

Introduction

This is a book about tacc: the TAC language, its compiler, and the runtime framework.

The book will eventually be both a tutorial and comprehensive reference.

There is lots of [example code](#) that you can explore interactively. The code is compiled automatically as part of the development of the book to ensure that examples remain up-to-date and correct.

Please help us to make the book better by [contributing](#).

Interactive snippets

This book contains many "snippets" of code. You can interact with the C++ and TAC snippets through the book itself.

TAC snippets and C++ code generation

Hovering your cursor over the snippet below reveals an "upload" button in the upper-right corner.

Clicking it will redirect you to [Code Explorer](#), where you can see the C++ code that is generated.

```
Person : Tac::Type( name, age ) : Tac::Ordinal {  
    name : Tac::String;  
    age : U32;  
}
```

This is a convenient way to understand how to interact with TAC definitions in C++. It's also useful as a way to develop insight into how the compiler works.

C++ snippets and interactive compilation

The "play" button for C++ snippets compiles and executes them using Code Explorer.

The contents of the standard output and standard error devices are also included.

Try it yourself!

```
#include <iostream>  
  
int main() {  
    std::cout << "Numbers:\n";  
  
    for ( int i = 0; i < 5; ++i ) {  
        std::cout << i << '\n';  
    }  
}
```

Cookbook

This is a collection of "recipes" for working with tacc.

It was formerly a Google Doc hosted at [AID 482](#).

The contents of the document developed very organically. Thus, there is a lot of useful content but it's neither comprehensive nor organized as well as it could be.

Our objective is to develop other sections of the book based on the contents of the cookbook, and eventually to delete it.

Basic attributes

Attributes in TAC are stored as data members of a C++ class. The compiler will also generate accessors and mutators for the attributes. The convention is that an attribute `foo` can be accessed by invoking `foo()`, and `foo` can be set to a new value by calling `fooIs(newVal)`.

```
taccExtension << "attrLog";

BasicAttributes : Tac::Namespace {

    // `BasicEntity` is a type derived from `Tac::Entity` and contains multiple
    // basic
    // attributes. `Tac::Type( name )` means that we are declaring a type that
    // takes a
    // single constructor argument called `name`. `name` is not declared in
    // `BasicEntity`
    // directly; it's declared in the base type, `Tac::Entity`.
    BasicEntity : Tac::Type( name ) : Tac::Entity {
        unsignedInt8Bits : U8;
        unsignedInt16Bits : U16;
        unsignedInt32Bits : U32;
        unsignedInt64Bits : U64;

        signedInt8Bits : S8;
        signedInt16Bits : S16;
        signedInt32Bits : S32;
        signedInt64Bits : S64;

        float32Bits : F32;
        float64Bits : F64;

        boolean : bool;
        string : Tac::String;
    }
}
```

Properties

"Properties" are special keywords that can change the behavior of attributes, types, and constraints. They are defined in [AID 6915](#).

Collections

The size of a collection

A collection attribute with the name `attr` will generate a member function `attrs()` (note the "s") which returns the size of the collection.

For example, given this type

```
MyType1 : Tac::Type() : Tac::PtrInterface {
    item : U32[ U16 ];
}
```

`Size.tac`

the size of the `item` collection can be accessed in C++ like this:

```
#include "Size.h"

int main() {
    MyType1::Ptr myType = Tac::allocate< MyType1 >();
    std::cout << "There are " << myType->items() << " items\n";
}
```

Externally-keyed collections

Externally-keyed collections use a key type that is external to the type stored in the collection.

```
MyType2 : Tac::Type() : Tac::PtrInterface {
    // A collection of `U32` keyed by `U16`.
    record : U32[ U16 ];
    // An ordered collection of `U64` keyed by `Tac::String`.
    age : ordered U64[ Tac::String ];
}
```

`External.tac`

For example:

```
#include "External.h"

int main() {
    MyType2::Ptr myType = Tac::allocate< MyType2 >();

    // Insert 10 with the key "joe".
    myType->ageIs( "joe", 10 );

    // Delete the value with the key 0.
    myType->recordDel( 0 );

    // Delete all values.
    myType->recordDelAll();

    // Print every value in the `record` collection.
    for ( auto iter : myType->recordIteratorConst() ) {
        std::cout << *iter << '\n';
    }

    // Access a collection value.
    std::cout << myType->age( "joe" ) << '\n';
}
```

Internally-keyed collections

Internally-keyed collections use a specific attribute of a type as the key. This is useful because no additional storage is necessary for the key: it's already stored as part of the collection value.

Internal.tac

```
Value : Tac::Type( a ) : Tac::PtrInterface {
    a : U32;
    b : U32;
}

MyType3 : Tac::Type() : Tac::PtrInterface {
    // A collection of pointers to `Value`, keyed by its `a` attribute.
    //
    // The `a` attribute must be a constructor argument of `Value` and must be
    // immutable.
    record : Value::Ptr[ a ];
}
```

Instantiating and non-instantiating collections

An instantiating collection is one that creates new instances of pointer types (like `Entity` or `PtrInterface`). A non-instantiating collection stores pointers to instances that were created elsewhere.

An instantiating collection must always be internally-keyed.

For example:

Instantiating.tac

```

MyRecord1 : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    a : U32;
    b : U32;
    c : U32;
}
MyType4 : Tac::Type() : Tac::PtrInterface {
    // An instantiating collection.
    foo : MyRecord1[ name ];
    // A non-instantiating collection.
    bar : MyRecord1::Ptr[ name ];
}

```

and in C++:

```

#include "Instantiating.h"

int main() {
    MyType4::Ptr myType = Tac::allocate< MyType4 >();

    // Instantiate a new instance of `MyRecord1` and insert it into the
    // collection.
    myType->fooIs( "first" );

    // Add a pointer to the instance to the `bar` collection.
    myType->barIs( myType->foo( "first" ) );
}

```

Fixed-size arrays

In the following type definition, `MyType5::entry` is an array of `U32` values with valid indices ranging from 32 to 40 (both inclusive). Storage will be allocated for $40 - 32 + 1 = 9$ values.

`Array.tac`

```

Index : Tac::Type( value ) : Tac::Nominal, U8 {
    value {
        `range = 32 .. 40;
    }
}
MyType5 : Tac::Type() : Tac::PtrInterface {
    entry : array U32[ Index ];
}

```

Here is an example of using the array from C++

```
#include "Array.h"

int main() {
    MyType5::Ptr myType = Tac::allocate< MyType5 >();

    // Accessing an entry requires first creating an instance of `Index`.
    //
    // Set the entry at index 32 to 400, and then read it back.
    myType->entryIs( Index{ 32 }, 400 );
    std::cout << "The value at 32 is " << myType->entry( Index{ 32 } ) << '\n';

    // An exception is thrown when an out-of-bounds `Index` is created.
    try {
        Index{ 20 };
        tacAssert( false );
    } catch ( Tac::RangeException const & e ) {
        std::cout << e.what() << '\n';
    }
}
```

Stacks

`Stack.tac`

```
MyRecord2 : Tac::Type( name ) : Tac::PtrInterface {
    name : Tac::String;
    scope : stack Tac::String[];
}
```

In C++:

```
#include "Stack.h"

int main() {
    MyRecord2::Ptr myRecord = Tac::allocate< MyRecord2 >( "foo" );

    // Push onto the top of the stack.
    myRecord->scopePush( "foo" );

    // Pop from the top of the stack.
    myRecord->scopePop();

    std::cout << "There are " << myRecord->scopes() << " scopes\n";
}
```

Queues

`Queue.tac`

```
MyRecord3 : Tac::Type( name ) : Tac::PtrInterface {
    name : Tac::String;
    job : queue U32[];
}
```

In C++:

```
#include "Queue.h"

int main() {
    MyRecord3::Ptr myRecord = Tac::allocate< MyRecord3 >( "foo" );

    // Enqueue a job at the back.
    myRecord->jobEnq( 100 );

    // Dequeue a job from the front.
    std::cout << "The latest job ID is " << myRecord->jobDeq() << '\n';
}
```

There is no way to specify the priority of values in a queue. An ordered collection may be an acceptable alternative.

Sets

Set.tac

```
MyRecord4 : Tac::Type( name ) : Tac::PtrInterface {
    // A set of `U32`.
    name : Tac::String;
    id : set void[ U32 ];
}
```

In C++:

```
#include "Set.h"

int main() {
    MyRecord4::Ptr myRecord = Tac::allocate< MyRecord4 >( "foo" );

    // Add to the set.
    myRecord->idIs( 5 );

    // Remove from the set.
    myRecord->idDel( 5 );

    // Check for membership.
    if ( myRecord->idHas( 10 ) ) {
        std::cout << "Found id: 10\n";
    }

    std::cout << "There are " << myRecord->ids() << " IDs\n";
}
```

Iterators

Iterators over TAC collections provide much stronger guarantees than iterators in the C++ standard library. Iterators are not invalidated upon modification of the collection.

For example, it is safe to delete values from the collection while iterating over it: no values are missed.

```
#include "Internal.h"

void
doSomething( Value const & ) {
    // Nothing.
}

int main() {
    MyType3::Ptr myType = Tac::allocate< MyType3 >();

    for ( auto iter = myType->recordIteratorConst(); iter; ++iter ) {
        if ( iter.key() % 2 == 0 ) {
            doSomething( *iter.ptr() );
            myType->recordDel( iter.key() );
        }
    }
}
```

For more details and examples please see [AID 7104](#).

Output iterators for C++ algorithms

The standard library provides the `std::back_inserter()` and `std::inserter()` functions to create output iterators which insert into collections like `std::vector`.

The `TacCollInserter()` macro is similar, except that more information is necessary when invoking it in order to interoperate with the notification mechanism of the framework.

```
#include "Instantiating.h"

int main() {
    MyType4::Ptr myType = Tac::allocate< MyType4 >();
    std::vector< std::string > names{ "foo", "bar", "baz" };
    std::copy( names.begin(), names.end(), TacCollInserter( *myType, fooIs ) );
    // which is semantically equivalent to the following:
    // myType->fooIs("foo"s);
    // myType->fooIs("bar"s);
    // myType->fooIs("baz"s);
}
```

The user should be cognizant about the number of inserts they perform without yielding, as each insertion may potentially notify all the reactors of the collection. One 'workaround' is making the reactors to the collection that the output iterator is modifying use a CODEF. Then the risk of spending a large amount of time inserting into the collection is somewhat mitigated. For more details on CODEFs please see [AID 832](#).

`TacCollInserter` is a preprocessor macro that reduces redundancy when creating pointer to member functions by eliminating the need to spell out the containing entity's type name. If desired, the pointer to member function can be referenced directly in the the underlying `Tac::CollInserter` insert iterator constructor call.

```
#include "Instantiating.h"

int main() {
    MyType4::Ptr myType = Tac::allocate< MyType4 >();
    std::vector< std::string > names{ "foo", "bar", "baz" };
    std::copy(
        names.begin(), names.end(), Tac::CollInserter( *myType, &MyType4::fooIs )
);
}
```

The insert iterator can be created for most TAC collections: set, map, array, queue, etc. and both for their instantiating and non-instantiating variants.

```
#include "Instantiating.h"

int main() {
    MyType4::Ptr myType = Tac::allocate< MyType4 >();

    // Instantiate a new instance of `MyRecord1` and insert it into the
    // collection.
    myType->fooIs( "first" );
    auto iterator = myType->fooIterator();
    std::transform( iterator.begin(),
                   iterator.end(),
                   TacCollInserter( *myType, barIs ),
                   []( auto const & elem ) { return elem.ptr(); } );
}
```

When there are multiple constructor parameters of the destination type, the output iterator must be assigned a tuple of the desired constructor arguments. That tuple object is then `std::apply`-ed onto the insertion function behind the scenes. Given the following types:

```
MultiCtor.tac

MultiCtorParam : Tac::Type( name, val, val2 ) : Tac::PtrInterface {
    name : Tac::String;
    val : U32;
    val2 : Tac::String;
}

MyType6 : Tac::Type() : Tac::PtrInterface {
    // An instantiating collection, with multiple parameters
    foo : MultiCtorParam[ name ];
}
```

We can use the inserter as:

```
#include "MultiCtor.h"

int main() {
    MyType6::Ptr myType = Tac::allocate< MyType6 >();
    std::vector< std::tuple< std::string, int, std::string > > triplets{
        { "a", 1, "A" }, { "b", 2, "B" }, { "c", 3, "C" }
    };
    std::copy( triplets.begin(), triplets.end(), TacCollInserter( *myType, fooIs )
) );
```

Reactors

Basic reactors

At Arista, most of the "business logic" is performed by state machines. These have the type `Tac::Constrainer` in TAC. Notifications are always immediate.

`.tac:`

```
BasicRecord : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    accelerated : bool;
}
// `Tac::Constrainer` can be thought of as a state machine that takes one or
more
// inputs and implements a reactor.
BasicSm : Tac::Type( entity ) : Tac::Constrainer {
    entity : in BasicRecord::PtrConst;
    // A non-const function that is implemented via `.tin` and `.`itin` files. It
    // returns no values and takes no parameters.
    reactor : extern invasive void();
    // Invoke `reactor()` whenever `entity::accelerated` changes.
    entity::accelerated => reactor();
}
```

`.tin:`

```
void
BasicSm::reactor() {
    if ( entity()->accelerated() ) {
        // ...
    } else {
        // ...
    }
}
```

Collection reactors

Changes to collections are handled with a reactor that gets invoked with a specific key. The user-provided implementation must perform a collection membership check to determine if the value was added or deleted.

`.tac:`

```

MyRecord : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    a : U32;
    b : U32;
    c : U32;
}
MyType : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    // An instantiating collection.
    foo : MyRecord[ name ];
    // A non-instantiating collection.
    bar : MyRecord::Ptr[ Tac::String ];
}
CollSm : Tac::Type( input ) : Tac::Constrainer {
    input : in MyType::PtrConst;
    // The `handleFoo()` reactor is invoked whenever a value is added or removed
    from
    // the collection.
    handleFoo : extern invasive void( name : Tac::String );
    input::foo[ name ] => handleFoo( name );
    // This overload of `handleFoo()` is invoked when `CollSm` is created.
    handleFoo : extern invasive overloading void();
    input::foo[] => initially handleFoo();
    handleBar : extern invasive void( name : Tac::String );
    input::bar[ name ] => handleBar( name );
}

```

.tin:

```

void
CollSm::handleFoo() {
    // ...
}
void
CollSm::handleFoo( Tac::String const & name ) {
    // Since `foo` is an instantiating collection, if the key is in the
    collection
    // then it must have been added.
    if ( input()->fooHas( name ) ) {
        // ...
    } else {
        // `name` was just removed from the collection.
        // ...
    }
}
void
CollSm::handleBar( Tac::String const & name ) {
    if ( input()->barHas( name ) ) {
        // Either a value corresponding to the `name` key has been added, or an
        // existing value has been modified.
        // ...
    } else {
        // The value corresponding to `name` has just been removed from the
        collection.
        // ...
    }
}

```

Note that reactors with empty braces get invoked during the state-machine's creation. They may be used to handle keys already present. Without the `initially` keyword, the reactor gets

invoked when the notifier is updated after the constrainer has started. Therefore, such a reactor is a no-op if the notifier is immutable. The compiler signals an error for such cases.

Derived entities

An entity with the `extensible` keyword makes it possible to derive from it.

```
Shape : extensible Tac::Type( name ) : Tac::Entity {}
Circle : Tac::Type( name ) : Shape {
    radius : F32;
}
Square : Tac::Type( name ) : Shape {
    width : F32;
}
```

From C++:

```
void
example() {
    Shape::Ptr circlePtr = Circle::CircleIs( "circle" );
    Shape::Ptr squarePtr = Square::SquareIs( "square" );
}
```

Derived reactors

The `overridable` keyword makes a function `virtual` in C++.

.tac:

```
MyType : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    attr : U32;
}
BaseSm : extensible Tac::Type( input ) : Tac::Constrainer {
    input : in MyType::PtrConst;
    handleAttr : extern invasive overridable void();
    input::attr => handleAttr();
}
ChildSm : Tac::Type( input ) : BaseSm {
    // Override the base implementation.
    handleAttr : extern invasive overriding void();
}
```

.tin:

```
void
BaseSm::handleAttr() {
    std::cout << "baseSm\n";
}
void
ChildSm::handleAttr() {
    // Note: the parent reactor has to be invoked explicitly if we want the code
in
    // the base class to be executed.
    BaseSm::handleAttr();
    std::cout << "childSm\n";
}
```

From C++:

```
void
example() {
    MyType::Ptr myType = MyType::MyTypeIs();
    ChildSm::Ptr childSm = ChildSm::ChildSmIs( myType );
    // This will cause `ChildSm::handleAttr()` to be invoked.
    myType->attrIs( 42 );
}
```

Nested reactors

Nested reactors allow one to track changes in attributes of entities that are part of a collection. Essentially, the compiler will create a state machine per entry in the collection. Each individual SM can then react to the changes of the entity (collection element) that it is tied to.

.tac:

```
SmallConfig1 : Tac::Type( key ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    key : U32;
    attr : U32;
}
MyRecord1 : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    config : SmallConfig1[ key ];
}
MySm1 : Tac::Type( entity ) : Tac::Constrainer {
    entity : in MyRecord1::PtrConst;
    parentId : U32;
    // We want to react to the change of each `SmallConfig1` entity that is part
    // of
    // the `MyRecord1::config` collection.
    config : in SmallConfig1::PtrConst[ key ] {
        // This is the "source" collection for the constrainters.
        = entity::config;
        handle : extern void();
        config::attr => handle();
        handleDeletion : extern void();
        this => {
            `isAppliedonDelete;
            handleDeletion();
        }
    }
}
```

Note that `config : in SmallConfig1::PtrConst[key] { ... }` causes a nested C++ class `MySm1::NestedSm_Config` to be generated, along with a type alias named `MySm1::TacConfig` referring to it. Instances of this class are notified of changes in each `config` attribute. `handle` and `handleDeletion` become member functions of that nested class.

.tin:

```

void
MySm1::TacConfig::handle() const {
    // We can access our parent SM if we need to.
    std::cout << "MySm ID: " << mySm1()->parentId() << '\n';
    // The key that corresponds to this nested SM.
    std::cout << "key: " << fwkKey() << '\n';
    // Since the key is named `key`, `key()` is equivalent to `fwkKey()` .
    std::cout << "key: " << key() << '\n';
    std::cout << "new attr value: " << config()->attr() << '\n';
}
void
MySm1::TacConfig::handleDeletion() const {
    std::cout << "I'm being deleted\n";
}

```

Example:

```

void
basicExample() {
    MyRecord1::Ptr myRecord = Tac::allocate< MyRecord1 >();
    MySm1::Ptr mySm = Tac::allocate< MySm1 >( myRecord );
    mySm->parentIds( 5 );
    myRecord->configIs( 1 );
    myRecord->config( 1 )->attrIs( 20 );
}

```

Reactors with empty brackets

Such a reactor with **initially** gets called during the nested SM's creation, and it could be used to handle keys already present when the SM is starting. If you skip **initially**, the handler gets called only when the notifier is updated after the SM has started. For nested SMs, the notifier is the value corresponding to the keys in the "source" collection. So if you update the value for the same key, from one entity to another entity, you can handle the state inside the new entity by using such a reactor with empty brackets.

.tac:

```

SmallConfig2 : Tac::Type( key ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    key : U32;
    attr : U32;
    myColl : Tac::String[ U32 ];
}
MyRecord2 : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    // Externally-keyed collection.
    collAttr : SmallConfig2::PtrConst[ U32 ];
}
MySm2 : Tac::Type( entity ) : Tac::Constrainer {
    entity : in MyRecord2::PtrConst;
    parentId : U32;
    config : in SmallConfig2::PtrConst[ U32 ] {
        = entity::collAttr;
        handleMyCollInit : extern void();
        config::myColl[] => initially handleMyCollInit();
        handleMyColl : extern void();
        config::myColl[] => handleMyColl();
    }
}

```

.tin:

```

void
MySm2::TacConfig::handleMyCollInit() const {
    std::cout << "handleMyCollInit: handling initial values\n";
    for ( auto iter = config()->myCollIteratorConst(); iter; ++iter ) {
        std::cout << "key: " << iter.key() << " value: " << *iter << '\n';
    }
}
void
MySm2::TacConfig::handleMyColl() const {
    std::cout << "handleMyColl: handling initial values in new notifier\n";
    for ( auto iter = config()->myCollIteratorConst(); iter; ++iter ) {
        std::cout << "key: " << iter.key() << " value: " << *iter << '\n';
    }
}

```

Example:

```

void
emptyBracketsExample() {
    MyRecord2::Ptr myRecord = MyRecord2::MyRecord2Is();
    SmallConfig2::Ptr config = SmallConfig2::SmallConfig2Is( 1 );
    myRecord->collAttrIs( 1, config );
    config->myCollIs( 42, "42" );
    std::cout << "Starting SM\n";
    MySm2::Ptr mySm = MySm2::MySm2Is( myRecord );
    mySm->parentIdIs( 5 );
    SmallConfig2::Ptr newConfig = SmallConfig2::SmallConfig2Is( 1 );
    newConfig->myCollIs( 32, "32" );
    std::cout << "Updating notifier\n";
    myRecord->collAttrIs( 1, newConfig );
}

```

Reactors that only notify their parent

Nested reactors can directly invoke member functions defined in the parent SM. Ideally, all of the reactions in the nested reactors are such invocations. The implementation of the nested SM is simpler this way because all of the code lives in the parent SM, and tests don't have to refer to the nested SM.

.tac:

```
SmallConfig3 : Tac::Type( key ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    key : U32;
    attr : U32;
}
MyRecord3 : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    config : SmallConfig3[ key ];
}
MySm3 : Tac::Type( entity ) : Tac::Constrainer {
    entity : in MyRecord3::PtrConst;
    parentId : U32;
    // We want to react to the change of each `SmallConfig` entity that is part
    of the
    // `MyEntity::config` collection.
    handle : extern void( key : U32 );
    config : in SmallConfig3::PtrConst[ key ] {
        // This is the "source" collection for the constrainters.
        = entity::config;
        config::attr => handle( key );
    }
    handleConfig : extern void( key : U32 );
    config[ key ] => handleConfig( key );
}
```

.tin:

```
void
MySm3::handle( U32 key ) const {
    std::cout << "MySm ID: " << parentId() << '\n';
    // The key that the nested SM told us was changed.
    std::cout << "key: " << key << '\n';
    std::cout << "new attr value: " << config( key )->attr() << '\n';
}
void
MySm3::handleConfig( U32 key ) const {
    if ( configHas( key ) ) {
        std::cout << key << " has been created\n";
    } else {
        std::cout << key << " has been deleted\n";
    }
}
```

Example:

```
void
parentPokeExample() {
    MyRecord3::Ptr myRecord = Tac::allocate< MyRecord3 >();
    MySm3::Ptr mySm = Tac::allocate< MySm3 >( myRecord );
    mySm->parentIdIs( 5 );
    myRecord->configIs( 1 );
    myRecord->config( 1 )->attrIs( 20 );
    myRecord->configDel( 1 );
}
```

Introspection for nested reactors

Consider the following data model:

```
Person : Tac::Type( name, age ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    age : U32 {
        `=;
    }
}
Group : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    member : Person[ name ];
}
```

Similarly to the previous examples, **MySm4** defines a reactor for the age of each group member:

```
MySm4 : Tac::Type( group ) : Tac::Constrainer {
    group : in Group::PtrConst;
    person : in Person::PtrConst[ name ] {
        = group::member;
        handleAge : extern void();
        person::age => handleAge();
    }
}
```

The compiler generates a class **MySm4::NestedSm_Person**, and a type alias named **MySm4::TacPerson** referring to it:

.tin:

```
void
MySm4::TacPerson::handleAge() const {
    std::cout << person()->name() << " is now " << person()->age() << '\n';
}
```

An alternative way to define a reactor for each group member's age is with the addition of the **new** keyword:

```
MySm5 : Tac::Type( group ) : Tac::Constrainer {
    group : in Group::PtrConst;
    person : in Person::PtrConst[ name ] new MemberSm {
        = group::member;
        handleAge : extern void();
        person::age => handleAge();
    }
}
```

There are two consequences of the addition of `new MemberSm`:

- The name of the generated class is now `MySm5::MemberSm`. That is, it has been given a name directly instead of the compiler generating a name based on the attribute name
- `MySm5::MemberSm` is generated with support for *introspection*. This means, for example, that instances of it may be created from Python

This distinction is not necessarily desirable and it's somewhat surprising. [BUG 514368](#) is tracking the issue and the behavior of the compiler may change in the future.

Timer reactors

See also [Using a task for deferred reactors](#) and [AID 1088](#) (which describes `Ark::HTimerWheel`).

```
TimeReactor : Tac::Type( wheel ) : Tac::Constrainer {
    wheel : in Ark::TimerWheel::PtrConst;
    handleClock : extern invasive void();
    wheel::time => handleClock();
}
```

```
void
TimeReactor::handleClock() {
    doSomethingUseful();
    // Set the next timer. This function will run again 20 seconds from now.
    wheel_->timeMinIs( Tac::now() + 20 );
}
```

Your agent should just use one `Ark::TimerWheel` object. You should pass the pointer to it to all of the SMs that need to schedule timers. `Ark::TimerWheel` is capable of dealing with thousands of timers.

