thus providing encryption of both requests and their results, so as to avoid eavesdropping from external parties. HTTPS was chosen because it is a well-established protocol that is widely supported across languages and platforms. The context manager supports both self-signed and authorised certificates.

Secondly, all web accesses can be controlled by a user name and password that will be mediated to the context manager using Basic Authentication[19]. Control should occur at the (virtual) directory level so as to provide for finer grained access restrictions if necessary. The goal is to refrain some users from, for example, setting the values of some objects of the context. Again, basic authentication was chosen because it is widely supported across languages and platforms. Basic Authentication sends the password from the client to the server unencrypted, however it should be used in conjunction with HTTPS.

There might be cases where HTTPS encryption is too heavy for the client platform in terms of computing resources, for example if sensors send directly their data to the context and/or need to reason about other sensors in their vicinity to take decisions. For those cases, the context manager is able to provide regular HTTP access. This HTTP access should be secured by a set of firewalling rules that will prevent access to the context manager from any remote client except the ones that need to access the manager for the reasons detailed above. Since these cases are most likely to occur within home networks and since most current home installations and Internet accesses are based on NAT techniques, the security risks introduced by unencrypted access in those cases are deemed to be low. In those cases, wires or proximity ensures physical security. This security relies however on proper configuration of the Internet access and the different firewalls involved.

C. Startup and Initialisation

On startup, the context manager will perform the following operations in sequence:

1) The context manager will start a web server with the proper credentials (see V-B) and proper encryption settings. Alternatively, the context manager can be embedded in an existing server framework if more suitable.
2) The web server will expose the schema and model that will define the context of the building or the house that the manager is controlling and modeling.
3) It will read the schema (see VI-A) that will describe what classes of objects are allowed

to appear in the context. This includes possible access to remote schemas that might be included from the main schema. Reading of the main schema might be through accessing the internal web service if necessary[8].

4) It will then read the model (see VI-B) that describes the particular building that it is modeling and controlling. All constraints implied by the schema that has just been read will be applied as the model is being read.
5) All objects instantiated as part of the model are bound to the noSQL engine so that further write operations will automatically lead to new versions of the object being stored and so that later get operations will be able to get older data, whenever possible.
6) It will initialise all conduits that are accessible to this context manager. Conduits are conceptually separated from the remaining of the code and are plugins communicating with the remaining of the context manager through a tiny and well-defined (internal) API.
7) It will read an initial "pairing" state (see VI-C) that is used to initialise a number of conduits and to bind a number of objects to remote services. Pairing is explained later and mostly a helper functionality that aims at reinitialising the context manager every time that it starts and reaching a similar functioning state.

VI. File Interfaces

Instead of providing an entire specification of the file formats that are understood by the context manager, this section focuses on providing real-life (shortened) examples. These examples are annotated and explained, bringing further insights to the internal of the context manager and all the facilities that it offers.

[8]Actually, reading the main schema via the web is encouraged since this will enforce UUIDs that remain constant over time and are bound to the specific installation. Preferably, a hostname will be involved in the main URL to bind the instantiated objects, classes and their UUIDs to a specific and logical place.

A. Schema

As highlighted before, the context manager provides techniques to specify the schema that will be used to describe the context itself. A key requirement to the provision of this schema is that it should be easily approachable not only by IT specialists, but also by less-knowledgeable people. To this end the schema brings in object-orientation concepts but simplifies them to their outermost. For example, it provides simple inheritance and mixes both object field specifications and inheritance[9]. The schema does not provide concepts such as private variables or similar, once again for the sake of simplification.

Below is a cut-down example of a schema, providing a flavour of how a schema looks and feels like. Roughly, this example schema divides the space into a number of possible floors and rooms within a building, and enables each part of the space to carry a number of devices (inhabitants are left aside on purpose). The example sports a single type of device, namely a thermometer, which demonstrates the (definition and) use of constraints to provide for a richer expression of units and properties of the physical world. The constraint defines temperature (in Celsius) as a floating point value that always is above the 0K.

```
Space {
  name String
  contains Space[]
  devices Device[]
  Outside {
  }
  Building {
     address Address
     pos Coordinate
  }
  Apartment {
     number Integer
  }
  Floor {
     above Floor
     below Floor
  }
  Room {
```

[9]While mixing class hierarchy and description in the same flow might surprise, this solution was chosen for the sake of simplicity. It has the advantage of presenting all data relevant to a given schema at a glance.

```
      Kitchen {
      }
      Bedroom {
      }
      Office {
      }
      Bathroom {
      }
  }
}
Address {
  street String
  streetNumber Integer
  areaCode Integer
  city String
  country String
}
Coordinate {
  latitude Float
  longitude Float
}
Temperature:Float {
  intervals {[-273.15,[}
  unit "celsius"
}
Device {
  name String
  SensorDevice {
     Weather {
        Thermometer {

          value Temperature
        }
     }
  }
}
```

To simplify the approach by non-technical experts, no forward declaration of classes or constraints is necessary. All new "types" that are discovered will be understood as (empty) classes as a start and converted when their real definition occurs. While this has the drawback of more complicated parsing and the possibility of duplicates or of unknown state — what to do when a class with a given name is then specified as a constraint under the same name — these problems are considered minor compared to the necessity to forward declare classes or constraints before being able to use them.

B. Model

The schema only specifies and constraints the types of the objects that should be placed in a model. While the schema is essential to the context manager since it provides guidelines to what can be instantiated within the model, achieving a conceptual model of a home and all its online devices is the ultimate goal of the context manager. To this end, the context manager provides a file format that is easily approachable, allowing people to quickly model their own house. At later stages, and depending on the success of the approach, graphical tools would certainly provide help in specifying the final model, perhaps based on existing drawings (blueprints or CAD).

Below is an extract of a model, based on the example schema above. The purpose of this example is to set the scene and provide a flavour for how model files could look like. Complete models tend to be more extensive, so the example below is not complete.

```
Outside pHataren1 {
  name "PositivHataren1"
  contains {myHouse}
  devices {
     outsideTemp
  }
}
Address aSoderman10 {
  street "August Södermansväg"
  streetNumber 10
  areaCode 12938
  city "Hägersten"
  country "Sweden"
}
# Approximate center of our lot.
Coordinate myPosition {
  latitude 59.299428
  longitude 17.970209
}
# The house contains three floors,
# which will contain the rooms.
Building myHouse {
  name "House Frecon-Waller"
  address aSoderman10
  pos myPosition
  contains {
     ground cellar top
  }
}
```

```
# The different floors in the house,
# here only one for the sake of
# concision.
Floor ground {
  name "Ground Floor"
  contains {
     hall kitchen diningRoom
     livingRoom bath vilma
  }
  above cellar
  below top
}
#######
# Devices
Thermometer outsideTemp {
  name "Temp. sensor outside"
}
```

The model uses the schema to control the content of objects that are created within the model. Every instance of a class is referenced using an identifier. Using techniques similar to those used for the schema, objects can be referenced before they are actually used, but the model provides enough feedback whenever the data that is specified does not correspond to the schema that controls what can be specified.

In the resulting model, both instantiated objects within the model and classes are identified by a UUID[20]. The UUID is of type 3 or 5 and built using a concatenation of the URL to the model (or to the schema), the class name and (when relevant) the reference to the object. This ensures that, even upon restarts, objects and classes will keep their UUIDs as long as the file structure, content and location has not changed.

C. Pairing

In order to be able to restart from a similar state at all times, the context manager is able to read from a pairing configuration file once the schema and the model have been read. The purpose of this file is to establish all the necessary conduit connections to well-known services. Pairing is made at the conduit level, thus at the REST/JSON level. In other words, when initialising the pairing, the context manager behaves as if it was an external client to itself. This is to be able to support new conduits in the future and to fail nicely if some conduit initialisation did not succeed properly.

Below is an annotated example file showing how pairing can be initialised at start, the syntax provides some visual markup to highlight the source and destination objects and uses a number of heuristics to detect which conduit to use for data migration. An integer is understood as a COSM feed, a UUID as an object from the local context manager, a URL ending with a UUID as an object in a remote context manager, a URN starting with `gcal:` as a Google calendar and a URN starting with `UPnP:` as a UPnP service. The "arrow" of the markup can specify and/or force polling frequency and indentation is used to further specify how the value of fields are carried to the remote entity.

```
# Map my heat pump to the COSM feed with
# identifier 53880. Non-matching
# fields/datastream names will be ignored.
# API key is picked up from the configuration
# of the context manager.
55851044-b290-56a5-3c88-d64ffbfa75e9 -> 53880
# Another COSM mapping, making sure the COSM
# datastream "inside" is mapped to 4 times the
# value of the field "value" in the context
# object,
20ecdbe4-8459-5636-6146-71c618badc71 -> 53882
  %inside% = 4.0*%value%
# Reverse COSM mapping, datastream called "2"
# in feed 55180 at COSM is brought the field
# "value" in the context object.
55180 --> 929494fb-84e1-50cb-beea-c04aecda088a
  %value% = %2%
# Pick up the weather station of somebody else,
# do some field names / datastream mappings and
# force polling to occur every 180 seconds.
45036 -180-> 684c4e19-c4ed-5861-f127-59109a41bb56

  %temperature% = %OutsideTemprature%
  %pressure% = %ABSPressure%
  %humidity% = %OutsideHumidity%
  %rain% = %Rain%
  %windDirection% = %WindDir%
  %windSpeed% = %WindSpeed%
# Send status of context object to the UPnP
# service named "Dev"
5d9a66e5-9738-598c-d0b0-e707eb0e2a36 -> UPnP:Dev
```

VII. Applications and Examples

This work has been carried out within the framework of a European project looking into energy optimisation. The project uses traditional home automation in order to attain energy savings while still offering the same level of comfort. The project

also seeks to offer "soft" actuation mechanisms, i.e. providing enough (summarised) information about some of the decisions taken by devices in the home to let inhabitants take the final necessary steps. Ambient displays are used to carry out this type of information in a form that is aesthetically acceptable. A number of prototype pilot houses have been equipped with sensors and actuators of various forms in order to gather data for future data-mining activities, but also to experiment with how smart actuation can turn into energy savings or reduction of CO_2 emissions. This section describes one of these prototype installations, located in the outskirts of Stockholm. There are several other installations, featuring a slightly different feature set of software and hardware so as to adapt to the particularities of these households: type of heating, electricity meter, etc.

A. Heating and (Inner) Climate

1) Heat Pump Analysis (and Control): Live status from a heat pump (IVT Greenline HT+) is picked up via its service serial interface using software from a small Swedish company called Husdata[10]. This is connected to a PC running Windows sitting on top of the pipe system in the direct vicinity of the heat pump. The default settings within StatLink, the software provided by Husdata as part of their offering have been slightly modify to increase the number of sensors being read and to regularly dump sensor data to a particular location on the PC, a location that is served by a tiny web server[11]. Raw dump data is (remotely) polled by a Tcl script at regular intervals and pushed to the context manager after naming transformations. Within the context manager, a conduit forwards acquired data to two COSM feeds[12].

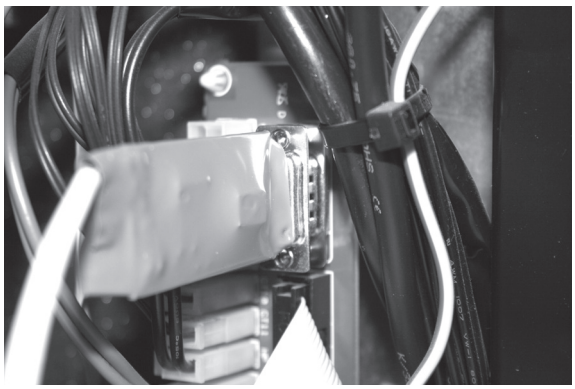


Figure 2. Husdata provides a hardware module to connect to the service serial interface of the heat pump, together with Windows software for analysing the decisions taken by the pump over time.

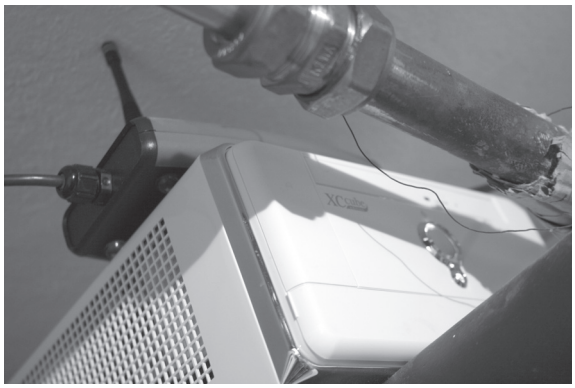


Figure 3. A Windows PC collects data from the heat pump, it is placed on top of a number of copper pipes in direct connection to the pump. The module with an antenna is the root node of the WSN network that collects temperature and main electricity data (see sections VII-A2 and VII-B1).

The current solution only provides gathering of heat pump sensor data. This has been immensely valuable since we can now perform long-term analysis of the behaviour of the heat pump using data-mining techniques, so as to be able to detect with

[10]Husdata http://www.husdata.se/ offers a number of hardware modules to connect a computer to a range of heat pump commonly found in Sweden, from several manufacturers.

[11]The current installation relies on mongoose, available at https://github.com/valenok/mongoose.

[12]Converted pump data is pushed to https://cosm.com/feeds/53880, temperature is pushed to one of the datastreams of https://cosm.com/feeds/53882, additionally raw data, as taken directly from the husdata software is pushed to https://cosm.com/feeds/52002. This is being used mainly for debugging purposes and for detecting possible failures in the context manager and in the surveillance PC network connection.

this given household will need warm water. It opens up for predicting when it will use warm water in the future, so as to shutdown warm water production during peak hours and start again at off-hours, before warm water is needed again. However, taking these last steps implies being able to control the internal logic of the heat pump, using the serial protocol described at http://rago600.sourceforge.net/.

2) Inner Temperature: A TinyNode[21] board running Contiki[22] and carrying a temperature sensor is hidden behind a photo frame. Measurements are sent along a mesh network at regular intervals, captured via the pump computer and sent on via UDP to a Tcl script. The script automatically pushes data into an object of the context manager, and further to COSM[13] via a conduit.

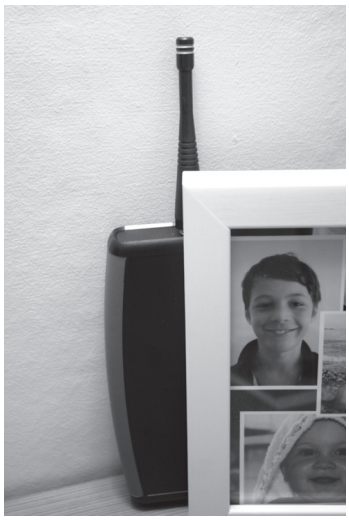


Figure 4. The TinyNode measuring temperature hides itself behind a photo frame in the living room, so as to break the aesthetics as little as possible. It has been slightly pushed aside for the sake of documentation and picture taking.

This particular heat pump installation only contains an outside temperature sensor, with which all decisions are taken when it comes to heating. The pump is able to host an inner temperature sensor (cabled) to take better decisions about when and when not to generate heat. Combined with the planned implementation of serial connection and control of the pump, the provision of a wireless inner sensor could provide for better inner climate without the wiring that is required by regular installations.

3) Weather: A specially written Tcl script can be used to update (in the future) an object of the context manager to reflect the weather forecast for a given location. The script uses the REST/JSON API from the Weather Underground[14] to access an hourly forecast for the coming 10 days. As updates are made in the future for those specific times, they are automatically stored in the noSQL cluster and set back as the "current" value as time passes. The script can be run once in a while or continuously. It will thus keep updating the object with an up-to-date weather forecast, allowing other applications using the context manager to reason about the current and future weather situation. For example, an application that would control heating could make the decision to accept temporary temperature drops if the outside temperature is only going to decrease for a few hours/days. The object that the script sends its data share the same model as a weather station, thus implementing a virtual private weather station in combination with the script.

B. Electricity and Energy Consumption

1) Total Measurement: The past decade has seen the progressive replacement of all electricity meters in Sweden in favours of so-called smart meters. These meters are able to report the hourly electricity consumption to the utility company as time passes, so as to bed for refined billing and better dimensioning of the grid. Pulses from the electricity meter are captured by another TinyNode sensor running Contiki, manufactured by CRL Sweden. Data is pushed out of the sensor network to the same Tcl

[13]Temperature is pushed to one of the feeds https://cosm.com/feeds/53882 that already is used to publish the outside temperature acquired via the heat pump, though to another datastream.

[14]Documentation for the API is available at http://www.wunderground.com/weather/api/. There are numerous other services offering the same type of data, Weather Underground was chosen because of its ability to chunk several questions into one request, but also because it is also uses ideas from the Internet of Things: forecast are improved using the data from private weather stations, whenever possible.

bridge as in section VII-A2. The bridge forwards to another object of the context manager, and thus automatically to COSM[15]. This publishes an history of the instantaneous power used by the household over time. As electricity is one of these hidden cost that is seldom understood, an ambient display is at the planning stage; a display that will both visualise how much electricity has been used so far, but also provides feedback to the instantaneous variations, thus to power, required by new devices being switched on (or off).

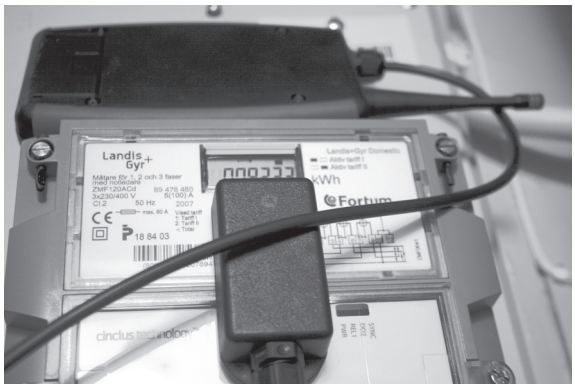


Figure 5. All electricity meters installed in Sweden host a LED (IR or visible) that flashes a number of time for each number of Watts used. The pulse metering node, sitting on top of the meter itself, continuously count these pulses and reports the total count for the latest period via the WSN network.

2) Measurement and Control at the Device Level: PlugWise are smart plugs manufactured by a Dutch company. They form a ZigBee mesh radio network, allowing access and control from a computer to which a specific USB key is connected. They offer three key features.

1) They host a relay, meaning that they are able to turn on or off all the devices connected to the plug and this from a distance. The state of the physical relay can be queried at any time.
2) They measure the instantaneous power being used by all devices connected to the plug, a value that can be requested from a distance.
3) They keep an hourly log of the electricity consumption related to the plug. As this log is kept in memory in the plug itself, historical data for the plug can be accessed from a distance at later time if necessary. The log rotates with time but keeps a few days worth of hourly data.

A Tcl script couples one or several objects from the context manager to as many smart plugs as there are objects. At the core of the Tcl bridge is a wrapper library around the command line interface of one[23] of the open source libraries created to access the PlugWise hardware and network. The bridging script connects to object representations of the plugs in the context manager using the WebSocket API[16] and pushes all information, including relay state and historical data, gathered from the plug. The state of the relay is represented by a `Boolean` and turning the field on and off in the context manager will be propagated to the physical plug, allowing to turn on and off connected electrical devices.

Figure 6. The form factor of the smart plugs from PlugWise (white plugs on the picture) make them easy to install across the house in order to measure the consumption of particular devices, but also to automatically turn on and off (sets of) devices based on heuristic such as the time of the day, the day of the week, or more advanced schemes in response to Demand/Response requirements from the grid.

[15]The COSM feed https://cosm.com/feeds/60040 is updated at two minutes interval with the current power consumption of the whole house

[16]In order to be able to resist to network equipment that restrict the use of WebSockets, the plugwise bridge is also able to poll at regular intervals for the desired state of the physical relay.

3) Spot Prices: The Swedish electricity market has been deregulated for a number of years and prices vary on an hourly basis, dividing Sweden in four different geographical regions. Nord Pool Spot runs the power market in Sweden and offers day-ahead prices to its customers. A Tcl script continuously acquire the prices[17] for all regions and updates one or several objects of the context to reflect the current price at that location.

C. Ambient Interfaces

An off-the-shelf multi-coloured lamp is put under the control of a REST-based server written in Tcl. Controlling of the lamp is via the IR from Dangerous Prototypes[18]. At present and for time reason, the solution only works on Windows, on top of WinLIRC. The lamp can take a wide number of colours and the REST interface accepts any RGB codes, approximating to the closest available colour on the lamp.

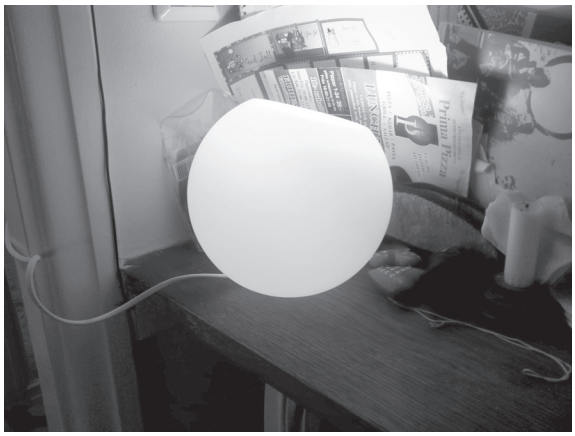


Figure 7. The design of the lamp makes it an acceptable display for home "events" in a number of cases.

A second Tcl REST server offers a Web interface to "tune" the lamp to various data sources present in the context manager. The user interface is kept to a bare minimum, but is easily accessible from both computers and mobile devices, which is the expected future scenario. Using colours, the lamp can visualise the live status of the heat pump (from green when not working to purple when using external heat), the temperature inside or outside, the price of the electricity on Spot, etc.

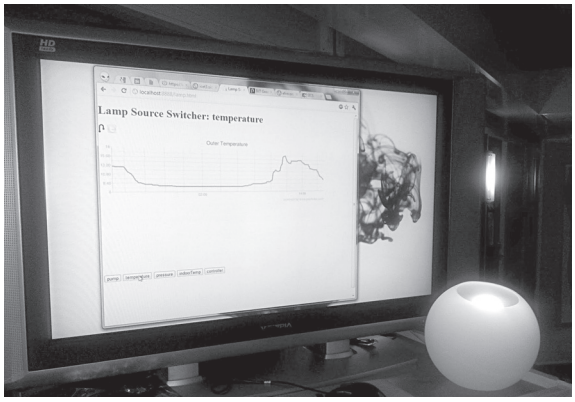


Figure 8. In the figure above, the lamp is tuned to the outside temperature and the user interface is shown on nearby TV for demo purposes. The UI uses the COSM connection to display relevant historical graphs.

VIII. Conclusions and Future Work

This paper has presented the context manager, a central hub designed both to provide contextual data to IoT applications and a storage for historical, present and future sensor and actuator data. The context manager is designed to lower the learning curve, letting less technically inclined people model and reason about their smart homes. Concepts such as pairing, conduits and triggers lean themselves easily to be controlled and specified via user interfaces rather than via the file formats that have been summarised in this document. As such, these concepts already contain parts of the logic that would control the flow of data between different objects, combined to both actuation and visualisation through external services. Possible extensions would consist in looking into visual programming efforts such as App Inventor[24] and ways to incorporate some of these ideas into the IoT domain. Already, colleagues have started to work on the

[17]Prices are scraped from http://www.nordpoolspot.com/ for historical reasons.

[18]The IR toy is a set of open source hardware and software available at http://dangerousprototypes.com/docs/USB_IR_Toy_v2 to record and (re)play the IR codes of most infrared-based remote controls.

building blocks of an "appification" of the home, i.e. the installations of "apps" that can control parts of your homes to attain some energy savings, while providing a (mobile) user interface to input settings and refine controlling. For this "appification" to take place, applications will need to be able to reason about the context in order to adapt to the specificities of as many homes as possible. They cannot rely on users or hard-coded objects, instead the location and search facilities of the context manager will be key to reasoning about the context and answers questions such as "give me all the lamp sources in that room" or "Have all inhabitants left home now?". In its current implementation, the context manager is starting to be able to provide an answer to this type of questions.

APPENDIX A
INTEGRATING WSN SENSORS AND ACTUATORS

Apart from the TinyNode deployment that has been described in section VII, additional Tcl scripts have been written in order to interface with IPv6-based WSN, one of the areas where the OS Contiki[22] is widely used. In meshing WSN, it is essential to restrict the size and number of packets to a strict minimum in order to keep power requirements low. The current implementation of these scripts relies on the HTTP capabilities of the motes. HTTP leads to sizable headers and the necessity to keep the TCP state across the network. Future directions will look into UDP[19] and WebSockets.

Scripts bridging motes to the context manager will receive or poll for mote data and push this data as updates to the field of one or several objects in the context manager. The simplest script will regularly poll for data at given motes with a given frequency. However, the more complex script is inspired by techniques initiated by CoAP[26]. It combines a UDP and HTTP servers, in order to both support regular HTTP POST and GET operations, but also to entertain WebSockets connections. On startup, the script contacts all relevant motes and subscribes itself (the proper root/details to the servers that it implements), together with a frequency for reception of data. Consequently, motes will, whenever needed push data to the script, which will forward it further to the appropriate objects of the context manager, depending on its configuration.

[19]Problems with the current UDP implementations in Tcl 8.6 (and in combination with IPv6) have unfortunately put part of the development on hold.

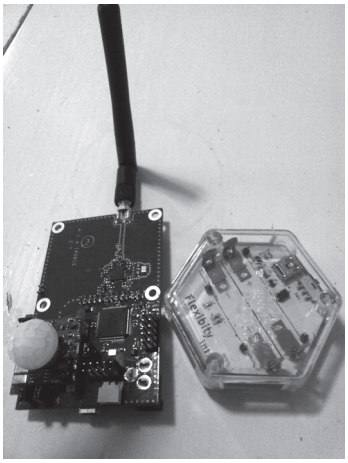


Figure 9. Two of the supported motes. To the left is a mote from Tyndall[25] that sports a stackable and pluggable interface for various sorts of sensors (temperature and humidity on the picture). To the right is a commercial mote from Flexibility (see http://www.flexibity.com/) implementing a thermometer, hygrometer and barometer.

ACKNOWLEDGMENTS

Most of the work has been sponsored by the European ARTEMIS project me^{3}gas, with valuable input from M. Westbergh (CRL Sweden), S. Duquennoy (SICS), J. Eriksson (SICS), P. Kool (CNet), P. Hansson(SICS) and L. Moore (Tyndall). Most of the code has been opened source at the following project location: http://code.google.com/p/efr-tools/

REFERENCES

[1] B. Welch, K. Jones, and J. Hobbs, *Practical Programming in Tcl and Tk*. Prentice Hall, 20 June 2003.

[2] K. P. Birman and T. A. Joseph, "Exploiting virtual synchrony in distributed systems," in *ACM Symposium on Operating Systems Principles (SOSP'87)*, pp. 123–138, 1987.

[3] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, ch. 5, pp. 76–106. 2000.

