Debugging
Tools & Tactics
Including NEW dff capability
Model Comparison

Sioadff — detailed Design deltas

Identify differences between Design revisions

- Verify that changes to a tool either
  - Leave the data unaltered
  - Change the data as expected

- Support automated ECO generation

- Generate a parseable delta report for any creative application’s purpose
Data-Driven Requirements

- Provide comparison of
  - Two CellViews (oaDesigns)
    - 1 level of parent hierarchy, including Occurrences already expanded
    - Just 2 like Objects within the same or different containers

- Assume a relatively small number of deltas

- Perform canonical traversal of data model
  - Use model’s owned-Object hierarchy
  - Hit every Object & Attribute once
  - Do not consider Logical/Physical equivalence
    - Electrical/Geometric/Behavioral equivalence is app’s responsibility
Extensibility Requirements

■ Enable application customization for:
  ● Model traversal
  ● Object comparisons
  ● Diff reporting
  Include demonstration examples of each of these in the release

■ Deliver usable default behavior for usage out-of-the-box
  ● Traversal: Complete model coverage
  ● Reporting: At least a human-readable output log

■ Configure as a linkable library to an application
Key Definitions

■ Owned Object hierarchy

- UML "black diamond" RTM owner relationship
  - Direct owner only
  - Collections of other refs treated as attributes
- Only one place in Owned Object Hierarchy for each RTM Object

■ Context
  – Either of the two datasets being compared

■ dff Signature ( a.k.a., dffsig )
  - Defines what Objects are a "match"
  - Traversal:
    Recursive descent only for matches
**dffsig Details**

**Type-level:**
- **dffsig** The *uniquely identifying subset of attributes* of an `oaType`
  - *dffsig* cannot be the OID handle ... WHY?!

**Instantiation-level**
- **dffsig** value  Tuple of *attribute values* for a particular instantiation

The *dffsig* values determine "a match" between 2 Objects
- Owned Object Collections are sorted by *dffsig* into arrays
- $1$ and $2$ array elements are compared by their *dffsigs*
  - Only elements with the same *dffsig* values are a "match"
    - *dff* compares (and reports) attributes of these matching Objects and then recurses into their owned Objects
    - if $2$ *dffsig* value has no $1$ match
      - *dff* fires "unique Object" events (reports)

- *dffsig* definitions *can be customized*
  - *dffsig* definition will depend on use cases
The name attribute for Types that have unique names is probably the right `dffsig` for all applications.

Unnamed Types require a compound key of attributes as the `dffsig` – the selection of which may well depend on the design data involved and/or application task at hand.
Type Inheritance, Owned Objects: scalarNet

- scalarNet1
  - name: oaScalarName

- busNet1
  - baseName: oaScalarName
  - start: oaUInt4
  - stop: oaUInt4
  - step: oaUInt4

Intertwined hierarchies:
- oaType inheritance
- owned-Objects

Recursion into owned-Object hierarchy
**Type Inheritance, OwnedObjects: scalarTerm**

- **ScalarTerm1**
  - Name: `oaScalarName`

- **BusTerm1**
  - Base Name: `oaScalarName`
  - Start: `oaUInt4`
  - Stop: `oaUInt4`
  - Step: `oaUInt4`

**Intertwined hierarchies:**
- **oaType inheritance**
- **owned-Objects**

**Reused for Terms & Nets**
- `inherイトObjectAtts()`
- `inherイトDesignObjectAtts()`
- `inherイトBlockObjectAtts()`

**Unique to Terms**
- `intPropA`
- `intPropB`
- `pin1`
- `pin2`

**Recursion into owned-Object hierarchy**

References to Objects
- Associated – but not owned
Algorithm

**Reporting**

- `reportAttsDiff()`: $[\$1 \neq \$2]$
- `reportUnique(1)`: $[\text{is a next } \$1 \text{ Object}]$
- `reportUnique(2)`: $[\text{is a next } \$2 \text{ Object}]$

**Traversal**

- `compare 2 Objects`
- `compare values`
- `reportAttsDiff()`: $[\text{for each attribute}]$
- `sort $1 \& $2$ owned Objects by sig` $[\text{for each Collection of owned Objects}]$
- `compare sigs` $[\text{for each } \$1 \text{ in sorted list}]$
- `toss $1$ off list` $[\$1 \leq \$2]$
- `toss $2$ off list` $[\$1 > \$2]$
- `drain $1$` $[\text{no more } \$1]$
- `drain $2$` $[\text{no more } \$2]$

*Not parallel if changes to Design allowed during diff*

---

**EVENT**

- `EVENT` $[\text{for each remaining } \$1]$
- `EVENT` $[\text{for each remaining } \$2]$

---

**Traversals intimately dependent on compare**
Reporting

- Any number of dffHandlers can be registered
- Each dffHandler can be dynamically
  - enabled
  - disabled
- Traversal/compare functions generate events
  - differentAttribute
  - uniqueObject
  - endObjects
- Report module calls registered event function virtuals
  - foreach dffEvent
  - foreach dffHandler[registered]->uniqueObject()
- Examples in the release:
  - Text logger
  - XML generator
Customization Uses

Example Uses for Customization

■ **Change the traversal**
  - For performance or special considerations
    - Skip some owned Objects in the model (like Props, or AppDefs)
    - Allow values of attributes to affect decisions on events or subsequent recursion

■ **Change the comparison**
  - Substitute a different signature for a Type
  - Install a more elaborate comparison function

  EXAMPLE: From rev1 to rev2 many Nets in a Design had "v2" appended to their names. This would mean *NONE* would correspond if the "name" used as dffsig.
  