Cancelling timers

The following snippet will cancel a pending timer:

```
clock_->timeMinIs( Tac::endOfTime );
```

Using a task for deferred reactors

The preferred way of handling the processing of modifications to a collection is to use a CODEF. CODEFs are described in [AID 832](#). To implement a codef, the recommended way is to use tasks. [AID 3070](#) includes details and more complete examples.

Here's a quick example:

```
TaskSm : Tac::Type( scheduler, input ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    input : in MyType::PtrConst;
    task : in Ark::Task = initially (
        scheduler, Ark::TaskPriority::PriorityEnum::normal, "SomeTaskName" );
    handleTaskRun : extern invasive void();
    task::taskRun => handleTaskRun();
    // Schedule the task to run when an attribute of the input changes.
    input::attr => task::scheduleTask();
}
```

```
void
TaskSm::handleTaskRun() {
    // React to `input::attr`. Note that it might have changed multiple times
    // before
    // the task ran, as in the inline notification we only schedule the task to
    run
    // later.
    doSomethingUseful();
}
```

Library support for codefs and backlogs

[AID7122](#) describes helper functions from the Ark libraries that provide simple APIs to efficiently process backlogs, and to walk over collections asynchronously.

Pragma support for backlog processing

The compiler provides support, through the use of the `backlog::task` pragma, to make the use of CODEFs more convenient.

Suppose a user has a small collection with rapidly changing values, and wanted to dampen the thrashing by reacting using a task. The user might use the backlog pragma, and write the following code:

```

BacklogSm : Tac::Type( scheduler, input, output ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    input : in AttrType::PtrConst;
    output : AttrType::Ptr;
    processMyBacklogElem : extern invasive void( k : U32 );
    [[ backlog::task =
        [ scheduler, "TaskPriority::PriorityEnum::normal", "SomeTaskName" ] ]]
    myBacklog : set void[ U32 ];
    myBacklog[ k ] => processMyBacklogElem( k );
    input::attr[ k ] => {
        myBacklog[ k ] = true;
    }
    handleInitialized : extern invasive void();
    this => initially handleInitialized();
}

```

along with a corresponding .tin file:

```

void
BacklogSm::handleInitialized() {
    FMTRACE0( "{}: starting up", __PRETTY_FUNCTION__ );
    for ( auto it = input()->attrIteratorConst(); it; ++it ) {
        processMyBacklogElem( it.key() );
    }
    FMTRACE0( "{}: finished", __PRETTY_FUNCTION__ );
}

void
BacklogSm::processMyBacklogElem( U32 k ) {
    FMTRACE8( "key: {}", k );
    if ( auto val = input()->attr( k ) ) {
        output()->attrIs( k, val );
    } else {
        output()->attrDel( k );
    }
}

```

The snippets above would automatically generate a task, and a top-level-method for the task that would implement the `processBacklogTask()` pattern from AID7122. Insertions into the backlog would also schedule the task to run.

(Note that the initial collection scan, as written in `handleInitialized()`, is not recommended practice. Rather than synchronously calling `processBacklogElem()` for every element in the collection, [AID3070](#) and [AID7122](#) describe better techniques.)

The pragma expects an expression list containing the scheduler attribute name, the desired priority for the task and the name for the created task.

The name of the method called for each element in the backlog will be derived from the reactor. In this case it will be `processMyBacklogElem()`.

In the future, if the `backlog::task` pragma is applied to a reactor instead of a backlog collection, then the compiler will automatically generate a backlog, and also will generate an asynchronous walk of the input collection (the trigger of the reactor). The walk over input will occur before any processing of the backlog occurs (unless the (not yet implemented)

`interleaveStartupScan` option is set to true). While the input scan is occurring, any modifications to the input should get stored in the backlog, for future processing.

Properties of the backlog

This section defines the `backlog::task` pragma, its 3 positional arguments, and its properties. The pragma may be applied to an attribute defining a backlog collection, or to a reactor to some other input collection. Some properties are only valid when applied to a reactor.

`[[backlog::task=(...),]]`

Accepts a scheduler, task priority level, and a task name string. If not specified, picks the "default scheduler", normal priority, and a name of `<smName>` + ":" + `<attrName>`.

The "default scheduler" is defined as either the single attribute of type TaskScheduler in the same type, the root scheduler if there is no such attribute, or an error if there is more than one attribute of type `TaskScheduler` (or derived classes, e.g. `TaskGroup`). (Default scheduler is not yet implemented, so the first argument is still required.)

If no priority is found, the backlog will assume `Priority::Normal`.

The task name string must be a literal string, for the moment.

Explicit properties are allowed (e.g. `traceLevel=2`) even if fewer than 3 positional properties are provided (they are described here as optional ("not specified"), with explicit default values).

`backlog::taskDelay`

Optional

When specified, accepts a double

If present, we use a TimerTask instead of a Task, and all calls to `scheduleTask` become `scheduleTaskLowestDelay`, passing the specified double.

`backlog::traceLevel`

Optional

Defaults to 8, otherwise can take an int, or `None`.

Controls trace level used in the top-level method of task. If equal to `None`, then no tracing is generated.

`backlog::traceCommand`

Optional

When specified, identifier

Form of TRACE command to use: TRACE (default), QTRACE, BTRACE, BTFMT, etc.

`backlog::backlog`

Optional

when specified, identifier

Only valid when pragma is on the reactor, not when applied to the backlog collection itself. This defines the name used by the backlog collection, which the compiler will automatically define. The type of the backlog will be a set `void[KeyType]`, where `KeyType` is the type of the key in the input collection (the trigger of the reactor). By default the backlog name is *input attribute name* concatenated with “Backlog”. This also serves as a mechanism to allow multiple input collections to fill the same backlog, provided all input collections sharing the backlog have the same key type, and all reactors share the same action (which typically is a call to a method to process the element).

backlog::scanCompletionMethod

Optional

When specified, identifier or `delete`

Only valid when pragma is on the reactor, not when applied to the backlog collection itself. This defines a method, with no args, to be called when the input scan of the input collection (trigger of the reactor node) is completed, and before backlog processing begins. (When there are multiple input collections associated with the same backlog, the method is called only when they are all complete. If this property is specified on more than one input for the same backlog, then the names must match). The specified method must be defined in the SM. By default (or if ScanCompletionMethod=delete is explicitly specified) then no method is called, and the user can (optionally) walk over the output and delete any entries explicitly in handleInitialized().

backlog::interleaveStartupScan

Optional

boolean

Only valid when pragma is on the reactor, not when applied to the backlog collection itself. When true, the SM begins reacting to input changes immediately, interleaving with the input scan. When false, the SM does not start reacting to input changes (although it records them in the backlog) until the input scan is complete. Interleaving makes depending on `withOldValue` more complicated. For now, the default behavior will be to not interleave, but the proper default value is still open to debate.

An example use of backlog::task

For example, the tac model with backlog pragma presented above:

```

BacklogSm : Tac::Type( scheduler, input, output ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    input : in AttrType::PtrConst;
    output : AttrType::Ptr;
    processMyBacklogElem : extern invasive void( k : U32 );
    [[ backlog::task =
        [ scheduler, "TaskPriority::PriorityEnum::normal", "SomeTaskName" ] ]]
    myBacklog : set void[ U32 ];
    myBacklog[ k ] => processMyBacklogElem( k );
    input::attr[ k ] => {
        myBacklog[ k ] = true;
    }
    handleInitialized : extern invasive void();
    this => initially handleInitialized();
}

```

would be desugared to:

```

BacklogSm : Tac::Type( scheduler, input, output ) : Tac::Constrainer {
    // User provides
    scheduler : Ark::TaskScheduler::Ptr;
    input : in AttrType::PtrConst;
    output : AttrType::Ptr;
    backlogMyProcessElem : extern invasive void( key : U32 );

    // -----
    // Implementation provides, based on syntax discussed above
    myBacklogTask : in Ark::Task = initially (
        scheduler, Ark::TaskPriority::PriorityEnum::normal, "someTaskName" );
    handleMyBacklogTaskRun : extern invasive void();
    myBacklogTask::taskRun => handleMyBacklogTaskRun();
    myBacklogBookmark : U32::Optional {
        `hasNotifyOnUpdate = false;
    }

    // Provided by user, modified by implementation:
    myBacklog : set void[ U32 ] {
        `hasNotifyOnUpdate = false;
        `fastButUnsafeIterator;
        `hasDelRangeMutator;
        `hasExternalMutator;
    }
    // User code:
    input::attr[ k ] => {
        myBacklog[ k ] = true;
    }
    handleInitialized : extern invasive void();
    this => initially handleInitialized();
}

```

with the name ‘backlogMyProcessElem’ being discovered by looking at the reactor to myBacklog, and backlogMyProcessElem specified by the user in a .tin file, and the names ‘myBacklogTask’ and ‘myBacklogBookmark’ derived from the modified attribute name (‘myBacklog’).

Precisely one reactor to myBacklog is required — we need at least one to derive the name of the method to process each element. More than one would create ambiguity. (Note that the method to process the element is called from the top-level method in the task. The reactor is removed in the generated code --- it would not be allowed now that the backlog has `hasNotifyOnUpdate = false`.)

The backlog also causes the compiler to generate the following C++ functions in addition to the already generated code for the above TAC file:

```
void
BacklogSm::myBacklogIs( U32 _tac_index ) {
    TacMyBacklog * m = myBacklog_[ _tac_index ];
    if ( m ) {
        return;
    } else {
        ( void )myBacklog_.newMember( _tac_index );
    }
    if ( myBacklogTask() ) {
        myBacklogTask()->scheduleTask();
    }
}

void
BacklogSm::handleMyBacklogTaskRun() {
    auto progress = Ark::processBacklogTask( myBacklogTask(),
                                              myBacklog_,
                                              myBacklogBookmark_,
                                              this,
                                              &BacklogSm::backlogProcessElem );
    TRACE8( "BacklogSm::handleMyBacklogTaskRun" << progress );
}
```

In the above code, `myBacklog_`, `myBacklogBookmark_`, `myBacklogTask_` and `handleMyBacklogTask()` are generated by `arcg`. In the case that `backlog::delay` is set, then `scheduleTaskLowestDelay(<delay>)` is called instead of `scheduleTask()`.

Collections of base types

At times, it is necessary to maintain a collection of base types but then cast the pointers to the derived type. If such a collection is used for types stored in Sysdb, tacc will know how to synchronize the actual derived type even though the collection contains the pointers to the base types. The `hasTemplateAccessor` property can be used to avoid making explicit calls to `dynamic_cast`. The property will instruct the compiler to generate a template accessor which will include `dynamic_cast` internally. If the desired type doesn't match the type stored at a given key then the result is `nullptr`.

For example:

```
Base : extensible Tac::Type( name ) : Tac::Entity {
    attr1 : U32;
}
Derived : Tac::Type( name ) : Base {
    attr2 : U32;
}
MyRecord : Tac::Type() : Tac::PtrInterface {
    base : Base::Ptr[ name ] {
        // This will create a templated accessor that performs a `dynamic_cast`.
        `hasTemplateAccessor`;
    }
}
```

```
void
example() {
    Derived::Ptr derived = Derived::DerivedIs( "derived" );
    MyRecord::Ptr myRecord = MyRecord::MyRecordIs();
    myRecord->baseIs( derived );
    // This will return `nullptr` in case `derived` doesn't point to a derived
    type.
    Derived::Ptr derived2 = myRecord->base< Derived >( "derived" );
    tacAssert( derived == derived2 );
}
```

Shadow maps

Shadow maps are typically used to get the value that was deleted given the key. They are often manually managed in each SM which duplicates the data. Since they hold the last state of the map before the deletion occurred, that state can be placed closer to the original map and not duplicated. An example of this is demonstrated in the [section on bi-directional maps](#).

Bi-directional maps

The use and creation of bi-directional maps in TAC needs some consideration when immediate reactors are taken into account. Given a model and constrainer like

```
Foo1 : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    aToB : U32[ Tac::String ];
    bToA : Tac::String[ U32 ];
}
FooSm1 : Tac::Type( foo ) : Tac::Constrainer {
    foo : in Foo1::PtrConst;
    handleA : extern invasive void( a : Tac::String );
    foo::aToB[ key ] => handleA( key );
    handleB : extern invasive void( b : U32 );
    foo::bToA[ key ] => handleB( key );
}
```

the order in which the `aToB` and `bToA` collections are updated will trigger a reactor where the reverse mapping does not exist due to immediate reactors.

For example,

```
void
example() {
    Foo1::Ptr foo = Foo1::Foo1Is( "foo" );
    // `handleA()` will be called immediately before the reverse mapping is
updated.
    foo->aToBIs( "aaa", 10 );
    foo->bToAIs( 10, "aaa" );
}
```

One solution is the following pattern, where the client reacts to a different collection other than `aToB` and `bToA`.

```
Foo2 : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    // Manage the mapping with these functions.
    addAtoB : extern invasive void( a : Tac::String, b : U32 );
    delA : extern invasive void( a : Tac::String );
    // When clients react to `notifyingA` or `notifyingB`, the state of `aToB`
    and
    // `bToA` will be valid.
    notifyingA : set void[ Tac::String ];
    notifyingB : set void[ U32 ];
    // Prevent the client from reacting to `aToB` and `bToA`.
    aToB : U32[ Tac::String ] {
        `hasNotifyOnUpdate = false;
    }
    bToA : Tac::String[ U32 ] {
        `hasNotifyOnUpdate = false;
    }
    // Shadow maps can also be maintained by the type. If you use them then you
    know
    // what they are for.
    shadowAToB : U32[ Tac::String ] {
        `hasNotifyOnUpdate = false;
    }
    shadowBToA : Tac::String[ U32 ] {
        `hasNotifyOnUpdate = false;
    }
}
FooSm2 : Tac::Type( foo ) : Tac::Constrainer {
    foo : in Foo2::PtrConst;
    handleA : extern invasive void( a : Tac::String );
    // `aToB` and `bToA` are in-sync.
    foo::notifyingA[ key ] => handleA( key );
    handleB : extern invasive void( b : U32 );
    foo::notifyingB[ key ] => handleB( key );
}
```

```
void
Foo2::addAToB( Tac::String const & a, U32 b ) {
    auto sB = aToB( a );
    if ( sB == b ) {
        // Idempotent NOP.
        return;
    }
    // Translate an update into a deletion and add to simply client reactors.
    delA( a );
    aToBIs( a, b );
    bToAIs( b, a );
    notifyingAIs( a );
    notifyingBIs( b );
    shadowAToBIs( a, b );
    shadowBToAIs( b, a );
}
void
Foo2::delA( Tac::String const & a ) {
    if ( !aToBHas( a ) ) {
        return;
    }
    auto b = aToBDel( a );
    bToADel( b );
    notifyingADel( a );
    notifyingBDel( b );
    shadowAToBDel( a );
    shadowBToADel( b );
}
```

Creating and initializing state in Sysdb

Entities can be created and initialized in Sysdb in many ways. This section provides a brief overview.

Preinit profiles

The easiest way to create entities statically in Sysdb at bootup (before any agents start up) is to add the paths in a preinit profile. This option is perfect for entities that are always needed, such as entities entities that hold configuration for agents that start unconditionally. Such entities should never be deleted to prevent difficult-to-debug issues.

See [go/preinitprofiles](#) for full details on using preinit profiles to create entities.

To initialize the attributes of entities created via preinit profiles to non-default values, see [DefaultConfigPlugin](#) below.

Create-on-mount

For entities that are not needed until a conditional agent is started, the "create-on-mount" option is most suitable. With this option, the entity is created when the path is mounted. To create an entity while mounting, the '`'c'`' mount flag should be used. Note that with [passive mounts](#), the creation will be whenever the mount profile entry is being processed/executed; and with [active mounts](#), the creation will be whenever the mount code is executed. This delayed creation can cause problems with other agents that attempt to mount these entities before they are created.

These entities can be deleted, but with caution. Since other agents are expected to mount these entities as well, deletion of a mounted entity could lead to it becoming an orphan (this is undesirable). See [go/orphans](#) for more details on orphan entities.

Dynamic entity creation in a `Tac::Dir`

This is a suitable option when entity state is created using dynamic system state. For example, when a vrf is configured (see: [Creating child entities](#)). Ideally, such entities should not be specified as mountpaths or the right checks must be put in place to prevent a race condition between entity creation and mounting by another agent. Also, entities created this way can be deleted (by convention) since these entities depend on the dynamic state of the system. So again, utmost caution must be taken if such entities are being mounted and deleted.

`DefaultConfigPlugin`

A `DefaultConfigPlugin` offers an opportunity to configure the attributes of an entity after it has been created using preinit profiles. Such plugins are typically used to initialize default state within an entity (for example, adding a few members to a collection before agent start up). The plugins

reside in `/src/<pkg>/DefaultConfigPlugin/` and are loaded by Sysdb before all agents are started.

Note: entity state may also be initialized by specifying default attribute values for built-in types using `initially`. A `DefaultConfigPlugin` is only needed for initializing complex types and collection members.

For example,

```
TestEntity : Tac::Type( name ) : Tac::Entity {  
    `hasFactoryFunction;  
    `allowsDirInstantiation;  
    pollInterval : U32 = initially 10;  
}
```

Tac::Dir - mounting, adding, removing, iterating, and reacting

A `Tac::Dir` has a few attributes of interest that you may want to react to. Some attributes are real collections while some others are `extern` collections. The attributes/collections to react to in a `Tac::Dir` are:

- `entityPtr` (a collection representing pointers of the entities)
- `entryState` (an internal collection that actually stores the entity type information/entry). Note that only directories mounted without the '`i`' (immediate) flag need to ever look at this collection

Mounting a Tac::Dir

One can mount a `Tac::Dir` with or without using the '`i`' flag. '`i`' can be combined with '`w`' or '`r`'. '`w`' or '`r`', as the names suggest, are writable or readable.

If a `Tac::Dir` is mounted using the '`i`' flag, it brings in (synchronizes) all the entities created that directory (ie., the entire subtree). If a directory is mounted without using the '`i`' flag, it does not bring in any entities created under that directory, unless you ask for it.

Sysdb:

```
// Create directories to test flags.
auto wDir = sysdb->createEntity< Tac::Dir >( "wDir" );
auto wiDir = sysdb->createEntity< Tac::Dir >( "wiDir" );
```

Agent (using '`i`'):

```
mg->doMount( "wiDir", "Tac::Dir", "wi" );
// ...
Tac::Dir::Ptr wiDir = em->getEntity< Tac::Dir >( "wiDir" );
auto wiDirChild1 = wiDir->entityPtr< Tac::Dir >( "wiDirChild1" );
```

Agent (without using '`i`'):

```

mg->doMount( "wDir", "Tac::Dir", "w" );
// ...
Tac::Dir::Ptr wiDir = em->getEntity< Tac::Dir >( "wDir" );
// Request a specific entity in the directory.
wiDir->syncEntity( "wDirChild1" );
// React to `entityPtr` and wait for entity to be synchronized:
//
//   wDir::entityPtr[ name ] => handleEntityPtr( name );
//
// In `handleEntityPtr()`:
//
//   auto wDirChild1 = wDir->entityPtr< EntityType >( "wDirChild1" );
//

```

Creating child entities

Use the `createEntity()` function for creating child entities:

```
dir->createEntity< Sysdb::BasicEntity >( "child1" );
```

`createEntity()` wraps two steps in one. The first step creates an `entryState` member with the specified name and type-name. The second step (`entityRef`) actually instantiates the entity and synchronizes it with Sysdb. Please note: if the same entity is present in Sysdb but not present on the agent side (`Tac::Dir` mounted without the `'i'` flag), the entity does not get populated into the `entityPtr` collection until it is fully synchronized. In general, mounting a `Tac::Dir` without `'i'` is not recommended.

To create a `Tac::Dir` inside a `Tac::Dir`, use the same `createEntity()` function:

```
dir->createEntity< Tac::Dir >( "child1" );
```

createEntity() on a path mounted with the '`'i'` flag

```

// `wiDir` is a "wi"-mounted path.
auto wiDirChildLocal = wiDir->createEntity< EntityType >( "wiDirChildLocal" );
tacAssert( wiDirChildLocal );

```

The entity is available to use immediately (fully synchronized) after creation.

createEntity() on a path mounted without the '`'i'` flag

```

// `wiDir` is a "w"-mounted path.
auto wDirChildLocal = wDir->createEntity< EntityType >( "wDirChildLocal" );
tacAssert( wDirChildLocal == nullptr );

// React to `entityPtr` and wait for the entity to be synchronized.
auto wDirChildLocal = wDir->entityPtr< EntityType >( "wDirChildLocal" );

```

The entity is *not* available to use immediately after creation. In fact, `createEntity()` will return `nullptr` for an entity created in a non-immediate mounted directory.

An agent needs to react to `entityPtr` and wait for the entity to be present in the `entityPtr` collection to access it.

Deleting child entities

To delete a child entity, use the `deleteEntity()` function:

```
dir->deleteEntity( "child1" );
```

Removing the `"child1"` entity causes the `Tac::Dir` code to remove it from the `entityPtr` collection as well.

Iterating over child entities

To iterate over the child entities of a `Tac::Dir`, create an iterator for the `entityPtr` collection and iterate as you would for any other collection:

```
for ( auto iter = dir->entityPtrIteratorConst(); iter; ++iter ) { handleEntityPtr(
```

or, with a range-based for-loop:

```
for ( auto iter : dir->entityPtrIteratorConst() ) { handleEntityPtr( iter.key() ) }
```

Below are a few simple use cases of `Tac::Dir` reactors. For full details of reacting to `Tac::Dir`, see [AID 49](#).

Reacting to child entities being created

To react to child entity objects being created under a `Tac::Dir`, the reactors should ideally be based on the `entityPtr` collection.

```
BasicDirSm1 : Tac::Type( inputDir ) : Tac::Constrainer {
    `hasFactoryFunction;
    inputDir : in Tac::Dir::PtrConst;
    reactor : extern invasive void( name : Tac::String );
    inputDir::entityPtr[ name ] => reactor( name );
}
```

```

void
BasicDirSm1::reactor( Tac::String const & name ) {
    if ( inputDir()->entityPtr( name ) ) {
        // Do something if the entity was just created in the directory.
        //
        //
    } else {
        // The entity was just removed from the directory.
        // See the note below.
        //
        //
    }
}

```

Please note that `entityPtr` is constructed from the pointer stored in `entityState`. However, the key is added to the `entityPtr` collection only when the entity is created in a `Tac::Dir` (and is fully synchronized with Sysdb for a non-'i' mounted directory). A key is never added to the `entityPtr` collection with a null value. This implies that if `entityPtr` contains a particular key, the entity with that name exists.

Reacting to a child's name being added to the `entryState` collection

This section is only relevant to a `Tac::Dir` when it's mounted without the 'i' (immediate) flag, so the child can be mounted once the name appears. For `Tac::Dir` mounted as 'i' see the section [above](#).

Here is how to react to a child's name being added to the `entryState` collection. The object may not be instantiated. This may be used if the `Tac::Dir` is being used to store multiple names, of which only some entities may be instantiated.

```

BasicDirSm2 : Tac::Type( inputDir ) : Tac::Constrainer {
    `hasFactoryFunction;
    inputDir : in Tac::Dir::PtrConst;
    reactor : extern invasive void( name : Tac::String );
    inputDir::entryState[ name ] => reactor( name );
}

```

```

void
BasicDirSm2::reactor( Tac::String const & name ) {
    if ( inputDir()->entryState( name ) ) {
        // The name was just added to the collection in the directory.
        //
        //
    } else {
        // The name was just deleted.
        //
        //
    }
}

```

If `BasicDirSm2::reactor()` calls `syncEntity(name)` to synchronize the entity for which the `entryState` was created, a request to synchronize the entity is sent to Sysdb. Once the request is received, the `entityPtr` collection is populated with the entity.

External types

Sometimes it is necessary to use a C or C++ type in a TAC model. This typically arises in platform packages that use various SDKs.

For example, `SomeType.h` defines `SomeType`:

```
#include <cstdint>
struct SomeType {
    std::int32_t x;
    std::uint64_t y;
};
```

and `SomeType` can be used in TAC as follows:

```
<<= CppInlineInclude( "SomeType.h" );
// Tell TAC that this type is defined elsewhere.
SomeType : extern Tac::Type;
Wrapper : Tac::Type( name ) : Tac::PtrInterface {
    `hasFactoryFunction = true;
    name : Tac::String;
    attr : SomeType {
        // This mutator is declared as `protected` in C++ and the GenericIf
mutator
        // will not be generated.
        `= : local;
    }
    attrPtr : SomeType::RawPtrConst;
}
```

Reacting to the previous value in a collection or singleton

There are several ways to do this.

Using a `when` statement

With a `when` statement, the old value can be directly provided to the reactor by the `when` statement in most situations.

The binding is always done in the order `old, new`, using a single name will only bind the new value.

The type of the old value will be `Optional` wrapped, just like the new value is. More information can be found in the section on [When statements](#).

```
handleLinkStatus : extern void(  
    intfId : IntfId,  
    prev : LinkStatus::Status::Optional,  
    curr : LinkStatus::Status::Optional );  
when linkStatus::status[ intfId ] as ( prev, curr ) =>  
    handleLinkStatus( intfId, prev, curr );  
  
handleColor : extern void(  
    prev : Color::Optional,  
    curr : Color::Optional );  
when linkStatus::color as ( prev, curr ) =>  
    handleColor( prev, curr )
```

On initialization, the value of `prev` will be `nullopt`, while `curr` will contain the value present in the collection or singleton.