[4] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)." RFC 4627 (Informational), July 2006.
[5] M. Banzi, *Getting Started with Arduino*. Make:Books, O'Reilly Media, Inc., Aug. 2011.
[6] S. Monk, *Getting Started with .NET Gadgeteer*. Make:Books, O'Reilly Media Inc., 4 May 2012.
[7] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, 17 May 2001.
[8] K. Aberer, M. Hauswirth, and A. Salehi, "A Middleware For Fast And Flexible Sensor Network Deployment," in *Proceedings of VLDB'06*, pp. 1199–1202, 2006.
[9] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, "IrisNet: An Architecture for a Worldwide Sensor Web," *IEEE Pervasive Computing*, vol. 2, pp. 22–33, Oct-Dec 2003.
[10] D. Halvik, G. Schimak, R. Denzer, and B. Stevenot, "Introduction to SANY (Sensors Anywhere) Integrated Project," in *Proceedings of ENVIRONINFO*, Sept. 2006.
[11] P. Kostelnik, M. Sarnovsk, and K. Furdik, "The Semantic Middleware for Networked Embedded Systems Applied in the Internet of Things and Services Domain," *Scalable Computing: Practice and Experience*, vol. 3, no. 12, pp. 307–315, 2011.
[12] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Frystyk Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." W3C Recommendation, 27 Apr. 2007. http://www.w3.org/TR/soap12/.
[13] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)." W3C Recommendation, 26 Nov. 2008. http://www.w3.org/TR/xml/.
[14] G. Klyne and C. Newman, "Date and Time on the Internet: Timestamps." RFC 3339 (Proposed Standard), July 2002.
[15] E. Rescorla, "HTTP Over TLS." RFC 2818 (Informational), May 2000. Updated by RFC 5785.
[16] S. Sanfilippo and P. Noordhuis, "Redis," 2012. http://redis.io/.
[17] I. Fette and A. Melnikov, "The WebSocket Protocol." RFC 6455 (Proposed Standard), Dec. 2011.
[18] A. Presser, L. Farrell, D. Kemp, W. Lupton, S. Tsuruyama, S. Albright, A. Donoho, J. Ritchie, B. Roe, M. Walker, T. Nixon, C. Evans, H. Rawas, T. Freeman, J. Park, C. Chan, F. Reynolds, J. Costa-Requena, Y. Ye, T. McGee, G. Knapen, M. Bodlaender, J. Guidi, L. Heerink, J. Gildred, A. Messer, Y. Kim, M. Wischy, A. Fiddian-Green, B. Fairman, J. Tourzan, and J. Fuller, "UPnP Device Architecture 1.1," tech. rep., UPnP Forum, 15 Oct. 2008.
[19] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication." RFC 2617 (Draft Standard), June 1999.
[20] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace." RFC 4122 (Proposed Standard), July 2005.
[21] H. Dubois-Ferrière, L. Fabre, R. Meier, and P. Metrailler, "Tinynode: a comprehensive platform for wireless sensor network applications," in *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, (New York, NY, USA), pp. 358–365, ACM, 2006.
[22] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 20 Dec. 2004.
[23] S. Petai, "python-plugwise." Bitbucket Project, 20 Mar. 2011. https://bitbucket.org/hadara/python-plugwise/wiki/Home.
[24] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor*. O'Reilly Series, O'Reilly Media, Inc., 15 Apr. 2011.
[25] A. Lynch, K. Aherne, P. Angove, J. Barton, Harte S., D. Diamond, and F. Regan, "The Tyndall Mote. Enabling Wireless Research and Practical Sensor Application Development.," in *Adjunct Proceedings, Advances in Pervasive Computing*, pp. 21–26, May 2006.
[26] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," Internet Draft draft-ietf-core-coap-12, IETF, 1 Oct. 2012.

Dr. Emmanuel Frécon is a senior researcher at the Swedish Institute of Computer Science (SICS). He received his Ph.D. from the IT university of Gothenburg in 2004. He has (co-)authored a number of articles in books, refereed conferences and journals, as well as edited a book in the field of computer science. Across the years his research interests have slowly shifted from collaborative virtual environments to ubiquitous computing, not forgetting ambient displays and novel interaction techniques. He strongly believes in the feedback loop between technology and users.

Dr. Emmanuel Frécon is also an entrepreneur and has co-founded two companies. He is on leave from his second company, JoiceCare, where he worked as a system architect and CTO. JoiceCare sells products for the elderly market: a SIP-based video telephone and a video-based supervision system. He also believes that industry and research have a lot to bring to one another and intends to alternate workplaces as opportunities present themselves.

WTK for APWTCL

An implementation of TK like Widgets for APWTCL.

A paper for the Nineteenth Annual Tcl/Tk Conference

Abstract

During first half year of 2012 APWTCL (the successor of itcl in javascript) has been implemented and based on the javascript version two additional versions have been implemented for better native performance: APWTCL (Java) for Android based handhelds and APWTCL (Objective-C) for iPhone. To have running something comparable to Tk for APWTCL, there was the decision to use wtk from Mark Roseman (https://github.com/roseman/wtk) as a base. This package splits up the administration and data of a widget to be handled with Tcl (snit classes and objects) from the representation (displaying on the screen), which uses native support from the environment it is running on. Original wtk was dedicated to support javascript only, my implementation inserts an environment independent message interface (using a string based protocol) in between and then uses native support for displaying and handling the widgets and events. The action handling on the events is done by passing the information back to the Tcl part, which is modifying the (Tcl side) data and calling Tcl callback scripts if necessary.

Contact information

Arnulf Wiedemann

Lechstr. 10

D-86931 Prittriching

Email: arnulf@wiedemann-pri.de

1 The Idea

Already during implementing itcl in javascript there was the decision to use wtk as the frontend for GUI building. WTK (WebTk) is a Tk like implementation of some widgets (frame, entry, button, label, checkbutton and canvas) and a rudimentary implemenation of a grid manager. It is based on the idea to separate the data and administration part of the GUI from the presentation part. The communication between the two parts can be done in different ways:

- Using a direct function call to the wtk client side functions
- Using a client/server solution which is running the client part (the presentation part in javascript) in the browser and the server part (the Tcl part) on the serving machine

The base for the above implementation is a snit class widget, which can create and administrate a widget. It is (from the comment of the implementation of Mark Roseman):

A 'generic' widget object, which handles routines common to all widgets like assigning it an id, keeping track of whether or not it has been created, etc. Purely for convenience, we also include some code here that manages widgets that use -text or -textvariable, though not every widget will do so.

The “mega”widgets like frame, button, entry etc. are built with snit classes, which delegate a lot of functionality to the widget base class. Again a comment from Mark Roseman from the implementation:

Stuff for defining different widget types here. Note that all widgets are expected to implement the "_createjs" method. This is called by the generic widget code, and should return a Javascript command that can be used to create the widget on the web side of things (i.e. calls routines in wtk.js).

Widgets that support -text and -textvariable are expected to implement the "_textchangejs" method, which is called by the text handling pieces of the generic widget code, and should return a Javascript command that will change the text of the widget on the web side to match the current internal state of the widget here.

Widgets that receive events from the Javascript side are expected to implement the "_event" method, which is passed the widget-specific type of event and any parameters.

Wtk.js is a set of functions building the base of the repesentation/displaying (client side) part.

Communication between the administrative side and the displaying side is done using two global procs:

- toclient
- fromclient

These classes and procs have been used to implement a running version in itclinjavascript using a few javascript classes to allow the direct communication with the interpreter written in javascript.

2 How the current version started

As can be seen in my presentation APWTCL at the 10th European Tcl/Tk User Meeting ths year and on the wiki page there has been a reimplementation of itclinjavascript based on JimTcl. This version was the first version wtk from Mark Roseman as an interface for handling basic widget support. Following that APWTCL (Javascript) version there was another implementation of APWTCL in Java for supporting Android smartphones and an implementation in Objective-C for support of iPhones.

When arriving at the point for building support for wtk widgets the first approach was to use the administrative part of wtk mostly as it was and to replace sending of javascript code to the displaying part by a generic message (string based) interface.

On the client side there should be a small interpreter for decoding the messages and for switching and dispatching to the appropriate functions for doing the displaying and event handling work.

This was first implemented for the iOS version, later on there was a port of that code to Java to support Android.

3 The Message Interface

For the message interface a simple string based protocol was implemented, which had the the message itself and the parts encoded as length and info parts. A Message is generally built similar to a Tcl proc call in having a command and some parameters for the command. The command is normally a class object and the first parameter is the action to be executed on a GUI element. The parameter is normally a handle for the GUI element to be worked on and the other parameters are additional info for the action like option and value pairs. The first character of the message shows the type of the message normally M and there will eventually be a type E for end of a message block. That way the protocol is extensible with other types. Right now there exist no other types.

The layout of the message interface is as follows:

M<length of message>:<message>

<length of message> is the length of the following message text (not including the ":") as an integer number

and a message is composed of 1 .. n parts with the following layout:

<length of part>:<part>

<length of part> is the length of the following message text (not including the ":") as an integer number

<part> is a sequence of printable ASCII characters.

These messages are interpreted on the client side (or the part which acts as a client for example some class methods).

Some examples:

M53:9:wtkclient11:createLabel4:obj120:label: Hello Chicago

M33:9:wtkclient7:newGrid4:obj05:grid0

M38:9:wtkclient4:grid5:grid09:insertRow1:0

M48:9:wtkclient4:grid5:grid03:row1:010:insertCell1:0

M70:9:wtkclient4:grid5:grid03:row1:04:cell1:011:appendChild7:widgets4:obj1

M47:9:wtkclient12:createButton4:obj213:Hello Chicago

M70:9:wtkclient4:grid5:grid03:row1:04:cell1:011:appendChild7:widgets4:obj2

M38:9:wtkclient12:createButton4:obj34:Quit

M70:9:wtkclient4:grid5:grid03:row1:04:cell1:011:appendChild7:widgets4:obj3

4 The Client Side

The client side is responsible for creating and displaying the GUI elements like a button or a label.

The implementation of the GUI part started with the iPhone version, the Java version was done some time later.

Some details of the client side:

The client side is implemented as a class with methods for the GUI elements and other parts. There is one object of that class instantiated at the beginning and when starting the application the implementation of the toclient and fromclient methods is defined.

Toclient encodes the message and uses the instantiated client object as the object and calls the decode message implemented there.

A reference to the fromclient method is set by an appropriate setter call to the client object.

The client side decode method decodes and interprets the messages sent via the message interface

For example for this message:.

M53:9:wtkclient11:createLabel4:obj120:label: Hello Chicago

After decoding we get a Tcl like list with the following contents:

{wtkclient createLabel obj1 {label: Hello Chicago}}

The first two parts build the client objects method to be called (after some mangling): wtkclientCreateLabel and there are two parameters: obj1 and {label: Hello Chicago} for that message.

As iOS and Java both can call class method using a text string with reflection/selectors this is the technique used.

Method wtkclientCreateLabel is responsible for creating a GUI element label with the text: “label: Hello Chicago”

First approach for crating GUI elements was, to use the native GUI elements available on iPhone namely the UI* classes. Using that approach, there is a rather limited implementation of a button and label support. Rather limited in that respect only means there is no completely compatible environment available as for a Tk Button.

Instead of a mouse click there is the possibility to hit the button using a touch screen event. When this event fires a native method is called, which in turn can call another method (in our case come method inside the client class. This method is implemented to forward that “event” to the Tcl wtk part in calling the fromclient method with parameters. Via that way the notification for an event is reaching the Tcl part, which in turn can handle the administrative part of the event and eventually is sending back some other message to be handled, for example to change the text of a button when the button is hit.

5 Different Approach for GUI Elements

Very soon the implementation reached a point (at least on the iPhone side) where it was obvious, that there is a lot of functionality missing, when looking for Tk like widgets. That might be partially because Apple wants to look all their GUI stuff look like they want it to be dispayed.

At that time some experiments with OpenGL ES started. OpenGL ES is a cut down version of OpenGL running on iOS and on Android and as part of WebGL in javascript for browsers too.

The experiments were based on the idea to use screen buffer implementation of OpenGL ES as a base for displaying pixels on the screen and to use some primitive functions of OpenGL ES like drawing a line or a rectangle or a polygon and filling some area with colors. For displaying text the idea is to use a freetype font implementation, also available for the iPhone.

Using that approach, it would be possible to use 3-D elements to be shown and also rotation of text would be very easy using OpenGL Es functionality.

As OpenGL ES is also used as a base for some games on iOS the guess was, that it should be fast enough for the implementation of a Tk GUI.

Having some small knowledge of OpenGL from the implementation of ntk_widget the decision was made to give OpenGL ES a try, to see, what can be done using that.

6 Use of OpenGL ES

The implementation of OpenGL ES for iOS (iPhone) has a rather simple interface to work with. There is an OpenGL graphic context, which can be used to display OpenGL ES primitives like a screen buffer.

OpenGL ES also offers primitives for drawing lines and polygons and to fill areas. Areas are built using for example triangles (there is no support for rectangles, which can be built using two triangles). Lines and triangles are built using vertices. There is an advanced interface available for using arrays of vertices for building graphics elements.

There were some successful experiments in building a Tk button using two triangles for the inner rectangle part and using a combination of some lines for building borders.

It is possible to add an event handling function to be called when a user fires a touch screen event in hitting at some point on the screen. This event also contains the x and y coordinates, where the event happen, Using that information and knowing where on the screen is the area of the simulated button it is possible, to detect when the touch screen event was fired inside the button rectangle area.

When the touch event happened inside the button area it is possible to emulate the Tk like press and release events of a button in setting different border colors for the four borders built using some lines. That way it is possible to make the button look sunken or raised. Depending on the touch screen event.

Experimenting with that implementation the idea came up to eventually use the ideas behind themed Tk (tile or ttk). Looking at the implementation of ttk it seems to be feasible, to implement some modified version of themed Tk widgets using OpenGL ES as the graphic context for displaying the stuff on the screen. Going that way, it would be rather simple too to rotate elements including text. For displaying text there are still some experiments necessary as there is nreal experience yet on how freetype2 fonts are supported on the iPhone (iOS). There is a port/adaption af freetype fonts called freetype2 from David Petrie for iOS and there if a ftgl library called ftgles also from David Petrie which can be compiled, but I was not yet able to make a demo running also it can be linked and started, but the display stays black. Seems to be a problem of adapting to iOS5, as the original was designed for iOS4.

The implementation of themed Tk functionality itself seems to be straight forward, just a matter of doing the work in Objective-C respectively Java.

During testing the implementation there were some problems in building rounded corners for button corners. There have been implemented some different algorithms, but without final success. There were always problems with rendering in getting something looking nice. There is some more time needed to find something suitable, as it seems to be possible looking at iOS button with rounded corners. It has to be found out, if there is a problem with the algorithms used or with OpenGL ES or how to use the same rendering as iOS UI functions/algorithms are using.

7 Status

It seems the suggested way is doable.

It also seems, that using OpenGL ES as the base is a rather platform independent way.

Making work freetype fonts and ftgles to be done.

The complete implementation of themed Tk support is not yet started, it should be relatively easy using enough time to do the work, as the existing implementation for Tk is available.

It seems not to be possible to use native fonts with OpenGL ES.

The implementation of the widgets using OpenGL ES commands is at the beginning.

There is the need for test cases..

There is also the need for examples/demos.

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 2
November 14, 13:15-14:15

Toward RESTful Desktop Applications

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

Abstract

The REpresentational State Transfer (REST) architecture includes**:** the use of Uniform Resource Locators (URLs) to place a universe of data into a single namespace; the use of URL links within the data to allow applications and users to navigate the universe of data; HTML/CSS for the presentation of data; a limited set of operations that are available for all URLs; multiple content types; and content negotiation when retrieving data from a URL. REST is primarily used in web applications; however, pure desktop applications can also benefit from RESTful concepts and technologies, and especially from the integration of web-like technologies with classic application software. This paper describes how REST concepts and technology have been used in the Athena simulation to present a vast sea of heterogeneous data to the user.

1. Background

The Athena Stability & Recovery Operations (S&RO) Simulation is a model of political actors and the effects of their actions in a particular region of the world. The region is divided into neighborhoods, in which reside various civilian groups. The actors have a variety of assets, including money, military and police forces, and means of communication, which they use to achieve their political ends. The extent to which they succeed depends on the attitudes of the civilians, which change in response to current events. The model runs for a period of months to years, and produces a vast quantity of data, all of which needs to be presented to the analyst in some form or other.

The Problem

Athena stores most of its data in an SQLite3 *run-time database* (RDB). In Athena V2.0 most data was made available to the user by taking the output of a particular database table or view and throwing it into a `tablelist`-based browser.[1] Such a tabular display is useful; but when the information about a particular entity, an actor, say, is extremely heterogeneous, one tabular display cannot tell the whole story. It is possible to collect together the information about the actor by looking across a number of tabular browsers…but not surprisingly our users thought that the application ought to be doing this for them.

If only there was an easy way of presenting heterogeneous data to the user, while taking advantage of relationships within the data as an aid to navigation….

The Solution

HTML/CSS is a powerful, well-understood means of presenting heterogeneous data to the user. Uniform Resource Indicators (URIs) are a powerful means of identifying specific resources to present to the user from within a vast sea of such resources. Links to URIs embedded in the data are a powerful means of allowing the user (or the application) to navigate the sea of data. The resource pointed at by a URI can exist in multiple content-types; through content negotiation, the client can retrieve the content-type that is most useful for its purposes. These have generally been used in web applications. However, there is no reason why these concepts cannot be fruitfully used in the desktop environment within the context of a single application with no network interfaces, when the application's data model calls for it.

2. The Desktop REST Architecture

HTML, URIs, and the rest of the web technologies described above were created to support an architecture called REpresentational State Transfer (REST) [2]; an application that uses REST is called a *RESTful application*. REST is a web architecture; this section describes how we have modified the basic concept to create a desktop REST architecture within our application.

REST: A Summary

A RESTful application, or client, accesses *resources*: collections of data, or indeed any kind of entity, by means of *Uniform Resource Indicators* (URIs), of which there are two kinds, *Uniform Resource Locators* (URLs), for resources that can be located and retrieved on-line, and *Uniform Resource Names*, which are unique names for entities that exist off-line.

The client accesses these resources by means of a handful of verbs, which in principle apply to all resources. In a traditional REST app, which uses HTTP for its transport, these are usually GET, PUT, POST, and DELETE.

The resources are provided to the client by a *server*, and the server provides the data in a form called the *content type*. Content types are typically expressed as MIME types such as `text/plain` and `text/html`. A single resource might be available in any number of content types, and the precise data returned for the resource might differ from one content-type to the next. (E.g., `text/html` contains structure in a way that `text/plain` does not.)

The client accesses a server using an *agent*. The client gives the agent the URI of a resource, and a verb, and the agent locates the server and accesses it on the client's behalf. In particular, the agent handles *content negotiation*: given the content types the client is prepared to handle, the

agent works with the server to provide the resources to the client in the content type it would most prefer.

A resource's content frequently contains URIs linking to related resources. The client can make use of these URIs to navigate the sea of resources.

The most common content type is `text/html`, because it provides a way to display the resource data attractively and allows the user to navigate the data space by clicking on links. These days, HTML documents typically use Cascading Style Sheets (CSS) for formatting and Javascript for interactivity. In a Tcl/Tk application, naturally, Tcl replaces Javascript.

These concepts and technologies provide just the thing to display heterogeneous, highly linked data to the user.

Why Not a Web App?

The advantages of the REST architecture would seem to be an argument for implementing Athena as a web application, yet there are compelling reasons for not doing so.

- Athena already exists as a single-user desktop application; moving to the web would change the architecture considerably.
- Network interfaces come with security headaches. And although Athena is not classified, it is often used in classified environments where network resources are tightly controlled and security is taken *very* seriously.
- Ease of installation is key; we do not want to require the users to install a web server. We could work around the installation issue by embedding something like TclHTTPD in Athena; but that still leaves us with the security headaches.
- We've not been asked to, nor do we have funding to make such significant changes, or to come fully up to speed on robust, secure web applications.

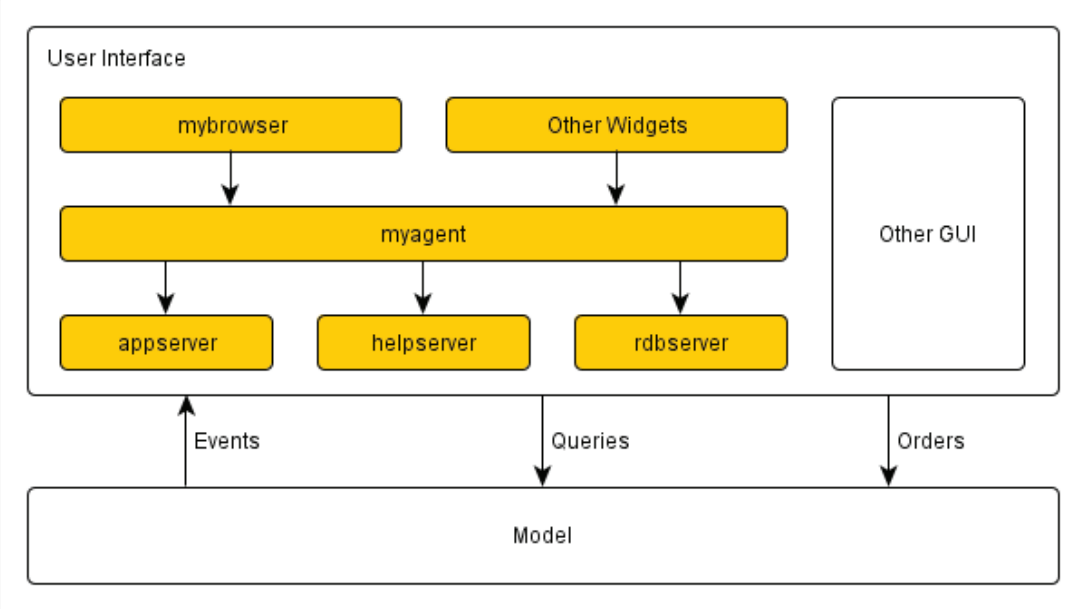
Adapting REST to the Desktop

So the question becomes, how do we use these RESTful concepts in a desktop application? We need to:

- Define a set of URIs that give access to various application resources.
- Determine the relevant content types. We use standard content types like `text/html`, but also types relevant to the desktop environment, such as `tk/image` and `tk/widget`.

- Implement a content server, and an agent with which to access it. Because the server resides within the application itself, access can be synchronous; the protocol reduces to a set of procedure calls.
- Specify tools for parsing URIs. We use the uri package from Tcllib.[3]
- Create tools for generating HTML output. (Yes, I wrote yet another HTML-formatting module. It's just something I do.)
- Choose a widget for displaying HTML/CSS
- Implement a web-browser-like mega-widget on top of TkHTML 3.0.
- Implement other widgets that can take advantage of server content.

With the RESTful components added, Athena's architecture is as shown in the following diagram; the new components are shown with a shaded background.



The Model represents the non-GUI portion of the application, including all management of scenario data and the simulation proper. As described in [4], Athena's User Interface interacts with the Model via three mechanisms. First, the UI can query the Model in any way it likes, provided that the queries do not affect the content of the Model in any way. Second, it can send *orders* to the model; all changes to Model content and operation are triggered by these orders. Third, the Model can send *events* to the UI, to notify it of particular happenings within the model. This portion of the Athena architecture remains unchanged from previous versions.

As a consequence of this existing architecture, we have not implemented the PUT, POST or DELETE verbs of the REST architecture; the existing mechanisms handle these operations

perfectly well. Instead, we have focused on the GET operation, which is what we chiefly need to present information to the user.

At present, Athena includes three servers. The `helpserver` serves up on-line help pages from a pre-compiled help database. The `rdbserver` provides access to the schema and content of the application's run-time database as an aid to development and debugging. The `appserver` is the most important of the three, as it provides access to the Model's resources. These servers are all instances of the `myserver` type.

Each of these servers is registered with the `myagent` module; instances of `myagent` provide GET access to the servers, and also do content negotiation.

Instances of `mybrowser` can be used to browse the content of these servers in the usual way; and there are other widgets that access the servers as well.

3. Displaying HTML/CSS

Desktop REST stands or falls on the application's ability to display HTML content. And in order to display HTML content, or at least HTML-like content, in a Tcl/Tk application, you need to have an HTML widget. There is no perfect choice; this is a place where Tcl/Tk is sadly lacking. The available options are these:

- Solutions based on the Tk `text` widget
- TkHTML 2.0
- TkHTML 3.0 [5]
- A wrapper around Gecko or some similar engine HTML engine.

It is possible to do a mostly adequate job of displaying an early version of HTML in a Tk `text` widget; it handles links and interaction perfectly well, and it can even display images and embedded widgets. HTML-style tables are a problem, however, and tools to position images and embedded widgets precisely relative to the text (e.g., wrapping paragraphs around an image) are lacking. In short, the Tk `text` widget is a solution, but only a mediocre one for this purpose. (Were we to use it, we'd probably abandon HTML in favor of a Tcl-based presentation language, to avoid parsing.)

Athena 1.0 and 2.0 had a help browser based upon TkHTML 2.0. It is stable, having been abandoned long ago, but it is highly quirky and its HTML support is archaic. Font support is problematic; for example, you can have monospace type or bold type, but not both at the same time. It claims to support embedded widgets but in our experience all attempts to do so end in a

crash. In our experience TkHTML 2.0 edges out the Tk `text` widget for display of rich content, primarily due to its support for tables, but it is not very satisfactory.

Another option is TkGecko [6], a Tk wrapper for Mozilla's Gecko HTML engine. It is clear from the TkGecko paper that Gecko is very much a moving target, and that wrapping it in a robust way is by no means easy. It would be an interesting choice if we wished to display live web content from over the network, but we do not; and stability is crucial.

TkHTML 3.0 is an HTML/CSS renderer implemented as the basis for a Tcl/Tk web browser. Abandoned some years ago, it has not kept up with the latest web standards. It has more than enough horsepower for displaying application data, however, including tables, embedded images, embedded widgets, and complex formatting. The bare widget lacks event bindings and other features that were provided by the web browser within which it was to be embedded, but once these are provided it becomes quite satisfying to use. It is fast, versatile, and sufficiently stable for our use, and is what we have opted to use.

4. The URI Scheme

Athena uses two distinct URI schemes, neither one of which is found in the wild: the `my://` scheme and the `gui://` scheme.

The `my://` Scheme

The most usual URI scheme used in Athena is the "`my://`" scheme, which is a simplification of the familiar `http://` scheme. `my://` URLs have the same syntax as `http://` URLs, with the unnecessary parts (port numbers, passwords, etc.) omitted:

`my://`*`server/path…?query#anchor`*

Here the server is the name of a `myserver` registered with `myagent`, and the path, query, and anchor are defined as usual.

We chose the name "`my:`" for this scheme because the named resource belongs to the application itself, rather than to some other entity out in the network. We considered abusing the `http://` scheme but rejected this for two reasons. First, we wanted to make it absolutely clear that Athena has no network interface; it is not pulling resources down from the web. Second, it allows us to modify the standards for `http://` URLs without causing confusion to future developers.

The `gui://` Scheme

The `gui://` scheme is a set of Uniform Resource Names (URNs) for entities in the Athena GUI. Links using this scheme are not handled directly by the `myagent`/`myserver` infrastructure; instead, the `mybrowser` widget hands them to its parent object via a callback, which hands to the application for handling. The upshot is that the user can click on a link in a browser, and the application will take them to some other tab in the GUI, or pop up an order dialog. For ease of parsing, the `gui://` scheme also uses a subset of the usual `http://` syntax.

5. Content Types

The `myserver` component allows each instance of the server to serve up content of any imaginable type. The standard MIME types `text/html` and `text/plain` are used for HTML and plain text context respectively; for consistency, application-specific content types are named in the same style, with "`tk/`*`type`*" used for Tk-specific content and "`tcl/`*`type`*" used for other kinds of data. The application-specific content types currently in use described in the following subsections.

The `tk/image` Content Type

The content consists of the name of a Tk image. An instance of `mybrowser` can display `tk/image` content directly and as the `src` of an HTML `<img>` tag.