  So customize either the
  - `compare()` function to delete "v2" from the name *AFTER* the dffsig has been retrieved.
  - `dffsig` definition to filter out that suffix

■ **Select/Define different reporters**
  - Add Object or attribute "equal" events
  - Enable new `dff` event handling functions
Customization Architecture

- Architecture
  - Derive from existing classes
  - Override/extend existing implementations
  - Register derived code in master indirection tables

mydff.h

mydff.cpp

mydwBusNet

\ldots
- Finish the OA model coverage
- Customer testing of customization architecture
- Develop report handler to perform TCL ECOs
- GUI implementation
DETAILS
Problem Scoping

- **OA Object Model is the focus**
  - Logical/Physical equivalence is *not* addressed
  - Electrical/Geometric/Behavioral equivalence is app's responsibility
  - Many possible interpretations of "equivalence": depends on context
  - User customizations can perform the necessary analysis

- **Solution is optimized for small Object deltas**
  - Only Objects of the same `oaType` are accepted
  - Really different Objects allowed but probably not very efficient

- **Solution is optimized for memory, speed**
  - Dff event data is not saved after event processing
  - Up to an individually registered report handler to:
    - Archive differences, if needed
    - Process dff events out of order
    - Buffer event data to influence future event processing

- **With a dff in progress: No messing with the RTM !**
- **CAYOR**: Customizers must consider inter-Object dependencies
Challenges

■ Abstract owned Objects and Collections contain mixed Types
  ● `oaCollection<oaNet, oaBlock>` contains `ScalarNets, BundleNets`, etc.
  ● `oaBlockObject` is returned by `oaRoute::getBeginConn()`

■ Collection Iters return Objects in random order

■ Only a run-time call to `getType()` will reveal the actual leaf `oaType`

■ Customization requirements include:
  ● Leaf-type granularity for traversal/compare overrides
  ● Ability to inherit some or all of the default traversal/compare processing

■ Sigs are reused many times: sorting Collections, final `compare()`

■ `dff` metadata must be propagated to/from the lowest levels of owned Object recursion, so that event handlers (reporters) can function
  ● Depth of the owner hierarchy
  ● Parent owner Object chains
  ● Counts of differences
Implementation Tricks

- **dfw Wrapper class (dw) around each oaObject**
  - Encapsulates metadata needed through recursion into owned Collections
  - Replaces repeated Type checks with run-time (*vtable*) dispatch
  - Enables registration table when comparing Collections of mixed oaTypes
  - Destroy wrapper as soon as Object is processed

- **Separate comparisons:**
  - First compare attributes
  - Then compare owned Objects – recurse into those with matching sig values

- **Collection Sorting**
  - Use sigs cached in dfw wrappers to line up those Objects that are "the same"
  - Common sort and compare function descends into those like Objects

- **Reporter list**
  - Register any number of reporters
  - Each handles events as needed for a particular purpose (text log, ECO, etc.)
**dff Wrapper (a.k.a. dw)**

- **dw** {abstract parent class} is a **wrapper for**:
  - Stashed context info
    - `_obj` (The `oaObject` being wrapped)
    - `_ownerLevel` (Depth of recursion into owner hierarchy)
    - `_dwParent` (The wrapper of the `_obj`'s owner)
  - Statistics aggregation
    - `_nDffs` (Attribute difference count)
    - `_extraObjs[0]` (Owned Objects in $1 not in $2)
    - `_extraObjs[1]` (Owned Objects in $2 not in $1)
  - Default implementations for compare functions

- **Leaf dw**<typename T> **wrapper types**
  - Derive from the abstract dw parent each `dwObject`
  - Implement the **actions**
    - Compare atts
    - Report dffs

---

**Solo, kamsart.com**

Stashes all the details necessary for any report handler to do its job

Where the traversal, compare, and event generation happen
dff Wrapper (dw) Derivation

**Abstract**

```cpp
<<constructor>> dw()
<<destructor>> ~dw() 
compareAtts ( dw *dw2 ): void 
compareOwnedObjs ( dw *dw2 ): void
```

**Default Implementations**

- Default implementations for `compare*()` are warnings!