Limitations:

- `when` cannot provide an old value for attributes marked with `publishOld=false`
- `when` cannot provide an old value for Smash collections or attributes in Shark types

Using the `prev` mechanism for immediate reactors

This will only work for reactors of attributes defined within the same type!

```

Foo : Tac::Type() : Tac::PtrInterface {
    coll : U32[ U32 ];
    handleColl : extern void( key : U32, old : U32 );
    coll[ key ] => handleColl( key, coll`prev );
    foo : U32;
    handleFoo : extern void( f : U32 );
    foo => handleFoo( foo`prev );
}

```

For a collection, if the value was not previously present, the default value will be passed (0, empty string, `nullptr`, default constructed Nominal, etc).

Using the `captureNotifyValues` mechanism

In a `.tac` file:

```

MyType : Tac::Type() : Tac::PtrInterface {
    `isNotifyByDefault;
    record : S32[ U32 ] {
        `captureNotifyValues;
    }
}

MySm : Tac::Type() : Tac::Constrainer {
    attr : in MyType::Ptr;
    handleEntity : extern void( idx : U32 );
    attr::record[ idx ] => handleEntity( idx );
}

```

PrevValue.tac

We then obtain the previous and the current (new) value by calling the generated function as in the snippet below, in the `handleEntity` function, usually defined in a `.tin` file.

```
#include "PrevValue.h"

void
MySm::handleEntity( U32 key ) const {
    auto [ prev, curr ] = attr()->recordNotifyValues();

    if ( curr && prev ) {
        // we replaced a value in the collection
        printf( "Value %d replaced by %d at key %u\n", *prev, *curr, key );
    } else if ( !curr ) {
        // we deleted a value from the collection
        printf( "Key, value ( %u, %d ) deleted from collection\n", key, *prev );
    } else if ( !prev ) {
        // we added a value to the collection
        printf( "Key, value ( %u, %d ) added to collection\n", key, *curr );
    }
}

int main() {
    auto ptrType = Tac::allocate< MyType >();
    auto sm = Tac::allocate< MySm >();
    sm->attrIs( ptrType );

    ptrType->recordIs( 0, 12 );
    ptrType->recordIs( 1, 15 );
    ptrType->recordIs( 1, 5 );
    ptrType->recordDel( 1 );
}
```

In the example above, the `prev` and `curr` variables are raw pointers to, in this case, the type `U32`. Note that if `prev` is a `nullptr`, it means that a new key was added to the collection with the value that `curr` points to. If `curr` is a `nullptr`, it means that we deleted an entry from the collection. If both are non-`nullptr`, then it means that at `key`, we replaced the value pointed-to by `prev`, with the value pointed-to by `curr`.

Note that the above mechanism can also be applied to singleton attributes.

For more uses of this mechanism as well as limitations, please see [AID 9718](#).

Ensuring that some attribute is sent last

Normally, when Scheduled AttrLog is not in use, notifications are received in the order the attributes were modified across all entities. Once congestion occurs and Scheduled AttrLog is enabled, notification ordering across entities becomes relaxed. No particular order is defined or imposed for updates across entities and ordering will vary dynamically. For a single entity, notifications always occur in the order the attributes are declared if congestion occurs. Some notifications may be coalesced if the same attribute is updated multiple times in a short time period.

At times, it is desirable to have the reader not react to all intermediate state updates and instead wait for a "done/ready" signal to indicate that it is okay to process the received state. As noted above, Scheduled AttrLog complicates things because the updates can get reordered and the "ready" signal can be received before all of the required state was received. The `hasBarrierSemantics` property ensures that the designated attribute is sent last even in the presence of Scheduled AttrLog.

```
Foo : Tac::Type() : Tac::Entity {
    name = "";
    attr1 : U32;
    attr2 : U32;
}
Bar : Tac::Type( name ) : Tac::Entity {
    x : U32;
    done : U32 {
        `hasBarrierSemantics;
    }
}
```

Let's imagine that things are congested or the '`'S'`' mount flag is used. The user does `bar->xIs(...)`, then `foo->attr1Is(...)`, then `foo->attr2Is(...)`. Also, the user wants `done` to be received by the other side only after all the updates to `foo` go out. If the user does `bar->doneIs(...)` without `hasBarrierSemantics`, the FlowOut for `bar` will be processed before `foo` because of the queue insertion order. Effectively, the reader will see `x`, `done`, `attr1`, and `attr2` (in this order). If `done` has the `hasBarrierSemantics` property, the framework will move the FlowOut for `bar` to the back of the list. This will have the effect that the update to `done` will be seen as the last update. (The reader will see the order: `attr1`, `attr2`, `x`, `done`).

Note that this property can only be applied to the last attribute in an entity. This is enforced by the compiler. If multiple entities have attributes marked with `hasBarrierSemantics`, all unmarked attributes will be updated before any of the marked ones.

If the amount of state involved is small, it is often easier to use a nominal to contain all the state since nominal values are sent in their entirety - eg, by value - when they are updated, rather than using `hasBarrierSemantics` to constrain notification ordering across entities.

Performing an action during object construction/destruction

Sometimes it is useful to perform some action when an object gets created or destroyed. The pattern below demonstrates how to accomplish this:

```
MyType : Tac::Type() : Tac::PtrInterface {
    // ...
    handleInitialized : extern void();
    this => initially handleInitialized();
    handleDeleted : extern void();
    this => {
        `isAppliedInDestructor;
        handleDeleted();
    }
}
```

Order of initialization

A TAC model may have multiple attributes which have an initially clause. This raises the question as to what order in which those `initially` clauses are executed. They are executed *in the order in which the attributes are declared*, and not in the order that the `initially` clauses appear. Additionally, the `this` "attribute" is considered to be *first*.

Consider:

```
MyComplexType : Tac::Type() : Tac::PtrInterface {
    a : U32;
    handleInitialized : bool() {
        return true;
    }
    initialized : bool;
    initialized => initially handleInitialized();
    handleThis : void() {}
    otherFn : U32() {
        return 2;
    }
    a => initially otherFn();
    this => initially handleThis();
}
```

The generated constructor body will be:

```
MyComplexType::MyComplexType():
    a_(0),
    initialized_(false) {
    handleThis();
    otherFn();
    handleInitialized();
}
```

Here is why one observes that particular constructor:

1. `a` and `initialized` are constructed with their initial values in the order which they were declared
2. `handleThis()` is called as the `this` attribute is "first"
3. `otherFn()` is called since `a` is the first user-supplied attribute
4. `handleInitialized()` is called since `initialized` is declared after `a`

The recommended solution is:

- If there is only one `initially` clause, then using the `this =>` form is sufficient
- If there are more than one `initially` clauses, use the `initialized => initially handleInitialized();` form, and ensure that the `initialized` attribute is the *last* attribute

By pulling all of the initialization logic into a single function, or one `initially` block, one can authoritatively control in what order the initialization occurs. This order would also be maintained even if the order of the attributes are changed at some point in the future.

To force initialization to be some arbitrary method, followed by `b`, followed by `a`:

```
MyImperativeComplexType : Tac::Type() : Tac::PtrInterface {
    a : U32;
    b : U32;
    handleThis : void() {}
    otherFn : U32() {
        return 2;
    }
    otherBFn : U32() {
        return 3;
    }
    this => initially {
        handleThis();
        otherBFn();
        otherFn();
    }
}
```

Note on the destructor

It is important to be aware that the body of a destructor is only executed when the underlying C++ object is actually destroyed. Some implications of that behaviour:

- If a btest obtains a reference to an object inside an instantiating container that is then removed from the container, the destructor would not be invoked. The btest is still keeping the object alive due to that reference.
- If the agent is crashing, destructors aren't invoked
- If one has some cleanup code invoked from the destructor, and one wants to test an unclean restart:
 - Agents normally get restarted by being killed and restarted
 - Such restarts don't get a chance to execute destructors
 - Dropping the last reference to an object in a btest will allow destructors to execute

- The btest has to be carefully constructed to prevent that cleanup from happening

Iterating over an enumeration

This pattern lets you iterate over all members of a non-invasive enumeration without having to explicitly provide the range or test for a sentinel value.

For example:

```
VlanTagFormat : Tac::Enum {
    all;
    untagged;
    priority;
    tagged;
}
```

Enum.tac

```
#include "Enum.h"

int main() {
    for ( auto tagFormat : Tac::Range< VlanTagFormat >() ) {
        std::cout << tagFormat << '\n';
    }
}
```

Inverse

The "inverse property" pattern used to be supported but is now deprecated. It was replaced by a functionally equivalent pattern which is described below.

This is the *deprecated* pattern:

```
Value1 : Tac::Type( name ) : Tac::PtrInterface {
    name : Tac::String;
    // ...
}
// This doesn't work anymore!
//
//Owner : Tac::Type( name ) : Tac::Entity {
//    value : Value[ name ] {
//        `inverse = owner;
//    }
//}
```

This is the new pattern:

```
Owner : Tac::Type : Tac::PtrInterface;
Value2 : Tac::Type( name ) : Tac::PtrInterface {
    name : Tac::String;
    owner : Owner::RawPtr;
}
Owner : Tac::Type( name ) : Tac::PtrInterface {
    name : Tac::String;
    value : Value2[ name ];
    value::owner = this;
}
```

Explicitly declaring an `owner` attribute and constraining it behaves similarly as the old `inverse` property. However, it does *not* cause the `owner` attribute to be cleared when a child is removed from the collection or the parent is destroyed. This can cause the `owner` pointer to be misleading and could cause it to point to invalid memory.

Foolproof options are limited, and depend on whether the child is a `Tac::Entity` or a `Tac::PtrInterface` and whether it's an instantiating attribute or a regular pointer. In the case of an entity with an instantiating attribute (like in the example above), each child object has a built-in `parent` pointer which is maintained by the generated `valueIs()` and `valueDel()` member functions. This means that if `parent` is non-null, it's guaranteed that the parent object exists and it still owns (via shared ownership) the child value. If it's `nullptr`, it's guaranteed that either the parent object is destroyed or it no longer owns the child object.

For the rest of the cases (a `Tac::Entity` or `Tac::PtrInterface` pointer attribute), there's no way to achieve all of the guarantees above using any TAC constructs. It then boils down to carefully managing parent and child lifetimes in agent code.

Abstract functions

The idea behind an abstract function is that it is declared in a type but not implemented. All derived types are expected to provide an implementation of it. TAC abstract functions behave like C++ virtual functions, with the main difference being that TAC abstract functions can't have any implementation.

TAC abstract functions have the following constraints:

- They can only be used on functions that are overridable. In fact, using `abstract` automatically makes the function overridable
- They can't be used on inline functions, functions with an implementation, or functions declared as `extern`
- They can't be used in value types, types that allow direct instantiation, or types that have a factory function
- If a type has at least one abstract function then it becomes an abstract type and can't be instantiated
- Abstract functions can't be used inside types that aren't extensible, and non-extensible types must implement all abstract functions they inherit

Here's an example:

```
// `Base` has an abstract function so it becomes an abstract type and can't be
// instantiated. `abstract` automatically implies `overridable`, but you can
// also
// explicitly specify it (as seen below).
Base : extensible Tac::Type() : Tac::PtrInterface {
    func : abstract overridable U32();
}
// Implements the abstract function, so it can be instantiated.
ChildA : Tac::Type() : Base {
    func : overriding extern U32();
}
// Doesn't implement the abstract function and since it's not extensible this
// produces an error!
//ChildB : Tac::Type() : Base {
//}
```

Optional attributes

Optional attributes manage an *optional* contained value, i.e., a value that may or may not be present. This is useful when the default value of a type also has some meaning and it's necessary to distinguish whether the attribute was set by user code or not. An optional value can be returned from a function as well as used as a function argument.

```
MyEnum : Tac::Enum {
    one;
    two;
}
BitSet : Tac::DenseEnumBoolSet {
    five;
    six;
}
MyValue : Tac::Type( a, b ) : Tac::Ordinal {
    `hasDefaultHash;
    a : U32;
    b : U32::Optional;
    c : MyEnum::Optional;
    d : BitSet::Optional;
    e : Tac::String::Optional;
}
Foo : Tac::Type() : Tac::PtrInterface {
    valOpt : MyValue::Optional;
    val : MyValue = initially( 1, nullopt );
    collVal : MyValue::Optional[ U32 ];
    collU32 : U32::Optional[ U32 ];
    func1 : extern U32::Optional();
    func2 : extern invasive U32( arg : U32::Optional );
}
```

The new `nullopt` keyword was added to the TAC language in order to initialize optional values, or for function calls within `.tac` files.

The implementation of the `Tac::Optional` type wraps `std::optional` and is defined in `/src/tacc/Fwk/Cpp/Tac/Optional.h`. Most used methods:

```

template< class T >
class Optional {
    //...
    // return true if Optional contains value
    bool hasValue() const noexcept;

    // same as hasValue(), this operator is useful in logic expressions
    explicit operator bool() const noexcept { return hasValue(); }

    // allows access to value via member access operator
    // Optional< Tac::String > opt( Tac::String( "hello" ) );
    // opt->size();
    T * operator->() { return &value(); }
    T const * operator->() const { return &value(); }

    // fetch value via dereference operator
    // Optional< Tac::String > opt( Tac::String( "hello" ) );
    // (*opt).size()
    T & operator*() & { return value(); }
    T const & operator*() const & { return value(); }

    // get contained value, exception will be thrown if no contained value
    // Optional< Tac::String > opt( Tac::String( "hello" ) );
    // std::cout << opt.value();
    T & value() &;

    // get contained value when Optional has a value, otherwise get defaultValue
    // Optional< Tac::String > opt;
    // std::cout << opt.value_or( "world" ); /* "world" will be outputted */
    template< typename U >
    constexpr T value_or( U && defaultValue ) const &;

    // ...
}

```

Some examples:

```

Tac::Optional< U32 >
Foo::func1() const {
    return 42;
}
U32
Foo::func2( Tac::Optional< U32 > arg ) {
    // Alternatively: `valOpt().hasValue()`
    if ( valOpt() ) {
        // Access a sub-attribute.
        collU32Is( valOpt()->a(), arg );
        // Access the contained value. Throws if empty.
        valIs( *valOpt() );
    }
    return val().b().hasValue() ? *val().b() : val().a();
}

```

```
void
example() {
    MyValue myValue;
    myValue.eIs( Tac::Optional< Tac::String >() );
    // Alternatively:
    myValue.eIs( std::nullopt );
    // value_or example
    auto str = myValue.e().value_or( Tac::String( "str" ) );
    if ( str.startsWith( "str" ) ) {
        // do something
    }
}
```

From Python:

```
myValue = Tac.Value( 'OptionalAttributes::MyValue', a=1, b=None )
if myValue.b is None:
    # Set a value for an optional attribute.
    myValue.e = "abc"
# Remove a value for an optional attribute
myValue.e = None
```

AttrLog serialization and GenericIf (with Python) are supported. Collections with optional keys are not allowed: the compiler throws an error.

More details can be found in the [original proposal](#).

Dynamic optional attributes

For large value types, a `Tac::Optional` value can potentially be very expensive memory-wise. This is because it still allocates and reserves the memory required for the embedded type. So, if we have a type where a certain attribute is in fact used only some of the time, we will waste memory for all the cases where this attribute isn't used, even if we use a `Tac::Optional` for the attribute.

The solution to the above problem is to use a "dynamic" optional value: `Tac::DynOptional`. It can be used as follows:

```
Foo1 : Tac::Type() : Tac::Ordinal {
    a : U64;
    b : U64;
}
Bar : Tac::Type() : Tac::Ordinal {
    optAttr : Foo1::DynOptional;
}
```

Internally, `Tac::DynOptional` uses `std::unique_ptr` to stored its embedded value on the heap. This allows us to allocate memory only when a value is stored. So, when a value isn't stored in `Tac::DynOptional`, we only use 4 bytes for the pointer (on 32 bit systems, 8 bytes on 64 bit systems). Only when a value is stored, do we allocate `sizeof(Foo)` on the heap via `std::unique_ptr`. Of course, the effect of `DynOptional` on the enclosing type's size is always `sizeof(void*)`.

Note that `Tac::DynOptional` can only be used on user-defined value types, unlike `Tac::Optional`, which can be used on enumerations, bit-sets, strings, and so on. Otherwise, `Tac::DynOptional`'s interface is identical to that of `Tac::Optional`: to get the value stored, for example, we also use the dereference operator or the `value()` method. If we try to get a value, but none is stored, we also get a `BadOptionalAccessException`. And, to check if a value exists, we also use the `hasValue()` method.

Setting and unsetting a `Tac::DynOptional` attribute

The generated mutator for a `Tac::DynOptional` attribute is roughly of the following form:

```
void Bar::optAttrIs( Foo::DynOptional _optAttr ) {
    optAttr_ = std::move( _optAttr );
}
```

Since we don't have access to the private `optAttr_` member from outside the class, we have to reset the `Tac::DynOptional` via this mutator. The best way to do this, which also clearly shows the intent and is reasonably efficient, is by passing `std::nullopt` to the mutator:

```
optAttrIs( std::nullopt );
// An alternative way to do the same.
optAttrIs( {} );
```

See [go/dynoptional](#) for more details.

Applying `Tac::DynOptional` to save memory

For certain TAC types, we often don't use many of the attributes in instances of the type. A good example of this can be found in the Acl package. For example, in `/src/Acl/Config.tac`, we have a type called `IpFilterBase`, used as a base type for many other Acl types. This type contains many value type attributes that are optionally used as a configuration for the rule. As a result, in many instances, our Acl types use more memory than they should.

We can mitigate this problem by placing the larger optional value type attributes in a dynamic optional type. However, in many cases, a lot of code will depend on this attribute's current interface and placing it in a dynamic optional will change its interface. This will potentially break a lot of code that depends on it, and it will also cause `operator==` to behave unexpectedly in certain situations (shown later in this example). To avoid this problem, we can define the large attribute as an `extern` attribute, and add a dummy attribute that stores the actual dynamic optional type. For example, if we have the following:

```
Foo2 : Tac::Type : Tac::Nominal {
    attr1 : LargeValueType;
}
```

then we can change it to the following:

```
Foo3 : Tac::Type : Tac::Nominal {
    dynOptAttr : LargeValueType::DynOptional;
    attr1 : extern LargeValueType;
}
```

Then we can define custom accessors and mutators for `attr1` like so:

```

LargeValueType
Foo3::attr1() const {
    if ( dynOptAttr_ ) {
        return *dynOptAttr_;
    }
    // If no memory is allocated, the assumption (as far as the interface is
    // concerned) is that we are storing a default value.
    return {};
}
void
Foo3::attr1Is( LargeValueType const & _attr1 ) & {
    if ( _attr1 == LargeValueType{} ) {
        // We free the memory if we want to assign a default value. The
        assumption is
        // that when no memory is allocated, `attr1` is (as far as the interface
        is
        // concerned) a default value.
        dynOptAttr_.reset();
        return;
    }
    if ( !dynOptAttr_ ) {
        // Memory not allocated yet, so allocate it.
        dynOptAttr_ = LargeValueType::DynOptional::makeDynOptional();
    }
    *dynOptAttr_ = _attr1;
}

```

The reason we have to be so careful with clearing the dynamic optional when we have a default assigned is because `operator==` will behave strangely if we do the conversion above if we don't care about whether a default value or a cleared `Tac::DynOptional` is stored. For example, before making the change above, we expect the following behavior:

```

Foo2 foo;
Foo2 foo2;
foo2.attr1Is( someValueWhichIsNotADefaultProbably );

// Do stuff, but don't modify `foo2`.
// ...

foo2.attr1Is( LargeValueType{} );
bool shouldBeTrue = foo == foo2;

```

This makes sense, because `foo` gets default-constructed, which means it holds a default `LargeValueType` value. We then assign a default `LargeValueType` to the `attr1` attribute of `foo2` again, without modifying `foo`. Now `foo` and `foo2` should be equal. However, after converting `foo` to use a `Tac::DynOptional` for `attr1`, if we don't take care to clear the `Tac::DynOptional` when assigning a default value, `foo` and `foo2` won't be equal. This is because a `Tac::DynOptional` that has memory allocated is never equal to a `DynOptional` that has no memory allocated to it. The behavior we expect above, therefore, will break if we don't make sure we free the memory that the `Tac::DynOptional` holds consistently if it holds a default value. Besides correctness, we also save more memory, because "no memory" becomes code for "default value" in this case.

The above will only save memory for large values types, though, since the `Tac::DynOptional` itself stores a pointer. Therefore, using an extra 4 bytes to store a `U32`, for example, will be self-

defeating.

However, even if we have a lot of small attributes that are used only occasionally (like in the `IpFilterBase` type in the `Acl` package), we can group these attributes into a new value type, create a new `DynOptional` attribute that holds this new value type, and declare all of the attributes we store in this type as `extern` so that the interface is unaffected. We then use only a pointer size to avoid in some cases storing all these attributes, which in sum might take up a lot of memory. An example of this has already been applied to the `Acl` package. We show a part of this example below. Only part of the `IpFilterBase` type is shown here for brevity.

Original `IpFilterBase`:

```
IpFilterBase : extensible Tac::Type() : Tac::Ordinal {
    ... // other attributes
    innerSource : Arnet::IpAddrWithFullMask;
    innerDest : Arnet::IpAddrWithFullMask;
    innerSource6 : Arnet::Ip6AddrWithMask;
    innerDest6 : Arnet::Ip6AddrWithMask;
    ... // more attributes
}
```

A new value type `InnerFilter` groups the `inner*` attributes together. `IpFilterBase` after applying `Tac::DynOptional`:

```
InnerFilter : Tac::Type : Tac::Ordinal {
    `hasDefaultHash;

    innerSource : Arnet::IpAddrWithFullMask;
    innerDest : Arnet::IpAddrWithFullMask;
    innerSource6 : Arnet::Ip6AddrWithMask;
    innerDest6 : Arnet::Ip6AddrWithMask;
}

IpFilterBase : extensible Tac::Type() : Tac::Ordinal {
    ... // other attributes
    innerFilter : InnerFilter::DynOptional;

    innerSource : extern Arnet::IpAddrWithFullMask;
    innerDest : extern Arnet::IpAddrWithFullMask;
    innerSource6 : extern Arnet::Ip6AddrWithMask;
    innerDest6 : extern Arnet::Ip6AddrWithMask;
    ... // more attributes
}
```

Here are the custom accessors and mutators for `InnerSource` (the other attributes' custom accessors and mutators are similar). Note that these have slightly different logic from when our `DynOptional` directly stores the large value type. However, the idea is the same.

```
// custom mutator
void
IpFilterBase::innerSourceIs( Arnet::IpAddrWithFullMask const & _innerSource ) &
{
    if ( !innerFilter_ ) {
        if ( _innerSource == Arnet::IpAddrWithFullMask{} ) {
            // no memory allocated, and we want to assign a default value - so
            // don't allocate and return immediately
            return;
        }
        innerFilter_ = InnerFilter::DynOptional::makeDynOptional();
    }

    innerFilter_->innerSourceIs( _innerSource );
    if ( _innerSource == Arnet::IpAddrWithFullMask{} &&
        *innerFilter_ == InnerFilter{} ) {
        // if we already had memory allocated, and it turns out our InnerFilter
        // object in the DynOptional is now a default value, we clear out the DynOptional
        // memory
        innerFilter_.reset();
    }
}

// custom accessor
Arnet::IpAddrWithFullMask
IpFilterBase::innerSource() const {
    if ( innerFilter_ ) {
        return innerFilter_->innerSource();
    } else {
        // no memory allocated, so we pretend we are storing a default value
        return {};
    }
}
```

Queue-based reactors

TAC has queue-based collections. The queue collections have an index that describes the position of an element in the collection.

```
MyNominal : Tac::Type( attr ) : Tac::Nominal {
    attr : U32;
}
MyRecord : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    element : queue MyNominal[];
}
MySm : Tac::Type( entity ) : Tac::Constrainer {
    entity : in MyRecord::PtrConst;
    handle : extern void( index : U32 );
    entity::element[ index ] => handle( index );
}
```

```
void
MySm::handle( U32 index ) const {
    std::cout << "index: " << index;
    if ( entity()->elementHas( index ) ) {
        std::cout << "enqueue\n";
    } else {
        std::cout << "dequeue\n";
    }
}
```

Here's an example:

```
void
example() {
    MyRecord::Ptr myRecord = MyRecord::MyRecordIs();
    MySm::Ptr mySm = MySm::MySmIs( myRecord );
    // The index is 0.
    myRecord->elementEnq( MyNominal( 1 ) );
    // The index is 1.
    myRecord->elementEnq( MyNominal( 2 ) );
}
```

Embedding a name attribute

`name` is an implicit attribute of `Tac::Entity` but it is possible to redefine it by constraining it to the empty string and embedding a type with its own `name` of type `Tac::String`.

```
Foo : Tac::Type() : Tac::Nominal {
    name : Tac::String;
}
Bar : Tac::Type() : Tac::PtrInterface {
    foo : embedded Foo;
}
```

Enabling AttrLog

By default, AttrLog support is not enabled when a TAC file is compiled. The rationale is that not all TAC modules need AttrLog and it would be wasteful to generate serialization and deserialization code as it will increase the size of `EOS.swi`. To enable AttrLog, the following must be added to the top of the TAC file:

```
taccExtension << "attrLog";
```

This extension will enable AttrLog for all the models in the file. It is a bad practice to mix TAC models that need to be in Sysdb with TAC models and constrainters that are purely local to the agent. The correct thing to do is to create separate TAC files.