The `tk/widget` Content Type

The content consists of a Tcl script to create the widget so that it can be displayed in an HTML page. The HTML `<object>` tag is used to embed widgets in pages; for example, the following HTML embeds a time plot in the page:

```
<object data="my://app/plot/time?start=2+vars=basecoop"
width="100%" height="3in"></object><p>
```

The query portion of the URL specifies the variables to plot, and the start time of the interval for which they should be plotted. The server uses these to customize the widget options, and then returns the script to create the widget. (The TkHTML 3.0 widget handles the width and height itself.) For example:

```
timechart %W -vars basecoop -start 2
```

The server doesn't know the window name to use, so it inserts a "`%W`" in place of the window name. The mybrowser substitutes in the window name and creates the widget, which then appears in the web page.

The Netscape Tcl plugin was never so easy.

The `tcl/enumlist` Content Type

This content type is simply a Tcl list of enumerated values; it is usually used to populate pulldowns in HTML forms, but can also be used by non-browser widgets.

The `tcl/enumdict` Content Type

This content type is similar to `tcl/enumlist`, but the value is a dictionary of enumerated values and their human-readable equivalents. It is also used to populate pulldowns in HTML forms.

The `tcl/linkdict` Content Type

This content type is used to represent trees of links. A `tcl/linkdict` is a nested dictionary mapping URLs (relative to the current server) to link metadata, primarily a human readable `label` and a `listIcon`, a Tk image to display next to the label. As such it represents one node in the tree, and its immediate children. By recursively retrieving `tcl/linkdict`s for the URLs, a component like the `linktree` widget can build up a tree of model entities or help pages.

6. Software Components

The Athena infrastructure includes the following software components.

The `myagent` Component

The `myagent` component is responsible for managing all interaction between clients and the various `myserver` instances. Servers register themselves with the `myagent` module, and instances of `myagent` retrieve data from the servers, doing all necessary URI resolution and content negotiation.

When creating an instance of `myagent`, the client specifies the content types it is prepared to handle, and the default server to contact:

```
myagent $agent \
    -defaultserver app \
    -contenttypes {text/html text/plain}
```

The client can then retrieve a URI's content as follows:

```
set cdict [$agent get $url]
```

The agent will throw a NOTFOUND error if the data cannot be retrieved; otherwise, it returns a dictionary with three keys: `url`, `contentType`, and `content`, which the client can do with as it pleases. If desired, the client can specify the desired content type or types explicitly:

```
set cdict [$agent get $url tk/widget]
```

Instances of `mybrowser` will normally accept `text/html`, `text/plain`, and `tk/image`, but will explicitly ask for `tk/widget` when handling an `<object>` element.

The `myserver` Component

Instances of the `myserver` component are registered with `myagent`, and thus become accessible to the application. Each instance of `myserver` defines the set of URLs that it can handle, and the content types for each:

```
myserver ::appserver
myagent register app ::appserver

appserver register / {/?}   \
    text/html [list /:html] \
    {Athena Welcome Page}

appserver register /actor/{a} {actor/(\w+)/?} \
        text/html [list /actor:html]          \
        "Detail page for actor {a}."
```

Each of these calls technically registers a pattern, rather than a specific URL; the handler handles all URLs that match the pattern. The first pattern registered above is simply "/", the top-level page for the server; the second registers a URL with a place holder for an actor's symbolic name.

For each pattern, we specify a unique name, e.g., `/actor/{a}`, and a documentation string; these are used in the server's `/urlhelp` page, which every instance of `myserver` provides automatically. Next, we provide a regular expression, which matches URLs of the correct

pattern. (Note that "^" and "$" are added to the expression automatically.) The regular expression may include parentheses to indicate match parameters; these will be provided to the handler. Finally, for each URL we specify a set of content types and handler commands.

Thus, when the server is given a URI it matches it against the registered resources; if a match is found, and the URI has a compatible content type, the handler for that content type is called. For example:

```
proc /actor:html {udict matchArray} {
    upvar 1 $matchArray ""

    set actor [string toupper $(1)]

    if {![actor exists $actor]} {
        return -code error -errorcode NOTFOUND \
            "Unknown entity: [dict get $udict url]"
    }
    .
    .
    .
    return $content
}
```

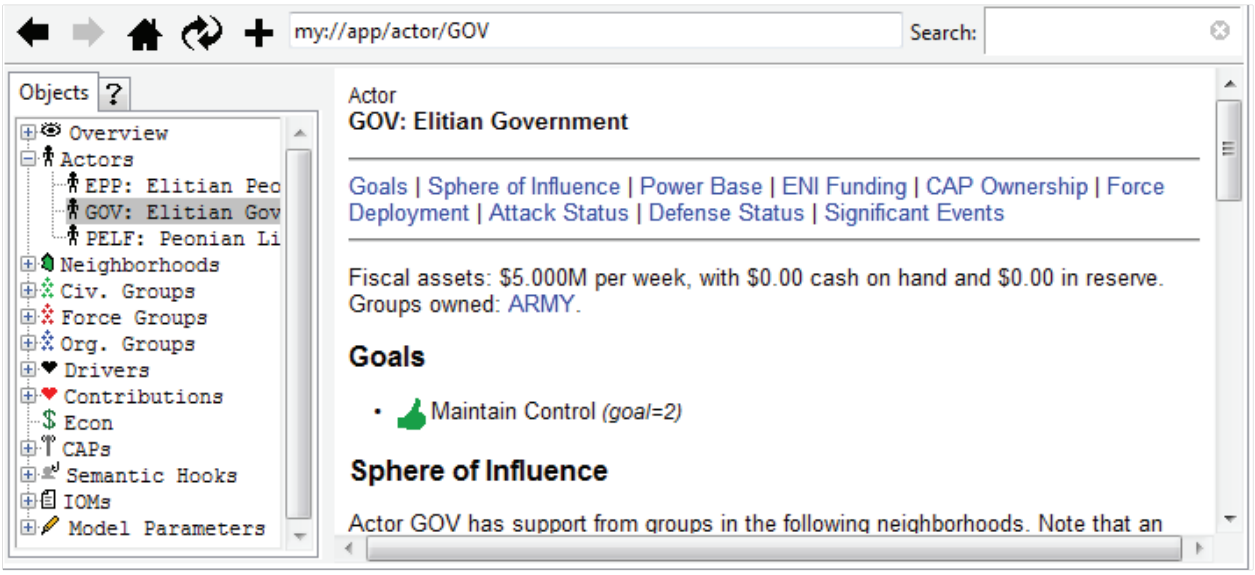
The *udict* parameter is a dictionary of the components of the URI: the path, the query, and so forth, as returned by `uri::split`. The *matchArray* parameter is the name of an array variable containing the matches from the regular expression; in this case, the actor's symbolic name. The handler may make use of both the *udict* and the *matchArray* or neither.

The `mybrowser` Component

The `mybrowser` component is a web-browser-like widget built on top of TkHTML 3.0. It has its own instance of `myagent`, and thus can retrieve resources from servers. In addition to the normal browser navigation tools, it has the following capabilities:

- Display `text/html`, `text/plain`, and `tk/image` resources.
- Embed `tk/widget` content in `text/html` pages, when specified using the `<object>` tag.
- Support HTML forms.

The following figure shows an instance of `mybrowser`. The toolbar, scroll bars, html pane, and the paned window widget that allows the side bar to be resized, are all provided by `mybrowser`; the sidebar itself is an instance of `linktree` (see Section The linktree Component).



The browser's support for HTML forms is robust but idiosyncratic. Athena has its own set of data entry field widgets which do not entirely match up to the standard HTML form fields; consequently, it provides its own mapping of `<input>` types and attributes to data entry fields, ignoring the standard HTML input types completely. For example, this HTML creates a form consisting of a single "enum" field, essentially a pulldown containing items from an enumerated list. The list of values comes from URL `my://app/enum/sortby`, which must provide content type `tcl/enumdict`. The default value for the pulldown is "`name`".

```
<form action="my://app/page/Cal" autosubmit="yes">
<label for="sortby">Sort Cells By:</label>
<input name="sortby" type="enum" content="tcl/enumdict"
       src="my://app/enum/sortby" value="name">
</form>
```

The form looks like this in use:

When the form is submitted, which will happen automatically when the user selects a new value from the pulldown, the form's values will be appended to the `action` URL as a query, and the URL will be retrieved:

```
my://app/page/Cal?sortby=name
```

At present, `mybrowser` supports `enum`, `text`, and `submit` input types.

The `myhtmlpane` Component

The `myhtmlpane` component is essentially a `mybrowser` without the navigation controls. It is intended to display a single page, retrieved from a `myserver`, as an alternative to a window defined using normal Tk widgets. If the user clicks on a link on the page, the URI is passed along to the application for display in the application's main browser.

The linktree Component

The `linktree` component is a Tk treectrl widget configured to display a tree of resource links retrieved from a given URL. The widget retrieves its top-level items from the URL, and then works its way recursively down the tree, retrieving `tcl/linkdict` content at each node. The descent ends when a leaf no longer has any `tcl/linkdict` content associated with it. Optionally, the `linktree` can retrieve content for non-leaf nodes when they are first expanded.

The sidebar in the browser screenshot in Section The mybrowser Component shows a linktree of simulation entities.

The htmlframe Component

Although not actually part of the RESTful infrastructure, the `htmlframe` widget has proven to be a useful addition to the toolkit. It is simply a TkHTML 3.0 widget configured to layout its children according to an HTML layout string. For example:

```
htmlframe .f
ttk::entry .f.first
ttk::entry .f.last

.f layout {
    First Name: <input name="first"><p>
    Last Name: <input name="last"><p>
}
```

It includes a `set` method to set attributes of HTML elements by `id`; thus, the application can customize the appearance by setting CSS classes or styles on particular elements dynamically, or simply by providing a new layout. And since the TkHTML 3.0 widget supports scrolling, it is easier to create a scrolling window than it is using a standard `frame` widget.

This can be a much simpler way to create a complicated GUI layout than using the normal Tk geometry managers.

7. Status and Future Work

The infrastructure described in this paper is currently in use in two applications: the Athena simulation proper and in a separate development tool used to debug certain kinds of models. It has proven to be powerful, effective, and easy to use. The Athena application defines three servers and over sixty distinct URL patterns, many of them with placeholders. Many pages use forms and embedded objects, and that number is expected to increase over time.

It is possible that future applications may opt to extend the `myagent`/`myserver` pair with PUT, POST, and DELETE operations, and make use of these instead of Athena's existing "order" mechanism for editing and creating application data. Such an application would be truly RESTful, rather than merely "accidentally RESTful", as now.

8. A Bit of Advocacy

Tk needs a robust, solid, well-documented HTML widget for uses like those shown here, and the existing TkHTML 3.0 widget makes a good starting point. The secret is to stop chasing the big browsers; we will never have enough development horsepower to keep up with Mozilla, Microsoft, and Google, and even if we could produce a widget that was completely up to date and could display any HTML page on the web, it would be out-of-date in months, if not weeks.

But this is OK. An HTML widget need not be capable of doing everything Firefox does to be useful to the application.

9. References

[1] Nemethi, Csaba, Tablelist Widget, http://www.nemethi.de/.

[2] Sletten, Brian, "Resource-Oriented Architectures: Being 'In the Web'," in *Beautiful Architectures*, pp 89-109, 2009, O'Reilly & Associates, ISBN: 978-0-596-51798-4.

[3] Kupries, Andreas, and Ball, Steve, `uri` URI Utilities package, found in Tcllib, http://tcllib.sourceforge.net/doc/uri.html.

[4] Duquette, William H., "The State Controller Pattern: An Alternative to Actions", 17th Tcl/Tk Conference, http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/WillDuquette/Statecontroller.pdf.

[5] Kennedy, Dan, TkHTML 3.0 Widget, http://tkhtml.tcl.tk/tkhtml.html.

[6] Petasis, Georgios, "TkGecko: Another Attempt for an HTML Renderer for Tk", 17th Tcl/Tk Conference, http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/GeorgePetasis/TkGecko.pdf

10. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 4
November 15 10:45-12:15

KineTcl

Andreas Kupries ActiveState Software Inc. 409 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

This paper describes a package enabling Tcl scripts to talk to Microsoft's `Kinect` and related devices.

Technically `KineTcl` is a binding to the `OpenNI` framework and thus provides access to all depth sensor devices for which a sensor plugin exists. The best known device so far in that category is the `Kinect`.

The paper will describe the internal structure of the package (i.e. how it matches to the `OpenNI` API, and weaves both C and `Tcl` [12] together to make use of each others strengths) and point to supporting packages and tools used in the implementation.

1. OVERVIEW

`KineTcl` [2] is a new Tcl package providing a binding to Microsoft's `Kinect` [9], and related devices.

The project was started at the behest of the National Museum of Health and Medicine, Chicago[1] (short: `NMHMC`) for use in its exhibition space as one of the pieces of software linking real world activities and actions to interactive virtual displays.

Research into existing open source software for `Kinect` located two existing projects, `OpenKinect` [5] (aka `libfreenect`), and Open Natural Interaction (`OpenNI` [6]).

`OpenKinect` was created by the OSS and OSH communities through reverse engineering the Kinect's USB protocol. It is a low-level library providing access to the device without having to care about this USB protocol and the like. While not quite as low-level as a driver, it is not much higher. The developers have planned an analysis library for higher level operations (e.g. user detection and gesture recognition) but this was not yet implemented at the time of the research.

`OpenNI`, is a framework abstracting away from hardware devices and image processing for particular tasks (like user-, hand-, and skeleton-tracking). It was created and is maintained by `PrimeSense` [8], the developer and manufacturer of the depth sensor used in the Kinect. `OpenNI` is also "an industry-led, not-for-profit organization formed to certify and promote the compatibility and interoperability of Natural Interaction (NI) devices, applications and middleware."[6]. Both the framework itself and a generic sensor driver "node" for the `PrimeSense` sensor are available in source, under the LGPL. A derivative of the latter, specialized to the `Kinect` is available on github[10].

At this point of the research both possibilities were seen as roughly equivalent.

`OpenNI` was chosen because of the existence of the `NITE` [11] extensions, encapsulating all of the necessary higher level algorithms (i.e user detection, skeleton/joint tracking, gesture recognition, etc).

Given the time frame of the project (started in January 2012, a working system needed by May) it was considered difficult or impossible to invent and write such algorithms from scratch, as would be needed when using `libfreenect`. Having access to these through `NITE` outweighed the consideration that this part of the system is only available in binary, and not in source.

The next chapter gives a general overview of `KineTcl`'s design, implementation, and features. Following that, chapters 3 and 4 discuss limitations, possible applications and future directions for the package.

2. DESIGN & IMPLEMENTATION

2.1 OpenNI

`OpenNI`'s API is written in C, with an underlying class hierarchy [1] where the leaves represent the various data streams coming from a depth sensor, and the higher classes provide the general functionality and APIs. This is shown in figure 1. Note that the classes not only represent data streams from physical sensors, but also data coming out of higher level algorithms like user detection and tracking (i.e. virtual sensors).

These APIs contain the mandatory minimum supported functions for each class. For sensors going beyond these, `OpenNI` defines a series of standard "capabilities" they may provide. From a different point of view these could be called aspects, or mixins. As an example, figure 2 shows the capabilities which are defined for user detection and tracking.

For full details, see `OpenNI`'s reference documentation[7].

2.2 Basic Design

Generally all `OpenNI` classes and instances are represented as classes and instances to the Tcl script as well. Whenever

[1] Underneath the C API is actually C++

Tcl '2012 Chicago, IL, USA

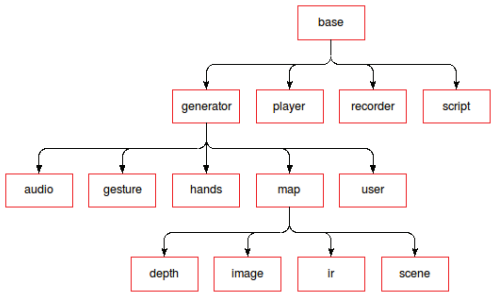


Figure 1: OpenNI class hierarchy

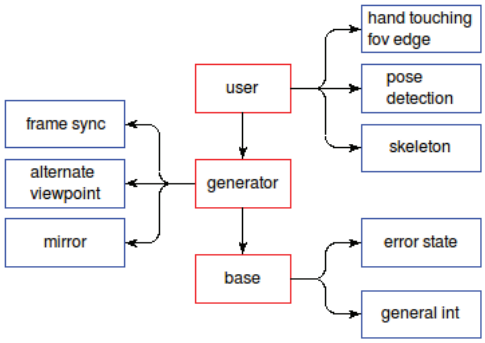


Figure 2: OpenNI User Tracking Capabilities

we mention a class in the future, we will also specify which of the three layers (`OpenNI`, C, or Tcl) we are talking about if it is not clear from the context.

Following the spirit of Poli-C [13] the binding is written using layers, with a low-level C layer implementing only the bare necessities which are then glued by the Tcl layer into the final user-visible API.

As mentioned, the C layer wraps each `OpenNI` "class" (which includes capabilities) into a Tcl class command whose methods map pretty much directly to `OpenNI` API functions. This is very much like Tk widgets. However, these classes do not know about the class hierarchy and superclasses. Each C class implements a binding to just the methods of their `OpenNI` class without regard for inherited methods.

This layering and the connections between the parts in the different layers is shown in figure 3, using the stack of classes for "depth image generator" nodes as example. We see not only the classes, but also the inheritance relationships (in blue), including the fact that `KineTcl`'s C layer does not use inheritance, and the use of instances (in red). The Tcl level depth image instances contain the C level instances of their class and all the required superclasses, which share the `OpenNI` handle for the node. This last point will be explained further in section 2.3.

This, and the mixin of the supported capabilities, is all handled in the Tcl layer. Here all the underlying classes are wrapped by `TclOO` [14] classes which instantiate all the required C classes so that the user may have access to the full set of methods, direct and inherited. The connection from the externally visible methods to the C methods is done through TclOO forwards, which also allows us to hide

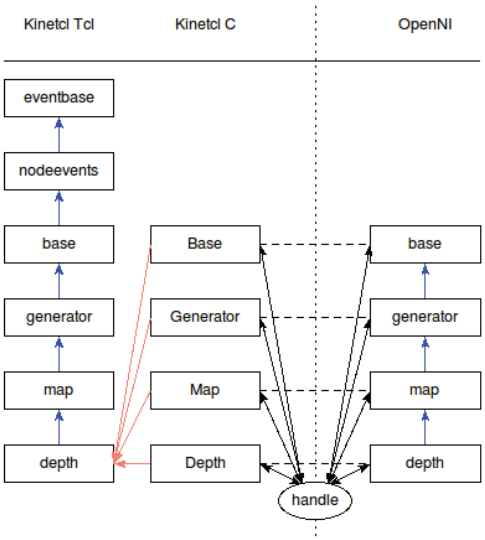


Figure 3: Package Layering

all special C methods needed by the Tcl layer which are irrelevant from the user's perspective. This includes, for example, the various introspection methods used to manage callbacks/events and capability mixing.

2.3 Object Construction

One tricky point in all this is that the various C instances constructed for the Tcl instance all have to operate on a single `OpenNI` handle for the object in question (see figure 3). How do we disseminate this information?

First, only the leaf C classes can create a new handle, a property the binding inherits directly from `OpenNI`. Knowing that the Tcl glue will construct the leaf first then walk up the Tcl class hierarchy to construct the required C level superclass instances, the code for a leaf class saves the obtained handle into a per-interpreter structure of the package. The superclasses' code then retrieves the handle from there. Doing things in this manner avoids having to expose and pass a C level pointer through the Tcl layer.

It should be further noted that the C base class provides a special method (`@unmark`) to explicitly clear this handle store. This is not done automatically by the C base class during its construction, because of the capabilities. The handle storage has to be kept around until the Tcl glue has mixed them in, thus the responsibility to signal its release falls to the Tcl layer.

2.4 Object to Handle Conversion

Another issue which has to be solved in the cooperation of C and Tcl layers is that various `OpenNI` (and thus C) methods take a second handle as input, requiring us to convert from a Tcl object command to the underlying `OpenNI` handle.

At the C level, this is managed by calling out to the Tcl procedure `::kinetcl::Valid` which performs both validation of a Tcl_Obj* as a proper Tcl object (command) and its conversion, leaving the resulting `OpenNI` handle in the same storage area as used during object construction. The

caller can retrieve it from there after the procedure returns.

At the Tcl level, `::kinetcl::Valid` uses a dictionary of the active instances managed by the base class to validate the argument as a Kinetcl object. For the arguments passing this test `::kinetcl::Valid` then uses its knowledge of the Tcl object internals, namely the existence and name of the C base class instance in the object to directly access it and invoke the special C method (`@mark`) which will store the desired handle in the storage area for the C level to pick it up from.

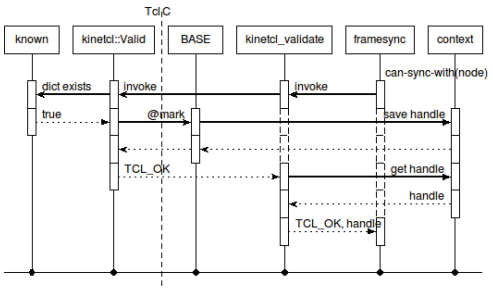


Figure 4: Object to Handle validation and conversion

Figure 4 shows all of the above in a UML sequence diagram.

2.5 Events and Callbacks

The last area of cooperation to talk about are the 34 `OpenNI` callbacks. Unfortunately, they are invoked from `OpenNI`'s internal threads, making it impossible to use them "as is" (i.e. let them directly call up into Tcl).

This issue was mainly solved by converting the callbacks into events, for which we have Tcl API functions to safely enqueue them regardless of which thread they come from and are going to. However, even with that we had two problems left.

First, one of the callbacks is very high-rate, generated several times per second. I am talking about the 'new frame' event for all the map generators, signaling the presence of a new image frame (image, depth, IR, ...). Because a single such signal is good enough this event is throttled by allowing only one per object into the event queue and discarding the remainder until the event in the queue has been processed.

The other remaining issue arises again from the fact that events are generated by threads outside of Tcl's control. It means that new events not only can, but will arrive while Tcl is processing the queued events. Without safeguards Tcl's event queue will never be empty, and the processing loop will never end, starving out idle-events processing.

While a solution was found for this, it doesn't look very nice. Readers of the example applications will see code like that shown in listing 1. This is essentially an emulation of Tcl's event loop using `while` and `update,` and inserting the necessary calls to (a) drive `OpenNI`'s processing (`waitUpdate)` and (b) safeguard (estart, estop) Tcl's event loop while processing events. `estop` causes the system to defer incoming events into a spill-over queue, whereas `estart` restores regular processing and moves all defered events into the main Tcl event queue.

With the pressure for getting a working system now gone,

Listing 1: Event loop

```
while {1} {
   kinetcl waitUpdate
   kinetcl estop
   update
   kinetcl estart
}
```

better solutions for the event integration should be investigated (e.g. Tcl's API for "Event Sources").

While `OpenNI`'s C API for callbacks allows the registration of an arbitrary number of actual callbacks for a specific event the C classes were kept simpler, handling only one actual callback per specific event, managed by associated set and unset methods.

The distribution of events to many observers is then again handled by the Tcl glue code, in two TclOO classes which are superclasses to the nomimal Tcl base class for `OpenNI` instances (see figure 3). These two classes, `kinetcl::eventbase` and `kinetcl::nodeevents,` provide a more event-like API, where users can `bind` to and `unbind` from events. The various Tcl sub-classes register the events they support with them, after using the C classes' method introspection facilities to determine this set. A small detail of the implementation is that a C level callback is set if and only if observers have been bound to the event it will be invoked for. This part of the functionality relies on a feature of the internally used `uevent` [15] package. That is, its ability to watch for and invoke commands when event bindings are set and removed (available since version 0.3.1).

2.6 Implementation

Now, how do we implement 39 C classes (14 core, 25 capabilities) quickly yet safely, especially in light of the large amount of virtually identical boilerplate needed to manage the class and instance commands and associated data structures?

By automating as much as possible.

Thus, a significant part of the time was not spent on writing the binding directly, but on writing the `critcl::class` generator package to encapsulate all the boilerplate and its templating. Having this generator in place, writing the binding became almost trivial, at least in most places. An only slighly abbreviated example is shown in listing 2.

Please note that the code in this listing represents the state of the Kinetcl head and of the `critcl::class` head officially released with `critcl 3.1 [4],` which also makes use of the additional features for custom argument and result type processing.

The code currently in use by the NMHMC, found at the tag "nmhmc" in the `KineTcl` and `critcl repositories` is less streamlined, containing various argument- and result-processing C code fragments multiple times. For the class shown, the difference is only about half a kilobyte (4 versus 4.5 KB). This class gets converted into roughly 25 KB of C code. From this we can estimate that about 84% of the result is boilerplate code, generated, instead of manually written.

This was further simplified by agressively using Tcl's meta coding abilities to factor out the common parts of the various classes (leaf vs inner classes, the integer capability classes),

Listing 2: kinetcl::map implementation excerpt

```
critcl::class def ::kinetcl::Map {
    ::kt_abstract_class

    method bytes-per-pixel proc {} int {
        return xnGetBytesPerPixel (instance->handle);
    }

    method modes proc {} ok {
        XnStatus s;
        int lc;
        Tcl_Obj** lv = NULL;
        XnMapOutputMode* modes;

        lc = xnGetSupportedMapOutputModesCount (instance->handle);
        if (lc) {
            int i;

            modes = (XnMapOutputMode*) ckalloc (lc * sizeof (XnMapOutputMode));
            s = xnGetSupportedMapOutputModes (instance->handle, modes, &lc);
            CHECK_STATUS_GOTO;

            lv = (Tcl_Obj**) ckalloc (lc * sizeof (Tcl_Obj*));
            for (i = 0; i < lc; i++) {
              ...
            }

            ckfree ((char*) modes);
        }

        Tcl_SetObjResult (interp, Tcl_NewListObj (lc, lv));

        if (lc) {
            ckfree ((char*) lv);
        }

        return TCL_OK;
    error:
        ckfree ((char*) modes);
        return TCL_ERROR;
    }

    method @mode? proc {} ok {
        XnStatus        s;
        XnMapOutputMode mode;
        Tcl_Obj* mv [3];

        s = xnGetMapOutputMode (instance->handle, &mode);
        CHECK_STATUS_RETURN;

        ...

        Tcl_SetObjResult (interp, Tcl_NewListObj (3, mv));
        return TCL_OK;
    }

    method @mode: proc {int xres int yres int fps} XnStatus {
        XnMapOutputMode mode;

        mode.nXRes = xres;
        ...

        return xnSetMapOutputMode (instance->handle, &mode);
    }

    ::kt_callback mode \
        xnRegisterToMapOutputModeChange \
        xnUnregisterFromMapOutputModeChange \
        {} {}

    support {
        #define kinetcl_NUM_PIXELFORMATS (5)
          ...
    }
}
```

and generating the whole of the callback support from short descriptions as seen in listing 3.

Listing 3: Callback definition

```
::kt_callback user-enter \
    xnRegisterToUserReEnter \
    xnUnregisterFromUserReEnter \
    {{XnUserID u}} {
        CB_DETAIL ("user", Tcl_NewIntObj (u));
    }
```

This last was made relatively simple by the very regular nature of `OpenNI`'s API for the (de)registration of callbacks, including the callback signatures. Even the places where two or even three callbacks were managed by a single pair of (de)registration functions could be fitted in.