**Specialization Imps**

- Specialization imps exist for each leaf:
  - `compareAtts()`
  - Inherit `compareOwnedObjs()`

**Customization Example**

- Override only `compareAtts()`
- Inherit `compareOwnedObjs()` from the `dw` for `oaBusNet`
**Factory Pattern**

- **dff** manipulates abstract interface of FactoryBase to create specific **dw** wrappers

```
<<interface>> FactoryBase
create(key): T*
```

- **dff** engine sees array of abstract **dw**, sorted by sig

```
<<interface>> dw
```

```
<<instantiates>>
```

- Run-time dispatch to the right **oaType** wrapper

```
<<uses>>
```

```
<<uses>>
```

```
0..*
```

```
Factory
create(): T*
```

```
dwObject
```

```
oaType
```

```
oaType
```

```
key
```

```
<<interface>>
```

```
<<uses>>
```
Leaf `oaType` Registration Granularity

**FactoryBase**
- static `registrationList[]`
- First-time automated `registerAll()`
- Default error message for unimplemented wrapper types
- Common constructor registers new wrapper generator (after unregistering old one)

**Factory**
- Template for specific `dw` wrapper types
- Enables dynamic vtable selection of right type when pawing through a list of `ownedObjects`
Leaf Wrapper Registration

$registrationList: vector<dwFactoryBase*>$

create(...): dw*

dwFactoryBase (abstract)

doFactory creates:
  - dwFactory<oaLib> create(): dw<T>*
  - dwFactory<oaTech> create(): dw<T>*
  - dwFactory<oaDesign> create(): dw<T>*
  - dwFactory<oaPath> create(): dw<T>*

<<register>>

registrationList

<<create>>

default-registration

dwFactoryBase creates:
  - dw<oaLibType> create(): dw<T>*
  - dw<oaTechType> create(): dw<T>*
  - dw<oaDesignType> create(): dw<T>*
  - dw<oaPathType> create(): dw<T>*

default-registration

default dff wrapper implementations

dw (abstract)

<<constructor>> dw()
Customized Leaf Wrapper

void dw::dffOwnedObjPair( char const *attName, 
    oaObject *ownedObj1, 
    oaObject *ownedObj2, 
    dw *parent2 ) { 
    vector<dw*> dwsSorted1, dwsSorted2; 
    dw *c1wrap = dwFactoryBase::registrationList[ownedObj1->getType()],
        *c2wrap = dwFactoryBase::registrationList[ownedObj1->getType()]; 
    if (ownedObj1) dwsSorted1.push_back( c1wrap->create(ownedObj1,this,parent2,1) ); 
    if (ownedObj2) dwsSorted2.push_back( c2wrap->create(ownedObj2,parent2,this,2) ); 
    dw::dff2arrays( attName, &dwsSorted1, &dwsSorted2, !ARE_REFS ); 
}
**Inherited Associations**

---

### $dw$ (abstract)

- dwParent: $dw^*$
- dwOtherParent: $dw^*$
- obj: $oaObject^*$
- sig: $oaString^*$
- objOwnerLevel: int
- nDiffs: int
- context: int
- extraObjs: vector<int>[2]

**<<constructor>>** $dw()$

**<<destructor>>** $~dw()$

**compareAtts** ($dw *dw2$): void
**compareOwnedObjs** ($dw *dw2$): void

---

### $ inheritAttObj$ (abstract)

- $\$inherit$: $ inheritAttObj^*$

**compareObjectAtts** ($dw *dw1, dw *dw2$): void
**compareObjectOwnedObjs** ($dw *dw1, dw *dw2$): void
**compareDesignObjectAtts** ($dw *dw1, dw *dw2$): void
**compareDesignObjectOwnedObjs** ($dw *dw1, dw *dw2$): void
**compareBlockObjectAtts** ($dw *dw1, dw *dw2$): void
**compareBlockObjectOwnedObjs** ($dw *dw1, dw *dw2$): void

...
Inherit (abstract oaType) Registration

```cpp
void
dqBlock::compareOwnedObjs( dq *dq2 ) {
    CAST_OBJs_FROM_DQs( oaBlock, this, dq2 );
    DFF_COLLECTION( this, dq2, oaInst , "insts" , getInsts () );
    DFF_COLLECTION( this, dq2, oaNet  , "nets" , getNets () );
    DFF_COLLECTION( this, dq2, oaShape, "shapes" , getShapes () );
    inheritAttObj::inherit->compareBlockObjectOwnedObjs( this, dq2 );
}

inheritAttObj::inherit

Single pointer registration for whole inherit class

class myInheritAttObj : public inheritAttObj {
public:
    virtual void compareBlockObjectAtts ( objDffRec &odr, dq *dq1, dq *dq2 );
    virtual void compareBlockObjectOwnedObjs ( dq *dq1, dq *dq2 );

    virtual void compareNetAtts ( objDffRec &odr, dq *dq1, dq *dq2 );
    virtual void compareNetOwnedObjs ( dq *dq1, dq *dq2 );

    virtual void compareBitNetAtts ( objDffRec &odr, dq *dq1, dq *dq2 );
    virtual void compareBitNetOwnedObjs ( dq *dq1, dq *dq2 );

    ...
}

Inherit (abstract oaType) Registration
Compilation Modules

mydff.o

App-registered modules
diff events

mydff.cpp
overrides

driver.cpp

si2dff.so
traverse + compare

si2dff.cpp
si2dffinherit.cpp

registration

si2dfffactory.cpp

report

si2dffreport.cpp

report handlers

si2dffxml.cpp
si2dfflog.cpp

reporting
diff events

diff events