Individual attributes

Typically, you don't have to specify whether individual attributes are AttrLog-enabled or not. The compiler determines it for you based on the kind of attribute. At times, when you want to override the compiler's decision, you can use the `isLoggedAttr` attribute property.

One example of such a scenario is that you want a protected mutator for an AttrLog-enabled attribute. Using

```
`= : local;
```

makes the mutator protected, but also disables AttrLog. Add `isLoggedAttr` to enable AttrLog. You can read more about the `local` keyword on its corresponding [page](#).

Flushing AttrLog

The agent (`CAgent` and `PyAgent`) infrastructure provides a mechanism to ensure that AttrLog updates have been sent and acknowledged by the peer. This is an asynchronous mechanism.

To use this mechanism in C++:

1. Implement `CAgent::handleFlushEntityLogComplete()`
2. Call `CAgent::flushEntityLog()`
3. The `handleFlushEntityLogComplete()` callback provides a way to perform tasks (such as exiting an agent) after the AttrLog has been acknowledged

To use this mechanism in Python:

1. Implement a callback function
2. Call the `Agent.flushEntityLog()` method and pass in the callback function
3. Perform tasks in the callback (such as exiting an agent) which gets called after the AttrLog flushing has completed

Note that that mechanism is *not blocking* (ie., the agent won't wait for `handleFlushEntityLogComplete()` to happen then keep working), and *doesn't handle multiple requesters* (ie., `handleFlushEntityLogComplete()` has to be the same for the whole agent). Also, using this mechanism implies passing a pointer to the `CAgent` all the way down to the SM that needs to flush the entity log and this can easily lead to misuses of that pointer.

To avoid this, `Agent::FlushEntityLogSm` can be used:

1. Reuse `CAgent::flushEntityLogConfig()` or create a new `Agent::FlushEntityLogConfig`
2. Instantiate `Agent::FlushEntityLogSm` in the `CAgent` and pass in the `FlushEntityLogConfig`
3. Pass the `FlushEntityLogConfig` all the way down to the SM that needs to flush the entity log, instead of passing a pointer to the `CAgent`
4. Whenever it's necessary to flush the entity log, call `FlushEntityLogConfig::addRequest()`. This will return an `Agent::FlushEntityLogResponse`
5. `FlushEntityLogResponse::flushEntityLogComplete` will be set to `true`. The caller SM can react to this attribute and trigger its own `handleFlushEntityLogComplete()` callback

The functions below provide half-baked functionality and we cannot support them in product code. These are to be used for tests only:

- Do not use `Tac.flushEntityLog()` or `Tac.flushEntityLogAsync()` in Python
- Do not use `DemuxerManager::flushEntityLog()` or `DemuxerManager::flushEntityLogAsync()`

Another note: `Sysdb::TacUtils::flushAttrLogConnections()` can be used for flushing the buffers when an acknowledgement of the state being received is not required. It only ensures the tacc connection buffer (user-space) contents have been written into the socket buffer but *does not guarantee the state is received by the peer.*

File types

.tac

A TAC model description. From this, the compiler generates a `.h` and `.cpp` file.

.tin

The C++ implementation of `extern` functions in the `.tac` file.

In the generated `.cpp` file, there is a line

```
#include "Foo.tin"
```

.itin

Like `.tin`, but these files are included in the generated `.h` file:

```
#include "Foo.itin"
```

These files are used to define inline `extern` functions.

.h

These are C++ header files. This may be generated from a `.tac` file, or be part of native C++ implementation of a component.

.cpp

These are C++ source files. They may be generated from a `.tac` file, or be part of a native C++ implementation of a component.

File structure

Inside `.tac` files, one may reference the contents of other files in many different ways.

TacModule

```
<<= TacModule( "Foo" );
```

Here, `Foo` is a model name.

This is similar to the `#include` directive in the C++ preprocessor.

It's used to make definitions in another TAC model available in the definition of this model.

These declarations must appear at the top of a `.tac` file.

CppInlineInclude

```
<<= CppInlineInclude( "Foo.h" );
```

This causes `#include "Foo.h"` to be added to the generated C++ *header* file.

Note that `CppInlineInclude` differs from `TacModule` in that declarations in the included header file will not be available for defining the TAC model unless they're declared as `extern`.

These declarations must appear at the top of a `.tac` file.

CppInclude

```
<<= CppInclude( "Foo.h" );
```

This causes `#include "Foo.h"` to be added to the generated C++ *source* file.

These declarations must appear at the top of a `.tac` file.

CppBlock

```
<<= CppBlock( "Foo.tin" );
```

This causes `#include "Foo.tin"` to be added to the end of the generated C++ *source* file.

These declarations must appear at the bottom of a `.tac` file.

CppInlineBlock

```
<<= CppInlineBlock( "Foo.itin" );
```

This causes `#include "Foo.itin"` to be added to the end of the generated C++ header file.

These declarations must appear at the bottom of a `.tac` file.

Example

The following example illustrates these declarations.

`Example.tac`:

```
// Copyright (c) 2022 Arista Networks, Inc. All rights reserved.
// Arista Networks, Inc. Confidential and Proprietary.

<<= TacModule( "Other::Include" );
<<= CppInclude( "Other/Foo.h" );

FileStructure : Tac::Namespace {
    B : Tac::Type( a ) : Tac::PtrInterface {
        a : A::Ptr;

        banana : extern U32 {
            `= : immutable;
        }

        handleInitialized : extern void();
    }
}

<<= CppInlineBlock( "Example.itin" );
<<= CppBlock( "Example.tin" );
```

`Example.tin`:

```
// Copyright (c) 2022 Arista Networks, Inc. All rights reserved.
// Arista Networks, Inc. Confidential and Proprietary.

namespace FileStructure {

void
B::handleInitialized() const {
    // Nothing.
}

} // namespace FileStructure
```

`Example.itin`:

```
// Copyright (c) 2022 Arista Networks, Inc. All rights reserved.
// Arista Networks, Inc. Confidential and Proprietary.

namespace FileStructure {

inline U32
B::banana() const {
    return a()->attr1()->count();
}

} // namespace FileStructure
```

`Other/Include.tac`:

```
// Copyright (c) 2022 Arista Networks, Inc. All rights reserved.
// Arista Networks, Inc. Confidential and Proprietary.

FileStructure : Tac::Namespace {

Attr : Tac::Type() : Tac::PtrInterface {
    `hasFactoryFunction;

    count : U32;
}

A : Tac::Type() : Tac::PtrInterface {
    `hasFactoryFunction;

    attr1 : Attr::Ptr;
}

}
```

Generated file structure

The generated header file has this structure:

`Example.h`:

```
// ...
#include "Other/Include.h"
// ...
#include "Example.itin"
```

The generated source file has this structure:

`Example.cpp`:

```
// ...
#include "Other/Foo.h"
// ...
#include "Example.tin"
```

`TacModule` results in an `#include` at the top of the header file, while `CppInclude` ends up at the top of the source file. The `CppBlock` and `CppInlineBlock` declarations, on the other hand, always end up at the end of the source and header files, respectively.

For those familiar with development in C and C++, this has straightforward implications. Given the mantra that types should expose the smallest necessary and sufficient API, one should include only as much type information as is needed. For example, it is correct to `TacModule`-include all files which contain a type needed by the given `.tac` or `.tin` file. That is, a type defined in some `Foo.tac` may only be needed in a `.tin` file. It is not *wrong* to add `TacModule("Foo")` to the `.tac` file; however, this can lead to larger than necessary binaries after compilation, and so is discouraged.

In general:

- If a type is needed explicitly in the `.tac` file, it must be `TacModule`-included. This includes the case where the `.tac` file refers to attributes of types defined in other `.tac` files
- If a type is only needed in the `.tin` file, one should directly `#include` the needed file in the `.tin` file. If for some reason this is insufficient, consider a `CppInclude`
- `CppInlineBlock` should be used for `.itin` files. These are useful to allow the compiler to inline small methods
- `CppBlock` is the standard addition to any `.tac` file requiring a `.tin` file (any custom C++), as this will `#include` the `.tin` file at the end of the generated C++ source file

How to use a `std::vector` in TAC

In some cases using a container from the C++ standard library like `std::vector` instead of TAC collections is useful. In particular, `std::vector` has properties not well-matched by existing collection types.

In the case of standard containers like `std::vector`, please use the `AllocTrack` wrappers, like `Tac::AllocTrackVector`. This allows us to track the memory usage in tools like

```
show agent <NAME> memory allocations
```

Example

`MyVector.h`:

```
#pragma once
#include <Tac/AllocTrackVector.h>
#include "SomeType.h"
using MyVector = Tac::AllocTrackVector< SomeType >;
```

In this example, the contents of a `MyVector` attribute are converted to a "snapshot" that is accessible from Python for the purpose of writing tests.

Here's the TAC model and implementation:

```
<<= CppInlineInclude( "MyVector.h" );
SomeType : extern Tac::Type;
MyVector : extern Tac::Type;
Snapshot : Tac::Type() : Tac::PtrInterface {
    element : queue SomeType[];
}
Foo : Tac::Type() : Tac::PtrInterface {
    data : MyVector;
    snapshot : extern Snapshot::PtrConst();
}
```

```
Snapshot::PtrConst
Foo::snapshot() const {
    auto snapshot = Tac::allocate< Snapshot >();
    for ( SomeType const & e : data_ ) {
        snapshot->elementEnq( e );
    }
    return snapshot;
}
```

However, be aware that `Foo::data()` will return a *copy* of the `std::vector`, not a reference to it. Therefore, when operating on the attribute you will typically directly access the member variable `Foo::data_`. It's `protected` in the generated C++, so it's only accessible inside the type or from derived types.

The data member can be made public like so

```
Foo2 : Tac::Type() : Tac::PtrInterface {
    data : MyVector {
        `hasPublicDataMember;
    }
    // ...
}
```

or you can make the accessor return a const-reference via

```
Foo3 : Tac::Type() : Tac::PtrInterface {
    // TACO GetByRef [name=vector entity=Foo]
    data : MyVector;
    // ...
}
```

Another possibility is to return a non-const reference but at that point, a public data member might be the best option.

```
Foo4 : Tac::Type() : Tac::PtrInterface {
    // TACO GetByRef [name=vector entity=Foo const=False]
    data : MyVector;
    // ...
}
```

Boolean, integer and floating point types

Boolean, integer and floating point types are the basic numerical value types in TAC.

Boolean type

There is a single boolean type called `bool`. It can be either `true` or `false`. The default value of a boolean type is `false`. The underlying implementation is modeled after a U8 with 0 representing false and 1 representing true.

Integer types

There are several basic integer value types, reflecting intrinsic hardware sizes of 8, 16, 32 and 64 bits, and signed/unsigned variants of each. Signed types begin with the letter 'S', unsigned types with the letter 'U'. Thus, `U32` is a unsigned integer 32 bits in size, and `S8` is a signed 8 bit value. The default value of all integer types is 0.

To simplify calculations and error handling, derived types of unsigned integer types of length 1 to 63 bits are also available. For example, `U17` denotes a unsigned type 17 bits long. These types may be used by including `Tac::UintTypes.tac` in TAC files or `Tac/UIntTypes.h` in C++ files. These types will raise a `Tac::RangeException` if attempts are made to store values out of range for the type.

Floating point types

Two floating point value types are supported in TAC, `float` and `double`, implemented by the homonymous C++ types. Alternatively, the name F32 and F64 denote the same types.

TAC

This is an example of a `PtrInterface` with various boolean, integer and floating point attributes.

`BoolIntFloat.tac`

```
NumericalExample : Tac::Type : Tac::PtrInterface {
    `hasFactoryFunction;
    maybe : bool; // boolean type
    uchar : U8; // unsigned byte
    sshort : S16; // signed short (16 bits)
    sllong : S64; // signed 64 bit int
    fvalue : float; // 4 byte floating point
    dvalue : double; // 8 byte floating point
}
```

Boolean, integer and floating point types can be used as literals in TAC files.

```

<=< TacModule( "Tac::UIntTypes" );
NumericalLiterals : Tac::Type : Tac::PtrInterface {
    constTrue : bool : true;
    constFalse : bool : false;
    ud : U8 : 42; // 42
    uo : U8 : 052; // 42 in octal
    ux : U8 : 0x2a; // 42 in hex
    ub : U8 : 0b101010; // 42 in binary
    uX : U6 : 0b101010; // 42 in special 6 bit
    sd : S8 : -42; // -42
    so : S8 : -052; // -42 in octal
    sx : S8 : -0x2a; // -42 in hex
    sb : S8 : -0b101010; // -42 in binary
    constPi : double : 3.141592;
    mole : double : 6.02E23;
}

```

Supported arithmetic operators for booleans, integers and floats in imperative TAC code include:

| operator name | syntax | notes |
|---------------------------|--------|-----------|
| unary plus | +a | |
| unary minus | -a | |
| addition | a + b | |
| subtraction | a - b | |
| multiplication | a * b | |
| division | a / b | |
| assignment | a = b | |
| addition assignment | a += b | |
| subtraction assignment | a -= b | |
| multiplication assignment | a *= b | |
| division assignment | a /= b | |
| remainder | a % b | not float |
| bitwise NOT | ~a | not float |
| bitwise AND | a & b | not float |
| bitwise OR | a b | not float |
| bitwise XOR | a ^ b | not float |
| bitwise left shift | a << b | not float |
| bitwise right shift | a >> b | not float |

C++

The boolean, integer and floating point types in TAC map directly onto the corresponding C++ types.

The TAC compiler will generate functional accessors and mutators for attributes of Tac bool, integer and float types, with the accessors named after the attribute, and the mutator function having an appended `Is`.

Both the accessors and mutators have visibility `hidden` so can be accessed only from within the same shared object into which they're linked.

BoolIntFloat.tac

```
#include "Access.h"
void
access( NumericalExample * ptr ) {
    // arbitrary example of use of accessors and mutator
    if ( ptr->maybe() ) {
        double newValue = ptr->uchar() * ptr->sshort() * ptr->fvalue();
        ptr->dvalueIs( newValue );
    }
}
```

Python

Tac Python bindings allow the attributes of an entity to be accessed and modified directly. All integer types in Tac map to `class<int>`, all floats to `class<float>`, and booleans to `class<bool>`.

```
a = Tac.newInstance( 'BoolIntFloat::NumericalExample' )
assert bool == type( a.maybe )
assert int == type( a.sllong )
assert float == type( a.dvalue )
a.dvalue = a.fvalue * a uchar
```

Attempts to set attributes to values outside their ranges will result in Tac::Type exceptions.

String and Bytes

Values of type `Tac::String` and `Tac::Bytes` are used to store signed character strings and unsigned character data, respectively. `Tac::Bytes` are expected to contain binary data; there's no expectation of the data they contain being representable as a string.

Unsurprisingly, the default value of both `Tac::String` and `Tac::Bytes` are arrays of length 0.

TAC

`StringBytes.tac`

```
HostInfo : Tac::Type : Tac::PtrInterface {
    `hasFactoryFunction;
    hostName : Tac::String;
    macAddr : Tac::Bytes;
}
```

String and Bytes types can be used directly as literals in TAC files. C++ `escape` sequences prior to C++23 are supported. Non-unicode compatible string data should be stored in `Tac::Bytes` to avoid problems with conversions during Python accesses.

The `+` operator can be used in imperative code.

`StringBytes.tac`

```
StringExample : Tac::Type : Tac::Nominal {
    alwaysStr : Tac::String : "Str"; // always has value Str
    alsoAlwaysStr : Tac::String : alwaysStr; // also has same value
    escapedString : Tac::String : "\n\t\v"; // some random escape chars
    // Now add a method that adds a string to its input argument
    isAlwaysHungry : Tac::String( arg0 : Tac::String ) {
        return arg0 + "is Always Hungry";
    }
    // And add a method to add two string arguments
    addStrings : Tac::String( arg0 : Tac::String, arg1 : Tac::String ) {
        return arg0 + arg1;
    }
}
```

AttrLog

AttrLogged String and Bytes types are sent as contiguous messages, so while local (non-AttrLogged) lengths may range up to 2^{32} , an AttrLogged string or bytes is limited to 32K due to buffer limitations. As a result, AttrLogging very long strings may quickly require flow outs to be enabled.

C++

`Tac::String` and `Tac::Bytes` support almost all the same interfaces, and share the same implementation.

`Tac::String` supports the interfaces provided by `std::string`, except that only `const` (read-only) access to the underlying character array is permitted. This supports the underlying shared storage model. In addition, `Tac::String` also supports the interfaces declared in `Marco/StringCommon.h`.

An example of using `Tac::String` data types in a simple C++ program:

```
#include <Tac/String.h>
#include <iostream>

int
main() {
    Tac::String a = "\"My hovercraft is full of eels.\"";
    Tac::String b = " is a sentence from a Monty Python skit.";
    Tac::String c = a + b;
    if ( a.starts_with( "\"My" ) ) {
        std::cout << "std::string functions work!" << std::endl;
    }
    if ( a.startsWith( "\"My" ) ) {
        std::cout << "String functions from Tac/String.h work! " << std::endl;
    }
    assert( a.length() == a.bytes() ); // std::string and Tac::String funcs
    std::cout << c << std::endl;
}
```

Getting and setting values

As is the same with other attribute types, the TAC compiler will generate functional accessors and mutators for `Tac::String` and `Tac::Bytes` attributes, with the accessors named after the attribute, and the mutators having an appended `Is`.

Both the accessors and mutators have visibility `hidden`, so can be accessed only from within the same shared object into which they're linked.

Accessing the underlying data

It is sometimes necessary to directly access the data that comprises a `Tac::String`, for example to call an operating system routine that expects a `const char *`. Such access is always read-only, of course.

You can use either the `const char *charPtr()` or `const char *c_str()` methods to access the data in a `Tac::String`; `const unsigned char * bytesPtr()` is the routine for a `Tac::Bytes` object. The data for `Tac::String` types is always terminated with a null byte value so that the returned pointers may be used as a conventional 'C' string.

****Carefully note the following, however:****

The lifetime of the pointer returned from either method is only guaranteed to be valid so long as the String object exists.

Since the accessors for an attribute return a new `Tac::String` or `Tac::Bytes` object, once that object goes out of scope the pointer returned from the `charPtr()` method called on the temporary object can no longer be used safely. This can be the source of well-camouflaged bugs.

Accessing the storage of `Tac::String` attributes safely is made easier and more efficient with the additional accessor function `**attributeName**CharPtr()` generated by the TAC compiler

```
#include "Access.h"
size_t
accessExample( HostInfo * ptr ) {
    // An example of using charPtr() incorrectly, with correct alternatives

    // Here we use a temporary Tac::String which goes out of
    // scope as soon as we've completed that line of code;
    // s is no longer valid on the next line.
    // Yes, strlen() is a silly example here.

    const char * s = ptr->hostName().charPtr();
    size_t i = strlen( s ); // BUG: s is no longer valid.

    // Instead, we can anchor the Tac::String so it doesn't disappear:
    Tac::String const & name = ptr->hostName();
    s = name.charPtr();
    i = strlen( s );
    // We can also do the operation in a single line:
    i = strlen( ptr->hostName().charPtr() );
    // Or we can use the alternative accessor that directly
    // returns a charPtr() to the underlying attribute store:
    i = strlen( ptr->hostNameCharPtr() );
    return i;
}
```

C++ implementation

The `Tac::String` class is actually now `Marco::String`. As noted earlier, the `Tac::String` type uses a smart pointer to share storage between replicated strings.

With EOS moving to 64 bit and the number of short strings in EOS code, significant memory savings were possible by implementing a short strings optimization (SSO). We store strings shorter than 8 bytes in the 64 bit environment directly in the pointer itself. This is what generates the hazard with `charPtr()` calls on temporary `Tac::String` values noted above.

```
#include <Tac/String.h>
#include <iostream>

int
main() {
    Tac::String s = "short"; // fits in pointer
    Tac::String l = "longerString"; // doesn't fit in pointer

    // Note that the returned charPtr() for a short string is the
    // containing object itself.
    printf( "Size of Tac::String object is %zu bytes\n", sizeof( s ) );
    printf( "Address of short string 's' is 0x%p\n", ( void * )&s );
    printf( "Address of short string storage 's.charPtr()' is %p\n",
            ( void * )s.charPtr() );
    // for long strings, the string storage is allocated elsewhere.
    printf( "Address of long string 'l' is 0x%p\n", ( void * )&l );
    printf( "Address of long string storage 'l.charPtr()' is %p\n",
            ( void * )l.charPtr() );
    // copied long strings share the same storage
    Tac::String c = l;
    printf( "Address of long string storage 'c.charPtr()' is %p\n",
            ( void * )c.charPtr() );
}
```

In cases where strings are received as data to be stored and may be identical, directly calling the `dedup()` method on the string before storing it can result in significant memory savings. Strings short enough to benefit from the SSO are silently ignored.

```
#include <Tac/String.h>
#include <iostream>

int
main() {

    auto showSharing = []( Tac::String msg, bool share ) {
        std::cout << msg << " do " << ( share ? "" : "not " ) << "share storage."
            << std::endl;
    };

    // show sharing due to assignment
    Tac::String x = "Hungarian phrasebook";
    Tac::String y = x; // duplicated from the start;
    showSharing( "x and y", x.charPtr() == y.charPtr() );

    // dedup can either return a duplicated string given
    // an argument, or can insert a string into the dedup
    // table directly depending on how it's called.
    Tac::String a = "\\\"My hovercraft is full of eels.\\\"";
    Tac::String b = "\\\"My hovercraft is full of eels.\\\"";

    showSharing( "a and b first", a.charPtr() == b.charPtr() );

    a.dedup(); // enter 'a' into dedup table via String method
    b.dedup(); // enter 'b' into dedup table via String method

    showSharing( "a and b after dedup", a.charPtr() == b.charPtr() );

    // Example of using dedup via a static String method
    Tac::String c = Tac::String::dedup( "\\\"My hovercraft is full of eels.\\\"" );
    showSharing( "a and c and c",
                a.charPtr() == b.charPtr() && a.charPtr() == c.charPtr() );
}
}
```

Python

The Python support for `Tac::String` and `Tac::Bytes` objects maps naturally onto Python strings and Python bytes. Values read from attributes of type `Tac::String` and `Tac::Bytes` appear as object of Python `class<str>` and `class<bytes>` respectively. Likewise, attributes of types `Tac::String` and `Tac::Bytes` can be set using Python objects of `class<str>` and `class<bytes>`.

```
hostInfo = Tac.newInstance( 'StringBytes::HostInfo' )

assert isinstance( "string test", type( hostInfo.hostName ) )
assert isinstance( b"\x01", type( hostInfo.macAddr ) )

# attributes of type Tac::String can be set via assignment from Python str
# class.
hostInfo.hostName = "test"
# attributes of type Tac::Bytes can be set via assignment from Python bytes
# class.
hostInfo.macAddr = b"\x00\x1B\x63\x84\x45\xE6"
```

Sensitive

Values of type `Tac::Sensitive` are used to store sensitive information, like passwords and secret keys.

They are sequences of bytes, like `Tac::Bytes`, but with a number of restrictions which are intended to prevent sensitive information from being accidentally disclosed.

Since `Tac::Sensitive` is a distinct type from `Tac::Bytes` and `Tac::String` (with no implicit conversion), it's possible to:

- handle sensitive data uniformly in the system (instead of having special rules for particular attributes in models)
- restrict the allowed operations on sensitive data
- prevent sensitive data from being used in contexts where it shouldn't (for example, logging messages)
- easily identify sensitive data while humans are reading code

Importantly, `Tac::Sensitive` attributes are *invisible to the introspection system*.

Note that `Tac::Sensitive` is intended to help engineers treat sensitive data correctly in the code that they write, but that there is no protection from *active attackers*. The bytes are still present, unobfuscated, in memory.

TAC

This is an example of a type with a `Tac::Sensitive` attribute:

```
Credentials : Tac::Type( user ) : Tac::PtrInterface {
    user : Tac::String;
    password : Tac::Sensitive;
}
```

Credentials.tac

`Tac::Sensitive` values are *immutable* and there is no way to access their data in TAC.

`Tac::Sensitive` attributes are also invisible to introspection.

There are a number of restrictions for `Tac::Sensitive` values in TAC itself, which are described below.

It's still possible to access the contents of `Tac::Sensitive` from target languages like C++, but doing so intentionally involves some "friction".

Default value and initialization

The default value of `Tac::Sensitive` is the "empty" instance: a sequence of zero bytes.

There is no way to initialize a `Tac::Sensitive` value to anything other than the empty instance.

Comparison and hashing

TAC requires that built-in types are equality-comparable, but if the equality of `Tac::Sensitive` values was based on the data they stored then sensitive information would be disclosed.

Therefore,

`Tac::Sensitive` values are equal when they refer to the same memory location. Empty instances are always equal.

Allowing `Tac::Sensitive` values to be compared according to an ordering or included in a type's hash would similarly disclose sensitive information.

Thus,

`Tac::Sensitive` attributes are not allowed in ordinal types or types with a hash.

These restrictions mean that `Tac::Sensitive` values and types with `Tac::Sensitive` attributes cannot be used as collection keys.