3. LIMITATIONS

A number of `OpenNI`'s features were not given full attention, or not implemented at all, because `KineTcl`'s intended use in the NMHMC did not require them. These are:

1. The audio, player, recorder, and script classes are mainly shells without full implementation. They are certainly not tested.

2. Instances are constructed using only default arguments. `OpenNI` actually has an API allowing the user to configure a query object/structure to limit the search for the type of instance to specific vendors, versions, and the like. None of this is used.

   Create a "user generator", for example, and the system will simply provide a handle it believes is the best.

3. Similarly `OpenNI` has functionality to query it for the set of installed modules, their vendors, versions, provided node types, etc. This also includes the ability to query what node stacks exist (i.e. coherent collections of nodes able to perform a task). For example, a "hands tracker" may need a "user generator" and if multiple modules provide implementations of either, `OpenNI` can construct different processing networks (node stacks) by mixing and matching them.

   None of this functionality is exposed by `KineTcl`.

4. FUTURE DIRECTIONS

Some of the things we can/may do in the future of `KineTcl` are obvious. Just look at the limitations listed in the previous chapter.

Another relatively obvious direction is to write additional processing classes directly in Tcl (e.g. implement various types of gesture recognition). Some work on this has actually been done, but is not complete (and buggy). See the files `stance.tcl` and `examples/dance` for the experiment with a `FAAST` [16] inspired system.

Finally, there is the currently used hack for the final integration of events. Better solutions for this, such as Tcl's API for "Event Sources", should be investigated.

APPENDIX

A. REFERENCES

[1] National Museum of Health and Medicine, Chicago `http://www.nmhmchicago.org/`

[2] Andreas Kupries, KineTcl. `https://chiselapp.com/user/andreas_kupries/repository/KineTcl`

[3] Andreas Kupries, CRIMP. `http://wiki.tcl.tk/crimp`

[4] Andreas Kupries, Steve Landers, Jean-Claude Wippler, CriTcl. `http://jcw.github.com/critcl/`

[5] Various. OpenKinect, libfreenect. `http://openkinect.org/wiki/Main_Page`

[6] PrimeSense. OpenNI organization and framework. `http://www.openni.org`

[7] PrimeSense. OpenNI API Reference. `http://openni.org/Documentation/Reference/index.html`

[8] PrimeSense. `http://www.primesense.com`

[9] Microsoft. Kinect. `http://www.xbox.com/en-US/kinect/`

[10] Avin. SensorKinect. `https://github.com/avin2/SensorKinect`

[11] PrimeSense. NITE. `http://www.primesense.com/technology/nite3`

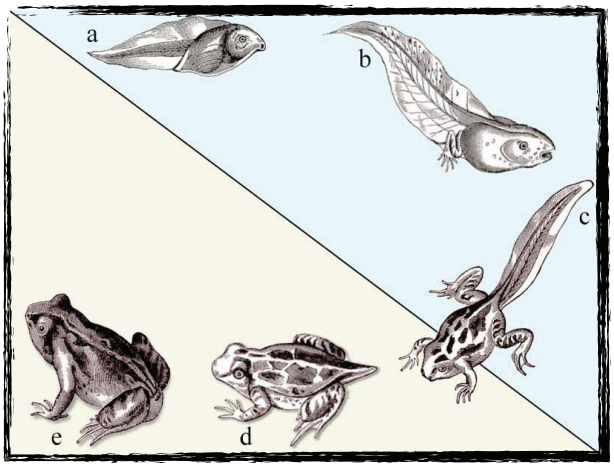
[12] Various, Tcl. `https://tcl.sourceforge.net`

[13] Jean-Claude Wippler, Poli-C. `http://wiki.tcl.tk/polic`

[14] Donal Fellows, TclOO `http://core.tcl.tk/tcloo`

[15] Various, Tcllib. `https://tcllib.sourceforge.net`

[16] ICT, Flexible Action & Articulated Skeleton Toolkit `http://projects.ict.usc.edu/mxr/faast/`



Lifecycle Object Generators (LOG)

Presented to the 19th Annual Tcl Developer's Conference (Tcl'2012)
Chicago, IL
November 12-14, 2012

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 22185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Abstract:

This paper describes a design concept call "Lifecycle Object Generators", or LOG for short. It involves a combination of coroutines, TclOO, and basic data structures to create objects that can readily transition from one class to another throughout the course of an application. This paper will describe the basic mechanisms required, and how this architecture can be applied to any complex problem from GUI design to Artificial Intelligence.

This paper is based on experience developing the Integrated Recovery Model for T&E Solutions.

Introduction

Most interesting computer models try to describe the actions and interactions of living, or at the very least animate, things. (The study of most dead an inanimate objects requiring a bit less computer power.) Living things have a tendency to change behavior. Until now modeling that change in behavior has required keeping track of state as variables and encoding every method with a patchwork of if/then/switch statements.

This paper will describe a new technique that exploits the ability of an object in TclOO to change class dynamically. TclOO is available as a package for Tcl 8.5, and is integrated into the core of the upcoming Tcl 8.6.

Style Guide

In this paper, I will be using the following style conventions:

Built in Tcl command/ keyword	`oo::class`
Name of an class, object, or variable	*`class_bar`*
Block of example code	`# Comment` `set foo bar`

Nickel Tour of TclOO

This paper exploits many advanced features of TclOO. But before we play with the advanced features, it may be helpful to go back over the basic ones.

A new class is declared with the `oo::class` command:

```
oo::class create classname {
 superclasses ancestor ancestor ...

 method methodname arguments {
  # Body of method
 }
}
```

Within the body, one declares the structure of the class. The keywords we'll be focusing on in this paper are:

`constructor`	Defines the constructor
`destructor`	Defines the destructor
`forward`	Forward calls for a method to another command
`method`	Define a method
`superclass`	Define the ancestors of this class

Once created, a class is a command. A command with several methods, the most important is **`create`**.

```
# Create a new object with a known name
classname create objectname

# Create a new object with a
# dynamically generated name
set obj [classname new]
```

And once an object is created, it lives as a command. To call a method:

```
objectname method $arg1 arg2 [arg3]
# Save a value returned from a method
set var [objectname method $arg]
```

If methods look and act a lot like procedures, that is by design. They can return a value, just like a standard Tcl proc. They can also call several built in commands, specific to the TclOO environment:

`my`	Exercise a method of the current object
`next`	Call on an ancestor's implementation of this class
`self`	Returns a the fully qualified name of this object

The **my** command is an unambiguous way for the Tcl parser to discern what commands are local to the object, and what commands should be resolved globally. It also makes for easier reading on the part of the programmer.

```
proc noop {string} {
  puts “global - $string”
}
oo::class create noop {
  method noop string {
    puts “[self] - $string”
  }
  method test {} {
    my noop “Hello World”
    noop “Hello World”
  }
}
noop create testobj
testobj test
testobj - Hello World
global - Hello World
```

In addtion to the **oo::class** command, TclOO provides **oo::define** and **oo::objdefine**. **oo::define** is used to modify a class dynamically. **oo::objdefine** is used to modify an object dynamically. TclOO also enhances the **info** command with two new methods: **info class** and **info object**, As you can imagine, **info class** provides introspection for classes, and **info object** provides introspection for objects.

Destroying a class

Classes in TclOO are implemented as objects, with their own constructors, destructors, and methods.

If you destroy a class, you automatically destroy any classes or objects derived from that class. And of course for every class that is destroyed as a result of destroying a class you destroy all of its derivatives, and so on. Taking our example from above:

```
info command obj*
obja objb objc objd obje objf
a destroy
info command obj*

# ^ Empty ^
```

Be careful though, *destroying objects by destroying their class prevents the object destructor from being called*.

Multiple Inheritance

One matter that will come up as we develop complex hierarchies of classes will be multiple inheritance. Given a choice between method implementations, TclOO will always choose the latest one defined.

```
oo::class create a {
 method noop {} { return a }
}
oo::class create b {
 superclass a
 method noop {} { return b }
}
oo::class create c {superclass a b}
oo::class create d {superclass b a}
oo::class create e {
 superclass c
 method noop {} { return e }
}
oo::class create f {superclass a b e}
oo::class create g {superclass a b c d e}
oo::class create h {superclass a b d c e}
oo::class create i {superclass e d c b a}
```

a is a common ancestor to the rest, and it provides an basic implementation of a method called *noop*. *b* is a descendent of *a* that provides its own implementation of *noop*. *c* and *d* inherit both *a* and *b* explicitly, but in a different order. *e* is a descendent of *b* that provides its own implementation of *noop*. *f* is a descendent of all of the classes *a-e*. *g-i* demonstrate various combinations of *a-e*.

```
foreach class {a b c d e f} {
 $class create obj$class
 puts [list obj$class [obj$class noop]]
}
obja a
objb b
objc b
objd b
obje e
objf e
objg e
objh e
obji e
```

You will see that in every example, the latest version of the *noop* that is defined is the one that is used. Since *b* is a descendent of *a*, given a choice between *b*'s implementation of a method and *a*'s implementation of a method, *b* will always be preferred. Likewise, *e* is a descendent of *b*. *e*'s version of a method will always be preferred to *b*'s.

If we do the example differently, sans b inheriting a and e inheriting b, we would get a different results, and the order in which classes are specified in the superclasses keyword becomes more important:

```
oo::class create a {
 method noop {} { return a }
}
oo::class create b {
 method noop {} { return b }
}
oo::class create c {
 superclass a b
}
oo::class create d {
 superclass b a
}
oo::class create e {
 method noop {} { return e }
}
oo::class create f {superclass a b e}
oo::class create g {superclass a b c d e}
oo::class create h {superclass a b d c e}
oo::class create i {superclass e d c b a}
```

```
foreach class {a b c d e f g h i} {
 $class create obj$class
 puts [list obj$class [obj$class noop]]
}
obja a
objb b
objc a
objd b
obje e
objf a
objg b
objh a
obji e
```

Objects Changing Classes

Within the **oo::objdefine** command is the ability for an object to change class:

```
oo::objdefine $object class $newclass
```

An object can even alter it's own class from within a method:

```
oo::class create moac {
 method morph newclass {
   oo::objdefine [self] class $newclass
 }
}
```

To demonstrate this process in action, imagine two classes, *classa* and *classb*:

```
oo::class create classa {
 superclass moac
 method testfunc {} {
   return "I am a classa object"
 }
}
oo::class create classb {
  superclass classa
  method testfunc {} {
   return "I am a classb object"
  }
}
```

Both classes have their own implementation of *testfunc*. The value that *testfunc* returns isn't as important as the fact that the values returned are different for the two different classes. Now with the help of a sufficiently rigged demo:

```
classb create test
test testfunc
I am a classb object
# Change class with oo::objdefine
oo::objdefine testfunc class classa
test testfunc
I am a classa object
# Ask the system what class test is
info object class test
::classa
```

```
# Change class with the morph method
test morph classb
test testfunc
I am a classb object
# Ask the system what class test is
info object class test
::classb
```

You can see that **oo::objdefine $*object* class** takes effect immediately. And it doesn't matter whether the call to change class occurs from within the object or externally. We can even change class several times during the execution of a method:

```
oo::define classb {
  method confusing_demo {} {
    # Store our present class
    set myclass [info object [self] \
      class]
    puts "Start"
    puts "1 - [my testfunc]"
    # Become a different class
    my morph classa
    puts "2 - [my testfunc]"
    # Return to our original class
    my morph $myclass
    puts "3 - [my testfunc]"
    puts "Done"
  }
}
```

The *classb* class now has an additional method, *confusing_demo*. Note, that through the miracle of modern science, changes to the class automatically apply to all objects that are instances of that class. So we can now call on this new method from our existing *test* object.

```
test confusing_demo
Start
1 - I am a classb object
2 - I am a classa object
3 - I am a classb object
Done
```

The body of *confusing_demo* is simply calling the same method three times. In between the calls, we change the class of the object with the *morph* method. The different implementations of *testfunc* give different output.

Beware of Disappearing Methods

There are plenty of ways to confuse matters by swapping an object's class. In this scenario, we have an event that is programmed to go off when an object changes class.

```
oo::class create  baz {
 method do_something {} {
   puts "Meh"
 }
 method morph newclass {
   oo::objdefine [self] class $newclass
   my do_something
 }
}
oo::class create  fubar {
 method event_morph {} {
   puts "I have morphed"
 }
 method morph newclass {
   oo::objdefine [self] class $newclass
   my event_morph
 }
}
```

Now, suppose we convert this object from *fubar* to *baz*:

```
fubar create test
test morph baz
error: Unknown method "event_morph"
```

We get an error! And we get that error because the object assumes the new class instantly. We just happened to pick a class that doesn't implement the *event_morph* method, which the script the object is running through tries to call on the next line.

Note, even though we encountered an error, *test* remains class *baz*. So if we run the *morph* method again:

```
test morph baz
Meh
```

It runs successfully. We can even make *test* back into a *fubar*:

```
info object class test
baz
test morph fubar
Meh
info object class test
fubar
test morph fubar
I have morphed
```

What Happens [next]

Another interesting wrinkle in changing classes is how the **next** keyword resolves within a method that changes the object's class. Lets say we have an class that uses **next** to exercise the ancestral implementation of the same method.

```
oo::class create a {
 superclass moac
 method testfunc {} {
  puts "a - [info object [self] class]"
 }
}
oo::class create b {
 superclass a
 method testfunc {} {
  next
  puts "b - [info object [self] class]"
 }
}
oo::class create c {
 superclass b
 method testfunc {} {
  my morph a
  next
  puts "c - [info object [self] class]"
 }
}
```

For interactions between *a* and *b*, things are quite straightforward.

```
a create test
test testfunc
a - a
test morph b
test testfunc
a - b
b - b
```

c is our complex case. Its implementation of *testfunc* changes the class of the object. And worse, it changes the class to one in which there is no ancestor for the **next** operator to hop to.

You would expect the system to die horribly along the lines of:

```
c create test
test testfunc
no next method implementation
    while executing
"next " ...
```

Instead we see:

```
c create test
test testfunc
a - a
b - a
c - a
# Note, the object really has changed
# class
info object class test
a
```

The pathway through the **next** calls is computed before the method is invoked.

Design Patterns

Now that we have covered the basics, it is time to start to develop the **LOG** framework.

Storing Properties

When an object expects to change class, there is often information specific to that class that we would like to access. A variable isn't a good fit for this purpose as its value doesn't change when the class changes. So I like to employ methods that return hard coded values.

The simplest way would be to declare a method for every value we would want to return:

```
oo::class create a {
 method color {} { return green }
 method flavor {} { return lime }
}
oo::class create b {
 method color {} { return green }
 method flavor {} { return apple }
}
oo::class create c {
 method color {} { return red }
 method flavor {} { return cherry }
}
```

For the lazy programmer this system has several drawbacks. First, it is difficult to distinguish between a method that is a property and a, shall we say, livelier method. Second, the notation is verbose. It introduces the temptation to cut and paste. Third, we have no fallback mechanism should a part of the system call for a property that has not been configured yet, or is simply not applicable to the object in question.

LOG adds two new methods: *property_define* and *properties*.

property_define creates a single value.

properties allow us to specify a key/value list. We can do this easily within TclOO because, behind the scenes, classes are merely a special kind of object. The just happen to be of class **oo::class**.

```
oo::define oo::class {
  method property_define {field value} {
    oo::define [self] method prop_$field \
       {} [list return $value]
  }
  method properties dict {
    foreach {var val} $dict {
      my property $var $val
    }
  }
  method property {field args} {
   set methods [info object methods [self] \
       -all -private]
   if {"prop_$field" in $methods } {
     return [my prop_$field {*}$args]
   }
  }
}
```

We also need to configure all of our client classes with a version of the property method.

```
oo::class create moac {
  method property {field args} {
   set methods [info object methods [self] \
       -all -private]
   if {"prop_$field" in $methods } {
     return [my prop_$field {*}$args]
   }
  }
}
```

So to configure a class:

```
oo::class create a {superclass moac}
a property_define color green
a properties {
 flavor lime
}
a create test
test property color
green
```

At the same time, if I ask for an item that is not configured (or configured yet), I get back an empty list instead of an error.

```
test property speed

# ^ Empty List ^
```

And if we are modeling a system worthy of a Lewis Carroll novel, we can alter the property of a class on the fly too.

```
a property_define speed very_fast
test property speed
very_fast
```

Using Classes to Represent State

State machine code becomes notoriously complex when there are more than a handful of states. I am going to introduce an easier way: create a separate class for each state an object can be in. Thus, if a method has to behave differently, we can just define that change for the particular state.

Let us begin with a few ground rules for changing an object's class. Even better, let's have a library of base classes that enforce those rules. All classes that are eligible to change class will be descendants of a common baseclass: *state_machine*.

state_machine provides several methods:

state_change	Change the class (and this state) of an object. Takes an additional argument which can pass additional data to event scripts.
state_current	Return the current class (thus state) of an object
state_enter	Script to run when an object enters the configured state
state_exit	Script to run when and object exits the current state

```
oo::class create state_machine {
  superclass moac ; # For "property" method
  constructor {} { my state_enter {} }
  # Return the current state
  method state_current {} {
    return [info object class \
     [self object]]
  }
  # Actions when we exit state
  method state_exit {} {}
  # Actions when we enter state
  method state_enter {} {}

  # Returns 1 if state changed
  # Returns 0 otherwise
  method state_change {newstate} {
   if { $newstate eq {} } { return 0 }
   set oldstate [my state_current]
   if { $newstate eq $oldstate } {
    # In the desired state, do nothing
    return 0
   }
   # Run cleanup from old state
   my state_exit
   oo::objdefine [self] class $newstate
   # Run setup from new state
   my state_enter
   return 1
  }
```

Example: Lifecycle of a Frog

Let is show off our newly developed *state_machine* with a demonstration: The lifecycle of a frog.

```
oo::class create frog {
  superclass state_machine
  method state_exit info {
    puts "Leaving [my state_current]"
  }
  method state_enter info {
    puts "Entering [my state_current]"
    next $info
  }
}
frog properties {
  has_tail 0
  respiration lung
  state_next {}
  color green
}
```

The baseclass *frog* is a series of general assumptions one could make about any frog, stored as properties. One of those properties *state_next* tells us what developmental state

follows the current state. For an adult *frog*, we have no *state_next*, so we configure an empty set.

To model our frog's lifecycle, a program can simply walk from one state to another, reading the properties as it goes.

```
oo::class create frog.egg {
  superclass frog
}
frog.egg properties {
  has_tail 0
  respiration none
  state_next frog.tadpole
}
oo::class create frog.tadpole {
  superclass frog.egg
}
frog.tadpole properties {
  has_tail 1
  respiration gill
  state_next frog
}
```

```
frog.egg create hypno
set changed 1
while {$changed} {
 foreach fld {
   has_tail respiration state_next
 } {
   puts " * $fld [hypno property $fld]"
 }
 set newstate [hypno property state_next]
 set changed [hypno state_change $newstate]
}
Entering ::frog.egg
* has_tail 0
* respiration none
* state_next frog.tadpole
Leaving ::frog.egg
Entering ::frog.tadpole
* has_tail 1
* respiration gill
* state_next frog.tadpole
Leaving ::frog.tadpole
Entering ::frog
* has_tail 0
* respiration lung
* state_next frog
```

Discrete Time Phases

Discrete time simulations are similar to tabletop games. Actors (or players) take turns. And the rules of the game govern which interactions are valid during which part of a game turn.

In [1]Risk™ , each turn has three phases: placing reinforcements, attack, and fortifying. Players are only allowed to add troops to the battlefield at a certain time. There is only one phase in which we would expect troops to be removed from the battlefield (as casualties.) And there is only one point in the turn where troops can move. Phases make the outcome of a series of events more consistent.

Table games are engineered to have a definite "winner". The actor with priority is allowed to have a significant impact on the outcome of the scenario.

```
turn 1
 Player 1 - Reinforce Phase
 Player 1 - Attack Phase
 Player 1 - Fortify Phase

 Player 2 - Reinforce Phase
 Player 2 - Attack Phase
 Player 2 - Fortify Phase
```

With scientific simulations, we don't want a "winner." We want to devise a series of rules such that we get the same outcome whether the actors are run in sorted order, reverse sorted order, random order, or whatever that subtle, non-random, but sufficiently inscrutable order we get from `[array names]`.

We also want to create the illusion that all of the actions in a given time phase occur simultaneously. So rather than let one actor run through all of the phases, followed by another, we give each actor an opportunity to act during every phase.

[1] Risk™, Trademark Parker Brothers

```
turn 1
 Player 1 - Reinforce Phase
 Player 2 - Reinforce Phase
 Player 1 - Attack Phase
 Player 2 - Attack Phase
 Player 1 - Fortify Phase
 Player 2 - Fortify Phase
turn 2
  ...
```

In simulators which allow objects to change class, I found it best to restrict any such changes to a specific phase in the time step. Preferably one in which nothing else is going on.

```
Agent Timestep
 phase_physics
 phase_observe
 phase_plan
 phase_action
 phase_reaction
 phase_morph
```

When an object wants to change state, the new state is recorded as a local state variable. The actual change does not take place until the morph phase comes around.

```
oo::define state_machine_discrete {
 method state_change newstate {
  if {[my state_current] eq $newstate } {
    return 0
  }
  my variable next_state
  set next_state $newstate
  return 1
 }
 ###
 # Called by the driver of the simulation
 ##
 method phase_morph {} {
  my variable next_state
  if { $next_state eq {} } {
   return
  }
  my state_exit
  oo::objdefine [self] class $next_state
  my state_enter
  set next_state {}
 }
}
```

Example: Agent Based Modeling

The Integrated Recovery Model simulates a ship and her crew during a shipboard catastrophe. Part of the simulation entails crew members changing roles. In the model, each role is represented by a distinct class.

Any number of events can lead to a crew member changing role. The most common role changes are in response to an order. Some orders are direct. For instance, a leader telling a crew member under his/her command “You do this.” Other orders are indirect. When a crew member hears the call to go to General Quarters, he/she switches from whatever they were doing to their assigned role at GQ.

But the hardest ballet to choreograph by far was the transitions that occur when a crew member is assigned to a fire team. Most crew don’t wear a fire suit as part of their regular duties. Thus a crew member newly assigned to a fire team must find a set of gear, put it on, and connect with a team that may already be on scene. Those behaviors were complex enough to merit a separate role.

```
Crew starts as role human
Crew receives order to join Team
 > Crew becomes role team.prospect
Crew member gathers equipment
Crew walks to location of Team leader
Crew joins Team
 > Crew becomes role team.member
Team battles fire
Team dissolves
 > Crew becomes team.dismissed
Crew returns equipment
Crew walks back to assigned station
 > Crew becomes human
```

In IRM, each agent is configured with a property that lists what tasks they want to perform, and in what priority. Each task, in turn, has criteria that govern when it should activate, when it should abort, and a coroutine to carry out once activated.

Our *team.prospect* class has the following task list:

action-station	Gather tools, report to action station
safety-check	Reflexes for fleeing from danger
join-team	Join the team we are assigned to
go-home	Return to action station (only called if join-team fails)

Every agent has an *action-station* task. It has a method that produces a list of equipment required for the role assigned. It checks to see that the agent has a working version of each. And if a device is missing, exhausted, or damaged, the agent gets a new one.

Normally agents produce their own list of needed equipment, based on information configured by the model maker. For this paper, the pseudocode uses a simple property.

```
agent::class human {
 method ensemble {} {
  return [my property equipment]
 }
 method ensemble_missing {} {
  set result {}
  foreach device [my ensemble] {
   if {[my device_working $device]!=1} {
     lappend result $device
   }
  }
  return $result
 }
 task action-station {
  begin {
    return [llength [my ensemble_missing]]
  }
  ... # Define the rest of the task ...
 }
}
```

```
agent::class fireteam {
  superclasses human
  properties {
   equipment { nfti scba ppe radio }
   member_equipment {scba ppe}
  }
}
agent::class rescueteam {
 superclasses human
 rescueteam properties {
   equipment { radio stretcher scba }
   member_equipment {scba medkit}
 }
}
# One team.prospect class suffices
# to join either team
agent::class team.prospect {
 superclasses human
 method ensemble {} {
  my variable team
  return [$team property member_equipment]
 }
}
```

Because the *team.prospect* role is it's own class, we can override the standard *ensemble* method with one that queries the team this agent will join.

```
# One team.prospect class suffices
# to join either team
agent::class team.prospect {
 superclasses human
 method ensemble {} {
  set team [my knowledge get team]
  return [$team property member_equipment]
 }
}
```

Thus:

```
fireteam create crew1
rescueteam create crew2
team.prospect create crew3
team.prospect create crew4
crew3 knowledge put team crew1
crew4 knowledge put team crew2
crew3 ensemble_missing
scba ppe
crew4 ensemble_missing
scba medkit
oo::objdefine crew3 human
crew3 ensemble_missing

# ^ Empty we are back to the human class
```

Application State

When designing a GUI, we also wrestle with state. Whether it be a megawidget, or a toplevel object that is managing the application, LOG can help.

In IRM our principle display interface is managed through a Tk canvas. Onto that canvas, we draw objects, color them, and respond to mouse gestures.

We divide our model's world into drawing layers. There are specific rules for rending a wall that are different than, say, a piece of equipment. Likewise, a user double clicking on a wall expects a different dialog box if clicking a crew member versus a portal.

Window objects call out which layers are active and in which state as a method of the window:

```
irm::class modelwindow {
 superclasses [redacted]
 method active_layers {
   return {
    wall  layer.wall.basic
    compt layer.compt.basic
    portal layer.portal.basic
    eqpt  layer.eqpt.basic
    crew  layer.crew.basic
   }
 }
}
```

When devising a set of visuals, I put together two sets of classes. One is the application window, the other is a drawing layer that is modified to produce the visual.

```
irm::class modelwindow.damage {
 superclasses modelwindow
 method active_layers {
   return {
    wall  layer.wall.damage
    compt layer.compt.damage
    eqpt  layer.eqpt.damage
    crew  layer.crew.damage
    portal layer.portal.damage
    holes  layer.holes
   }
 }
}
```

In this case we are putting together a special mode that highlights damaged objects with a special color.

When applying a new state, the window object will call forth into being an object to represent each layer, and configured with the appropriate class. If the layer already exists it simply changes class.