Ignoring `Tac::Sensitive` values for comparison

The previous restrictions don't apply if all `Tac::Sensitive` attributes in a type are marked with `excludeFromComparison`.

For example, the `sessionId` and `user` fields can be used to compare instances of `UserSession` while the `password` attribute is ignored:

```
UserSession : Tac::Type( user, sessionId ) : Tac::Ordinal {
    `hasDefaultHash;
    user : Tac::String;
    sessionId : U32;
    password : Tac::Sensitive {
        `excludeFromComparison;
    }
}

Server : Tac::Type() : Tac::PtrInterface {
    sessionExpiry : Tac::Time[ UserSession ];
}
```

C++ API

Linking requirements: `libMarco.so`, `libTacMarco.so`.

`Tac::Sensitive` is an immutable reference-counted sequence of bytes.

On destruction, the memory allocated by `Tac::Sensitive` is filled with zero prior to being deallocated.

This is a small example. A detailed description of the API is below.

```
#include "Credentials.h"

int main() {
    auto credentials = Tac::allocate< Credentials >( "joe" );

    // The attribute is initialized to the empty instance.
    std::cout << credentials->password() << '\n';
    assert( credentials->password().size() == 0 );

    // Set a password based on a C string.
    credentials->passwordIs( Tac::Sensitive( "mypassword", 10 ) );

    // Retrieve the data manually.
    [[maybe_unused]] unsigned char const * passwordData =
        credentials->password().sensitiveData();
    assert( credentials->password().size() == 10 );
}
```

Constructors

In general, instances of `Tac::Sensitive` are constructed by copying a sequence of bytes.

```
// The empty instance.
Sensitive() noexcept;

// Copy a span of bytes.
Sensitive( unsigned char const *, U32 );
Sensitive( char const *, U32 );

// Copy and append two byte spans.
Sensitive( unsigned char const *, U32, unsigned char const *, U32 );
Sensitive( char const *, U32, char const *, U32 );

// Produce a copy. This is inexpensive.
Sensitive( Sensitive const & ) noexcept;

// Move.
Sensitive( Sensitive && ) noexcept;
```

Assignment

```
Sensitive & Sensitive::operator=( Sensitive const & );
Sensitive & Sensitive::operator=( Sensitive && ) noexcept;
```

Accessing sensitive data

`Tac::Sensitive` exposes a pointer to the sensitive data and the number of contained bytes.

```
unsigned char const * Sensitive::sensitiveData() const noexcept;
U32 Sensitive::size() const noexcept;
```

Python API

`Tac::Sensitive` values are not accessible from Python because they're invisible to the introspection system.

AttrLog

`Tac::Sensitive` values have the same AttrLog encoding as `Tac::String` and `Tac::Bytes`.

The number of bytes is first sent as a 32 bit unsigned integer, followed by the bytes themselves.

Time

Values of type `Tac::Time` represent points in time based on the epoch of the system monotonic clock. `Tac::Time` supports two approaches to marking points in time:

- The traditional Arista approach stores time points using floating-point numbers, with any fractional part of a value representing subseconds. The `Tac::Seconds` type, for example, which is used to represent time durations, is implemented using a `double`.
- Systems such as `YANG` use 64-bit integers to store nanoseconds.

`Tac::Time` takes the second approach, storing nanosecond values in a 64-bit integer type, but its interface also supports setting and getting its value as a double-precision floating-point number.

`Tac::Time` also supports arithmetic operations involving durations and other time points:

- A duration, which is a value that represents the difference between two points in time, can be added to or subtracted from a `Tac::Time` instance to get a new `Tac::Time` value.
- A `Tac::Time` instance can be subtracted from another to get the duration between them.

A `Tac::Time` instance can store at most `std::numeric_limits< U64 >::max()` nanoseconds. Attempting to store a value larger than that results in an exception, as does any attempt to create a negative `Tac::Time` instance. `Tac::Time` does not support the concept of infinity.

The `Tac::Time` type is conceptually similar to the `std::chrono::time_point` C++ type.

TAC

In TAC, `Tac::Time` is a value type. This is an example of a type with `Tac::Time` attributes:

TimeExample.tac

```
Tracker : Tac::Type( start ) : Tac::Constrainer {
    `hasFactoryFunction;
    start : Tac::Time;
    end : Tac::Time;
    duration : Tac::Seconds;
    duration = end - start;
}
```

Default value and initialization

The default value of a `Tac::Time` instance is zero nanoseconds, representing the starting point of the system monotonic clock.

To initialize a `Tac::Time` instance with some other time point value, use a numeric value representing the number of seconds since the `Tac::Time` epoch. The numeric value can have either an integral type or a floating-point type, must not be negative, and also must not exceed the maximum timespan value.

Comparison and hashing

Two `Tac::Time` instances are considered equal when they both store the same number of nanoseconds.

The `Tac::Time` type supports normal hashing and so is usable for attributes within other hashable types.

C++ API

Linking requirements: `libtac.so`, `libMarco.so`.

Under the covers, `Tac::Time` is implemented using an unsigned 64-bit integer storing nanoseconds, but its value can be set or retrieved as different C++ types.

Constructors

`Tac::Time` can be constructed from non-negative integral and floating-point numeric values, and from C++ `std::chrono::duration` and `std::chrono::time_point` values. For constructors, numeric values are treated as seconds, whereas C++ `duration` and `time_point` values can specify different time units ranging from years to nanoseconds. Alternatively, use the static `fromNanoseconds` function to create a `Time` instance from an integral nanoseconds value.

```

// Default instance is zero.
Time() noexcept;

// Create a Time instance representing the current value of the system
monotonic
// clock.
static Time now() noexcept;

// Create a Time instance from the given nanoseconds value.
template< typename T >
static Time fromNanoseconds( T nanoseconds );

// Create a Time instance from the specified timespan value in seconds. Type T
// must be a numeric type and its value must be non-negative and must not
// exceed
// the maximum timespan value Time can store.
template< typename T >
Time( T seconds );

// Create a Time instance from the specified std::chrono::duration value. The
// duration value must be non-negative and must not exceed the maximum timespan
// value Time can store. The template parameter type Rep indicates the
// representation type for the count of time ticks stored in the duration, and
// the Period type indicates the number of seconds between ticks. Please see
// the
// std::chrono::duration documentation for more details
// (https://en.cppreference.com/w/cpp/chrono/duration).
template< typename Rep, typename Period >
Time( std::chrono::duration< Rep, Period > const & duration );

// Create a Time instance from the specified std::chrono::time_point value. The
// time_point value must be non-negative and must not exceed the maximum
timespan
// value Time can store. The std::chrono::steady_clock template argument
indicates
// the system monotonic clock. Please see the std::chrono::time_point
documentation
// for more details (https://en.cppreference.com/w/cpp/chrono/time\_point).
Time( std::chrono::time_point< std::chrono::steady_clock > const & time_point
);

// Copy a Time value. This is inexpensive.
Time( Time const & ) noexcept;

// Move.
Time( Time && ) noexcept;

```

A `Tac::Time` instance is the same size as a `U64`, so copy construction, passing by value, and assignment are efficient.

Assignment

```

Time & Time::operator=( Time const & ) noexcept;
Time & Time::operator=( Time && ) noexcept;

```

Static functions

```
// Return the maximum number of nanoseconds a Time instance can store.
static U64 maxNanoseconds() noexcept;
```

```
// Return the maximum number of seconds a Time instance can store.
static U64 maxSeconds() noexcept;
```

```
// Create a Time instance representing the current value of the system
// monotonic
// clock.
static Time now() noexcept;
```

```
// Create a Time instance from the given nanoseconds value.
template< typename T >
static Time fromNanoseconds( T nanoseconds );
```

Retrieving a time value

Tac::Time supports a number of ways to retrieve the stored time value:

- The `nanoseconds()` function returns a `U64` nanoseconds value.
- The `seconds()` function returns a `double` seconds value.
- Both `asDuration()` and `timeSinceEpoch()` allow a caller to specify a `std::chrono::duration` type as a template argument to control the time units of the return value. By default, `timeSinceEpoch()` returns a whole number of seconds as a signed integer type.

```
// Return the Time value as seconds, including subseconds.
double seconds() const noexcept;
```

```
// Return the Time value as nanoseconds.
U64 nanoseconds() const noexcept;
```

```
// Return the Time value as a std::chrono::duration of type Duration. This
// allows
// a caller to retrieve the Time value in whatever time units Duration
// supports,
// such as milliseconds, hours, days, years, etc.
template< typename Duration >
Duration asDuration() const noexcept;
```

```
// Return the Time value as the Duration representation type. By default, the
// return type is the representation type for std::chrono::seconds, which is a
// signed integer type, and so the return value is a whole number of seconds.
The
// caller may specify a different Duration type if they need a time unit other
// than seconds.
template< typename Duration = std::chrono::seconds >
Duration::rep timeSinceEpoch() const noexcept;
```

Arithmetic operators

Tac::Time supports addition and subtraction with duration values of integral and floating-point types representing seconds, as well as `std::chrono::duration` types representing various

`Tac::Time` also supports the subtraction of one `Tac::Time` value from another to get a duration value as a `double` representing their difference in seconds.

For all arithmetic operators that return or mutate `Tac::Time` values, if the result would be negative or greater than `maxNanoseconds()`, a `Tac::IllegalValueException` is raised.

```
// Subtract a Time value from the target Time value and return their difference
// as a double duration. If the argument Time value is greater than the target
// Time value, the result duration will be negative.
double operator-( Time t ) const;

// Create a new Time by adding the timespan argument to the target Time value.
template< typename T >
Time operator+( T timespan ) const;

// Create a new Time by subtracting the timespan argument from the target Time
// value.
template< typename T >
Time operator-( T timespan ) const;

// Add the specified timespan to the target Time instance.
template< typename T >
Time & operator+=( T timespan );

// Subtract the specified timespan from the target Time instance.
template< typename T >
Time & operator-=( T timespan );
```

Comparison

`Tac::Time` is essentially a numeric value and so it supports the usual comparison operations:

- equality
- less than
- less than or equal to
- greater than
- greater than or equal to
- three-way comparison via `operator<=`, with a `std::strong_ordering` result

A `Tac::Time` instance can also be compared for equality with a time duration value, such as a value of type `Tac::Seconds` or a C++ `std::chrono::duration` value. For such a comparison, a `Tac::Time` instance is constructed from the duration value and is then used for the equality check.

Conversions

`Tac::Time` supports two conversions:

- An explicit conversion to `bool`, which returns `true` for a non-zero `Tac::Time` instance or `false` otherwise.
- A non-explicit conversion to `double`, which is equivalent to calling the `Tac::Time::seconds()` function.

```
explicit operator bool() const noexcept;
operator double() const noexcept;
```

Formatting

`Tac::Time` supports formatting via the `fmt` library and also supports `std::ostream` insertion. In both cases, the default is to format the result of calling the `Tac::Time::seconds()` function, as this follows typical Arista practice for formatting floating-point time values.

Numeric limits

The `std::numeric_limits::max()` function is specialized for `Tac::Time` to return the maximum number of nanoseconds an instance can store. It equals the maximum value of an unsigned 64-bit integer.

C++ example

The example below uses the `Tracker` type, which has three attributes:

- `start`: a `Tac::Time` starting time.
- `end`: a `Tac::Time` ending time.
- `duration`: a `Tac::Seconds` value.

The `duration` attribute is constrained to `end - start`, which is the difference between the ending time and starting time.

```
#include <fmt/printf.h>
#include "TimeExample.h"

int main() {
    auto tracker = Tracker::TrackerIs( Tac::Time::now() );
    fmt::print( "tracker started at {}\n", tracker->start() );
    sleep( 2 );
    tracker->endIs( Tac::Time::now() );
    fmt::print( "tracker ended at {}\n", tracker->end() );
    fmt::print( "tracker duration: {} seconds\n", tracker->duration() );
}
```

First the code creates a `Tracker`, initializing its `start` attribute to `Tac::Time::now()`. It then sleeps, and then sets the `end` attribute to `Tac::Time::now()`. The constraint on the `duration` attribute means that when we set `end`, `duration` gets set to the difference between `end` and `start`. Printing `duration` shows that it's roughly equal to the sleep time.

While the example uses the C++ `fmt` library for formatting and output, `Tac::Time` supports formatted output with standard `iostream` types as well.

Python API

Import requirements: `Tac`.

In Python, `Tac.Time` is a class type that supports the following operations:

- initialization
- setting and accessing time values
- comparisons
- arithmetic
- conversions

Initialization

The default `Tac.Time` initializer creates an instance holding a value of zero nanoseconds. To initialize a `Tac.Time` instance with some other timespan value, pass it as a constructor argument:

```
import Tac

zero = Tac.Time()
t1 = Tac.Time( zero )
t2 = Tac.Time( 1234567 )
```

A timespan value passed to the `Tac.Time` constructor can be an integral value or a floating-point value. For both cases, passing a negative value or a value that exceeds the `Tac.Time.maxNanoseconds()` value results in a `ValueError` exception. Attempting to create a `Tac.Time` instance from a value of a type other than an integral or floating-point type results in a `TypeError` exception.

To get a `Tac.Time` instance initialized to the current value of the system monotonic clock, call the `Tac.Time.now()` function:

```
now = Tac.Time.now()
```

Construction of `Tac.Time` instances from the maximum value functions is guaranteed to work:

```
max1 = Tac.Time.fromNanoseconds( Tac.Time.maxNanoseconds() )
assert max1.nanoseconds() == Tac.Time.maxNanoseconds()
max2 = Tac.Time( Tac.Time.maxSeconds() )
assert max2.nanoseconds() == Tac.Time.maxNanoseconds()
```

Assignment

`Tac.Time` instances can be assigned from numeric duration values or from other `Tac.Time` instances:

```
# Assign the value of now() to t2.
t2 = Tac.Time.now()
# Copy t2 to t3
t3 = Tac.Time( t2 )
```

Comparison and hashing

Instances of `Tac.Time` can be compared for equality and ordering:

```
if t1 == zero:
    print( 't1 is 0!' )
if t1 <= t2:
    t1 = t2
```

`Tac.Time` supports the usual numeric binary comparison operations:

- equality
- less than
- less than or equal to
- greater than
- greater than or equal to

`Tac.Time` also supports hashing, and so can be used with types that require hashing support.

Arithmetic operations

`Tac.Time` supports addition and subtraction with duration values, and also supports the subtraction of one `Tac.Time` value from another to get a duration value representing their difference.

```
t4 = Tac.Time( 12345678 )
t5 = Tac.Time( 12345689 )
assert t5 - t4 == 11.0
assert t4 - t5 == -11.0
origT5 = Tac.Time( t5 )
t5 += 123
assert t5 - 123 == origT5
assert t5 - t4 == 134.0
t5 -= 123
assert t5 == origT5
assert t5 - t4 == 11.0
# The subtraction below, where the duration value
# subtracted from t5 is greater than t5, raises
# a ValueError exception.
try:
    d = t5 - 99999999
except ValueError as e:
    assert e.args[ 0 ] == 'Tac::IllegalValueException("Time cannot be negative")'
```

For all arithmetic operations that return or mutate `Tac.Time` values, if the result would be negative or greater than `maxNanoseconds()`, a `ValueError` exception is raised to indicate a `Tac::IllegalValueException` failure.

Conversions

`Tac.Time` supports the following conversions:

- `bool`
 - a `Tac.Time` instance holding a non-zero nanoseconds value converts to `True`
 - a `Tac.Time` instance holding a zero nanoseconds value converts to `False`
- A `Tac.Time` instance can be converted to a `float` value of seconds.

Python example

The example below uses the `Tracker` type.

```
import time
import Tac

tracker = Tac.newInstance( 'TimeExample::Tracker', Tac.Time.now() )
print( f'tracker start time is {tracker.start.nanoseconds()}' )
time.sleep( 2 )
tracker.end = Tac.Time.now()
print( f'tracker end time is {tracker.end.nanoseconds()}' )
print( f'tracker duration is {tracker.duration} seconds' )
```

Enumerations

An enumeration (or "enum") defines a set of related integral constants.

TAC

Enumerations are defined in TAC with the `Tac::Enum` keyword:

```
Fruit : Tac::Enum {
    apple;
    banana;
    cherry;
}
```

Fruit.tac

Here, `apple`, `banana`, and `cherry` are assigned integral values beginning from zero.

Each of these named constants is called an "enumerator".

Invasive enumerations

It is possible to assign specific integral values to each enumerator:

```
Vehicle : invasive Tac::Enum : U32 {
    bicycle : 100;
    car : 2;
    train : 2000;
}
```

For example, `train` is a value of type `U32` which is equal to `2000`.

With an invasive enumeration, nothing prevents two enumerators from being assigned the same value.

C++ API

For the example above, the following `enum` is generated:

```
enum Fruit : U8 {
    apple_ = 0,
    banana_ = 1,
    cherry_ = 2
};
```

These names shouldn't normally be used directly in C++ code. The following accessors are generated and should be used instead:

```
constexpr Fruit apple();
constexpr Fruit banana();
constexpr Fruit cherry();
```

In addition, the following C++ functions are generated:

```
// Convert an integer to an enumerator.
Fruit FruitInstance( U8 );

// Parse an enumerator name into an enumerator.
Fruit FruitInstance( Tac::String const & );

// Get a string representation of an enumerator's name.
Tac::String stringValue( Fruit );

// Write an enumerator to an output stream.
std::ostream & operator<<( std::ostream &, Fruit );
```

Conversions

`FruitInstance()` create an enumerator from either its integral value or the string representation of its name.

`stringValue()` is the string representation of an enumerator's name.

The conversion functions throw `Tac::RangeException` exceptions when they're given invalid input.

For example:

```
#include "Fruit.h"

int main() {
    // Accessors.
    assert( apple() == 0 );
    assert( banana() == 1 );
    assert( cherry() == 2 );

    // Enumeration values from integers.
    assert( FruitInstance( 0 ) == apple() );
    assert( FruitInstance( 1 ) == banana() );
    assert( FruitInstance( 2 ) == cherry() );

    // An exception is thrown for an invalid integer.
    try {
        FruitInstance( 5 );
        assert( false );
    } catch ( Tac::RangeException const & e ) {
        std::cerr << e << '\n';
    }

    // Convert a value to a string.
    assert( stringValue( apple() ) == "apple" );

    // Parse a string into a value.
    assert( FruitInstance( "banana" ) == banana() );

    // An exception is thrown for an invalid string.
    try {
        FruitInstance( "spinach" );
        assert( false );
    } catch ( Tac::RangeException const & e ) {
        std::cerr << e << '\n';
    }
}
```

Enumeration ranges

All non-invasive enumerations support `Tac::Range`. This can be a convenient way to iterate over enumerators.

Python API

The following examples demonstrate how to interact with a TAC enumeration in Python:

```

from enum import IntEnum
from TacMarco import tacPyEnum
import Tac

Fruit = Tac.Type( 'Enumerations::Fruit' )

# Specific enumerators.
assert Fruit.apple == 'apple'
assert Fruit.banana == 'banana'
assert Fruit.cherry == 'cherry'

# Get an enumerator's integral value from its name.
assert Tac.enumValue( 'Enumerations::Fruit', 'apple' ) == 0
assert Tac.enumValue( 'Enumerations::Fruit', 'banana' ) == 1
assert Tac.enumValue( 'Enumerations::Fruit', 'cherry' ) == 2

# Get an enumerator from its integral value.
assert Tac.enumName( 'Enumerations::Fruit', 0 ) == Fruit.apple
assert Tac.enumName( 'Enumerations::Fruit', 1 ) == Fruit.banana
assert Tac.enumName( 'Enumerations::Fruit', 2 ) == Fruit.cherry

```

To get a Python standard library `Enum` from TAC enumeration, use `TacMarco.tacPyEnum`:

```

# Get enum.Enum equivalent for underlying Tac::Enum
pyFruit = tacPyEnum( 'Enumerations::Fruit' )
# Use of Enum
# .value for underlying integer
# .name for enumerator name
assert pyFruit.apple.value == 0
assert pyFruit.banana.value == 1
assert pyFruit.cherry.value == 2
assert pyFruit.apple.name == "apple"
assert pyFruit.banana.name == "banana"
assert pyFruit.cherry.name == "cherry"

```

It is also possible to specify subclasses of `enum.Enum`:

```

# Use second arg to get any subclass of enum.Enum
pyVehicle = tacPyEnum( 'Enumerations::Vehicle', IntEnum )
# Use of IntEnum
# enum can be used like integer implicitly
# .name for enumerator name
assert pyVehicle.bicycle == 100
assert pyVehicle.car == 2
assert pyVehicle.train == 2000
assert pyVehicle.bicycle.name == "bicycle"
assert pyVehicle.car.name == "car"
assert pyVehicle.train.name == "train"

```

Iterating over enumerations

A tuple containing strings for all the attribute names in an enumeration can be obtained like this:

```
Tac.Type( <enum> ).attributes
```

This can be iterated over, like any standard Python tuple:

```
Fruit = Tac.Type( 'Enumerations::Fruit' )
Vehicle = Tac.Type( 'Enumerations::Vehicle' )
assert Fruit.attributes == ( 'apple', 'banana', 'cherry' )
assert Vehicle.attributes == ( 'bicycle', 'car', 'train' )
fruitValues = [ Tac.enumValue( Fruit, attr ) for attr in Fruit.attributes ]
assert fruitValues == [ 0, 1, 2 ]
vehicleValues = [ Tac.enumValue( Vehicle, attr ) for attr in Vehicle.attributes ]
assert vehicleValues == [ 100, 2, 2000 ]
```

Bit sets

A bit set is a collection of named bits with an efficient implementation based on integers.

TAC

Bit sets are defined with `Tac::DenseEnumBoolSet`.

For example,

```
FileProperties : Tac::DenseEnumBoolSet {
    readable;
    writeable;
    executable;
}
```

Values of type `FileProperties` store three independent bits identified by name.

There is no limit to the number of bits in a set.

Memory efficiency

Bit sets use integers to store the underlying bits and this is more memory-efficient than defining models with `bool` attributes.

The size of the integer (and the number of required integers) depends on the number of bits. For example, when there are eight or fewer bits, the implementation is a single `u8` value.

Designing types with bit sets to save memory

Consider a model `WastefulConfig` with multiple `bool` attributes, and a better model that instead uses bit sets:

```

WastefulConfig : Tac::Type() : Tac::Nominal {
    name : Tac::String;
    legacy : bool;
    accelerated : bool;
    max : U64;
    observed : bool;
}

// A better model:
//

Properties : Tac::DenseEnumBoolSet {
    legacy;
    accelerated;
    observed;
}

Config : Tac::Type() : Tac::Nominal {
    name : Tac::String;
    max : U64;
    properties : Properties;
}

```

In C++, each `bool` attribute typically requires at least a byte and alignment requirements can quickly balloon the size of the type.

```

#include "Memory.h"

int main() {
    std::cout << "wasteful size: " << sizeof( WastefulConfig ) << '\n';
    std::cout << "size: " << sizeof( Config ) << '\n';
}

```

Reading, writing, and manipulating

Individual bits may be read and written by referring to them by name.

Also, `FileProperties::readable_` is a `FileProperties` value with just the `readable` bit set.

A bit set may also be manipulated using "bitwise" arithmetic operators like `&` and `^`.

The `-` operator performs a "set-difference" operation.

This example demonstrates these features and how they're useful:

```
File : Tac::Type() : Tac::PtrInterface {
    defaultProperties : FileProperties :
        FileProperties::readable_ | FileProperties::writeable_;
    properties : FileProperties = initially defaultProperties;

    restrict : invasive void() {
        properties -= FileProperties::writeable_ | FileProperties::executable_;
    }

    // To demonstrate referring to a particular bit.
    readable : bool : properties::readable;

    // To demonstrate writing a particular bit.
    setWriteable : invasive void() {
        properties::writeable = true;
    }
}
```

C++ API

This is a simplified summary of the C++ class generated for the `FileProperties` example:

```

class FileProperties final {
    static constexpr FileProperties readable_() noexcept;
    static constexpr FileProperties writeable_() noexcept;
    static constexpr FileProperties executable_() noexcept;

    // Constructors.
    constexpr FileProperties() noexcept;
    explicit FileProperties( U8 ) noexcept;

    // Bit accessors.
    constexpr bool readable() const noexcept;
    constexpr bool writable() const noexcept;
    constexpr bool executable() const noexcept;

    // Bit mutators.
    constexpr void readableIs( bool ) & noexcept;
    constexpr void writeableIs( bool ) & noexcept;
    constexpr void executableIs( bool ) & noexcept;

    // Comparison and equality.
    constexpr bool operator==( FileProperties const & ) const noexcept;
    constexpr bool operator<( FileProperties const & ) const noexcept;
    constexpr bool operator<=( FileProperties const & ) const noexcept;
    constexpr bool operator>( FileProperties const & ) const noexcept;
    constexpr bool operator>=( FileProperties const & ) const noexcept;

    // Arithmetic.
    constexpr FileProperties operator&( FileProperties const & ) const noexcept;
    constexpr FileProperties operator|( FileProperties const & ) const noexcept;
    constexpr FileProperties operator^&( FileProperties const & ) const noexcept;
    constexpr FileProperties operator-( FileProperties const & ) const noexcept;

    // Mutating arithmetic.
    constexpr FileProperties & operator&=( FileProperties const & ) & noexcept;
    constexpr FileProperties & operator|=( FileProperties const & ) & noexcept;
    constexpr FileProperties & operator^=( FileProperties const & ) & noexcept;
    constexpr FileProperties & operator-=( FileProperties const & ) & noexcept;

    // Integral and Boolean conversion.
    constexpr U8 value() const noexcept;
    constexpr operator bool() const noexcept;
};

```

A default-constructed bit set has all zero bits.

The comparison, equality, and arithmetic functions behave as though they were operating directly on the underlying integers in the bit set.

A small example

This example demonstrates some of the above functions, which should mostly be self-explanatory.

```
#include "FileProperties.h"

int main() {
    FileProperties properties;

    // Initially, all the bits are zero.
    assert( !properties.readable() );
    assert( !properties.writeable() );
    assert( !properties.executable() );

    // Properties may be composed by combining bits.
    properties = FileProperties::readable_() | FileProperties::writeable_();
    assert( properties.readable() && properties.writeable() );

    // Individual bits may be written.
    properties.executableIs( true );
    assert( properties.executable() );
}
```

More than 64 bits

When there are more than 64 bits in a bit set, the underlying implementation uses a sequence of integers instead of just one.

In that case, the functions that treat a bit set like a *single* integer are omitted.

For example, given

```
LotsOfBits : Tac::DenseEnumBitSet {
    b1;
    b2;
    // ...
    b100;
}
```

then `LotsOfBits::value()` will not be generated and neither will the non-default constructor.

Python API

The Python support for bit sets is very similar to the support in TAC and C++, as the following example demonstrates:

```
import Tac

FileProperties = Tac.Type( 'BitSet::FileProperties' )

# Bit sets may be created by specifying some of the bits.
properties = FileProperties( readable=True, executable=True )
assert properties.readable
assert not properties.writeable
assert properties.executable

# Individual bits may be written and read.
properties.executable = True
assert properties.executable

# There are factory functions for each bit.
assert FileProperties.readable_ == FileProperties( readable=True )

# Arithmetic is supported.
properties -= FileProperties.readable_
assert not properties.readable
```

Union

A union type holds one of a fixed set of alternative values.

A similar type in C++ is `std::variant`.

TAC

First, some examples.

A person may either be referred to by their full name or by their unique ID number:

```
Person : Union {
    name : Tac::String;
    id : U32;
}
```

Person.tac

A shape is either a circle, a rectangle, or an ellipse:

```
Rectangle : Tac::Type( width, height ) : Tac::Ordinal {
    width : F32;
    height : F32;
}

Ellipse : Tac::Type( major, minor ) : Tac::Ordinal {
    major : F32;
    minor : F32;
}

Shape : Union {
    // A circle has a radius.
    circle : F32;
    rectangle : Rectangle;
    ellipse : Ellipse;
}
```

The different cases of a union are called its "alternatives".

A union value only requires as much memory as the size of its largest alternative and the size of the internal "tag" which identifies which of the alternatives is populated.

Some or all of the alternatives of a union may be of the same type. However, the alternative *names* must be unique.

Supported types

All value types are allowed as union alternatives, irrespective of whether they are built-in or user-defined.

Non-instantiating pointers are also supported. For example,

```

Address : Tac::Type() : Tac::PtrInterface {
    country : Tac::String;
    municipality : Tac::String;
    street : Tac::String;
}

Destination : Union {
    box : PostOfficeBox;
    address : Address::Ptr;
}

```

Collections can't be union alternatives.

Hashing

Union types support hashing when they are annotated with the `hasDefaultHash` property.

These types may be used as keys in unordered collections.

```

Letter : Union {
    `hasDefaultHash;
    a : U32;
    b : F32;
    c : Tac::String;
}

LetterNames : Tac::Type() : Tac::PtrInterface {
    name : Tac::String[ Letter ];
}

```

Default values

The first declared alternative is populated when a union is default-constructed.

It's possible to change the default alternative:

```

Pet : Union {
    cat : Cat;
    dog : Dog = default;
    parrot : Parrot;
}

```

Default-constructed instances of `Pet` will have a default-constructed `dog` alternative populated.

Imperative context

New instances of union types may be created in the imperative context, but accessing and mutating the alternatives is not supported.

For example, given the definition of `Person` above:

```
Team : Tac::Type() : Tac::PtrInterface {
    leader : Person = initially Person::name( "Joe Smith" );

    favoriteMember : Person() {
        return Person::id( 10 );
    }
}
```

Other restrictions

Unions are supported in Smash collections as long as the union alternatives are statically-allocated POD types.

If a YANG pragma is present on a union declaration then only gNMI scalar types are allowed as alternatives.

This union with YANG pragmas is valid:

```
[[ yang::description="type description" ]]
Record : Union {
    [[ yang::description="description for a" ]]
    a : U32;
    [[ yang::description="description for b" ]]
    b : Tac::String;
}
```

C++ API

Consider again the example of the `Person` union:

```
Person : Union {
    name : Tac::String;
    id : U32;
}
```

This is a simplified summary of the generated C++ `Person` class. More details and examples are provided below.

```

class Person {
    struct Tag {
        enum Enum { name, id };

        using Name = std::integral_constant<Enum, name>;
        using Id = std::integral_constant<Enum, id>;
    };

    // Constructors.
    Person();
    Person( Tag::Name, ... );
    Person( Tag::Id, ... );

    // Factory functions by alternative name.
    static Person Name();
    static Person Name( Tac::String const & );
    static Person Id();
    static Person Id( U32 );

    // Factory function by type.
    template< typename T >
    static Person make( ... );

    // Accessors for `name`.
    Tac::String const & name() const;
    Tac::Optional< Tac::String > nameOpt() const;

    // Accessors for `id`.
    U32 id() const;
    Tac::Optional< U32 > idOpt() const;

    // Mutators.
    void nameIs( Tac::String const & );
    void IdIs( U32 );

    // Query for the populated alternative.
    constexpr bool containsName() const noexcept;
    constexpr bool containsId() const noexcept;
    constexpr Tag::Enum tag() const noexcept;

    // Equality comparison.
    bool operator==( Person const & ) const;
    bool operator!=( Person const & ) const;

    // Ordering.
    bool operator<( Person const & ) const;
    bool operator<=( Person const & ) const;
    bool operator>( Person const & ) const;
    bool operator>=( Person const & ) const;

    // Default-initialization check.
    explicit operator bool() const noexcept;

    // Visitation.
    template< typename Visitor >
    decltype( auto ) applyVisitor( Visitor && ) const;
};

std::ostream & operator<<( std::ostream &, Person const & );

Tac::String valueToStrep( Person const & );

```

Constructors

The default constructor, `Person()`, default-initializes the default alternative. For `Person`, that is `name`.

```
#include "Person.h"

int main() {
    Person p;
    assert( p.name() == "" );
}
```

The constructors with a first parameter like `Person::Tag::Name` forward their arguments to the constructor for the alternative corresponding to the tag.

```
#include "Person.h"

int main() {
    // Default construct a Tac::String.
    Person p1( Person::Tag::Name{} );
    assert( p1.name() == "" );

    // Provide a name.
    Person p2( Person::Tag::Name{}, "Joe" );
    assert( p2.name() == "Joe" );

    // Default construct a U32.
    Person p3( Person::Tag::Id{} );
    assert( p3.id() == 0 );

    // Provide an ID.
    Person p4( Person::Tag::Id{}, 42 );
    assert( p4.id() == 42 );
}
```

Factory functions

Factory functions are defined for convenience.

The first form is named after the alternatives of the union:

```
#include "Person.h"

int main() {
    Person p1 = Person::Name();
    Person p2 = Person::Name( "Joe" );
    Person p3 = Person::Id();
    Person p4 = Person::Id( 42 );
}
```

It's also possible to construct a union based on the type of its alternative instead of the alternative's name.

```
#include "Person.h"

int main() {
    Person p1 = Person::make< U32 >( 42 );
    assert( p1.id() == 42 );

    Person p2 = Person::make< Tac::String >( "Joe" );
    assert( p2.name() == "Joe" );
}
```

If there are multiple alternatives with the designated type then the first-declared one is selected.

Setting.tac

```
Setting : Union {
    a : F32;
    b : U32;
    c : U32;
}
```

```
#include "Setting.h"

int main() {
    Setting s1 = Setting::make< U32 >( 10 );
    assert( s1.b() == 10 );
}
```

Accessors

There are two kinds of accessors.

The first kind assumes that the caller knows which alternative is populated. For example, `U32 Person::id() const` assumes that the `id` alternative of `Person` is currently populated. If this kind of accessor is invoked for an alternative that is not populated then a `Tac::RangeException` is thrown.

```
#include "Person.h"

int main() {
    Person p = Person::Name( "Joe" );
    assert( p.name() == "Joe" );

    try {
        p.id();
        assert( false );
    } catch ( Tac::RangeException const & e ) {
        std::cerr << e.what() << '\n';
    }
}
```

The second kind of accessor returns a value of type `Tac::Optional`. If the alternative is not populated then the result is `std::nullopt`.

```
#include "Person.h"

int main() {
    Person p = Person::Name( "Joe" );
    Tac::Optional< Tac::String > maybeName = p.nameOpt();
    assert( *maybeName == "Joe" );

    assert( p.idOpt() == std::nullopt );
}
```

Mutators

A union value's mutators switch the currently-populated alternative.

```
#include "Person.h"

int main() {
    Person p = Person::Name( "Joe" );
    assert( p.name() == "Joe" );

    p.idIs( 42 );
    assert( p.id() == 42 );

    p.nameIs( "Sally" );
    assert( p.name() == "Sally" );
}
```

Querying

It is possible to query if a particular alternative is populated via, for example, `Person::containsName()`:

```
#include "Person.h"

int main() {
    Person p = Person::Id( 42 );
    assert( p.containsId() );
    assert( !p.containsName() );

    p.nameIs( "Joe" );
    assert( !p.containsId() );
    assert( p.containsName() );
}
```

Also, the union value's tag indicates the index of the currently-populated alternative:

```
#include "Person.h"

int main() {
    assert( Person::Name( "Joe" ).tag() == Person::Tag::name );
    assert( Person::Id( 42 ).tag() == Person::Tag::id );
}
```

Ordering and equality

Two union values are equal if they both have the same alternative populated and the underlying values of the alternatives are equal.

```
#include "Person.h"

int main() {
    assert( Person::Name( "Joe" ) == Person::Name( "Joe" ) );
    assert( Person::Name( "Joe" ) != Person::Name( "Sally" ) );
    assert( Person::Name() != Person::Id() );
    assert( Person::Id( 42 ) == Person::Id( 42 ) );
    assert( Person::Id( 42 ) != Person::Id( 10 ) );
}
```

The ordering of two union values is defined first by the declaration order of the alternatives and then by the ordering relation of the alternative's value.

```
#include "Person.h"

int main() {
    assert( Person::Name( "Joe" ) < Person::Id( 42 ) );
    assert( Person::Name( "Joe" ) < Person::Name( "Zachary" ) );
    assert( Person::Id( 42 ) < Person::Id( 100 ) );
}
```

Boolean conversion

A union value converted to a Boolean value is `false` when its default alternative is populated with its default-initialized value.

```
#include "Person.h"

int main() {
    Person p;
    assert( !p );

    p.nameIs( "" );
    assert( !p );

    p.nameIs( "Joe" );
    assert( p );

    // Not the default alternative, even though it's default-initialized.
    p.idIs( 0 );
    assert( p );
}
```

String representation

The string representation of union values indicates the particular alternative that is populated.

```
#include "Person.h"

int main() {
    assert( valueToStrep( Person::Name( "Joe" ) ) ==
           "Value('BookUnion::Person', name='Joe')");
    assert( valueToStrep( Person::Id( 42 ) ) == "Value('BookUnion::Person',
id=42)");
}
```

Visitation

A visitor may be used to fold over a union value based on the particular alternative that is populated.

```
#include "Person.h"

int main() {
    // Compute a numeric value for a person. The formula differs depending on
    // whether
    // they are referred to by name or by their ID.
    struct Visitor {
        U32 operator()( U32 id ) const { return 10 * id; }

        U32 operator()( Tac::String const & name ) const { return name.size() +
5; }
    } visitor;

    assert( Person::Name( "Joe" ).applyVisitor( visitor ) == 8 );
    assert( Person::Id( 42 ).applyVisitor( visitor ) == 420 );
}
```

Python API

The support for unions in Python is more limited than in C++.

Creating values

Union values are created with `Tac.Value()` by naming the alternative to be populated. If the alternative is not specified then the default alternative is default-initialized.

```
p1 = Tac.Value( 'BookUnion::Person' )
p2 = Tac.Value( 'BookUnion::Person', name='Joe' )
p3 = Tac.Value( 'BookUnion::Person', id=42 )
```

Accessors

Accessing the populated alternative is straightforward.

If the alternative is not populated then `NotImplementedError` is raised.

```
p = Tac.Value( 'BookUnion::Person', name='Joe' )
assert p.name == 'Joe'

try:
    print( p.id )
    assert False
except NotImplementedError as e:
    print( e )
```

Mutators

The value of the populated alternative may be changed by simple assignment.

Assigning to a different alternative will switch to it.

```
p = Tac.Value( 'BookUnion::Person', name='Joe' )
p.name = 'Sally'
assert p.name == 'Sally'

p.id = 42
assert p.id == 42
```

Querying

Checking which alternative is populated in a union value is similar to how it's done in C++.

```
p = Tac.Value( 'BookUnion::Person', name='Joe' )
assert p.containsName()
assert not p.containsId()
```

Split-inheritance types

A split-inheritance type wraps another type in order to enrich it with additional functions, properties, or restrictions.

Split-inheritance types have no memory overhead compared to the built-in value that they wrap.

One way that split-inheritance types are useful is to distinguish particular kinds of values in the type-system in order to avoid programmer errors. For example, an attribute of type `Temperature` is more meaningful than one of type `U16`.

What types can split-inheritance types wrap?

- Basic types (bool, integers, floats, `Tac::String`, `Tac::Bytes`, `Sensitive`)
- `Tac::Enum`

TAC

The following TAC type is a split-inheritance type that wraps type `U16`.

```
Temperature : Tac::Type( value ) : Tac::Ordinal, U16 {
    `hasDefaultHash;
    operator U16;
    value {
        `=;
        `range = 0 .. 100;
        = initially 0;
    }
}
```

`value` attribute

The value of the underlying (wrapped) type is available as the `value` attribute, and that's the only stateful attribute allowed. `value` attribute is implicitly declared, so explicit declaration is considered an error.

Restrict value ranges

One common use case of split-inheritance type is to restrict value ranges. Here, `min` and `max` attributes are used in the properties of the `value` to specify its possible value ranges.

A different way to define the same restriction can be:

```
value <= max;
```

Mutability

`value` by default is immutable, meaning `valueIs` mutator will not be generated. However, a mutator can be requested using attribute property:

```
value {`=; }
```

Constructor parameter

`value` attribute does not have to be constructor argument. It can be non-arg, but an initial value must be provided. In the example, we use `= initially min;` to specify the initial value to be `min`.

Additional attributes

Users can add additional attributes to the types. For example, here, `min` and `max` attributes are added and initialized to `0` and `100` respectively, indicating minimum temperature value and maximum temperature value. `value` range restrictions can also use `min` and `max` attributes instead.

```
Temperature : Tac::Type( value ) : Tac::Ordinal, U16 {
    `hasDefaultHash;
    operator U16;
    min : U16 : 0;
    max : U16 : 100;
    value {
        `=;
        `range = min .. max;
        = initially min;
    }
}
```

Using split-inheritance type as array index

One more use case of split-inheritance types is to define a key type for a static array as shown in the array chapter.

See [Array](#) for details.

C++ API

Constructor

```
Temperature temp(); // default construct
Temperature temp(100); // construct with value argument
```

value accessor and mutator

```
Temperature temp(100);
// access value
U16 tempValue = temp.value();
// static_cast is OK because `operator U16`
U16 anotherTempValue = static_cast<U16>( temp );
// change underlying value, because `value` is marked to have `= operator
temp.valueIs(200);
```

Type conversion

A wrapped type can be implicitly converted to more restricted wrapper type. For example, following code is valid:

```
int predict( const Temperature & t );
predict( 123 );
```

A wrapper type cannot be converted to underlying type, by default.

```
// Rank -> U16 conversion can be applied implicitly
Rank : Tac::Type( value ) : Tac::Ordinal, U16 {
    operator U16;
}
```

Value types: nominal and ordinal

Two flavors of value types exist: `Tac::Nominal` and `Tac::Ordinal`. They are identical except that nominal types only support equality comparison, while ordinal types additionally support comparison operators such as `<=`.

In general, value types:

- are passed by value or constant reference as parameters to functions
 - are returned by value from functions
 - have no virtual function support
 - must be modified as a single unit
-

Be careful when defining a value type as a subclass of another value type, since slicing concerns are applicable just as with C++. For example, if a derived value type is passed by value to a function taking its base value type, the derived type will be sliced down to the base value type when it's copied. The same can occur when assigning a derived value type to an instance of the base value type. In many cases, using composition and not inheritance is preferable.

Value types support the notification mechanism and AttrLog, with a caveat: an update is at the granularity of the entire type. Notifications are not supported for individual attributes. For example, any change to an attribute within a value type causes the whole value to be sent over AttrLog.

```
// This type will have equality operators.
NominalType : Tac::Type() : Tac::Nominal {
    attr : U32;
}
// This type will have equality and comparison operators.
OrdinalType : Tac::Type() : Tac::Ordinal {
    attr : U32;
}
```

Value types used as indices into collections (even unordered ones) need to support comparison operators, so they need to be `Tac::Ordinal`.

Value types used as indices into unordered collections need to support hashing. The recommended way to do this is to use ``hasDefaultHash``.

```
OrdinalTypeWithHash : Tac::Type() : Tac::Ordinal {
    `hasDefaultHash;
    attr : U32;
}
```

Constructor Arguments

Value types have two related ways of defining constructor arguments, both around the Tac::Type portion of their declaration.

These are:

- Explicit constructor arguments
- Implicit constructor arguments

All Value Types additionally generate a no-argument constructor, which is used to produce the somewhat special "default value" for the type.

The following is an example of a type with explicit constructor arguments:

```
ExplicitCtorArgs : Tac::Type( a, b ) : Tac::Nominal {
    a : U32;
    b : Tac::String;
    c : S64;
}
```

This will lead to generating a constructor that requires two arguments, a U32 and a string. These attributes of the Value Type will also be immutable after construction (No als or bls generated).

As a minor note, one-argument constructors are not marked as explicit, and may lead to implicit conversions.

If we look at a type with implicit constructor arguments instead:

```
ImplicitCtorArgs : Tac::Type : Tac::Nominal {
    a : U32;
    b : Tac::String;
    c : S64;
}
```

Note how here, we do not have any parentheses at all after the Tac::Type. This leads to all attributes in the type being included in the constructor, and all of these attributes also being mutable, as opposed to immutable (e.g. als, bls, and cls are all generated).

Arithmetic operators

Value types can support the following arithmetic operators: + - & | ^ *

These arithmetic operators (`<op>`) as well as their corresponding assignment operators (`<op>=`) can either be automatically generated or user defined.

In order to automatically generate arithmetic operators, the following statement needs to be added to the TAC definition of the value type: `operator <op>;`

The corresponding assignment operators can also be automatically generated using:
operator <op>=;

When automatically generating either the arithmetic operator or its corresponding assignment operator, TACC will also automatically generate the other operator if it does not have a user definition.

A full example can be seen below:

```
NominalTypeWithAutoOp : Tac::Type() : Tac::Nominal {
    attr : U32;
    operator |=; // This will also automatically generate the |= operator.
    operator +=; // This will also automatically generate the + operator.
    // This is more explicit and thus is preferred.
    operator &;
    operator &=;
}
```

In order to generate custom definitions for the arithmetic operators or their corresponding assignment operators, the following statement needs to be added to the TAC definition of the value type: **operator <op> : extern;**

Unlike the automatic arithmetic operator generation, using a custom definition of either the arithmetic operator or its corresponding assignment operator will not result in the automatic generation of the other operator.

The implementation of the automatically generated arithmetic operator relies on the corresponding assignment operator. Thus, users only need to specify custom definitions for the assignment operator and can rely on the automatic definition of the corresponding arithmetic operator.

For custom arithmetic operators, a C++ definition which implements the following method signature needs to be provided: **<Type> <Type>::operator<op>(<Type> const & t) const**

For the corresponding assignment operators, a C++ definition which implements the following method signature needs to be provided:

<Type>& <Type>::operator<op>=(<Type> const & t)

If **`passByValue=false;** is not specified and the size of the type is small enough, then the TACC compiler might opt to pass it by value rather than by reference as an optimization. In such cases, the operator signatures need to be modified accordingly (taking in **<Type>** instead of **<Type> const &**).

A full example can be seen below:

```
NominalTypeWithCustomOp : Tac::Type() : Tac::Nominal {
    attr1 : U64;
    attr2 : U64;
    operator |= : extern; // The |= operator will not be automatically generated.
    operator += : extern; // The + operator will not be automatically generated.
    // The automatically generated & operator will use the custom defined &=
operator.
    operator &= : extern;
    operator &;
}
```

```
NominalTypeWithCustomOp
NominalTypeWithCustomOp::operator|( NominalTypeWithCustomOp const & other )
const {
    NominalTypeWithCustomOp retVal{ *this };
    // This custom implementation only |'s attr1.
    retVal.attr1Is( attr1() | other.attr1() );
    return retVal;
}
NominalTypeWithCustomOp &
NominalTypeWithCustomOp::operator+=( NominalTypeWithCustomOp const & other ) {
    // This custom implementation "cross" adds attr1 and attr2.
    attr1Is( attr1() + other.attr2() );
    attr2Is( attr2() + other.attr1() );
    return *this;
}
NominalTypeWithCustomOp &
NominalTypeWithCustomOp::operator&=( NominalTypeWithCustomOp const & other ) {
    attr1Is( attr1() & other.attr1() );
    attr2Is( attr2() & other.attr2() );
    return *this;
}
```

Pointer types: `PtrInterface` and `Entity`

The `PtrInterface` and `Entity` types are closely related. `Tac::Entity` is a subclass of `Tac::PtrInterface` and shares many characteristics:

- pointers are passed as function parameters and return values
- virtual functions are supported
- inheritance is supported
- ownership is shared via reference-counting with `Tac::Ptr`

`Tac::PtrInterface`:

- is non-notifying by default, but it can be enabled
- doesn't support AttrLog

`Tac::Entity`:

- notifying by default
- supports AttrLog at the granularity of individual attributes or collection elements
- can be inserted into `Tac::Dir`
- uses more memory than `Tac::PtrInterface`
- has more generated code than `Tac::PtrInterface`

Choosing between `PtrInterface` and `Entity`

Use `Tac::Entity` when you need values of the type to be synchronized outside of the agent via AttrLog or need to store it in a `Tac::Dir`.

Also use `Tac::Entity` if needed for use with `when statements`, as these can leverage the parent pointer in entities.

`Tac::Entity` uses more memory and generates a lot more C++ code, resulting in longer compilation times and larger binaries.

Use `Tac::PtrInterface` for all objects local to the agent, enabling notifications if required.

| Type | Base size in bytes (32 b / 64 b) | AttrLog |
|---|----------------------------------|---------|
| <code>Tac::PtrInterface</code> | 8 / 16 | No |
| <code>Tac::PtrInterface</code> with <code>`isNotifyingByDefault;</code> | 12 / 24 | No |
| <code>Tac::Entity</code> with <code>`isNotifyingByDefault=false;</code> | 16 / 32 | No |
| <code>Tac::Entity</code> | 20 / 40 | Yes |
| <code>Tac::Nominal</code> | 0 | Yes |
| <code>Tac::Ordinal</code> | 0 | Yes |

Note that

```
taccExtension << "attrLog";
```

needs to be specified at the top of a `.tac` file to enable code-generation for AttrLog.

Tac::Ptr

`Tac::Ptr` is the pointer type used for `Tac::PtrInterface`, `Tac::Entity`, and `Tac::Constrainer` types. Each instance of a `Tac::Ptr` increments an intrusive reference-count of the object it points to. When the last reference goes away, the object is destroyed and the memory is released. A "raw" pointer is available via `Tac::Ptr::ptr()`.

Constructor arguments

The constructor for a Pointer type is defined based on the `Tac::Type` section of the type definition, and can be either explicit or implicit.

An explicit constructor will list the constructor arguments that must be passed when constructing an object of this type, inside parentheses. As an example:

```
PointerTypes : Tac::Type( a ) : Tac::PtrInterface {
    a : U32;
    b : Tac::String;
}
```

This type specifies attribute `a` as the only constructor argument. This attribute must be passed when constructing the value, or when creating this type via an instantiating collection or singleton.

The `a` attribute is also made immutable by default, allowing it to be used as a collection key for an internally keyed collection. (It can be made mutable again using the `\=;` trait).

For a type with no parentheses, we instead have implicitly defined constructor arguments, based on the parent type. As an example:

```
BaseType : extensible Tac::Type( a ) : Tac::PtrInterface {
    a : U32;
    b : Tac::String;
}
DerivedType : Tac::Type : BaseType {
    c : U64;
}
```

Here, `DerivedType` will have one constructor argument, `a`, inherited from `BaseType`.

Note that `PtrInterface` has no constructor arguments, so a type derived from it with implicit constructor arguments will have a no-arg constructor.

A type derived from Entity directly will have a single constructor argument, `name`. It is recommended to be explicit with types deriving from Entity, rather than implicit because of this.