In our example, the modified drawing layer colors all damaged components red.

```
irm::class layer.eqpt.damage {
 superclasses layer.eqpt.basic
 method node_is_damaged nodeid {
   # test for damage that returns 1 or 0
 }
 method node_style {nodeid} {
  if {[my node_is_damaged $nodeid]} {
    return {-fill red -outline -red}
  } else {
    return {-fill grey -outline grey}
  }
 }
}
```

Application window states can also specify bindings for the canvas. In the next example, upon entering the new state the canvas gets new bindings. Once the user clicks on an object the window translates motion to drag actions. When the user releases the dragged object, the window reverts back to its normal state.

```
irm::class modelwindow.drag {
 superclasses modelwindow
 method active_layers {
   return {
    wall  layer.wall.basic
    eqpt  layer.eqpt.editor
    crew  layer.crew.editor
   }
 }
 method state_enter {} {
   set canvas [my get canvas]
   bind $canvas <B1> \
     [list [self] drag_start %x %y]
   bind $canvas <B1-Motion> {}
   bind $canvas <B1-Release> {}
   my redraw
 }
 method drag_start {x y} {
  set obj [my object_at $x $y]
  if { $obj eq {} } { bell ; return }
  set canvas [my get canvas]
  bind $canvas <B1-Motion> \
    [list [self] drag_do $obj %x %y]
  bind $canvas <B1-Release> \
    [list [self] drag_done $obj %x %y]
 }
 method drag_done {obj x y} {
   set layer [my object_layer $obj]
   $layer move_to $obj $x $y
   my morph modelwindow
 }
}
```

Conclusion

Lifecycle Object Generators are not the solution to every problem in Object Oriented programming. But they are quite useful for complex state-based logic. I am developing these concepts into a fully featured toolkit, which is available for download at:

http://www.etoyoc.com/tcl

Image Credits:

Cover Image:
"Entwicklung des Krötenfrosches", By Meyers Konversations-Lexikon [Public domain], via Wikimedia Commons, accessed 17 October 2012, <http://commons.wikimedia.org/wiki/File%3AMetamorphosis_frog_Meyers.png>

Exploring Tcl Iteration Interfaces

By Phil Brooks

Presented at the 19th annual Tcl/Tk conference, Chicago Illinois November 2012

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

Abstract--- In Mentor Graphics' Calibre verification tool, Tcl is frequently used as a customer extension language - allowing customers to customize and drive the tool through various exposed interfaces. These interfaces are frequently used to access large collections of application data and provide a wide variety of mechanisms for iteration over that data. This paper will examine several interfaces that have been used for iteration over large C++ data structures along with the benefits and drawbacks of each method. Methods explored include Tcl lists, indexed array-like access, iterator object accessor (similar to C++ STL iterators), and specialized foreach style commands. Example stand alone implementations are provided and discussed from within the context of their original use in Calibre customer scripting interfaces. Ease of use and performance are considered.

1 Introduction

The simple task of iteration over each object in a container is one of the most common in programming. The task is so common that every programming language tends to develop common idioms for the form. Simple expression of the concept of iteration in a vernacular form aids readability and maintainability of code. In Tcl where the list is the most commonly used aggregate data structure, the foreach command is the standard for an iterative vernacular:

```
set test_list { a b c d }
foreach var $test_list {
    puts $var
}
```

When C based Tcl_Obj object interfaces that represent collections of underlying objects are being used, the foreach command itself is of little use since it works only with Tcl lists. So for iteration, either the Tcl object must convert its contents into Tcl list form, or another interface must be constructed for iteration over the object data sets. The remainder of this paper considers potential interfaces for this purpose.

2 Demo Environment

All of the demo interfaces used in the example program are providing access to the contents of a C++ array of doubles - or in C++ "std::vector<double>". A Tcl program is used to iterate over the contents and to accumulate a result which is returned to the C++ program. The context of the environment is that of a customized analysis routine that is called from the C++ application. The Tcl interface allows an end user to perform custom calculations without having access to the application C++ source code or having to manage a C compilation environment. The Tcl program has read only access to the C++ vector.

Since the Tcl routine is called directly from the C++ program, a record based user interface is provided so that the user can direct the application with the name of the Tcl script and the proc to call. In the Mentor Graphics Calibre environment, these Tcl calls are specified from the Standard Rulefile Verification Format (SVRF) language that makes up the bulk of the application's programming interface.

In this example program, a configuration file specifies the name of the Tcl script, a proc to call, and the iteration interface that is to be selected (from the 4 we are describing).

These fields are specified as simple text fields on a single line of the file:

```
<script_file> <called_proc> <interface>
```

For example:

```
list_user_script.tcl do_calculation list
```

describes list_user_script.tcl as the script file, calc_abmi as the Tcl proc, and the list generation interface as the interface to provide.

3 Loading the script file

After the config file is read, the script file itself is read and evaluated in the Tcl interpreter so that it can be called repeatedly as the application progresses through its data set. This is accomplished by first creating a Tcl_Obj that will contain the script:

```
Tcl_Obj* tcl_script = Tcl_NewObj();
Tcl_IncrRefCount( tcl_script );
```

(Note that code examples in the paper are sometimes slightly altered for brevity from the example program.) Then, the following code adds the script, line by line, to that object using Tcl_AppendStringsToObj:

```
std::ifstream file_loader( load_file.c_str() );
std::string load_line;
while( file_loader ) {
    std::getline(file_loader, load_line);
    Tcl_AppendStringsToObj( tcl_script,
        load_line.c_str(), "\n", NULL );
}
```

The script file itself is now loaded into the interpreter using Tcl_EvalObjEx:

```
rc = Tcl_EvalObjEx(interp,tcl_script,TCL_EVAL_GLOBAL);
```

Now the interpreter is ready to run the indicated proc for each vector in the analysis set.

4 Exploring the Iteration Interfaces

The main body of the paper explores several interfaces that an application can present to user through the Tcl C Tcl_Obj and Tcl_ObjType interfaces. The goal for these interfaces is to provide users simple and intuitive access to large native application datasets in an efficient manner that looks at least vaguely familiar and intuitive to Tcl users.

4.1 Accessing a Tcl List directly

The most natural and straight forward mechanism for iteration in Tcl is simple iteration through a Tcl list:

```
proc do_calculation input_list {
    # now iterate
    foreach var $input_list {
        puts $var
    }
}
```

The Tcl list is, then, a very straight forward mechanism to providing access to application data. The Tcl List interface is used in the Calibre product's LVS Device recognition application in order to provide access to a (usually) short set of numbers describing proximity of features near a transistor. Lists in the example program are constructed using the Tcl_ListObjAppendElement interface. The command is invoked by name (from the proc_name argument), with the list passed in the second field of the command:

```
//
// Tcl List construction from a C++ std::vector<double>
//
Tcl_Obj *command[2];
command[0] = Tcl_NewStringObj( proc_name.c_str(), -1 );
Tcl_IncrRefCount(command[0]);
command[1] = Tcl_NewObj(); Tcl_IncrRefCount(command[1]);
for( std::vector<double>::iterator i = data.begin();
     i != data.end(); ++i )
{
    Tcl_ListObjAppendElement(interp, command[1],
        Tcl_NewDoubleObj( *i ));
}
```

After construction of the list, the command text and the list are passed in to the calling script using Tcl_EvalObjv with the script text as the first argument and the list as the second argument.

```
int rc = Tcl_EvalObjv(interp,2,command,TCL_EVAL_GLOBAL);
```

Since the interface here is through a real Tcl list, this method presents the most natural interface to the Tcl programmer. Its main drawback is that the data structure must be fully copied from its

native C++ into the Tcl list. For applications that have very large datasets, or high performance goals, the overhead required to form the Tcl list may be unacceptable. For those applications, the other access mechanisms may be more appropriate.

4.2 Access through an Index

The second interface demonstrated uses an index for random access into the contents of the container:

```
proc do_calculation my_arr {
    # returns an object count
    set entry_count [ $my_arr entry_count ]
    # iterate using an index
    for { set i 0 } { $i < $entry_count } { incr i } {
        puts "my_arr $i => [ $my_arr value $i ]
    }
}
```

The interface to the array is provided through the Tcl_CreateObjCommand interface.. In order to construct that interface, the example program uses Tcl's Tcl_CreateObjCommand interface.

```
Tcl_CreateObjCommand(interp,"arg1",
    vector_interface,data,NULL);
```

This call creates a command object named “arg1”, bound with data pointer data, and implemented through the some_stats_vector_interface function. The name “arg1” is arbitrary and it is only used when inside the application as seen below. Inside the called proc, this command is bound to a parameter of the called proc. This technique allows the end user to select meaningful names for what are potentially a large number of parameters that all have real names that aren't very meaningful to the end user.

Next, the index_interface function provides implementation for the required commands:

```
int index_interface(
       ClientData cd,
       struct Tcl_Interp *interp,
       int objc,
       Tcl_Obj *CONST objv[] )
{
    std::vector<double>* data =
         static_cast<std::vector<double>*>(cd);
    const char* command =
            Tcl_GetStringFromObj( objv[1], NULL );
    if ( strcmp( command, "size" ) == 0 ) {
    f  size_t sz = data->size();
      Tcl_Obj *result=Tcl_NewLongObj(sz);
      Tcl_IncrRefCount(result);
      Tcl_SetObjResult( interp, result );
   } else if ( strcmp( command, "value" ) == 0 ) {
     ...
```

The object command is passed along with the name of the proc as an argument to Tcl_EvalObjEx. This is where the name “arg1” is used. It is not visible to the user (unless the user knows to look for it).

```
std::string invoke_line = procname;
invoke_line.append( " arg1" );
int rc = Tcl_Eval(interp,invoke_line.c_str());
```

The array index interface is used in the Calibre product's LVS Device recognition application in order to provide access to a randomly accessible array of measurement numbers related to a transistor. The advantage of this method over the constructed List method is mainly efficiency. The contents of the C++ vector are accessed directly by methods implemented through the Tcl_CreateObjCommand interface. The interface is not nearly as elegant as the list interface for simple iteration over the contents of a container. It also isn't suitable for data that doesn't fit an index->value retrieval model. The next interface extends the index to a more fully fledged iterator accessor.

4.3 Using an iterator interface similar to C++ iterators

The C++ standard library provides a convenient common mechanism for iteration through containers. That mechanism is called the 'iterator'. The code looks like this if you want to iterate through all members of an array of doubles called 'data' printing each item on a separate line:

```
std::vector<double>::iterator i = data.begin();
while ( i != data.end() ) {
   std::cout << *i << std::endl
   ++i;
}
```

We might construct a similar interface in Tcl where code could look like this:

```
set my_iter [ $data get_iterator ]

while { ! [$data at_end $my_iter] } {

    puts "my_arr $i => [ $data value $my_iter ]

    $data incr $my_iter

}
```

In the example program, the iterator interface is constructed from two parts. The record is accessed via a Tcl command object that is similar to the one used in the indexed interface. In the place of the index, the iterator is a full fledged Tcl_ObjType object. It can retain state and independent settings from the container itself. It is also more vulnerable to going out of synch with the container, so may require mechanisms to void its validity if the container changes state while the iterator is still in existence. The initialization of the command object is pretty much the same, using Tcl_CreateCommandObj, as it is for the indexed access. The commands supported by the implementation command are:

- get_iterator - returns an iterator to the beginning of the data container
- at_end - indicates the iterator is past the last data item in the data container
- incr - moves the iterator to the next item in the container
- value - retrieves the value represented by the iterator

The iterator itself represents the std::vector<double>::iterator and that is its only data member in this implementation. That is actually quite inadevalue quate since the vector iterator is represented by a raw pointer into the memory of the data vector. As long as the data vector remains in its original state, the iterator is fine. If the data container is altered or goes away, the iterator should, in fact, be invalidated immediately. This would normally be done with some sort of Observer pattern where the interface retains a list of active iterators and can void them when ever any operation occurs that would invalidate an iterator.

The Tcl_ObjType interface that contains the iterator pointer is implemented using the standard name and set of type handling functions for free, duplicate, update_string and set_from_any. These functions manage the access to the C++ iterator.

The iterator style interface is used in the Calibre product's Yield Server application to access a wide variety of EDA design data like electrical nets, devices, design cells, and geometries etc.

5 Exploring more consistent interfaces

The implementations explored thus far have resulted in vastly different Tcl code because of the mechanics of the underlying iteration mechanism and the fact that the Tcl foreach command is strictly a list-based iteration mechanism. In the next section, two methods of providing a more generic interface are explored. While the foreach command is strictly list based, a specialized foreach-like command can be used to soften the differences between the custom interfaces and the Tcl list interface. Coroutines, new to tcl, are also referred to as generators. They provide a potentially much more powerful and consistent interface to the problem of iteration.

5.1 Using a specialized foreach command

It is possible to adapt the interfaces presented earlier to get closer to the syntactic simplicity of the original foreach loop around the list. A specialized foreach-like command can be implemented that allows use of syntax that is very similar to the original foreach implementation on the list. The specialized foreach command can hide the differences between the various access interfaces allowing the user routine to The example program implements such a foreach_instance command on top of the indexed access method presented earlier. It does this with a specialized command “foreach_instance” which allows the following interface:

```
proc do_calculation record {
    foreach_instance value $record {
        puts $value
    }
}
```

which is very close to a Tcl List interface:

```
proc do_calculation record {
    foreach value $input_list {
        puts $value
    }
}
```

This foreach_instance command is implemented entirely in Tcl - and it hides the complexity of the index access interface. The foreach_instance proc is implemented as:

```
proc foreach_instance { var1 record body } {
    set vlen [ $record size ]
    upvar 1 $var1 value# now iterate
    for { set i 0 } { $i < $vlen } { incr i } {
        set value [$record value $i]
        uplevel 1 $body
    }
}
```

The foreach style top level command is used by the Calibre LVS Comparison application's device reduction application to give iterative access to a potentially large singly linked list of devices. While the two scripts are quite similar, they vary on the name of the critical foreach command itself. In the next section, use of a coroutine allows the difference to be obscured using another mechanism.

5.2 Using a Tcl coroutine with the index interface

Implementing a coroutine interface further explores the iterative style command in the context of coroutines. Using the coroutine, like the specialized foreach command, requires a specialized adapter routine that traverses the data structure for another command that is doing the calculation. One simple way to traverse a coroutine until it is empty follows. This example uses a coroutine to traverse a Tcl list:

```
proc do_calculation { record } {
  coroutine data_fetcher get_from_record $record
  while 1 {
    puts [ data_fetcher ]
  }
}
```

In this proc, the coroutine data_fetcher is created from the proc get_from_record (not shown) and its argument $record, the list of data. It then goes into a while loop that prints the value of each item in the list. The loop is broken when data_fetcher returns with a -code break return code that indicates the end of the list. Next is the implementation of a list iteration form of get_from_record:

```
proc get_from_record record {
    yield [ info coroutine ]
    foreach value $record {
        yield $value
    }
    return -code break
}
```

This calculation proc can remain unchanged while the get_from_record proc changes to cover a different interface - in this instance, the index interface shown above:

```
proc get_from_record record {
    set vlen [ record size ]
    yield [ info coroutine ]
    for { set idx 0 } { $idx < $vlen } { incr idx } {
        yield [ $the_record value $idx ]
    }
    return -code break
}
```

Tcl Coroutines are not currently used in the Calibre application family which is still using Tcl 8.4.

6 Conclusion

While Tcl provides a number of high performance adaptable interfaces to a C application programmer, iteration over a data collection is still quite cumbersome due to the differences in handling C object type collections and Tcl lists. These differences in interface are overcome in certain situations through the use of a customized foreach-like command, but that approach has a drawback in that the foreach-like command itself is specific to its data container. Coroutines provide promise for providing a common iteration mechanism within Tcl, though the language feature is new and idioms are not yet well developed.

7 Acknowledgments

Special thanks go to Donal K. Fellows for his assistance in writing the specialized foreach command and coroutine adapter interfaces.

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 5
November 15 13:30-15:00

Pulling Out All the Stops - Part II

By Phil Brooks

Presented at the 19th annual Tcl/Tk conference, Chicago Illinois November 2012

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

Abstract--- At the 12th annual Tcl/Tk conference in Portland, Oregon, I presented a paper entitled 'Pulling Out All the Stops' - which concerned using Tcl as a user interface custom calculation engine at the heart of a high performance electronic design analysis package. This talk discussed the efficiency concerns and implementation details that were considered during implementation of this package. One simplifying constraint applied to the design was that, though this is a multi threaded application, a single Tcl interpreter was used and individual threads would access that Tcl interpreter through a mutex lock. seven years later, customers are running more complex calculation codes on systems with more processors. Locking on the single Tcl interpreter now restricts scaling. This paper briefly reviews the original design and then discusses the conversion to a fully threaded design in which one Tcl interpreter per thread allows completely parallel execution of the Tcl calculator.

1 Introduction - A review of the 2005 paper

The 2005 paper discusses implementation of Calibre LVS's Device TVF feature, in which a highly efficient, though quite limited, calculation engine is given the ability to make calls to a Tcl program allowing for more sophisticated programming capabilities. Some examples of the calculation language to Tcl calling mechanism is described. Finally, a set of implementation details that wring maximum performance out of the Tcl interpreter in this situation are described. Specifically, the major performance related recommendations were:

- Make use of TCL_EVAL_GLOBAL.
- The user's Tcl programs are pre-compiled before calculations are started using Tcl_EvalObjv.
- Runtime data access Tcl_Obj objects for arguments passed into function and return values are set up once up front and are reused during each individual call to the device calculator.
- Use Tcl object commands to hand off performance critical processing from Tcl to C++.
- End users are strongly encouraged to write compact efficient Tcl code.

The end of the 2005 paper includes a brief mention comparing Tcl 8.4 MT vs. Tcl 8.3 non MT builds and shows a favorable scaling improvement especially when 4 or more processors are used. The performance scaling that was visible in toy test cases, where the multi-threaded C++ processing was made as simple as possible and the Tcl processing was made artificially complex, bore no relationship to the real world testing on real customer data where the MT C++ part took vastly more time and the Tcl - even locked and single threaded - took hardly any time at all. So, the originally shipped implementation didn't use MT Tcl interpreters due to time constraints, tcl version constraints (we originally released on 8.3), some lurking bugs in the Tcl MT package, and one critical Tcl threading implementation detail that we had yet to uncover!

2 What has happened in the mean time

Over the next several years, customers started using the interface, at first in exactly the way it was designed... The original interface was conceived as a simple functional interface in which customers would pass in a couple of vector style parameters, run through a loop, perform the calculations of their choosing, and return a single value to the calculation engine to be used in later processing. We found that after they got used to the interface, customers did far more clever things than that with it:

- They would pass in dozens of parameters and make complex multi-step calculations.

- They would calculate several different parameters using the same set of input parameters, often doing preliminary calculations repeatedly due to the fact that our interface would only return a single number at a time.
- To get around the single return number constraint, customers would put together string results that were actually a concatenation of numbers separated by '_' characters and then pass them back and reparse those strings in subsequent calls.

We also found, during performance analysis on 16, 32, and 64 way systems, that, especially when customers did these sorts of complex calculations, the single shared Tcl interpreter was having a throttling effect on overall MT scaling of the application.

3 Searching for Clues

Most of the documentation that I could find for using multi-threaded Tcl talked about using Tcl itself to start and manage the threads. There is one passage in the book "Practical Programming in Tcl and Tk" by Welch, Jones and Hobbs [Welch] that says:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be managed by only one thread." p. 322*

And that's about it. The rest of the chapter talks about all of the facilities that Tcl has at the scripting level for creating and maintaining multiple interpreters using the Thread package, how to make them communicate with one another, how to pass channels back and forth, how best to write to a common file, how to share data efficiently between them, synchronization between Tcl threads, but nothing else that talks about how to run multiple interpreters that don't have to interact with one another from C. I didn't know it at the time, but that simple statement contains the most important thing (and possibly the only important thing) that you need to know to create independent parallel Tcl interpreters running on separate threads in C.

4 More about Calibre threading

Calibre has been a multi-threaded application for a very long time. There are embarrassingly parallel problems of computational geometry to be solved, and so they have been solved and improved upon since at least April of 1998. Here is an interesting CVS checkin log:

```
revision 1.1
date: 1998/04/10 17:41:31;
author: ****; state: Exp;
routines related to flat drc thread are going be in this
file. Currently there is not much in it.
```

Needless to say, there is considerably more in that file today than there was and the threading capabilities in Calibre are very well entrenched, so changing the threading model and control of the entire application for one new feature in the LVS is not in the cards. So years after the initial paper on 'Pulling out all the stops', it appeared, in fact, that more stops needed pulling, I rolled up my sleeves and resurrected the notion of running the Tcl interpreters in threads.

5 Task queue and thread pool Pattern

Let's look briefly at this commonly used pattern for threading architecture. The task queue and thread pool pattern [Wikipedia] is also known as the Worker-Crew Model [Sharapov]. In this pattern essentially separates the management of processing threads in a system from the tasks that need to be performed. This means that the threads can be managed for one set of constraints (like hardware availability, licenses, performance management, etc.) while task creation is under the purview of the application algorithms (like what things do I need to do and in what order do they need to occur).

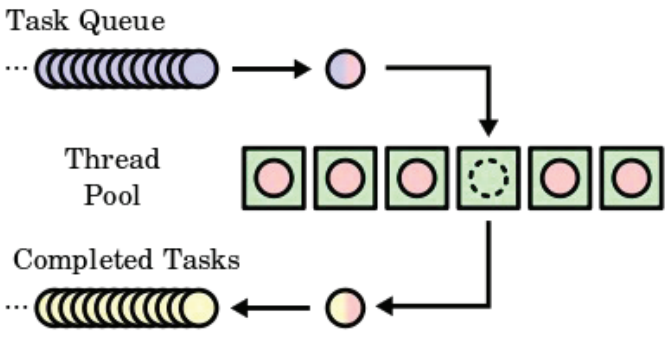


Figure 1. - Task Queue and Thread pool pattern - Wikipedia

Calibre uses just such a model for threading. This allows many disparate algorithms that may execute to solve a customer's particular problem to parcel their work out to a number of worker threads that complete the tasks according to hardware and licensing constraints that are managed by a completely different part of the system.

5.1 An Example Work Queue and Thread Pool API

So a work queue API generally has the following sorts of requirements and capabilities:

- it allows you to package up a unit of work including data and some direction for what needs doing.
- it allows you to ship that unit of work off to the work queue.

The thread pools then grab the pieces of work and execute them. At that point, the essential get-something-done part of the API comes in:

- it allows you to write some code that will do the work in a worker thread.

finally, once the work is done, the final piece:

- wake me up when its over.

5.2 Implementing work queue and thread pool in Calibre LVS

So, the Calibre device recognition algorithm follows the above prescription:

- In the main thread: Package up the work. Ship the tasks off to the thread pool.
- In a worker thread: Wake up and perform a task on a particular set of data.
- In the main thread: OK - wake up, everything is done now.

In my initial implementation, the following steps are taken in the main thread:

- Start setting up the algorithmic tasks. Ship the tasks off to the thread pool.

In a worker thread:

- Use thread specific storage to look for an existing interpreter
- if there is no interpreter create one. Also create the entire infrastructure for the interpreter, feed it the Tcl program text etc. Set up the argument objects, return objects, calling infrastructure etc. Store thread specific data inluding the pointer to the interpreter.
- Execute the local work task - it calls the thread's Tcl interpreter when needed.

At this point, everything is doing just fine. Now, once we finish all of the work, we need to clean up after ourselves.

- Wake up everything is done.
- Clean up the Tcl interpreters.

No big problem, just loop remember where all of the tcl interpreters are and delete them one at a time from the main thread!

5.3 Oops...

For the most part, the initial implementation worked fine. However, as this excerpt from my post to comp.lang.tcl points out, the cleanup at the end wasn't going so well:

> *At the start of the processing that was requiring multiple Tcl interpreters, I set up some thread specific storage in a vector of pointers, then as each thread got its first task, it would check for an interpreter there if there was no interpreter, it would create one under its thread index.*
>
> *Threads would individually calculate, using Tcl and they ran nicely in parallel. At the end of the processing, once all of the tasks were done and the threads all quietly parked in their Calibre threading model parking slots, I would loop through on the main thread and destroy each interpreter. This is where things would go terribly wrong. Crashes, memory leaks, unclosed files, etc..*

I finally reproduced the entire problem outside of my application in a small snippet of C++ and Tcl C api. While doing that, I found out that if I deleted the interpreter from inside the same thread that it was created in, all was fine, but if I did the same thing from the main thread in the same way that I was in the application, I got the same crashes that I had in the full application. I posted my findings to Comp.lang.tcl and received the following reply:

> *Gerald W. Lester: ...*

> *I also found that by deleting the interpreter inside the same*
> *execution thread that it was created in,*
Where else would you be deleting it from -- an interpreter is not supposed to be accessed by more than one thread. You may have many interpreters per thread (i.e. a thread can access/use many interpreters), but only one thread per interpreter (i.e. only a single thread should be accessing a given interpreter).

Let's reread the bit from the book:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be managed by only one thread." p. 322*

Ahh - so that's what it means! I suggest rewriting that sentence:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be created, used and destroyed by only one thread. Interpreters cannot be used across multiple threads."*

6 Conclusion - Tcl threading from C Threads

Using Tcl interpreters in separate threads requires, absolutely, that the threaded interpreters be started (Tcl_CreateInterp) and stopped (Tcl_DeleteInterp) from within the thread that they execute in. This all makes perfect sense if Tcl is managing all of the threads in the application and lower level bits of code go off and do various low level things in a thread safe manner. It is a disaster if you are trying to plug Tcl into an existing threading system that uses the thread pool pattern in which you don't have control of or access to the individual threads, but instead submit tasks to a work queue that eventually dispatches them to threads.

Construction of the interpreters following this rule is fairly easy. Simple lazy creation of the interpreter from inside the execution thread and access through a thread specific variable or array slot works just fine.

Cleanup is much more problematic. Luckily, the guy that writes the work queue and thread pool code for Calibre LVS is just four doors down from me. Also, the architecture of our application is such that there are natural sequence points after each high level operation during which all of the threads are quiet. In Calibre, a new API that implements a "perform this task on each thread" routine which I now use to clean up interpreters. If we had a more heterogeneous work queue, though, this approach might not be possible. It would be much cleaner for this type of usage if

Tcl_DeleteInterp were able to clean up a quiet interpreter regardless of which thread it was created on.

Having found a solution to the Tcl threading issues, we can examine the performance of multiple Tcl interpreters doing user defined calculations in parallel on large numbers of parallel processors where we look for the next performance bottlenecks.

7 Acknowledgements

Special thanks to Fedor Pikus for helping me understand the pthread library more fully. Thanks also to Gerald W. Lester for his response to my initial query on comp.lang.tcl

8 References

"Practical Programming in Tcl and Tk" - Brent B. Welch, Ken Jones, with Jeffrey Hobbs, Prentice Hall, 2003
"Techniques for Optimizing Applications – High Performance" - Garg & Sharapov, Prentice Hall 2002 p. 394
"Thread Pool Pattern" - Wikipedia article

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 6
November 15 15:15-16:15

editTableA Generic Display and Edit Widget for Database Applications

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
http://www.noucorp.com
clif at noucorp dot com

October 24, 2012

Abstract

An application that includes a database always requires a set of pages to edit the contents of the database.

The bulk of the edit pages are simple and easy to write. Even with Tk's ease in constructing simple data-entry pagespages, writing a dozen procedures takes time that could be spent on the more interesting parts of a project.

The goal of the EditTables package is to provide a set of good enough pages with no effort on the part of the programmer, better pages with a bit of effort, and a framework for building the fully featured pages an end user will demand.

The implementation uses TclOO for a base class with mixins to provide customized behavior for particular database engines. The test engines are sqlite3 and TDBC.

1 Introduction

One application of the 80/20 rule is that 80 percent of an application is the code that's boring to write, and 20 percent will be fun. In a database application with many data entry screens this ratio could push 95-5.

It would be nice if the 95 percent part of a database application could be generated without requiring 95 percent of the programmer's time.

It turns out that this can be done.

There is enough information in an SQL schema to create a simple entry screen. The following schema can be used to generate an adequate GUI:

```
table create phoneList {
  id integer unique,
  name text,  -- person
  phone text, -- phone number
  type text -- home, work, mobile
}
```

The GUI would resemble:

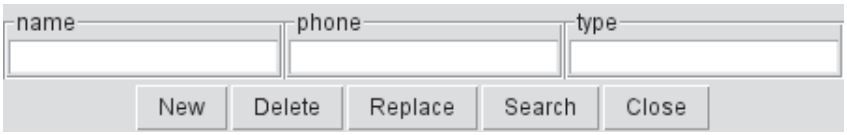


Figure 1: Simple, uninstrumented GUI

By adding some extra information about the fields, the GUI can be improved to resemble the following image:

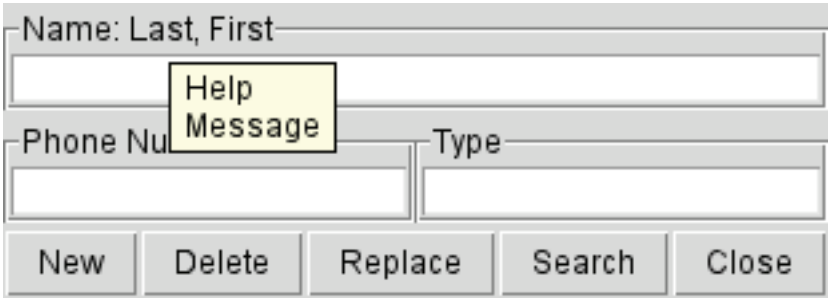


Figure 2: Simple, instrumented GUI

To paraphrase many lolcats, *Simple tables are Simple*. In order to be useful, the `EditTables` package needed handle one-to-one mapping schemas, such those containing references to other tables.

```
table create phoneList {
  id integer unique,
  name text,
  num  text,
  typeid integer references types
}
```

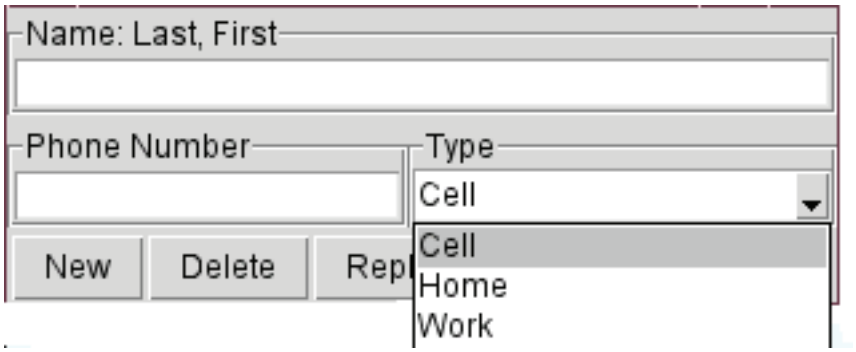


Figure 3: Reference, instrumented GUI

The package also needs to handle many-to-one mapping, perhaps a fixed number of fields as shown below:

```
table create person {
      id integer unique,
      name text,
      address text,
    }
  }
table create phone {
      id integer unique,
      num  text,
      personid integer references person,
      typeid integer references types
    }
```

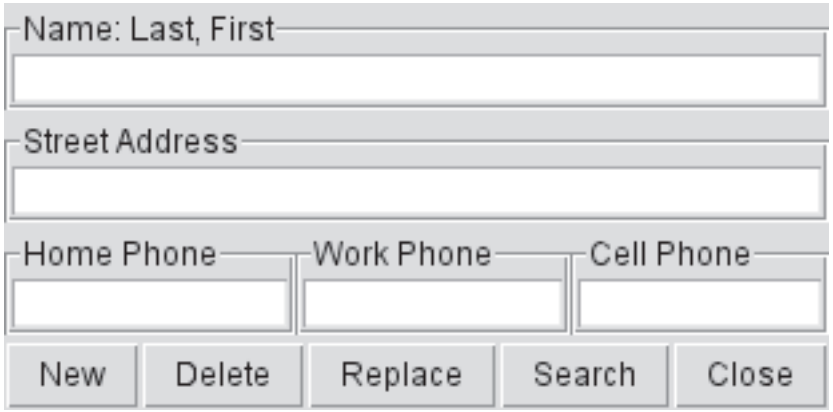


Figure 4: Reference to table, instrumented GUI

The trick to generating these sorts of GUI's instead of just a simple GUI from the pure schema requires adding some layout instructions.

The next requirement is to validate inputs. This may need to be done on a per-field basis, or when the user submits a page. Both of these options are supported in this package.

Finally, a useful package needs to work with multiple database back ends. It would be nice to only support TDBC, but in the real world, TDBC doesn't have enough penetration (yet).

There are features in Tcl 8.6 that allow all of the requirements to be met.

The package is written using TclOO. The base package understands Tk and how an SQL schema is laid out. It provides some primitives that the other classes can use to find a primary field, find references, etc.

If the project starts using the `EditTable` package, the layout instructions can be embedded into the SQL schema and extracted as needed. If `EditTable` is being shoe-horned onto an existing database, or if you find the idea of mixing layout commands and data definitions distasteful, the instrumentation can be provided by modifying the method that acquires the layout commands in the Tcl object.

Validation is accomplished by adding flags to fields with some common validation routines available in the main class, and the capability of adding custom validation if an application requires it.

Finally, support for multiple database backends is accomplished using TclOO's `mixin` feature.

The current version of `editTable` is the fourth or fifth iteration of ideas. It's being used in a commercial product. It's also likely to see a number of modifications before it gets used in another product.

2 Using `EditTable` package

2.1 Creation

The `EditTable` package is constructed using the TclOO megawidget design pattern described by Donal Fellows (Tcl/Tk Proceedings 2009) and in my book (`Tcl/Tk: A Developer's Guide`, Chapter 11). This pattern emulates a standard Tcl widget, allowing the new `editTable` object to be created like any other Tk widget.

Syntax: `editTable` *`widgetName mixin dbEngine args`*

`widgetName`	Name of the widget, normal Tk style
`mixin`	Name of the db engine mixin
`dbEngine`	Name of the db engine to use
`args`	Optional arguments for opening the db engine

Creating an `editTable` connected to an SQLite database named `test.db` would resemble:

```
set obj [editTable .t1 SQLITE3_support sqlite3 -dbArgs test.db]
```

Doing the same with a TDBC engine that's connected to the SQLite database would resemble:

```
set obj [editTable .t1 TDBC_support sqlite3 test.db]
```

Once an `editTable` object has been created, it can be used to query various parameters using the `config` or `configure` commands (both map to the same underlying code.)

Like Tk, the `configure` command will accept no arguments to return all configuration options and values, a single argument to report the value of an option, or a pair of arguments to set a value.

One of the options that can be extracted is the underlying database pointer. Extracting this allows an application to interact directely with the database object opened by `editTable`. This end-run functionality allows applications to easily extend the behavior of the `editTable` class.

The next snippet shows creating an `editTable` object, extracting the TDBC database object and using that to create a table in the underlying sqlite3 database.

```
set obj [editTable .t1 TDBC_support sqlite3 test.db]
set db [$obj config db]
$db allrows {
  CREATE TABLE person (
    id integer unique primary key,
    loginid text, -- loginID
    fname text, -- First name
    lname text, -- Last name
    addrRef integer references addr
    );
}
```

There are several methods defined for the `editTable` object. The most commonly used are:

- *editObj* `config`
- *editObj* `getSchema`
- *editObj* `makeGUI`
- *editObj* `populateBySearch`

The config command will let the application query or set configuration options. If it is called with no key, it will return a list of all currently defined keys and values.

Syntax: editObj *config ?key? ?value?*

editObj	A widget created with editTable command
config	Query or set a config option
key	Name of the config option or blank
?value?	Optional value to define an option

The getSchema method will retrieve a schema for a table from the database, or optional schema retrieval process. The default behavior is to query the database. This is used internally to build GUIs for tables containing relations.

Syntax: editObj *getSchema tableName*

editObj	A widget created with editTable command
getSchema	Return the schema for a table
tableName	Name of the table to return Schema for

The makeGUI method is the workhorse that builds a GUI within the editTable frame. It can build a GUI for any table defined within the database. The editTable object can rebuild itself to display a different table as necessary. The default GUI includes buttons to perform simple searches, and add, delete or modify a record.

Syntax: editObj *makeGUI schema*

editObj	A widget created with editTable command
makeGUI	Construct a GUI within the editTable object frame.
schema	a Schema - may be return from getSchema

The populateBySearch method will load values into the fields of a GUI. The query can be any valid query suitable for an SQL SELECT on the currently active table.

Syntax: editObj *populateBySearch query*

editObj	A widget created with editTable command
populateBySearch	Populates the values in a GUI
query	An SQL query

The next example shows initializing a sample database and creating a simple GUI for a table.

```
toplevel .tt
set obj [editTable .tt.t1 TDBC_support sqlite3 -dbArgs test2.db]
set db [$obj config db]

$db allrows {
  CREATE TABLE person (
    id integer unique primary key,
```

```
    loginid text, -- loginID
    fname text, -- First name
    lname text, -- Last name
    addrRef integer references addr
    );
    CREATE TABLE addr (
    id integer unique primary key,
    street text, -- Address
    city text, -- Address
    state text -- Address
  )
}

$db allrows {INSERT INTO addr VALUES (1, '123 St', 'Acity', 'AA')}
$db allrows {INSERT INTO addr VALUES (2, '234 St', 'Bcity', 'BB')}
$db allrows {INSERT INTO person VALUES (1, 'aaa', 'Alpha', 'Adam', 1)}
$db allrows {INSERT INTO person VALUES (2, 'bbb', 'Beta', 'Blocker', 2)}
$db allrows {INSERT INTO person VALUES (3, 'aa2', 'Abel', 'Adam', 2)}
$db allrows {INSERT INTO person VALUES (4, 'aa3', 'Aard', 'Adam', 2)}

$obj makeGUI [$obj getSchema person]
$obj config -table person
$obj populateBySearch "loginid = 'aaa'"

pack .tt.t1
```

The generated GUI resembles this:

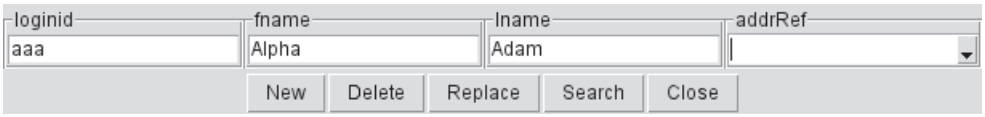


Figure 5: default GUI

2.2 Layout Information

Because we are polite and civilized, we will not call the previous example butt-ugly.

But it is.

An SQL schema is designed to convey the logical date relationships between fields and tables. It is not designed to convey any information about aesthetics.

The aesthetic information is conveyed as a six element list attached to each field that is to be displayed. The elements are shown in the following list. The first 2 are required, the others have default values that may be adequate.

`help`	A message to display in a popup help balloon
`label`	The label to display with this field
`reference`	If this field references another table, this element will contain the `table.field` name of the referenced table. If this field contains non-referential data, this element is blank.
`widget arguments`	arguments for the entry or combobox widget that will be created. These might include `-width` or `-background` arguments.
`row column`	A list of the row and column where this field is to be displayed.
`grid options`	Arguments to be attached to a grid command for this field. These might include `-columnspan` for example.

This technique for conveying layout info is sub-optimal. The requirements grew as the complexity of the application grew.

But it works.

The layout information is included in a Schema by adopting a modification of the standard SQL comment.

An instrumented schema has the layout string appended to a field definition with a triple-dash comment.

A simple GUI can be created like this:

```
set obj [editTable .t1 "" "" -table phoneList]

$obj makeGUI {
 table create phoneList (
 id integer unique,
 name text,  --- {First Last} {Name} {} {-width 40} {1 1} {-columnspan 2}
 num  text, --- {Number with area code} {Phone Number} {} {} {2 1}
 type text  --- {Type of phone} {Type} {} {} {2 2}
 );
}

grid .t1
```

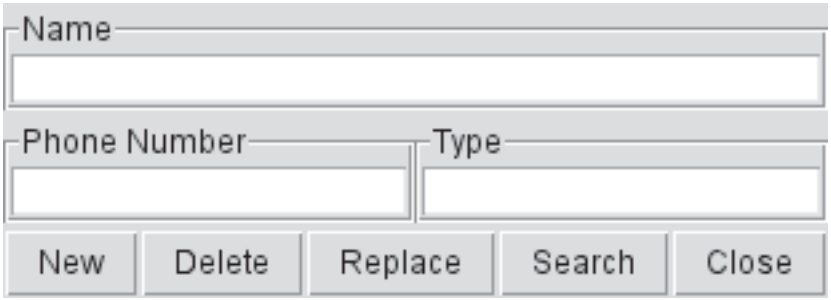


Figure 6: simple GUI with layout info

The `editTable` widget requires that the SQL schema follow strict rules and a field which references another table must include a *`references`* clause.

The presence or absense of the *`references`* clause determines how the third field - the `reference` element is to be treated.

If a *`references`* clause is present, then the `reference` element contains the database field name, or list of field names, to display in each element of a combobox. The fields to display must be named as `tableName.fieldName`.

One pattern used with references is for a table to reference one of several options. The next example is a database in which the book table has a string for title and a reference for author. The author table has separate fields for first and last name. The combination of first and last is displayed in the combobox for selecting an author.

```
set obj [editTable .book SQLITE3_support sqlite3 -dbArgs test4.db]
set db [$obj config db]

$db eval {
 CREATE TABLE book (
 id integer unique primary key,
 title text, --- {} {Title} {} {-width 40} {1 1} {-columnspan 2}
 authorid integer references author
     --- {} {Author} {author.first author.last} {} {2 1}
 );

 CREATE TABLE author (
 id integer unique primary key,
 first text, --- {First Name} {First} {}
 last text  --- {Last Name} {Last} {}
 );
}

$db eval "INSERT INTO author VALUES (1, 'Clif', 'Flynt')"
$db eval "INSERT INTO author VALUES (2, 'Mark', 'Twain')"
$db eval "INSERT INTO book VALUES (1,'Tcl/Tk: A Developer''s Guide', 1)"
```

```
$db eval "INSERT INTO book VALUES (2,'Tom Sawyer', 2)"
$db eval "INSERT INTO book VALUES (3,'Huckleberry Finn', 2)"

$obj makeGUI [$obj getSchema book]
$obj config -table book
$obj populateBySearch "title like '%Tcl%'"
grid .book
```

The GUI generated from this code looks like this:

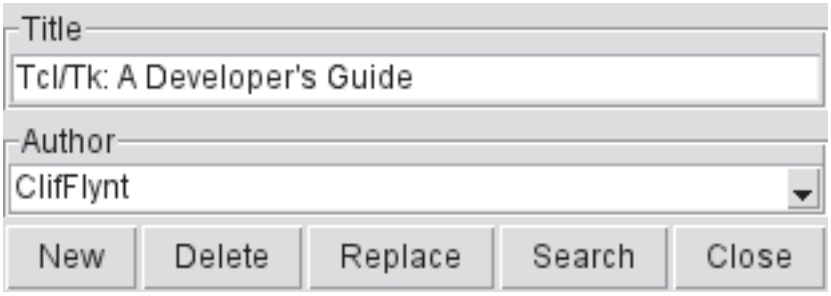


Figure 7: Schema with reference and layout info

Another common pattern is the many-to-one mapping implemented by having a field in one table point back to another table. In a book database, there are an undefined number of keywords that might be attached to a book.

The schema for this pattern would resemble:

```
CREATE TABLE book (
  id integer unique primary key,
  title text,
);
CREATE TABLE keyword (
  id integer unique primary key,
  key text,
  bookid integer references book
);
```

To generate a GUI for this database pattern, the `reference` element in the layout field in the `keyword` table is used to hold the name of a table. When a GUI is generated for a table listed in the `reference` element, an *`Associated values`* section is created in the GUI.

Expanding the previous example to include a reference table, the new table resembles this

```
CREATE TABLE keyword (
  id integer unique primary key,
```

```
   key text, --- {Keyword} {Category} {book} {} {}
   bookid integer references book --- {book} {Title} {book.title} {} {}
);
```

Which creates the following GUI:

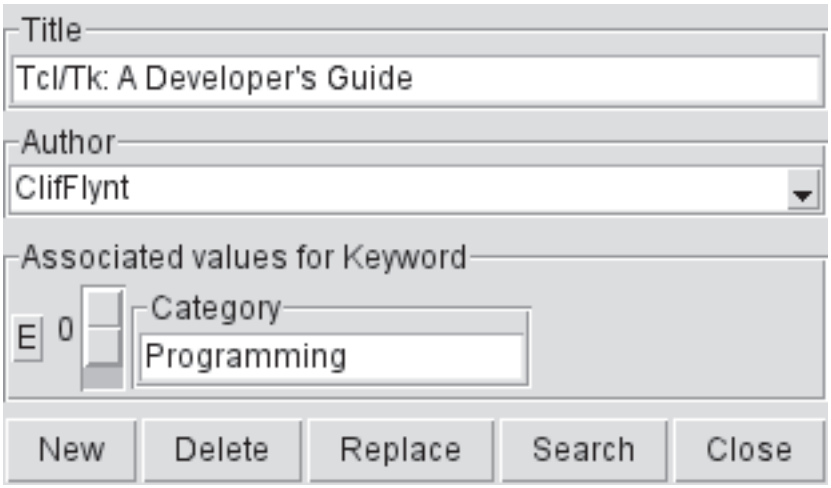


Figure 8: Schema with reference and layout info

The `E` button in the previous image opens a new toplevel to edit a keyword which will be associated with this table row.

2.3 Customization

The `editTable` widget follows the Tk dictum of being adequate with no tweaking, and open for customization if the application isn't suited to the base GUI.

Various degrees of customization are supported. These techniques include:

- using the layout elements
- configuring options for screen or field validation
- building a separate mix-in or inherited class
- tweaking the GUI before using it

Using layout elements to modify the appearance of the GUI works as described.

By default, a `editTable` GUI has no validation. Per-Field validation can be enabled by configuring the `validate,`*`fieldName`* attributes for a GUI. The validation attributes accept a script to evaluate when focus leaves that

field. The widget name (entry or combobox) and field name are appended to the script when it's invoked.

The `editTable` class includes some trivial validation methods including `validatePhone` and `validateNumeric`.

2.4 Extending the class

The current implementation of `editTable` has mixins for Sqlite3 and TDBC. A programmer can add a new mixin to support another database engine by coding the engine-specific methods:

`init`	Initialize a connection to the database.
`getSchema`	Retrieve the schema for a given table. Setting `save` sets this to be the active table.)
`doSQL`	Execute an SQL command and return whatever the command returns. This provides a generic access to the underlying database.
`getTables`	Return a list of the tables defined by this database.
`getPrimary`	Retrieve the name of the primary key.
`doReplace`	Update the DB row based on the contents of the GUI.
`doNew`	Create a new row in the DB based on the contents of the GUI.
`populateBySearch`	Populate the GUI based on an SQL query.
`getValues`	Return a set of values based on a list of *`tableName.fieldName`* values
`getValueByRef`	Returns a value from table *`$tbl`* for field *`$fld`* where the primary is *`$value`*
`populateWithFwdRef`	Populate a GUI that includes forward references based on an SQL query
`closeDB`	Close the connection to the DB

2.4.1 GUI Modification

Since the GUI follows a fixed pattern, it can be modified post-`makeGUI` and before display.

For example, if users are only allowed to view and edit fields, the the delete button can be removed with a command like:

```
$obj makeGUI $specialSchema noClose
$obj configure -table book
```

Getting more aggressive, rather than showing the *`Associated values`* for a row, as they are displayed by default, an application can define a fixed quantity of associated values and a custom display procedure while taking advantage of the bulk of the `editTables` widget.

The next example demonstrates a book database that emulates a card catalog via a slider on the side to select books, and a simplified display of keywords.

It does this by with a special procedure to construct the GUI and a new procedure to extend the normal `editTable` population methods. The new procedure for creating the GUI (`bldBookGUI`) defines a modified schema with a couple extra fields to hold the keywords. The `showItem` procedure uses the `populateBySearch` method to populate the fields that are defined in the `book` schema, and has extra code to populate the extra fields.

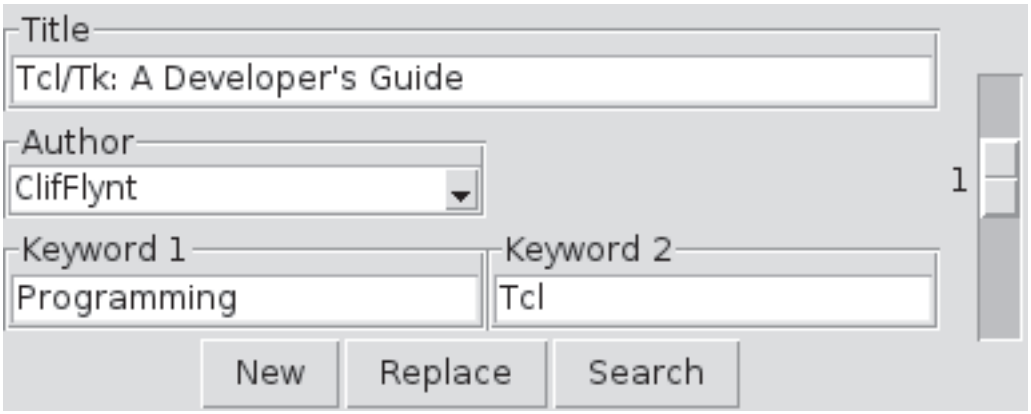


Figure 9: Schema with reference and layout info

```
toplevel .t3

set obj [editTable .t3.t2 SQLITE3_support sqlite3 -dbArgs test3.db]

proc bldBookGUI {obj} {
  set specialSchema {
 CREATE TABLE book (
   id integer unique primary key,
   title text, --- {Title} {Title} {} {-width 40} {1 1} {-columnspan 2}
   authorid integer references author, --- {Author} {Author} {author.first autho
   keyword1 text --- {Keyword} {Keyword 1} {keyword.key} {} {3 1}
   keyword2 text --- {Keyword} {Keyword 2} {keyword.key} {} {3 2}
 );
  }

  $obj makeGUI $specialSchema noClose
  $obj configure -table book
  destroy $obj.buttons.b_Delete
```

```
  set prim [$obj getPrimary book]
  set count [$obj doSQL "SELECT count($prim) FROM book"]

  set scale [scale .t3.sc -from 0 -to $count \
      -command [list showItem $obj $specialSchema .t3.sc ]]
  grid $obj $scale
}

proc showItem {obj schema scale num} {
  $obj populateBySearch "id=$num"

  set pos 0
  foreach l [split $schema \n] {
    lassign [$obj splitSchemaLine $l] def layout
    lassign $layout help label ref args rowcol gridO
    if {($ref ne "") && ([string first "references" $l] < 0)} {
      set dfld [lindex $l 0]
      lassign [split $ref .] tbl fld
      set lst [$obj doSQL "SELECT $fld FROM $tbl WHERE bookid=$num"]
      set item [lindex $lst $pos]
      incr pos
      $obj configure -value,$dfld $item

    }

  }
}
bldBookGUI $obj
```

3 Implementation

The `editTable` widget is implemented as a TclOO Megawidget with mixins to provide database engine customization.

Another standard pattern would have been to make each database engine a class that inherited the base functions from `editTable`. I considered and discarded this pattern in order to have a single widget class (`editTable`) rather than multiple classes (`editSQLite`, `editTDBC`, etc.).

The `editTable` class uses the technique for creating a megawidget described by Donal Fellows in his 2009 paper, and also described in my book (`Tcl/Tk: A Developer's Guide`, Chapter 11) (Shameless plug).

This technique uses a `classmethod` call to create an `unknown` method that checks to see if the first argument is a window name, and if so proceeds to create the new widget.

```
# Create a class method
# Avoid redefining classmethod if it already exists.
if {[info proc ::oo::define::classmethod] eq ""} {
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
   # Get the name of the class being defined
   set cls [lindex [info level -1] 1]
   # Make connection to private class "my" command by
   # forwarding
   uplevel forward $name [info object namespace $cls]::my $name
}
}

oo::class create editTable {
  classmethod unknown {w args} {
# puts "UNKL: $w -- $args"
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }
...
}
```

When a new instance is created, one of the arguments describes the mixin to be added. Since mixins don't have constructors, the `editTable` constructor calls a `init` method defined within the mixin to perform special initializations.

The `editTable` class can create objects that have lives of their own. These includes GUI widgets, open database connections and may include slaved `editTable` GUIs. Because of these additional elements to the `editTable` class, the `editTable` class requires a destructor.

Each `editTable` object contains a list of cloned and referenced objects. The `desstructor` descends upon these like a wolf upon the fold destroying them in gay abandon.

While the `editTable` class is happy to have multiple channels to databases, many database engines are less happy with this. The `init` method in the mixins avoids opening multiple channels. The `editTable` class must also avoid closing a database channel until all users have been destroyed.

A class variable is used to keep track of the number of open database connections. Again, this code was stolen from Donal Fellows' 2009 talk and my book.

```
if {[info proc ::oo::Helpers::classvar] eq ""} {
proc ::oo::Helpers::classvar {args} {
    # Get reference to classs namespace
    set ns [info object namespace [uplevel 1 {self class}]]
    # Double up the list of varnames
    foreach v $args {
      uplevel 1 namespace upvar $ns $v $v
    }
}
}
```

4 Future

The method of conveying the layout information, and what information is required has been evolutionary. It is subject to a rework before the this package gets used in another application.

The schema parsing is a quick and dirty approach that assumes each field definition is contained on a single line. The schema parsing will be enhanced as a method of the base `editTable` class.

It may be possible to reduce the number of methods in the `mixin` classes by better use of the basic methods in the `editTables` class.

5 Summary

The current status of the `editTable` class is *`functional`*. It's in use in a commercial product and is scheduled for a several other in-house and out-house projects.

What started as a simple way to generate a *good enough* GUI that reflected the underlying database schema has expanded into a package that can generate a commercial-grade GUI with capability of hiding the schema and providing a simple interface to a user.

The underlying simple display has been retained with potential tweaks to extend the behavior beyond simple.

As the class is force-fed to other applications I expect to discover more things it *should* do, and streamline the current feature set.

The current escape is available at `http://www.noucorp.com`.

Customizable Keyboard Shortcuts

Ron Wold
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070
503-685-0878

Abstract

Anyone that spends a lot of time using the same software tool becomes very familiar with it. They know how the tool works, what the tool's commands are and when to execute these commands. Coined a "power user", these type of users operates fast. Power users want quick access to commands, they do not want to navigate through a menu to access commonly used operations. User interfaces often address this issue by adding toolbar buttons, but, the fastest method of access is through a keyboard shortcut.

A keyboard shortcut refers to the association of a key sequence with an operation. User interfaces typically have a predefined set of keyboard shortcuts. However, a tool that runs on multiple platforms and that has dozens of windows and hundreds of operations will be unable to define a single set of shortcuts that is sufficient for all users.

Modelsim is a software program written in Tcl/Tk that has been recently enhanced to support customizable keyboard shortcuts. Users can associate a key sequence with a menu pick, a toolbar button, a CLI command or a custom tcl script. In addition, users can specify that the key sequence is applicable for the entire tool or for just a specific window. Implementing this functionality presented several technical challenges. Tcl/Tk has a unique methodology for processing keyboard events and a successful solution requires an architecture that functions within the bounds of this methodology. This paper will discuss the basic architecture as well as the technical challenges that were faced and how they were addressed.

Keywords

Tcl/Tk, keyboard shortcuts, bind, bindtags, customizable

1. Introduction

Modelsim is an integrated development environment (IDE) used by electronic designers to develop, debug, simulate and test electronic designs. It supports several different hardware description languages (HDLs) - such as VHDL [1]

and Verilog [2]. Modelsim's user interface is comprised of many unique windows, toolbars, menus, popups and a command line interface. The user interface is written entirely in Tcl/Tk.