Type alias

A type alias is a name for another type.

You can use a type alias name anywhere you would use the name of the type it aliases.

Similar features in C/C++ are also called **type aliases**.

TAC

There are three forms of type aliases.

The first form provides a new name for an existing type name. In the example below, we define **IntType** as an alias for type **S64**.

```
using IntType = S64;
```

IntType.tac

The aliased type, **S64** in this example, can be a simple type name or a scoped type name. If you're writing a TAC model and you're using a deeply-scoped type name, it can be handy to define a type alias for the type name to make the model easier to read.

With the second type alias form, you can alias the type of an accessor expression. For example, below we define **T1b::AType** as an alias for the type of attribute **a** within the **T1a** type.

```
T1a : Tac::Type() : Tac::PtrInterface {
    a : Tac::String;
}

T1b : Tac::Type() : Tac::PtrInterface {
    using AType = decltype( T1a::a );
}
```

AType1.tac

Both this form and the third type alias form described below require the **decltype** keyword. Here, we use the accessor expression **T1a::a** to indicate that we want to alias the type of that specific attribute within that specific type.

The next example is similar to the previous one, except that we use an attribute chain expression.

```
T2a : Tac::Type() : Tac::PtrInterface {
    a : Tac::String;
}

T2b : Tac::Type() : Tac::PtrInterface {
    f : T2a::Ptr;
    using AType = decltype( f::a );
}
```

AType2.tac

Here, within **decltype**, we use attribute **f** of type **T2b** to access attribute **a** so we can alias its type as **AType**.

The third type alias form is similar to the second form except that it uses *qualifiers* to access the desired type of a complex expression. In TAC, items like collections and functions are not first class; they cannot be passed as arguments, nor can they be specified as function return types. Therefore, TAC does not support aliasing the type of a collection or function. TAC does, however, support aliasing the types associated with these items by attaching a qualifier to the `decltype` expression.

In the example below, we use qualifiers to alias types associated with the internally-keyed collection attribute `c` and the function `f`.

Qualifiers.tac

```
T3a : Tac::Type( a ) : Tac::PtrInterface {
    a : Tac::String;
    f : inline T3a::Ptr( s : Tac::String ) {
        return T3a( s );
    }
}

T3b : Tac::Type() : Tac::PtrInterface {
    c : T3a[ a ];
    v : vector U32[];
    using CKey = decltype( c )::keyType;
    using CKeyOpt = CKey::Optional;
    using CVal = decltype( c )::valueType;
    using V = decltype( v )::elementType;
    using FRetType = decltype( T3a::f )::resultType;
}
```

There are several things to note about this example:

- Type alias `CKey` uses the `keyType` qualifier to obtain the type of the key of collection `c`. The `keyType` qualifier is valid only for `map` collections.
- Type alias `CKeyOpt` uses type alias `CKey` as part of its aliased type.
- Type alias `CVal` uses the `valueType` qualifier to obtain the type of the value of collection `c`. The `valueType` qualifier is valid only for `map` collections.
- Type alias `V` uses the `elementType` qualifier to obtain the type of the element of vector `v`. The `elementType` qualifier is valid for all collection types except `map` collections.
- Type alias `FRetType` uses the `resultType` qualifier to obtain the result type of function `T3a:f`. The `resultType` qualifier is valid only for functions.

C++ API

All TAC type aliases map to `using` statements in C++. Consider again the simple type alias example:

```
using IntType = S64;
```

In C++, this maps to the following `using` statement:

```
using IntType = S64;
```

For type aliases that use `decltype`, the TAC compiler generates an alias for the underlying type. Consider again the attribute accessor example:

```
T1a : Tac::Type() : Tac::PtrInterface {
    a : Tac::String;
}

T1b : Tac::Type() : Tac::PtrInterface {
    using AType = decltype( T1a::a );
}
```

In the generated C++ code, `AType` is declared as an alias of the type of the `a` attribute, which is `Tac::String`:

```
using AType = Tac::String;
```

C++ API for Enum Type Aliases

For the C++ mapping of a type alias of an `enum` type, the TAC compiler generates not only the C++ type alias, but also `enum` helper functions whose names are based on the name of the type alias.

Consider the following `enum` type and type alias definitions:

```
Food : Tac::Namespace {
Groups : Tac::Namespace {

Fruit : Tac::Enum {
    apple;
    banana;
    cherry;
}

} // namespace Groups

Pie : Tac::Namespace {

using Flavors = Groups::Fruit;

} // namespace Pie
} // namespace Food
```

Enum.tac

The C++ mapping for `enum` type `Fruit` provides the following helper functions with names based on the name of the `enum`:

```
Fruit FruitInstance( U32 v );
Fruit FruitInstance( Tac::String const & str );
```

For type alias `Flavors`, the generated C++ code includes the same functions except based on the type alias name:

```
Fruit FlavorsInstance( U32 v );
Fruit FlavorsInstance( Tac::String const & str );
```

Additionally, if the scope of the type alias is different than the scope of the enum, the enumerators and accessor functions normally generated for an `enum` are also generated into the same scope as the type alias.

The `Fruit` type lives within the `Food::Groups` namespace. The type alias `Flavors` lives within the `Food::Pie` namespace. The generated C++ code for `Fruit` includes not only the `enum` definition along with its enumerators, but also enumerator accessor functions for the enumerator values, the `FruitInstance` functions, and a `stringValue` function for converting a `Fruit` enumerator to its string name:

```
namespace Food::Groups {

enum Fruit : U8 {
    apple_ = 0,
    banana_ = 1,
    cherry_ = 2
};
constexpr Fruit apple() { return apple_; }
constexpr Fruit banana() { return banana_; }
constexpr Fruit cherry() { return cherry_; }

Fruit FruitInstance( U32 v );
Fruit FruitInstance( Tac::String const & str );

Tac::String stringValue( Fruit e );

} // namespace Food::Groups
```

Note that in the generated C++ code, the enumerators `apple_`, `banana_` and `cherry_`, the enumerator accessor functions, the `FruitInstance` functions, and the `stringValue` function all reside at the same scope as `Fruit`. Since `Flavors` is a type alias for `Fruit` residing in a different scope than `Fruit`, the generated code for `Flavors` also provides the `Fruit` enumerators, accessors, instance functions, and the `stringValue` function in the scope where `Flavors` is defined:

```
namespace Food::Pie {

using Flavors = Groups::Fruit;

// "using enum" pulls all the
// Fruit enumerators into this scope.
using enum Groups::Fruit;

constexpr Groups::Fruit apple() { return apple_; }
constexpr Groups::Fruit banana() { return banana_; }
constexpr Groups::Fruit cherry() { return cherry_; }

Fruit FlavorInstance( U32 v );
Fruit FlavorInstance( Tac::String const & str );

Tac::String stringValue( Fruit e );

} // namespace Food::Pie
```

The **C++** `using enum` construct introduces the `Fruit` enumerator names into the scope where `using enum` appears.

This section describes properties supported in a Tac::Type declaration.

allowsDirInstantiation

tacc has a special framework type called `Tac::Dir`, which allows storing the objects of different Entity types. If you are unfamiliar with `Tac::Dir`, see [this tacc-faq](#). An Entity by default is not eligible to be instantiated within a `Tac::Dir`. Using this property makes it eligible.

`Tac::Dir` is typically used as a mount point within an object tree, which is shared between the agents via Attrlog. As such, this property is commonly used on a root Entity anchored to a mount point.

Syntax

Add a new declaration statement within the type declaration context. It could be in one of the following forms:

```
`allowsDirInstantiation;           // enables instantiation
`allowsDirInstantiation = true;    // same as above
`allowsDirInstantiation = false;   // disables instantiation
```

The last form is a no-op, as instantiation is disabled by default. A common practice is to use the first form.

How to use

Let's create a small tac model with two Entities. One we want to instantiate within a `Tac::Dir`, and another one which contains our target `Tac::Dir`.

```
Fan.tac
<<= TacModule( "Tac::Dir" );

Fan : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    speed : U32 = initially 42;
}

Switch : Tac::Type() : Tac::PtrInterface {
    fru : Tac::Dir;
}
```

`Fan` in above example has dir instantiation enabled. See how it allows you to create an entry inside the `fru` dir in the C++ code below. In case you're wondering, `fru` stands for "Field Replaceable Unit" -- parts of a switch you can replace in the field without shutting down the switch.

```
#include "Fan.h"

int main() {
    // Create a new Switch, and a fru Dir inside it
    const auto mendo = Tac::allocate< Switch >();
    const auto fru = mendo->fruIs( "fru" );

    // Instantiate a new Fan entry under the Dir
    const auto fan = fru->createEntity< Fan >( "fan1" ); // Ok
    std::cout << "Fan speed: " << fan->speed() << std::endl;
}
```

Error cases

Compile time

Since Nominal and PtrInterface objects cannot be instantiated within a `Tac::Dir`, it is an error to use this property on such types. tacc throws an error for the following types:

```
Fan : Tac::Type() : Tac::PtrInterface {
    `allowsDirInstantiation; // Err
}

Cpu : Tac::Type() : Tac::Nominal {
    `allowsDirInstantiation; // Err
}
```

Run time

Trying to create an Entity, without dir instantiation support, inside a `Tac::Dir` results in a runtime exception.

Cpu.tac

```
taccExtension << "attrLog";

Cpu : Tac::Type() : Tac::Entity {
    cores : U32;
}
```

`Cpu` does not declare ``allowsDirInstantiation`` in the previous tac model. If you run the C++ code below, you should see an exception with a hint about adding this property.

```
#include "Cpu.h"

int main() {
    try {
        // Instantiate a new Cpu entry under global root
        Tac::globalDir()->createEntity< Cpu >( "cpu0" ); // Err
    } catch ( Tac::DirInstantiationException const & ex ) {
        std::cout << "Got an exception: " << ex.what() << std::endl;
        return 0;
    }
    assert( false );
}
```

gnmiBytes

gNMI is an open protocol for managing state on network devices. It is typically used to retrieve switch state for monitoring purposes and to modify switch configuration.

The AirStream package converts EOS-native state to/from the format used in gNMI. It does this transparently for all TAC types, without the authors of the types needing to do anything special.

Normally, AirStream treats values of type `Tac::Bytes` as an opaque sequence of bytes. These usually correspond to "bytes" or "binary" types in the industry (such as YANG's `binary` type).

Sometimes you may want AirStream to treat your custom TAC type also as a binary blob. In such cases, you can add the `gnmiBytes` property to the type. You have to provide an implementation of `valueToGnmiBytes()` so that AirStream knows how to convert it to a sequence of bytes.

There's no need to provide a conversion the other way around (from a sequence of bytes to your type).

Here's an example taken from the BgpStream package in EOS:

```
[[ yang::description = "", yang::explicit_type = "binary" ]]
ExtCommunityValue : Tac::Type( value ) : Tac::Nominal {
    `gnmiBytes;
    [[ yang::description = "" ]]
    value : U64;
}
```

BgpOpenConfig.tac

```
#include "BgpOpenConfig.h"

std::vector< std::byte >
valueToGnmiBytes( ExtCommunityValue const & extCommValue ) noexcept {
    std::vector< std::byte > result;
    result.push_back( std::byte( ( extCommValue.value() >> 56 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 48 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 40 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 32 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 24 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 16 ) & 0xff ) );
    result.push_back( std::byte( ( extCommValue.value() >> 8 ) & 0xff ) );
    result.push_back( std::byte( extCommValue.value() & 0xff ) );
    return result;
}
```

BgpOpenConfig.tin

gnmiString

gNMI is an open protocol for managing state on network devices. It is typically used to retrieve switch state for monitoring purposes and to modify switch configuration.

The AirStream package converts EOS-native state to/from the format used in gNMI. It does this transparently for all TAC types, without the authors of the types needing to do anything special.

Sometimes the author of a type may wish to override AirStream's default behaviour and specify that the type should be treated as a string. This is often the case when the type is usually represented as a string in some well-known format in the industry that doesn't match how its value is stored internally.

For example, an IPv4 address may be stored as a `U32`, but is usually represented as a string in the quad-dotted form, e.g., `127.0.0.1`.

Similarly, an IPv6 address may be stored as two `U64`s (or four `U32`s) but is usually represented as a string, e.g., `::1`.

In such cases, you can add the `gnmiString` property to the type to tell AirStream to not use the default representation and to treat it as a string instead. You have to provide implementations of `gnmiStringToValue()` and `valueToGnmiString()` to convert to/from your desired string format.

Example:

`IPv4.tac`

```
IPv4 : Tac::Type( value ) : Tac::Ordinal, U32 {
    `gnmiString;
}
```

```
#include "IPv4.h"
#include <TacMarco/Fmt.h>

Tac::String
valueToGnmiString( const IPv4 & ip ) noexcept {
    return ArFmt::format( "{}.{}.{}.{}",
        ( ip.value() >> 24 ) & 0xff,
        ( ip.value() >> 16 ) & 0xff,
        ( ip.value() >> 8 ) & 0xff,
        ip.value() & 0xff );
}

IPv4
gnmiStringToValue( std::string_view & str, IPv4 * /* ignore */ ) {
    Tac::CharRange cr{ str };
    U32 a = Tac::sequenceToValue< U32 >( cr );
    cr.needString( "." );
    U32 b = Tac::sequenceToValue< U32 >( cr );
    cr.needString( "." );
    U32 c = Tac::sequenceToValue< U32 >( cr );
    cr.needString( "." );
    U32 d = Tac::sequenceToValue< U32 >( cr );
    str = cr.remaining();
    return { ( a << 24 ) | ( b << 16 ) | ( c << 8 ) | d };
}

int
main() {
    IPv4 ip = 2130706433;
    std::cout << valueToGnmiString( ip ) << "\n"; // prints "127.0.0.1"
    std::string_view s = "127.0.0.1";
    IPv4 ip2 = gnmiStringToValue( s, nullptr );
    assert( ip == ip2 );
}
```

hasAttrLog

In rare occasions it is beneficial to declare a non-attrLogged type in a TAC file that contains the definitions of attrLogged types (e.g., will be synced with Sysdb). `hasAttrLog` is a type level property that disables attrLog code generation on a per type basis. An attribute of such a type in another type that is attrLogged will have to explicitly set `isLoggedAttr=false` to avoid a compilation error.

Example:

```
taccExtension << "attrLog";  
  
HasAttrLog : Tac::Namespace {  
  
    LocalStats : Tac::Type() : Tac::Nominal {  
        `hasAttrLog = false;  
        errCount : U64;  
    }  
  
    Speed : Tac::Type( speed ) : Tac::Nominal {  
        speed : U32;  
    }  
  
    Port : Tac::Type() : Tac::Entity {  
        speed : Speed;  
        duplex : bool;  
  
        stats : LocalStats {  
            // isLoggedAttr has to be explicitly set to false to avoid  
            // a compiler error  
            `isLoggedAttr = false;  
        }  
    }  
}
```

hasDefaultHash

All Nominal and Ordinal types that are to be used as part of indices into unordered collections need to have a hash function. Such a hash function should generate as even a spread of output values as possible over the set of possible inputs for good lookup performance. To defend against hash collision attack, the hash calculation should also be seeded, an operation similar to salting a password.

The `hasDefaultHash` property on a value type will cause the tacc compiler to generate a seeded hash function which combines the hashes of all attributes in that value type, aside from any marked with the `excludeFromComparison` property. The unseeded version is also generated.

Setting the `hasDefaultHash` property is strongly recommended for all new work as it avoids the problems experienced with manually defined hash functions such as uneven distributions, failure to update hash function when new attributes are added, etc.

One caution is that the iteration order over unordered collections with seeded hash functions is not necessarily consistent over all invocations. This can impact legacy tests that may rely on such ordering; these tests should be fixed. See description of `useDeprecatedHash_DoNotCopyThis` collection attribute.

There are some value types in EOS that need a user-defined hash function for some collections, but when used as index for other collections can use a better machine generated hash. In this case, setting `hasDefaultHash` to `addSeededHash` will automatically generate just the seeded version. This causes any collections with this type as index to use the user-supplied hash function if `useDeprecatedHash_DoNotCopyThis` is set for that collection, but if not set will use the machine-generated function.

Syntax

```
`hasDefaultHash;                                // Generate both seeded and unseeded hash
functions.
`hasDefaultHash=addSeededHash; // Generate just seeded hash function.
```

Example

```
taccExtension << "attrLog";

TypeWithDefaultHash : Tac::Type( anInt, aString, aDouble ) : Tac::Ordinal {
    `hasDefaultHash;
    anInt : U32;
    aString : Tac::String;
    aDouble : double {
        `excludeFromComparison; // don't compare/hash this
    }
}

TypeWithOwnHash : Tac::Type( one, two, three ) : Tac::Ordinal {
    `hasDefaultHash = addSeededHash;
    one : U32;
    two : Tac::String;
    three : U64;
    hash : extern U32(); // usr-defined hash for collections needing legacy
behavior
}

ExampleEntity : Tac::Type( name ) : Tac::Entity {
    mySet : set void[ TypeWithDefaultHash ];
    myColMyHash : U32[ TypeWithOwnHash ] { // uses our external hash function.
        `useDeprecatedHash_DoNotCopyThis;
    }
    myColMachineHash : U32[ TypeWithOwnHash ]; // uses machine generated hash
}
```

hasExternalStrep

The compiler normally produces the following functions to serialize or deserialize a type to string:

- `valueToStrep` converts a type T into a string representation
- `sequenceToValue` deserializes the string representation and creates an object of type T
- `operator<<` pretty-prints the type calling `valueToStrep`

Specifying `hasExternalStrep` lets you provide the implementation for those methods, which can be useful to customize how a type gets printed. These should be defined in the namespace in which the type is defined.

For instance, for the following Tac type:

```
HasExternalStrep : Tac::Namespace {
    Foo : Tac::Type( a, b ) : Tac::Ordinal {
        `hasExternalStrep;
        a : U32;
        b : U32;
    }
}
```

Example.tac

Here is how the methods can be implemented and used:

```
#include "Example.h"
#include <iostream>
#include <TacMarco/Fmt.h>

namespace HasExternalStrep {

    Tac::String
    valueToStrep( const Foo & f ) {
        return ArFmt::format( "Foo( a={}, b={} )", f.a(), f.b() );
    }

    Foo
    sequenceToValue( Tac::CharRange & charRange, Foo * /*ignore*/ ) {
        charRange.needString( "Foo( a=" );
        U32 a = Tac::sequenceToValue< U32 >( charRange );
        charRange.needString( ", b=" );
        U32 b = Tac::sequenceToValue< U32 >( charRange );
        charRange.needString( ")" );
        return Foo( a, b );
    }

    std::ostream &
    operator<<( std::ostream & s, const Foo & f ) {
        s << valueToStrep( f ) << "(stream version)";
        return s;
    }
} // namespace HasExternalStrep

void
hasExternalStrepDemo() {
    using namespace HasExternalStrep;
    Foo f( 1, 2 );
    Tac::String strep = valueToStrep( f );
    std::cout << strep << std::endl; // prints "Foo( a=1, b=2 )"
    Tac::CharRange cr( strep );
    Foo result = sequenceToValue( cr, ( Foo * )nullptr );
    std::cout << "Result=" << result
              << std::endl; // prints "Result=Foo( a=1, b=2 )(stream version)"
    assert( result == f );
}

int main() {
    hasExternalStrepDemo();
}
```

hasFactoryFunction

`hasFactoryFunction` type property causes two effects when used on a type declaration:

1. the generated C++ class has a global factory function that can be used to instantiate an object of that class,
2. object of the type can be instantiated through a mechanism known as "GenericIf" (Generic Interface), for example from Python.

Syntax

`hasFactoryFunction` can be used only on pointer types (`Tac::Entity` and `Tac::PtrInterface`) or types derived from `Tac::Constrainer`. Using this type property on a value type (`Tac::Nominal` or `Tac::Ordinal`) will result in an error.

Add a new declaration statement within the type declaration context. It could be in one of the following forms:

```
`hasFactoryFunction;           // enables factory function
`hasFactoryFunction = true;   // same as above
`hasFactoryFunction = false;  // disables factory function
```

The last form is a no-op, because factory function is disabled by default. A common practice is to use the first form.

Object creation without factory function

An object of a pointer type can typically be created through an `Is` method or by directly calling `Tac::allocate`, like in the example below.

```
Type1 : Tac::Type( attrX ) : Tac::PtrInterface {
    attrX : U32;
}
```

```
#include "Type1.h"

int main() {
    auto a = Type1::Type1Is( 100 );
    auto b = Tac::allocate< Type1 >( 200 );
    std::cout << "Address of object a: " << a << std::endl;
    std::cout << "Address of object b: " << b << std::endl;
}
```

Additionally, entities that allow dir instantiation (see `allowsDirInstantiation`) can be created with a `createEntity` call on a `Tac::Dir` object.

Object creation with factory function

`hasFactoryFunction` provides the class with a global factory function named `myTypeFactory` where `MyType` is the declared type name. The first character of the function is lowercase. If the type takes constructor arguments they become the arguments of the factory function. The function can be used as a shorthand for calling `Is` method or `Tac::allocate`.

`hasFactoryFunction` is also required to allow creating an object in Python using `Tac.newInstance` call.

Example use

An example below shows how a factory function can be used to create an object.

`Type2.tac`

```
Type2 : Tac::Type( attrY ) : Tac::PtrInterface {
    `hasFactoryFunction;
    attrY : U32;
}
```

```
#include "Type2.h"

int main() {
    auto c = type2Factory( 300 );
    std::cout << "Address of object c: " << c << std::endl;
}
```

Below Python code is possible only when `hasFactoryFunction` type attribute is used on type declaration.

```
import Tac

d = Tac.newInstance( "Type2", 400 )
```

Error cases

Using this type property on a value type (`Tac::Nominal` or `Tac::Ordinal`) will result in a compile time error.

`Type3.tac`

```
Type3 : Tac::Type : Tac::Nominal {
    `hasFactoryFunction;
    attrZ : U32;
}
```

hasPackedRep

`hasPackedRep` has two uses.

Used on a type, the generated C++ class is marked with the C++ `packed` attribute. This minimizes its size by omitting padding between members.

Used on a dynarray of bools, it enables a space-efficient bit array implementation.

Dynarray is deprecated. Please use `Vector`. `vector bool[]` is optimized for space by default.

Syntax

```
`hasPackedRep;           // enables packed representation
`hasPackedRep = true;   // same as above
`hasPackedRep = false;  // disables packed representation
```

The last form is a no-op, as `hasPackedRep` is disabled by default.

Examples

```
PackedType : Tac::Type() : Tac::Nominal {
    `hasPackedRep;
    first : bool;
    second : U32;
    third : bool;
    fourth : U32;
}
```

```
PackedBools : Tac::Type() : Tac::Nominal {
    bitArray : dynarray bool[] {
        `hasPackedRep;
    }
    useThisInstead : vector bool[];
}
```

Error cases

Because non-aligned data members could cause issues, it is an error to use `hasPackedRep` in the following contexts:

- Pointer types without `hasOwnPtrInterface`.
- Types containing any non-array `collection` attributes
- Types containing `Tac::String` attributes or `Tac::Bytes` attributes.
- Types containing smart pointer attributes

implAccessAllowed

`implAccessAllowed` can be used to declare a type as a friend of a given type. This let the C++ class, which is generated for a friend type, access private and protected members of the C++ class generated for a type declaring this property. It simply translates to a friend [declaration](#) in the generated C++ class.

Unlike C++, friend functions are not supported. Also, a type must be declared *before* it is used in an `implAccessAllowed` declaration.

Syntax

This is how a single type can be declared a friend:

```
`implAccessAllowed = <type-name>;
```

where the is the name of the friend. The named has to be qualified, depending on the scope at which the friend is defined.

This property can be repeated multiple times, once per friend type. A single instance can also declare multiple friends.

```
`implAccessAllowed = ( <comma separated type-names> );
```

How to use

Following example defines two types, one nested inside another. The nested type declares the enclosing type as a friend.

Led.tac

```
Led : Tac::Type() : Tac::PtrInterface {
    State : Tac::Type() : Tac::Nominal {
        // Declare enclosing type as a friend
        `implAccessAllowed = Led;

        Color : Tac::Enum {
            green;
            yellow;
            red;
        }
        color : Color {
            `= : local; // mutator is protected
        }
    }

    state : State;
    isError : extern bool();
    setError : extern invasive void();
}
```

The `Led` class methods can access protected and private members of the `State` class in the following C++ code:

```
#include "Led.h"

bool
Led::isError() const {
    return state_.color_ == State::red(); // private data-member access
}

void
Led::setError() {
    State newState;
    newState.colorIs( State::red() ); // protected mutator call
    stateIs( newState );
}

int main() {
    const auto led = Tac::allocate< Led >();
    std::cout << "Error: " << std::boolalpha << led->isError() << std::endl;

    led->setError();
    std::cout << "Error: " << std::boolalpha << led->isError() << std::endl;
}
```

Following example shows how to declare multiple friends:

```
Hw : Tac::Namespace {
Led : Tac::Type() : Tac::PtrInterface {}

Cpu : Tac::Type() : Tac::PtrInterface {
    Core : Tac::Type() : Tac::Nominal {}
}
}

Conf1 : Tac::Type() : Tac::PtrInterface {
    `implAccessAllowed = Hw::Led;
    `implAccessAllowed = Hw::Cpu::Core;
}

Conf2 : Tac::Type() : Tac::PtrInterface {
    `implAccessAllowed = ( ::Hw::Led, Hw::Cpu::Core );
}
```

Error cases

A type other than a `Tac::Type` (e.g. Enum or Fundamental types) cannot be declared a friend. tacc throws an error for such cases.

tacc also throws an error if the type is not previously declared, like in the following example:

```
Led : Tac::Type() : Tac::PtrInterface {
    `implAccessAllowed = State;
    State : Tac::Type() : Tac::Nominal {}
}
```

Forward-declared type (i.e. declared but not defined) can be used in `implAccessAllowed`.

supportsNonLocalNotification

`supportsNonLocalNotification` is a property used to allow the definition of a `when` statement on an object's state, inside a reference-counted type that does not derive from `Tac::Constrainer`.