Developing a functional, well tested electronic design can take several weeks to several months. Users that spend this much time working with the same tool become 'power users' [3]. A key behavior of a power user is their desire to perform operations quickly. Within a graphical user interface, there are many ways to perform an operation such as selecting a menu item from a popup, clicking a tool bar or by typing the command into a command line interface. While each of these methods has their advantages, none of methods can be executed as quickly as a keyboard shortcut.

A keyboard shortcut[4] is a key combination that performs a certain command. The efficiency of a shortcut key comes from that fact that it can be invoked entirely from the keyboard. A user isn't required to position the mouse cursor or click mouse buttons.

Keyboard shortcuts are not a new concept. On some platforms, such as Microsoft Windows ™, there is a standard set of keyboard shortcuts. Given an operating system, a tool developer can identify the common shortcuts and implement them within their tool. However, a conflict can arise if the tool supports multiple platforms, and the standard shortcuts are different between the two platforms. For many years Modelsim detected the platform that was in use and defined the shortcuts based on the platform. Although this is more flexible than a single shortcut definition, the solution is still incomplete. Users want a shortcut definition that matches their own expectations, and they want shortcuts for the commands that they most often use. Since there is not a single set of keyboard shortcuts that will appease all 'power users', the best solution is to allow users to define and customize their own shortcuts.

2. Shortcut Fundamentals

Implementing a shortcut in Tcl/Tk requires the bind[5] command. In its simplest form, the bind command associates an event, like a key stroke or mouse event, with an action, like a procedure call. For example, consider this bind command:

```
bind .a.b.c <control-key-x> "Control_X_pressed"
```

This bind will result in the procedure "`Control_X_pressed`" being called when the widget .a.b.c receives the control-x key. In this example, the binding is placed on the widget .a.b.c, but bindings can also be placed on the name of a widget class. When a binding is placed on the name of a widget class, all instances of that widget inherit the binding. A binding can also be placed on "all" which causes all widgets to inherit the binding. Widgets also have the notion of bindtags. A widget's bind tags is a list of tag names, which may include the widget's name, that class name and all.

```
bindtags .a.b.c {.a.b.c My_Class . all }
```

When an event occurs on a widget, it is applied to each of the window's bind tags, in the order in which they are defined. If the bind tag has a binding definition for the event, then the bindtag's script is executed.

3. Capturing keyboard events

Modelsim's user interface is comprised of many windows. These windows are actually just widgets that contain other widgets, but from a user's standpoint, a window is a self contained tool providing specific functionality.

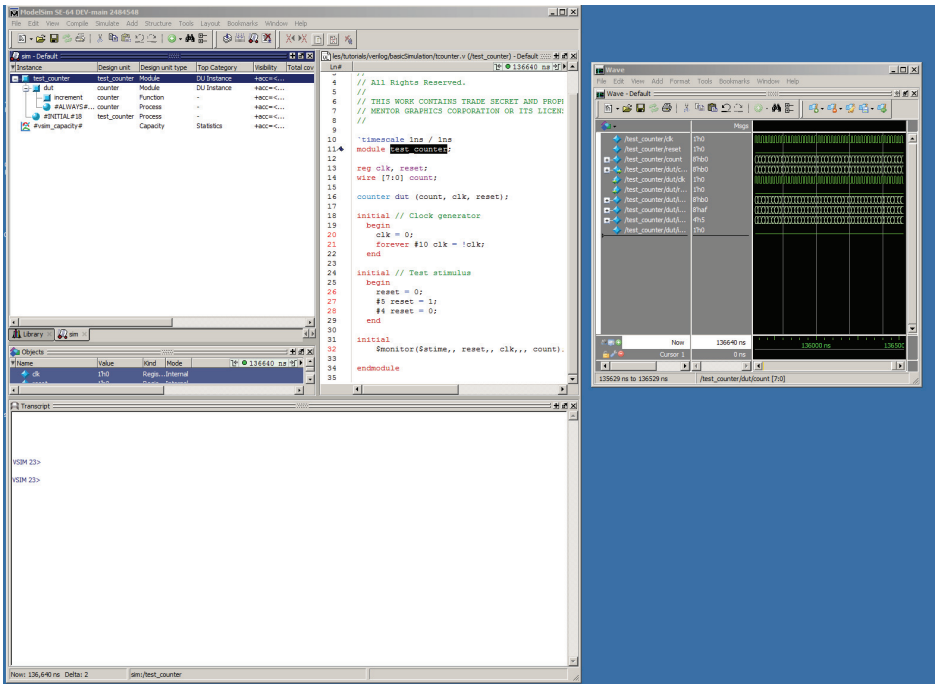


Figure 1 - Modelsim Windows

When a user activates a particular window and issues a keyboard event, such as key-delete, there is an expectation that whatever is selected in the window will be deleted. However, a window may be comprised of multiple subwidgets and each subwidget can take focus and thus be the target of the keyboard event. The user may have activated the window by clicking in any of the subwidgets. From the users standpoint, the shortcut key is defined by the window, not the individual subwidgets that make up the window. This creates a technical challenge in that every subwidget must have an identical binding. There are other possible solutions, such as forcing focus to a particular widget which contains the correct bindings, but these solutions require a detailed understanding of a window's construction. Modelsim has over 50 distinct windows, so defining a customizable binding architecture that requires significant window specific changes is not feasible given this project's time constraints.

Rather than managing the customizable bindings on a widget, an alternative approach was taken. Applying a binding to the 'all' tag provides a means of catching an event that is independent of the target widget. Given the following bind command:

```
bind all <key-delete> "Binding::ServiceBinding %W $key %k %x %y %X %Y %A"
```

a user can click anywhere in a window that is comprised of subwidgets, hit the delete key, and the procedure Binding::ServiceBinding will be executed. There are, however, two exceptions. The first is that all widgets must have the 'all' tag in their list of bindtags. This is not an issue with the Modelsim environment. Widgets receive the 'all' bind tag by default, and this bind tag is not removed from any widget. The second issue that can prevent the 'all' tag from receiving an event occurs if there is a binding for an event on one of the bind tags found earlier in the bind tag list and the binding script issues a break. For example, given these definitions:

```
bind .a.b.c <key-delete> "Delete_Something;break"
bind all <key-delete> "Binding::ServiceBinding %W <key-delete>"
bindtags .a.b.c { .a.b.c all}
```

the call to `Binding::ServiceBinding` will never occur. The binding tag .a.b.c is found earlier in the bind tag list than the bind tag all, so it is executed first. Since the binding script contains a break, all bindtag processes are halted. This behavior is fundamental to Tcl/Tk's bind processing algorthim, and the only way to assure that `Binding::ServiceBinding` is executed is by eliminating the break. The example below replaces the bind command on the widget with a procedure call, `Binding::DefineBinding:`

```
Binding::DefineBinding .a.b.c <key-delete> "Delete_Something"
bind all <key-delete> "Binding::ServiceBinding %W <key-delete>"
bindtags .a.b.c { .a.b.c all}
```

Replacing the bind call with a call to `Binding::DefineBinding` serves two purposes. First it eliminates the binding conflict between the `.a.b.c` tag and the `all` tag. More importantly, it captures and stores the *intent* of the original bind command. In this example, the intent can be described as *"if widget .a.b.c is the target widget and the delete key is hit, the procedure Delete_Something should be called".* When `Binding::ServiceBinding` recieves an event, it compares the target widget and the key event with the binding defintions that have been defined via `Binding::DefineBinding` command. If a match is found, the script associated with the binding definition is executed.

4. The Binding database

Replacing a bind command with a command that saves the bind's intentions results in a database of binding definitions. Not all bind commands use the bind database. Many bind commands are not candidates for shortcut keys, such as binds that are based on mouse buttons or motion events. For example, the right mouse button raises a window's popup. If a user changed this binding they could lose access to the popup menu. Only bindings that are intended as a shortcut key are stored in the binding database.

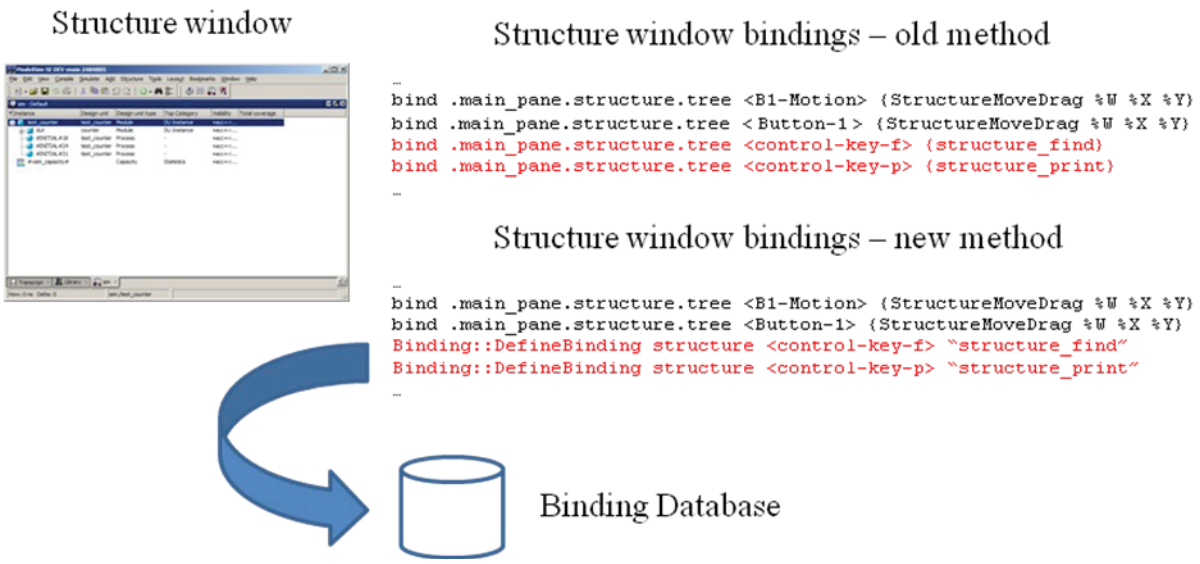


Figure 2

Storing binding definitions in a database has several advantages. First, changing a binding involves nothing more than changing the database. Since the bindings are no longer placed directly on widgets, knowledge of a window's widget hierarchy is not required when modify or adding a binding definition. The database also allows for persistent storage and binding definitions can easily be imported and exported. Lastly, determining what shortcuts are available in any given window becomes a trivial task.

5. Processing a keyboard event

The following tcl code creates a binding on the bindtag `all`. The binding will be in place for all widgets that currently exist as well as any widget created in the future.

```
foreach key [Supported_Shorcut_Keys] {
      bind all $key "Binding::ServiceBinding %W $key %k %x %y %X %Y %A"
}
```

When a shortcut key event occurs, regardless of the target widget, `Binding::ServiceBinding will` catch the event. `Binding::ServiceBinding` then examines the binding database and determines if there is a binding script associated with the event. If there is an associated script, the script is invoked.

6. Binding Priority

Modelsim is comprised of many windows. Each window has commands that are specific to only that window as well as keyboard shortcuts that reference these commands. Executing a window specific command requires that the window be activated. Likewise, a window's keyboard shortcut is only valid if the window that it is associated with it is active. Binding definitions that are associated with a particular window are called *window bindings*.

There are, however, commands that are not window specific. These commands are available without regard to the active window. For example, Modelsim's 'open' command will open a source file for editing and this command is always available. Binding definitions that are not associated with a specific window are called *global bindings*.

In addition to window and global bindings, there is also a distinction between intrinsic bindings and custom bindings. An *intrinsic* binding is one that is defined by a tool developer. An intrinsic binding is built into the tool and it is available to a user the first time they run Modelsim. A binding that a user adds is called a *custom* binding.

The distinction between window and global bindings and whether they are intrinsic or custom is important when an event matches more than one binding definition.

Consider the following scenario; a global intrinsic binding is added by a developer, say control-s, which raises a generic search dialog. In addition, a window intrinsic binding is added by a developer to the source window, it also uses control-s, but the bind script is different, it issues a command to save outstanding edits. Next, a user redefines the control-s binding definition for the source window to yet another command. When the user opens a source file and issues control-s, `Binding::ServiceBinding` will be called and it will examine the binding definitions database, looking for a match. This search will result in three matches, each with a different binding script. Only one of the binding scripts should execute, determining which binding definition to execute requires rules of priority. There are four categories of binding definitions, intrinsic global, custom global, intrinsic window and custom window. Given these four categories, we concluded that a window binding definition has precedence over a global binding definition and that a custom binding definition has precedence over an intrinsic binding definition. Implementing these rules of priority results in the following order of precedence. (Figure 3).

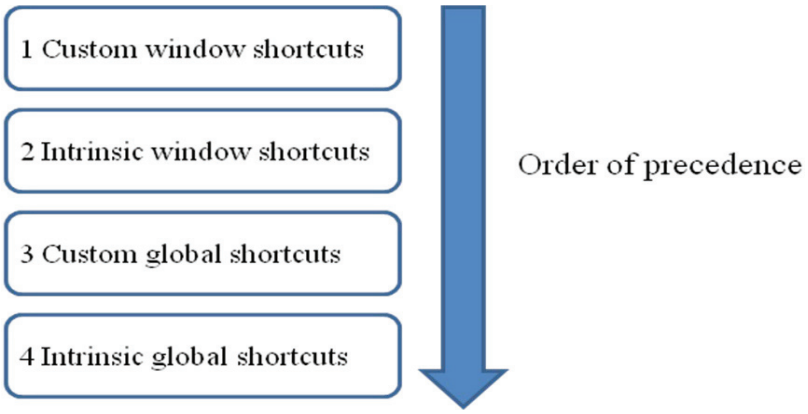


Figure 3

7. Editing the binding definition database

With an architecture that supports customized keyboard shortcuts, the final requirement is defining a user interface for adding, modifying and deleting keyboard shortcuts. The user interface must present the information found in the binding database in an understandable form, as well as provide operations for adding and modifying the database.

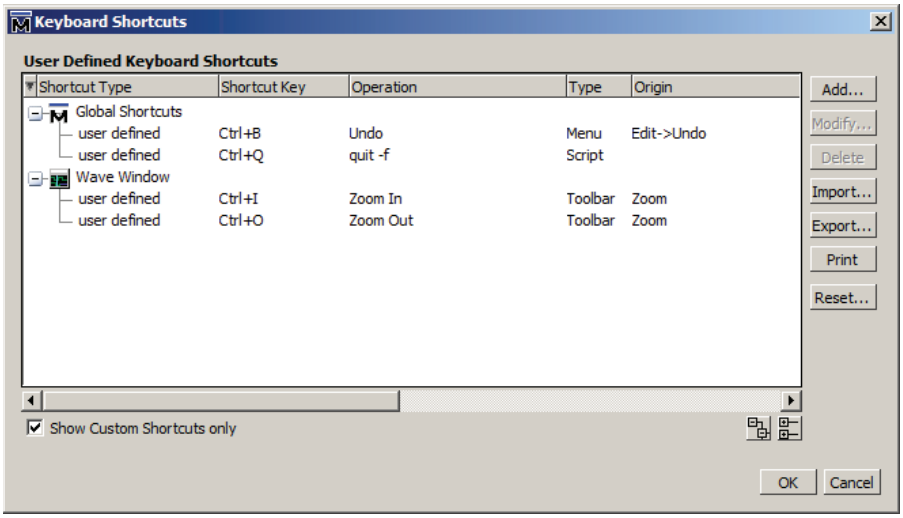
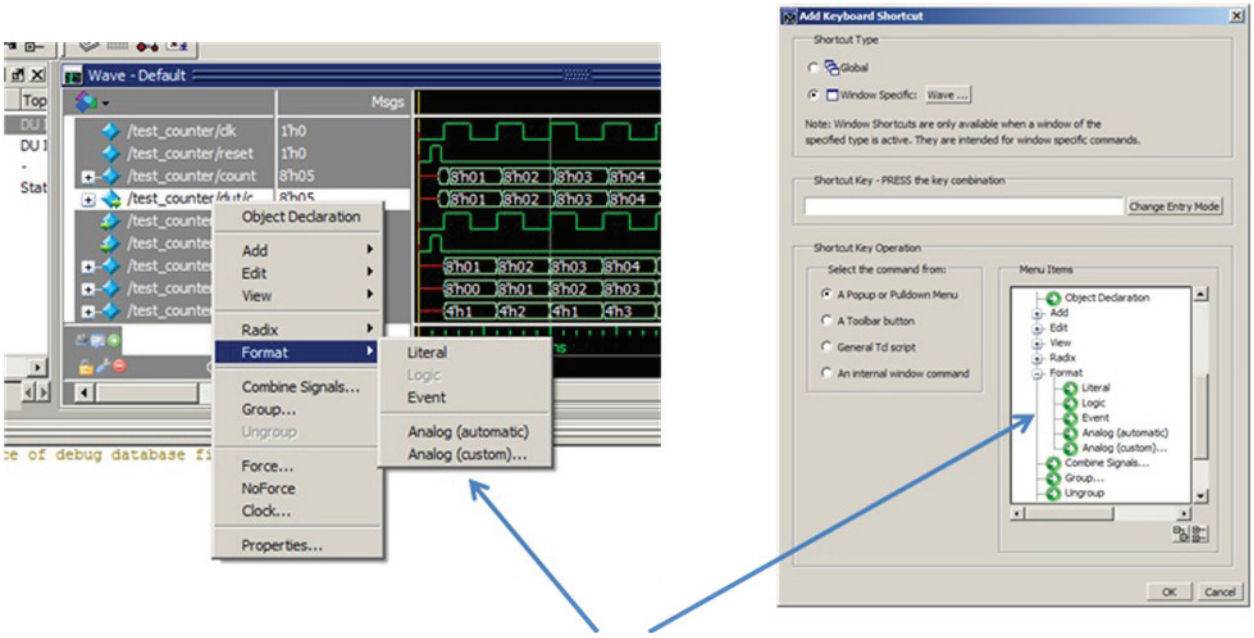


Figure 4

The keyboard shortcut dialog (Figure 4) lists the shortcuts found in the binding database. By default, only shortcuts added by the user are listed. Unchecking the check box 'Show Custom Shortcuts only' will cause intrinsic shortcuts to be displayed as well. Shortcuts are grouped by the window that they are associated with or as a global shortcut. A keyboard shortcut list item displays its shortcut type, the associated shortcut key, as well as information on the menu item or toolbar that the key is associated with.

Adding a keyboard shortcut is nothing more than adding a new entry in the binding database. From a user's standpoint, they simply want to associate a key sequence with a menu item, toolbar button or a tcl script. This simplistic requirement contains a technical challenge; specifically how does one present a user with a complete list of menu items and toolbar buttons?

A fewl years ago Modelsim's user interface underwent a re-architecture to address issues due to an ever increasing number of windows[7]. This re-architecture included a well defined API for creating menus, menu items and toolbars. The API is essentially a group of wrapper functions that embed the actual Tk menu functions. Since these wrapper functions are used for all menu and toolbar creation, they provide a single point for capturing and saving menu creation and hierarchy.



The menu items and their commands are captured at the time of creation. This data is used later by the "Add Keyboard Shortcut" dialog, providing a list of menu items that can be selected.

It is important to display menu items using the same name and hierarchy that is found in the menu itself. This is also true for toolbars, their name and listing order must also match the actual toolbar. Matching hierarchy makes finding the menu item or toolbar much easier.

8. Teaching the shortcuts

If a user is not aware of a keyboard shortcut, the shortcut will not be used. Although a listing of intrinsic shortcuts can be found in Modelsim's documentation, quite often users do not read the documentation. Modelsim has two features that are intended to help users learn the available shortcuts.

Keyboard Shortcut Quick Help

Modelsim has an intrinsic keyboard shortcut that will raise a temporary dialog. This dialog lists the keyboard shortcuts that are currently available for the active window.

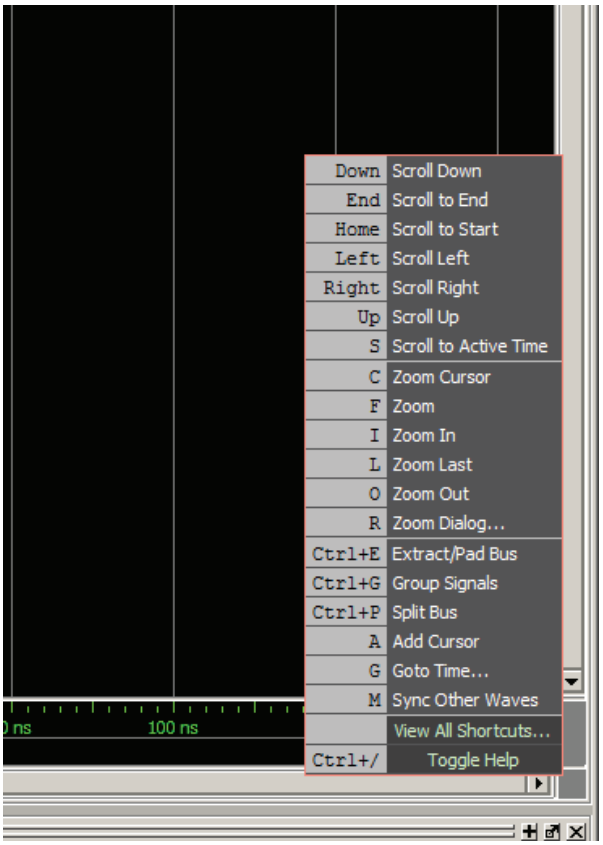
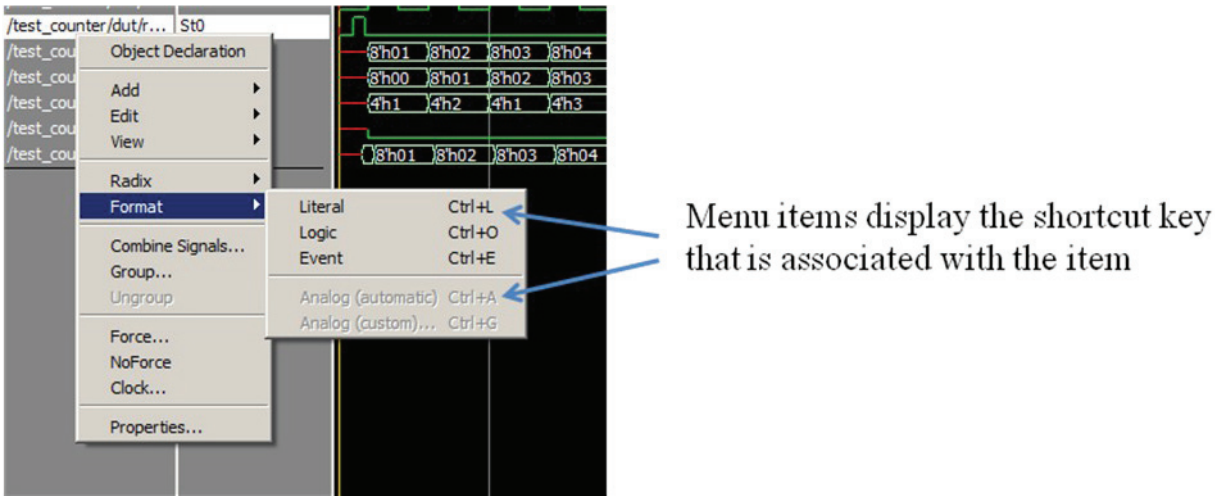


Figure 5 Keyboard Shortcut Quick Help

Dynamic Menu Item shortcut key association

If a menu item has an associated shortcut key, it has become common practice to display the key sequence to the right of the menu item text. We modified our default menu post command to query the default binding database before rendering each menu item. If a menu item has an associated shortcut key, the shortcut key is displayed to the right the menu text. The shortcut key display is dynamic in that it is not statically defined with the menu item. If a user changes or deletes a shortcut key, the associated menu item will reflect the change immediately.



9. Issues

Custom Window Bindings

When a user creates a custom window binding, they must specify the window type. If they are adding a custom binding to a menu or toolbar, the dialog provides a list of menu items or toolbars to select from. The list of menu items or toolbars is created when the menu item or toolbars is created. If a menu hasn't been created it will not be in the list that the user can select from. If a window has not been opened at least once, there will be no information to display in the add shortcut key dialog. We address this issue when the user selects the window type that they want to apply the binding to. The user selects from a list of window types and only windows that have been instanced at least once in the current session can be chosen. This solution is not ideal, but Modelsim's user interface must support 3rd party windows seamlessly and this approach achieves this.

Dialogs and in place edit boxes

Adding a binding to the "`all`" bindtag generates a lot of event traffic sent to `Binding::ServiceBinding`. The traffic does not generate a performance issue, but there are situations where the active window and the shortcut key match a binding definition, but the binding script should not be executed. A text entry box in a dialog is one example. Modelsim follows a model dialog model. When a dialog is raised, all key events are intended for the dialog, not the underlying window. The service binding routine must first detect whether a dialog is raised before processing a key event. When a dialog is raised and a key is detected, all key events received by the service routine are ignored..

An even more difficult issue occurs when in-place text entry boxes are used. Several of Modelsim's windows use in-place text boxes. For example, when the user double clicks on some text, a text box is placed directly on the window. Unlike a dialog, detecting a text entry box requires examining the widget class name. The service routine must exclude processing for certain class names.

Context specific menus

Context menus are created on the fly, the menu items are typically based upon a current state within the window, such as selection. Context specific menu items are cleared and recreated each time the menu is raised. For example, consider a debugger's breakpoint menu, when the user places the mouse over a visual break point and issues the popup, the menu items are created based upon the specific breakpoint. The menu items could have the name of the breakpoint in the menu text, as well as in the menu command. This type of menu item is not a candidate for a keyboard shortcut and special work is needed to prevent a user from binding to the menu item.

10.0 References

[1] Doulos, A Brief History of VHDL, http://www.doulos.com/fi/desguidevhdl/vb2_history.htm.
[2] Doulos, A Brief History of Verilog, http://www.doulos.com/fi/desguidevlg/vb2_history.htm
[3] http://en.wikipedia.org/wiki/Power_user
[4] http://www.techterms.com/definition/keyboardshortcut
[5] http://www.tcl.tk/man/tcl8.5/TkCmd/bind.htm
[6] http://www.tcl.tk/man/tcl8.5/TkCmd/bindtags.htm
[7] Too Many Windows, Ron Wold, 2010 Tcl/Tk Conference.

Tcl 2012
Chicago, IL
November 14-16, 2012

Session 7
November 16 10:45-12:15

A Guided debugging of EDA software with various components of Tcl/Tk GUI

Roshni Lalwani
roshni_lalwani@mentor.com

Amarpal Singh
amarpal_singh@mentor.com

Abstract

EDA software has various hardware design rule checks that can be debugged easily using schematic widget. The main objective of design rule checking (DRC) is to achieve a high overall yield and reliability for a hardware design. If design rules are violated the design may not be functional at all. This paper presents a flow of using some enhanced Tcl/Tk widgets in an innovate manner that can facilitate hardware designers in debugging various design issues of EDA tools.

1. Introduction

Our Tcl/Tk based GUI software provides a debugging environment to various EDA tools. It is built upon various widgets like schematic widget, dialog boxes, MTIwidgets etc. A schematic generator widget (Nlview) is a visualization software component that helps electronic design engineers to easily understand, debug, optimize and document electronic designs. A schematic window in a Tcl/Tk GUI is a simplified graphical representation of an electrical circuit .The schematic diagram consists of instances, pins and nets that are graphical representation of hardware design netlist. The schematic widget supports a number of features to navigate the user to **interesting parts** of the logic and to present engineering information in relation to the schematic. The Schematic Generator is not intended to extract any engineering data from the netlist - but is designed to generate a schematic as a "**skeleton"** for presenting these data. This implies the need of an engineering system that "feeds" Nlview with data. The data is provided by various EDA tools via our GUI interface. The interface between Schematic Generator and Our Tcl/Tk based GUI is defined by string based API (Application Programming Interface). These APIs provides a simple set of commands, callbacks and configuration properties and makes it easy to visualize and debug the EDA software backend data.

There are certain attributes associated with each HDL components/objects like instances, pins and nets. The idea here is to display this information in the callout box on various objects of the schematic window, in such a way that it will help the user in debugging the problematic design issues. A callout box in schematic window is sticky tag visually associated with each object in the schematic window. A callout box is a nothing but a pixmap formed by few rendering shape and text rendering APIs, so it is very fast and efficient. The call out box is a light weight object

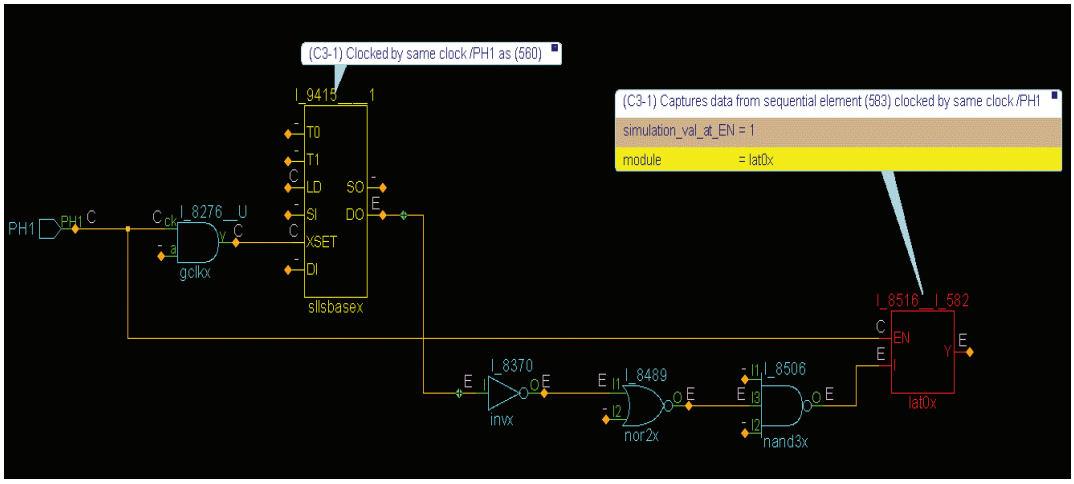
that can be displayed on any object in the schematic. It displays the text that gives a user a way to solve the DRCs and other design issues. The callout box is integrated in our Tcl/Tk GUI using Tcl/Tk interface provided by NLview widget.

Section 2 below covers callout box, highlighting how callout box helps in debugging design rules checks and **Section 3** outlines the enhanced dialogue box and its integration with the callout box functionality.

2. Guided debugging using Callout Box

A design rule error is often associated with one or two instances. There is an error instance where rule error occurs and there is source instance that is starting pointing for the error. For example, there is design rule error which reports a wrong simulation value at the input pin of error instance. The incorrect simulation value is because the source instance and error instance are clocked by same clock. The tool also displays callout box on source and error instance. The message displayed in callout box is an intuitive step to debug and resolve the design rule error. There are also attributes associated with source and error instances. This information is also displayed on the source and error instances in the same callout box.

For example the schematic view with callout box is as follows.



2.1. Callout box Integration in with GUI tool

The schematic generator component integrated with our GUI tool provides an API based mechanism to attach various kinds of attributes with Schematic objects like instances, pins nets. These attributes can be visual or only non-visual in nature. The various kind of visual attributes are like object's display name, object's border color, object's fill color

and object's line style (solid/dashed/thick/thin etc). Outbox is also one special kind of visual attribute attached to Nlview objects where application can along with specifying the text to be shown in an outbox, configure the outbox for its background color, foreground color and color of its various regions. Mostly, this whole configuration information about how an outbox should be rendered is provided via some options during setting outbox on an object.

Here is a simple API interface to demonstrate how an outbox is attached to a Schematic object and its configuration mechanism.

The add_outbox command allows user to add one or more outbox on objects.

add_outbox object_id -name n? ?-value value? ?-bgcolor n? ?-textcolor n? ?- -colorlist <string>? ?-separatorcolor n? ?-crosscolor n? ?-deltaX x? \
?-deltaY y?

The object_id addresses the data base object, one of:inst, net, netBundle,port, portBus,pin, pinBus, hierPin or hierPinBus.
Please note:

Option **-bgcolor** <number> option specifies the color of outbox region. (default value is 1)
For ex : -bgcolor 2 specifies that the color of outbox region will be taken from the property outboxcolor2.

Option **-textcolor** <number> option specifies the color of text for that particular outbox. (default value is 0)
For ex : -textcolor 1 specifies that the color of text for outbox will be taken from the property outboxcolor1.

Option **-crosscolor** <number> option specifies the color of cross for that particular outbox. (default value is 4)
For ex : -crosscolor 1 specifies that the color of cross for outbox will be taken from the property outboxcolor1.

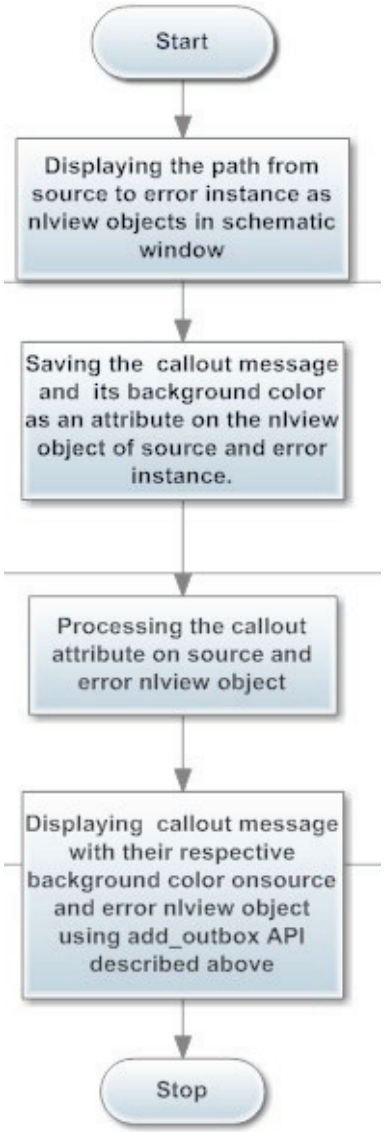
Option **-colorlist** <string> option specifies the in order list of colors of regions for that particular outbox. (by default colorlist is empty string)
For ex : -colorlist "3 5 4" specifies that the color of the first region of the outbox will be taken from the property outboxcolor3, color of second region from property outboxcolor5 and color of third region from property outboxcolor4.

Option **-deltaX** <number> specifies the horizontal shift.

Option **-deltaY** <number> specifies the vertical shift.

The text displayed in the callout box guides the user to resolve the problematic area of EDA design.

2.2. Algorithm for schematic view with callout box



The following pseudo code depicts the example usage of Nlview TCL API for displaying the instances in schematic window with callout box.

proc analyzeDrc { } {

- *Add objects to schematic window*
- *Add callout specific attributes to same objects*
- *Display the objects and its associated attributes in the schematic window.*

}

3. Section2 : Enhance TCL/TK dialogue box

There is an enhanced TCL/TK dialogue box that is used to modify the background color of the text displayed in the callout box. The user can add/delete the text from the callout box using this dialogue box. The dialog box and mainly consist of a combo box, two list boxes, an option menu and Add/Remove button.

Figure2: An innovative dialogue box

3.1. Creation of Enhanced Dialogue box

The dialog box is also enhanced and mainly consists of a Combo-box, two list boxes, an option menu and Add/Remove buttons.

3.1.1. Combo box

The combo-box box is constructed using IWidgets combo-box .The user can select the category from Instances/Pins/Nets by using combo box and the respective attributes gets displayed in left /right list box.

3.1.2. Two list boxes

The two list boxes are scrolled list boxes and are constructed using list box and scroll bar of Tk widget.

3.1.3. Option menu

The Option menu is constructed using tk_Option Menu.

3.1.4. Text Message

The text message area is constructed using text widget of TK.

3.1.5. Add/Remove button

3.1.6. OK/Cancel button

The Add/Remove OK and Cancel buttons are constructed using button widgets of TK.

All the items are arranged in the grid using grid of TK.

3.2. Integration of Enhanced Dialogue box with Callout box functionality.

The user can select the category from Instances/Pins/Nets by using combo box and the respective attributes gets displayed in left /right list box. The attributes which are displayed in right list box gets displayed in callout box. The remaining attributes, displayed in the left list box are associated with the object type but are not displayed in the callout box .The user can add/remove attributes from left/right list using Add/Remove buttons. The respective changes will get applied to the callout box. The user can also modify the background color of the text displayed in the callout box by selecting a color in the option menu. These changes will also get applied to the callout box displayed in the schematic window.

3.3. Pseudo code for Integration of Dialogue Box with Callout Box

proc modifyCalloutBoxMessage { } {

1. *Creating individual widgets like combo box, scrolled listboxes and buttons and arranging them in grid.*
2. *The user can select an item from Pin/Instances/Nets from Combo-box , and the attributes of the same will be displayed in left and right list box*
3. *The user can also modify the background color of the attribute by selecting the appropriate color from the tk_option menu*

   OR

   The user can add/delete the attributes from left to right list box to display/remove the attributes from callout box.

   OR

   The user can execute both the steps.
4. *When the user press Ok button , these changes will be reflected in the callout box*

}

4. Glossary

GUI: Graphical User Interface.

HDL: Hardware description language.

DRC: Design Rule Checks.

5. Summary

Thus we display the DRC error information to user in an intuitive way. We can also modify the background color of the callout box using enhanced TCL/Tk dialogue. Thus we provide

a guided debugging environment to the user by implementing and using the enhanced TCL/Tk widgets.

6. Bibliography

TCL wiki, http://wiki.tcl.tk

An Efficient Method for Rendering Design Schematics Using Tcl/Tk, and Distributed Relational Databases.

Manu Goel(manu_goel@mentor.com), Antara Ghosh(Antara_ghosh@mentor.com)
Mentor Graphics Corporation

Abstract:

Debugging a design in EDA is always a challenging and time consuming process. Designers need to have access to an efficient tool which can provide them the design connectivity in a logical and efficient manner. This paper discusses various challenges faced while writing such a tool for debugging a design and how were they handled to provide a fast and efficient solution. Schematic browser is a Tcl/Tk based GUI application, which user can use interactively to debug and understand the design.

Glossary:

Description of terms used in the paper:

Schematic Browser – Widget to view/trace the RTL level connectivity of a signal in a design
Incremental Mode – Browse the connectivity incrementally based on need
Full View Mode – View the connectivity of a particular segment of design in one go
Waveform Viewer – Widget to view the signal waveforms

Introduction:

The widget being discussed here is the schematic widget, which is a part of GUI provided with a typical simulator. The GUI is used to run simulation, view waveforms and then debug user design in case of any issues, using Schematic window/Wave window. Typically, user needs to compile the design with flags to turn on debugging and then use the GUI to verify/debug the design.

Schematic browser shows a graphical representation of a user's design. The tool converts the RTL constructs of user design, to their equivalent graphical symbols and presents them. The tool should be easy to use interactively to debug or understand the design. Viewing the design graphically, results into much faster debugging and clear insight in the design, making it easy for the user to correlate quickly about how his chip is going to behave.

Schematic window has two modes –

1. Interactive Design Mode: The purpose of this mode is to debug the design incrementally. For example, if the user finds any mismatch in his design output, they will start tracing the design in schematic window starting with the first signal which shows mismatch. User can then select any net and can choose to see the drivers or readers of the selected net to see the connectivity between various constructs around the net of interest. This mechanism can help him in identifying any misconnection or any unconnected logic.

2. Full Design mode: This mode is used to create full understanding of the design and it shows one full module at a time. This mode provides a compact view initially. User can expand to see more details on their area of interest and can compact that again whenever needed.

3. Batch Mode: There is a mode in this tool, where user can use the tool even without bringing up the GUI. This is called batch mode. In this mode, user gets an interactive prompt at the terminal itself, where user can perform certain operations. User can still use some of the above mentioned features in this mode. For example, they can request certain details through command and the details will be provided in text format.
   Even when the GUI is up, user still has access to the prompt, and from there as well user can perform certain operations without actually opening up the Schematic browser GUI.

The GUI and the Schematic widget in discussion here are based on Tcl/Tk and the debug information is stored in a database software tool. User has the flexibility to open multiple instances of the Schematic browser and can perform independent operations in all of them in parallel.

Problem Statement:

As discussed, schematic viewer should be a intuitive tool for debugging any design. The Schematic viewer must give absolute clarity and maximum insight in the design to user, in real time. However, when the design is big, so is the netlist debug database. To manage such huge amount of data, fetch relevant information and drawing it in real time is a daunting task. To achieve that one must make sure there is as little as possible database interaction. That is, same query should not go to the database again and again.

So in one hand, the data access should be managed in a way, so that user can open debug netlist in multiple windows separately. These windows should be completely modular in behavior. Any change in one window, should not affect other windows in any manner. Consider a typical schematic rendering flow. Whenever logic is drawn in

schematic window, there are safeguards to avoid painting same logic again. The scenario of repeated rendering can occur in two cases. One case is, the design has some looped logic, and while path browsing and incrementally drawing, the tool might go through same logic repeatedly. Second case is, the user has issued command to draw same logic more than once. To avoid these, there should be information present against every window, about what logic is already present in Schematic window. These caches of information is checked before drawing any logic, so that for an already drawn logic, the whole process of data fetching, processing and drawing is not repeated. Every netlist object must be processed (processing being the cycle from data fetching to netlist rendering) only once. However the tool must make sure, if the same net is drawn in different schematic window, which should be allowed. This is needed to maintain the window functional modularity as mentioned above.

On the other hand the tool needs to keep database interaction minimum. For example, as mentioned above, same logic should not be drawn in single Schematic window more than once, but same logic can be drawn in different schematic window. However, the effort of information fetching and processing should not be repeated for same logic. This is cardinal as, multiple accesses to debug database is costly and should be strictly guarded against.

The solution for this is to keep the data pool common among different Schematic window. Database access and initial data processing, which is same for all logic, regardless of which schematic window needs the information, should be done in a way so that the effort is not repeated. This is a must for good performance.

So the system has two apparently clashing goals. One is to keep the data model as mutually exclusive as possible to have correct functionality of multiple schematic viewers. The other is to have a common data pool and data processing algorithm.

Added to this is, another use of the system is working of Batch mode. As discussed in the introduction, this mode does not need any GUI window, so the system of information caching needed for schematic windows are not needed here. However this mode can also use the common data pool.

Lastly one must understand, as design gets bigger, DB size also gets bigger impacting the performance in multiple ways –

- Loading the whole DB may take a lot of time
- If full DB is loaded in memory, then memory footprint will increase causing the system to slow down.
- Fetching the required information will be slower

So handling of database also have to be clever. Creating a monolithic database for whole design, and loading the whole database in memory, irrespective of user debug interest locality is wasteful and will harm netlist drawer’s performance.

So to sum up, for schematic to be truly useful, it must have correct functionality, it must be reentrant and fast. The performance (time and memory) is almost as important as functionality is, for schematic debugging.

Solution

In order to create such a tool, the basic requirements are

- Tool should support multiple windows, which can provide similar functionality, but should be completely independent
- It should provide a clear interface to database from where all the necessary information can be fetched
- Whatever information is once processed should not be processed again.
- Non-GUI mode should also work

In order to provide the above functionality, advantage of **object oriented Tcl/Tk** is taken to create the main Schematic window widget. All common functionality that has to be provided and needed to be localized to a single window can be encapsulated inside a class. This class should have functionality of both, incremental and full view mode. Information, once loaded in a window needs to be cached, so that it can be brought back very quickly if user performs the same operation in that particular window. Such information is dependent on context of the window. So a localized caching is a must for such operations. This caching data structure, resides in the class created for the window. The class will also store all the user specified preferences for that particular window.

Second part of the problem is, to fetch the necessary information from the database to show the required functionality in GUI windows, as well as in non-GUI mode. Since a lot of information may be shared among various windows, keeping the code to fetch and store the information separate is a good idea. Since this information may be needed for non-GUI mode as well, it has to be outside the scope of main class creating the widget. Apart from this, since the design connectivity information will be same irrespective of the window from where the information is being requested, all information fetched for this is cached and can cater to future requests without having to go to debug database. So **Namespace** feature of Tcl/Tk came very handy here.

All interface APIs were protected inside the name space. The caching arrays were also protected inside the name space avoiding any misuse of these caches. Further, keeping these interfaces and caching outside the main class also helps batch mode, because that is not associated to any window, so the tool does not need to create any window object for non-GUI mode. It can simply work through fetching the information directly from these name space APIs.

Since the tool uses a lot of caching and the advantage of cache can be fully achieved only if the cached information can be fetched real quickly. The **associative arrays** of

Tcl were of great help for such a purpose. The logic of interest automatically became the key for such an array to store the information in the cache, and to fetch. One simply needs to check if such an entry exists in the concerned cache or not, and if it exists then the information is available very quickly. It does not require any hashing function implementation to store or retrieve the information form cache.

The information to be cached in these arrays is of the form of
`Readers_of_net(/top/mid/in1) {/top/mid/o1 /top/mid/o2…}`

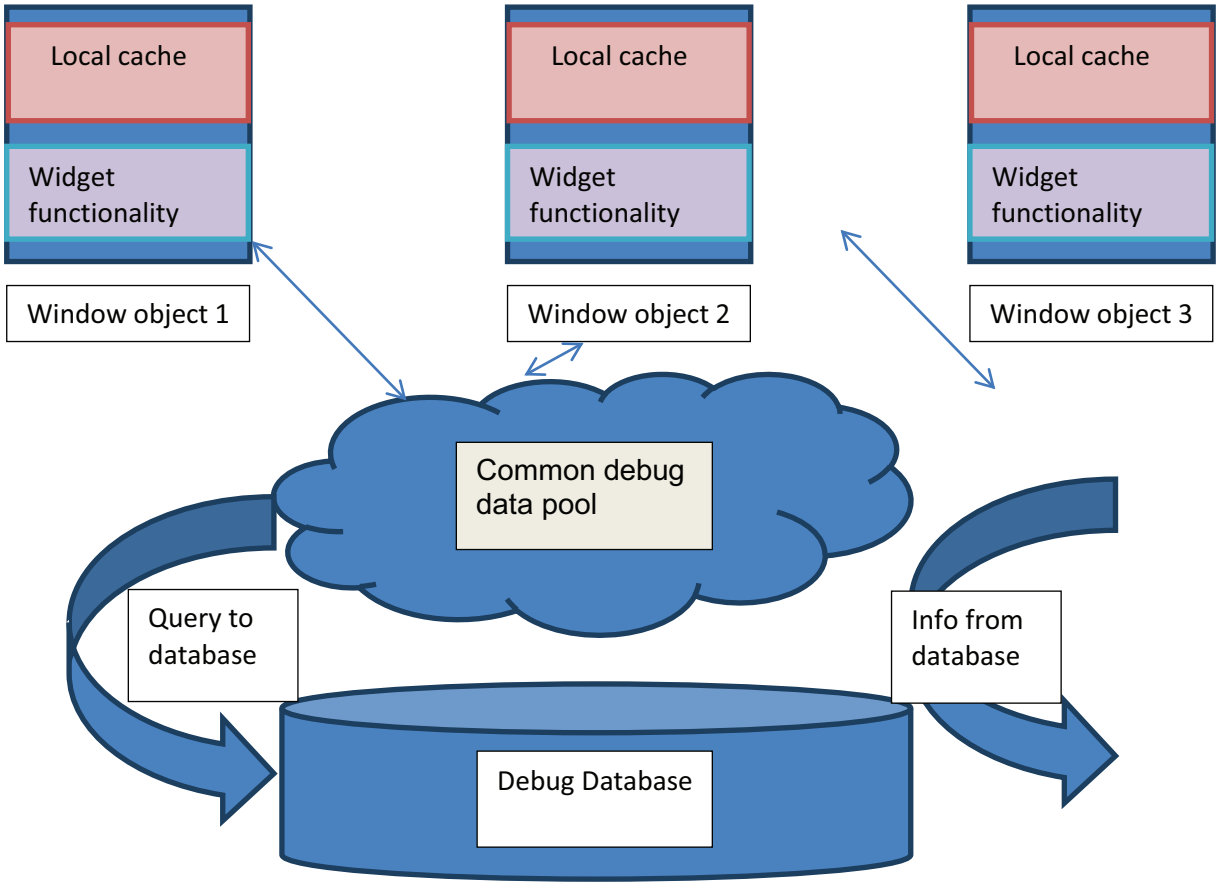


Figure 1: Window widget and debug data pool database

This is how the GUI side problem was solved, now comes the problem of managing the big database. Once that problem is solved, one can achieve a real efficient functionality that is needed. For this, again, Tcl came in very handy. The handling of a big database is divided into two parts. One, database should be modeled in such a way that data can be accessed efficiently. Two the data already fetched into memory should be cached and shared in judicious way among the different processes can access it independently.

The usual size of designs that simulator handle is hundreds of million gates. The debug database size can easily run into several hundred gigabytes. To keep a single monolithic database of that size and fetching information from it is time consuming. So instead of keeping single large design database, GUI should have several small databases for different parts of design.

Dividing databases into several parts also gives GUI the flexibility of generating different amount of debug information for different parts of the design, depending on the user's requirement. The simulator tool's debug database is arranged in a way that the tool maintains several databases for different parts of the design. These databases may have different amount of granularity of information on the parts of design it represents. This depth of the information for each part is dictated by the user. There is always one top node database that does the book keeping for the whole design. It would keep record of what part of design resides in which database. Also it keeps all the global information of the database.

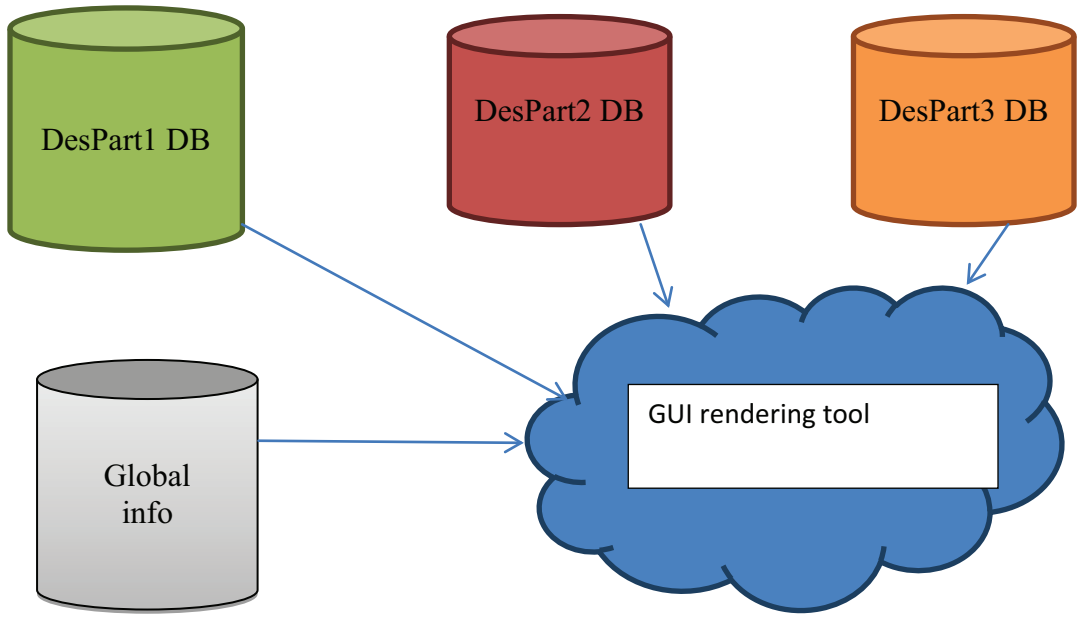


Figure 2: Debug database structure

The above picture shows flow of handling debug database by GUI rendering tool. The design is partitioned into Despart1, Despart2 and Despart3. The segregated parts of design produces separate debug databases. Different colors on the databases indicate that the debug data dumped for that part of the design are different. User has flexibility to modulate amount of database that will be dumped for different parts of design. This is the structure of database that the tool needs to handle.

Any standard database tool allows the user to access multiple databases through the handles they supply to user. The user must request the database tool to open a particular database (say db1) for information access; the tool does that and returns a handle (db1_handle). The user then must interact (execute queries to fetch relevant information) with the database (db1) through this handle (db1_handle).

The database structure for debugging as described above will have several databases and thus database handles for each. The top (or global) database will supply the databases with the path to each of these databases; however GUI must handle opening, accessing right database and closing them on its own. For this purpose the associative array of Tcl language comes in very handy once again. One can easily create an associative array for database handles. For example, for the above case discussed, the Tcl storage would be –

```
array set db_handle_array {}
set db_handle_array(db1)  db1_handle
```

Now, one can simply find out which queries should be carried out on which database (say db1) through the global database, and execute the queries on the database handle ($db_handle_array(db1) stored in the associative array. Because the array is associative, the worst time complexity to pick the correct handle is constant.

The second requirement of having a multiple database in a flexible GUI tool is, one opens a database only when that part of the design is accessed. So the database handles are created on the fly while drawing the part of the design that database holds. This again needs to be a fast (preferably in constant time) action for GUI. The GUI must in constant time determine if a database is already opened and handle is available if not then create that handle. Tcl gives a solution through where one can check if there is any value stored in an associative array against a particular key. So the algorithm of handling this flow would be -

```
If { [info exists db_handle_array(db1)] } {
     Use existing db handle
} else {
     Open db handle for db1
     Set db_handle_array(db1) db1_handle
}
```

The last requirement of handling multiple databases is, one must close all these handles before exiting GUI. Any database on which handle is kept open, might not behave correctly if a subsequent process tries to access it. However the flexibility that Tcl offers while accessing associative array fixes this problem. One can easily traverse an associative array like a list. The “array names” functionality brings all the keys of an associative array for the database handle arrays. The flow of closing all databases is –

```
foreach key [array names db_handle_array] {
     Close database whose handle is stored in
$db_handle_array($key)
}
```

Conclusion:

Using the above described approaches made as develop an efficient schematic widget tool. IncrTcl helped us in creating the main window providing the needed functionality and storing window specific information locally. Name spaced helped us in providing well defined interface to fetch the required data and to manage a common cache of data. This also helped in keeping the GUI side clean and clear from code to interact with database and keeping the GUI code thin. Associative arrays helped significantly in managing multiple databases parallel and at also helped in caching and retrieving the data quickly and easily.
When all of these constructs of Tcl gets combined, then comes the real power of Tcl through which, however complex the widget is, looks easy and trivial to create and maintain.

Bibliography:

[1] An Object Oriented Mega-Widget Set, Mark L. Ulferts, http://incrtcl.sourceforge.net/iwidgets/paper/paper.html

[2] TCL wiki, http://wiki.tcl.tk

[3] Can Distributed DB Provide An Effective Means Of Speeding Web Access Times, Christopher G. Brown, http://jitm.ubalt.edu/XVIII-1/article1.pdf

[4] Using [incr Tcl] to improve stability of a GUI – A Case Study http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2009/proceedings/guis/incrtcl-emulation-debug-gui.pdf