This property is intended for the [Config Model Views](#) types. Avoid using this property, and instead use `when` statements within a `Constrainer` type for other use cases.

The use of `supportsNonLocalNotification` is an error when it is used inside a value type.

Syntax

CMV.tac

```
EosEntCMV : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    `supportsNonLocalNotification;

    eosEnt : in EosEnt::Ptr;
    handleCol : void( i : U32 ) {}
        when eosEnt::col[ i ] => handleCol( i );
}
```

Set

A set is a collection of unique elements. Those elements may be also referred to as keys. The keys have no values associated with them, as opposed to a map that stores values associated with the keys.

The elements can be stored in two ways: unordered or ordered. Those two structures use different underlying container implementations, have different operation complexities and slightly different element type requirements. Search, insertion and removal in unordered sets have average constant-time complexity, whereas in ordered sets these are logarithmic.

TAC

An attribute storing a set collection is defined using the `set` keyword together with a `void` value type (sets do not store values associated with the keys as maps do) and element type in square brackets.

Sets with built-in types

Below are examples of set collection attributes with elements being of built-in types:

```
vlan : set void[ U32 ];
agentName : set void[ Tac::String ];
```

Defining an ordered set requires using an additional `ordered` keyword. For example:

```
orderedVlan : ordered set void[ U32 ];
orderedAgentName : ordered set void[ Tac::String ];
```

Element type requirements

Elements stored in the sets must be less-than-comparable. In C++ terms they need to have an `operator<` defined. Most built-in types (excluding `Tac::Sensitive` and `bool`) can be used for set elements. As for user-defined element types they have to be `Tac::Ordinal` value types.

For unordered sets, additionally the element type must provide a hash. So in case of `Tac::Ordinal` types they will usually have `hasDefaultHash` type property.

Pointers may also be elements, including `Ptr`, `PtrConst`, `RawPtr` and `RawPtrConst` of `Tac::Entity` or `Tac::PtrInterface` types.

Sets with user-defined types

Below are examples of set collection attributes with elements being of a user-defined type:

```
macAddr : set void[ MacAddr ];
orderedMacAddr : ordered set void[ MacAddr ];
```

The above element type may be defined as in the following example:

```
MacAddr : Tac::Type() : Tac::Ordinal {  
    `hasDefaultHash; // hash is only required for unordered sets  
    addr : array U8[ 6 ];  
}
```

Element ordering

For ordered sets the element comparison order defines in what sequence the elements are stored and iterated over. For unordered sets no specific arrangement should be assumed.

Elements stored in the sets may not be modified because modifications could change ordering and/or hashing of the elements and corrupt the underlying container.

C++ API

Below is a summary of the generated C++ API for a set attribute `item` storing elements of non-trivial type `T`.

```
item : set void[ T ];
```

item set attribute

▼ C++ API generated for item

```

// Check if the given element exists in the set.
bool itemHas( T const & element ) const;

// Get the number of stored elements.
U32 items() const;

// Add the given element into the set,
// if it doesn't already contain an equivalent element.
void itemIs( T const & elementToAdd );

// Delete the given element from the set, if it contains the element.
// Return true if the element was deleted, or false if it didn't exist in the
set.
bool itemDel( T const & elementToDelete );

// Delete the element pointed by the given iterator, if the set still has it.
// Return true if the element was deleted, or false if it didn't exist in the
set.
bool itemDel( ItemIterator const & iteratorPointingAtElementToDelete );

// Delete all elements.
void itemDelAll();

// Get an iterator to the first element in iteration order.
ItemIteratorConst itemIteratorConst() const;
ItemIterator itemIterator();

// Get an iterator to the element equivalent or greater than the provided one.
ItemIteratorConst itemIteratorConst( T const & elementToFind ) const;
ItemIterator itemIterator( T const & elementToFind );

// Access the underlying container directly.
// The type of the container is different for ordered and unordered sets.
ItemColl const & itemColl() const;

```

For ordered sets the C++ API is extended with the following:

▼ Additional C++ API generated for an ordered set

```

// Get a reverse iterator to the last element in iteration order.
ItemRevIteratorConst itemRevIteratorConst() const;
ItemRevIterator itemRevIterator();

// Get a reverse iterator to the element equivalent or less than the provided
one.
ItemRevIteratorConst itemRevIteratorConst( T const & elementToFind );
ItemRevIterator itemRevIterator( T const & elementToFind );

```

Example of basic operations

Consider the following type containing an unordered set attribute:

```
// A type with an unordered set attribute.
IntfVrfStatus1 : Tac::Type( id ) : Tac::PtrInterface {
    `hasFactoryFunction;
    id : U32;
    name : Tac::String;
    vlan : set void[ U32 ];
}
```

Below is an example showing basic operations like adding and deleting elements as well as querying for the number of stored elements or presence of a specific one.

```
#include "Set1.h"

int main() {
    IntfVrfStatus1::Ptr st = intfVrfStatus1Factory( 1 );
    // Populate the set.
    for ( auto id : std::vector< U32 >{ 9, 99, 999, 20, 30, 4091 } ) {
        st->vlanIs( id );
    }
    // Print out some information.
    std::cout << "Size of vlan set is " << st->vlans() << std::endl;
    std::cout << "Id 99 is " << ( st->vlanHas( 99 ) ? "" : "not " ) << "in the
set"
                << std::endl;
    std::cout << "Id 100 is " << ( st->vlanHas( 100 ) ? "" : "not " ) << "in the
set"
                << std::endl;
    // Delete some elements.
    st->vlanDel( 999 );
    st->vlanDel( 30 );
    std::cout << "Size of vlan set after deletions is " << st->vlans() <<
std::endl;
    // Clear the set.
    st->vlanDelAll();
    std::cout << "Size of vlan set after clearing is " << st->vlans() <<
std::endl;
}
```

Iterators

Elements in sets can be traversed using iterators. Iterators are obtained with API calls listed in the [C++ API](#) section. Methods are provided to start with an iterator from the first element in the set, or from an element equivalent to the given one (or, if not found, the next one in iteration order).

Iteration order

Unordered sets can be traversed only in one direction. No assumptions should be made related to the order of iteration. The order is the same for repeated iterations in a single process execution. However it can be different the next time the process is run, even if the data stored in the set is identical.

Ordered sets can be traversed either in ascending or descending order, as defined by the comparison `operator<` of the element type. In this case the order is deterministic.

Iterator guarantees

Iterators for sets have stronger guarantees than the typical C++ containers. Iterators are not invalidated upon modification of the collection. In particular, an iterator remains valid on deletion of any element, including the one at the position of the iterator. It is possible to get the contents of the deleted element through an iterator and it can be advanced to the next element in iteration order. Therefore it is safe to delete elements from the set while iterating over it. No elements are missed after such operation.

Iterator C++ API

Full iterator API depends on the underlying container which is different between ordered and unordered sets. However, the basic portion of the interface commonly used while traversing a set is the same across iterator types:

▼ C++ API for iterators

```
// Check if the iterator is valid.
// If true, the element stored under the iterator can be obtained
// and the iterator can be advanced to the next element in iteration order.
// If false, the iterator has completed iterating over the whole collection.
// A false iterator is returned when obtaining an iterator from an empty
collection
// or when looking for an element beyond the last one in iteration order.
// A true iterator becomes false when advanced past the last element in
iteration
// order. A true iterator doesn't mean that the element still exists in the
set.
explicit operator bool() const;

// Advance the iterator to the next element in iteration order.
// This can be done only for a true iterator.
ItemIterator &operator++();

// Get the contents of the element stored under the iterator.
// This can be done only for a true iterator.
T const &key() const;
```

Iterating examples

Consider the following types containing the same attribute as ordered or unordered set:

Set2.tac

```
// A type with an unordered set attribute.
IntfVrfStatus1 : Tac::Type( id ) : Tac::PtrInterface {
    `hasFactoryFunction;
    id : U32;
    name : Tac::String;
    vlan : set void[ U32 ];
}

// A type with an ordered set attribute.
IntfVrfStatus2 : Tac::Type( id ) : Tac::PtrInterface {
    `hasFactoryFunction;
    id : U32;
    name : Tac::String;
    vlan : ordered set void[ U32 ];
}
```

Below are examples showing ways of iterating over above sets using the aforementioned [Iterator C++ API](#).

▼ Iterating by manually advancing the iterator

```
#include "Set2.h"

int main() {
    IntfVrfStatus1::Ptr unordst = intfVrfStatus1Factory( 1 );
    IntfVrfStatus2::Ptr ordst = intfVrfStatus2Factory( 2 );
    // Populate the sets.
    for ( auto id : std::vector< U32 >{ 9, 99, 999, 20, 30, 4091 } ) {
        unordst->vlanIs( id );
        ordst->vlanIs( id );
    }

    // Iterate over whole collection.
    std::cout << "Elements in the unordered set: ";
    for ( auto iter = unordst->vlanIterator(); iter; ++iter ) {
        std::cout << iter.key() << ' ';
    }
    std::cout << std::endl;

    std::cout << "Elements in the ordered set: ";
    for ( auto iter = ordst->vlanIterator(); iter; ++iter ) {
        std::cout << iter.key() << ' ';
    }
    std::cout << std::endl;

    // Iterate over ordered collection in reverse.
    std::cout << "Elements in the ordered set reversed: ";
    for ( auto iter = ordst->vlanRevIterator(); iter; ++iter ) {
        std::cout << iter.key() << ' ';
    }
    std::cout << std::endl;
}
```

It is also possible to use range-based for loops with the provided iterators:

▼ Iterating with range-based loops

```
#include "Set2.h"

int main() {
    IntfVrfStatus1::Ptr unordst = intfVrfStatus1Factory( 1 );
    IntfVrfStatus2::Ptr ordst = intfVrfStatus2Factory( 2 );
    // Populate the sets.
    for ( auto id : std::vector< U32 >{ 9, 99, 999, 20, 30, 4091 } ) {
        unordst->vlanIs( id );
        ordst->vlanIs( id );
    }

    // Iterate over whole collection.
    std::cout << "Elements in the unordered set: ";
    for ( U32 val : unordst->vlanIterator() ) {
        std::cout << val << ' ';
    }
    std::cout << std::endl;

    std::cout << "Elements in the ordered set: ";
    for ( U32 val : ordst->vlanIterator() ) {
        std::cout << val << ' ';
    }
    std::cout << std::endl;

    // Iterate over ordered collection in reverse.
    std::cout << "Elements in the ordered set reversed: ";
    for ( U32 val : ordst->vlanRevIterator() ) {
        std::cout << val << ' ';
    }
    std::cout << std::endl;
}
```

Please note that when using range-based loops with sets storing elements of user-defined types or `Ptr`/`PtrConst` pointers it is recommended to explicitly specify the named variable in the range declaration as reference to const. This is to avoid unnecessary copies of the objects (in case of value types) or creating temporary `Tac::Ptr` objects (which involves increasing and decreasing reference counts in targeted objects).

▼ Recommended ways of writing range-based loops with non-trivial types

```
for ( MacAddr const & ma : ent->macAddrIterator() ) {
    ...
}

for ( SomeEntityType::Ptr const & ptr : ent->childIterator() ) {
    ...
}
```

The following example demonstrates how to store a bookmark while iterating and resume later.

▼ Iterate from bookmark

```
#include "Set2.h"

int main() {
    IntfVrfStatus2::Ptr ordst = intfVrfStatus2Factory( 2 );
    // Populate the set.
    for ( auto id : std::vector< U32 >{ 9, 99, 999, 20, 30, 4091 } ) {
        ordst->vlanIs( id );
    }

    std::cout << "Elements in the ordered set: ";
    std::optional< U32 > bookmark;
    auto iter = ordst->vlanIterator();
    for ( ; iter; ++iter ) {
        if ( iter.key() > 50 ) {
            break;
        }
        std::cout << iter.key() << ' ';
    }
    if ( iter ) {
        std::cout << "...paused iterating";
        bookmark = iter.key();
    }
    std::cout << std::endl;

    // The set can be modified here

    if ( bookmark ) {
        std::cout << "Resuming: ";
        for ( U32 val : ordst->vlanIterator( bookmark.value() ) ) {
            std::cout << val << ' ';
        }
        std::cout << std::endl;
    }
}
```

The following example shows how it is possible to modify the set while iterating over it.

▼ *Modify set while iterating*

```
#include "Set2.h"

int main() {
    IntfVrfStatus1::Ptr unordst = intfVrfStatus1Factory( 1 );
    // Populate the set.
    for ( auto id : std::vector< U32 >{ 9, 99, 999, 20, 30, 4091 } ) {
        unordst->vlanIs( id );
    }

    std::cout << "Iterating: ";
    for ( auto iter = unordst->vlanIterator(); iter; ++iter ) {
        std::cout << iter.key() << ' ';
        if ( iter.key() % 2 == 0 ) {
            unordst->vlanDel( iter );
            std::cout << "deleted ";
        }
    }
    std::cout << std::endl;

    std::cout << "Iterating again: ";
    for ( U32 val : unordst->vlanIterator() ) {
        std::cout << val << ' ';
    }
    std::cout << std::endl;
}
```

Notification

Set attributes defined in a type with notification support generate notifications when elements are added or deleted. Those operations notify with the affected element.

Consider the following example TAC model with a constrainer demonstrating reactor syntax for a set:

```
SwitchSm.tac
Switch : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    agent : set void[ Tac::String ];
}

SwitchSm : Tac::Type( sw ) : Tac::Constrainer {
    sw : in Switch::PtrConst;
    handleAgent : extern void( agentName : Tac::String );
    sw::agent[ agentName ] => handleAgent( agentName );
}
```

The reactor may check the set to determine whether the element was added or deleted, like in the following example. Using `captureNotifyValues` is also possible.

```
#include "SwitchSm.h"

void
SwitchSm::handleAgent( Tac::String const & agentName ) const {
    std::cout << "Notified of change in set: " << agentName
        << ( sw()->agentHas( agentName ) ? " <- added" : " <- deleted" )
        << std::endl;
}

int main() {
    Switch::Ptr sw = Switch::SwitchIs();
    SwitchSm::Ptr swSm = SwitchSm::SwitchSmIs( sw );

    sw->agentIs( "Fru" );
    sw->agentIs( "Sand" );
    sw->agentIs( "Strata" );
    sw->agentDel( "Sand" );
    sw->agentDelAll();
}
```

Introspection API

The following table summarizes supported generic `TacAttr` API calls and how they map to generated `C++ API`:

| TacAttr method | C++ | Notes |
|-------------------------------|-------------------------------|---|
| <code>genericIfExist</code> | <code>itemHas(element)</code> | returns a C++ <code>bool</code> value |
| <code>genericIfGet</code> | <code>itemHas(element)</code> | returns a <code>GenericIf::Ptr</code> wrapper over a <code>bool</code> value |
| <code>genericIfMembers</code> | <code>items()</code> | |
| <code>genericIfSet</code> | <code>itemIs(element)</code> | argument must be a <code>GenericIf</code> wrapper over <code>true</code> boolean; passing <code>false</code> will throw |
| <code>genericIfDel</code> | <code>itemDel(element)</code> | |

Python API

The following table summarizes how C++ calls map to Python:

| C++ | Python |
|-------------------------------|---|
| <code>itemHas(element)</code> | <code>element in item</code> |
| <code>items()</code> | <code>len(item)</code> |
| <code>itemIs(element)</code> | <code>item.add(element)</code> or <code>item[element]=True</code> |
| <code>itemDel(element)</code> | <code>item.remove(element)</code> or <code>del item[element]</code> |
| <code>itemDelAll()</code> | <code>item.clear()</code> |

An example use of a set collection in Python is shown below:

```
import Tac

st = Tac.newInstance( "Set::IntfVrfStatus1", 1 )

# Populate the set.
for v in [ 9, 99, 999, 20, 30, 4091 ]:
    st.vlan.add( v )

# Print out some information.
print( "Size of vlan set is", len( st.vlan ) )
print( "Id 99 is in the set:", 99 in st.vlan )
print( "Id 100 is in the set:", 100 in st.vlan )

# Iterate over whole collection.
print( "Whole set:", end=' ' )
for v in st.vlan:
    print( v, end=' ' )
print()

# Delete some elements.
st.vlan.remove( 999 )
del st.vlan[ 30 ]
print( "Size of vlan set after deletion is", len( st.vlan ) )

# Clear the set.
st.vlan.clear()
print( "Size of vlan set after clearing is", len( st.vlan ) )
```

An example output of the above python script is:

```
Size of vlan set is 6
Id 99 is in the set: True
Id 100 is in the set: False
Whole set: 20 9 99 999 4091 30
Size of vlan set after deletion is 4
Size of vlan set after clearing is 0
```

Array

Arrays are fixed-size containers whose elements are default-initialized.

TAC

There are three ways to define an array, depending on the kind of its indices.

Simple arrays

```
item : array T[ 16 ];
```

Here, `item` is an array of some TAC type `T` with an index of type `u32`. The first index is `0` and the last index is `15`.

Choosing an index range

Arrays may be defined with indices over ranges that don't begin at zero.

The key type must be a range-limited integral split-inheritance type.

For example,

```
Key : Tac::Type( value ) : Tac::Ordinal, U16 {
    value {
        `range = 10 .. 13;
    }
}

Thing : Tac::Type() : Tac::PtrInterface {
    label : array Tac::String[ Key ];
}
```

The smallest index is `Key(10)` and the largest is `Key(13)`.

Enumeration keys

Arrays may also be defined over an [enumeration](#).

For example,

```
Shape : Tac::Enum {
    circle;
    square;
    triangle;
}

Shapes : Tac::Type : Tac::PtrInterface {
    count : array U32[ Shape ];
}
```

C++ API

This is a summary of the generated C++ API for an array attribute `item` of any TAC type `T`:

```
// Access the element at index i.
T item( Key i ) const;

// Get the count of elements.
U32 items() const;

// Update the element at index i.
void itemIs( Key i, T element );

// Delete the element at index i by setting it to the default value of T.
T itemDel( Key i );
T itemDel( ItemIterator const & i );

// Delete all elements.
void itemDelAll();

// Get an iterator to the first element.
auto itemIteratorConst() const;
auto itemIterator();

// Get an iterator starting at the ith element.
auto itemIteratorConst( Key i ) const;
auto itemIterator( Key i );
```

Let's take an example of a TAC type containing array attributes named `reg` and `buf`:

`Driver.tac`

```
RegisterIndex : Tac::Type( value ) : Tac::Nominal, U16 {
    value {
        `range = 1 .. 5;
    }
}

Driver : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    `hasFactoryFunction;
    reg : array U32[ RegisterIndex ];
    buf : array U8[ 8 ];
}
```

Accessors

An array attribute generates an accessor which takes some index and returns a copy of the element at that index.

An out-of-bounds access throws `Tac::RangeException`.

```
#include "Driver.h"

int main() {
    // Populate the arrays
    Driver::Ptr dr = Tac::allocate< Driver >();
    dr->regIs( RegisterIndex( 3 ), 3 );
    dr->regIs( RegisterIndex( 4 ), 5 );
    dr->bufIs( 0, 'A' ); // buf's key type is U32

    // Accessor
    // The RegisterIndex key is implicitly created from integer 3
    assert( dr->reg( 3 ) == 3 );
    assert( dr->buf( 0 ) == 'A' );

    // Elements are default initialized
    assert( dr->reg( 5 ) == U32{} );
    assert( dr->buf( 1 ) == U8{} );

    // Out of bounds access throws exception
    try {
        auto const & last [[maybe_unused]] = dr->reg( 999 );
        assert( false );
    } catch ( Tac::RangeException const & e ) {
        std::cout << e.what() << std::endl;
    }
}
```

Mutators

An element at a given index can be updated in-place via `regIs`. An element at a given index can be deleted (reset to default value) via `regDel`. `regDelAll` sets all elements to the default value.

```
#include "Driver.h"

int main() {
    Driver::Ptr dr = Tac::allocate< Driver >();

    // update element at index 1, 2 and 3
    dr->regIs( RegisterIndex( 1 ), 404 );
    dr->regIs( RegisterIndex( 2 ), 404 );
    dr->regIs( RegisterIndex( 3 ), 404 );

    dr->bufIs( 1, 'x' );
    dr->bufIs( 2, 'y' );
    dr->bufIs( 3, 'z' );

    // delete (set to default value) and return the element at index 1
    auto deleted = dr->regDel( RegisterIndex( 1 ) );
    assert( deleted == 404 );
    assert( dr->reg( RegisterIndex( 1 ) ) == U32{} );

    assert( dr->bufDel( 3 ) == 'z' );
    assert( dr->buf( 3 ) == U8{} );

    // delete (set to default value) all elements
    dr->regDelAll();
    assert( dr->reg( RegisterIndex( 2 ) ) == U32{} );
    assert( dr->reg( RegisterIndex( 3 ) ) == U32{} );
}
```

Iterators

Array attributes follow the tacc iterator protocol. Specifically, they have the following properties:

- Iteration order is well-defined and is based on the key type
- Collections can be iterated starting from a given key or index. In other words, an iterator can be created from any arbitrary key.
- Iterators are always valid and are convertible to `bool`. If an iterator is `true` then it denotes an element in the collection. The `false` iterator is the end of the collection and should not be dereferenced.

The following example shows how to use iterators in the tacc style:

```
#include "Driver.h"

int main() {
    Driver::Ptr dr = Tac::allocate< Driver >();

    // Iterate from the start to end
    for ( auto it = dr->regIterator(); it; ++it ) {
        std::cout << "index: " << it.key() << std::endl;
        dr->regIs( it.key(), 404 + it.key().value() );
    }

    // Iterator starting from a stored bookmark
    auto bookmark = RegisterIndex( 2 );
    if ( auto const it = dr->regIteratorConst( bookmark ) ) {
        assert( *it == 406 );
    }
}
```

Notification

Array attributes defined in a type with notification support generate notification when modified. An operation which results in a change notifies with an index where the change occurred.

State machines (`Tac::Constrainer`s) can react to changes within an array using the same collection reactor syntax as shown in the following example.

```

RegisterIndex : Tac::Type( value ) : Tac::Nominal, U16 {
    value {
        `range = 1 .. 5;
    }
}

Driver : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    `hasFactoryFunction;
    reg : array U32[ RegisterIndex ];
    buf : array U8[ 8 ];
}

DriverSm : Tac::Type( dr ) : Tac::Constrainer {
    dr : in Driver::PtrConst;
    handleReg : extern void( index : RegisterIndex );
    dr::reg[ index ] => handleReg( index );
}

```

The index within square brackets binds to the notification index described above.

```

#include "DriverSm.h"

void
DriverSm::handleReg( RegisterIndex index ) const {
    std::cout << "Update at " << index << std::endl;
}

int main() {
    auto dr = Tac::allocate< Driver >();
    auto drSm = Tac::allocate< DriverSm >( dr );
    dr->regIs( RegisterIndex( 4 ), 404 );
    dr->regIs( RegisterIndex( 1 ), 404 );
    dr->regDel( RegisterIndex( 4 ) );
}

```

Introspection API

The following table shows how the generic `TacAttr` API maps to the C++ API which is generated:

| TacAttr method | C++ |
|------------------|-------------------------------|
| genericIfGet | reg(index) |
| genericIfMembers | regs() |
| genericIfExist | regColl().isMemberOf(index) |
| genericIfSet | regIs(index, val) |
| genericIfDel | regDel(index) |

Python API

The following script shows a few examples of how to interact with an array within a C++ TAC object:

```

import Tac

dr = Tac.newInstance( "Array::Driver" )

# update an element
dr.reg[ 1 ] = 404
dr.reg[ 2 ] = 404
dr.buf[ 3 ] = ord( 'a' )
dr.buf[ 4 ] = ord( 'z' )

# out of bounds accesses throw an IndexError
try:
    _ = dr.reg[ 999 ]
except IndexError:
    pass

try:
    _ = dr.buf[ 999 ]
except IndexError:
    pass

# clear an element
del dr.reg[ 1 ]
assert dr.reg[ 1 ] == 0

# get lists
assert dr.reg.keys() == [ 1, 2, 3, 4, 5 ]
assert dr.reg.values() == [ 0, 404, 0, 0, 0 ]
assert dr.reg.items() == [ ( 1, 0 ), ( 2, 404 ), ( 3, 0 ), ( 4, 0 ), ( 5, 0 ) ]
assert dr.buf.items() == [ ( 0, 0 ), ( 1, 0 ), ( 2, 0 ), ( 3, 97 ),
                           ( 4, 122 ), ( 5, 0 ), ( 6, 0 ), ( 7, 0 ) ]

# string representations
assert "Tac._Collection 'reg'" in str( dr.reg )
assert "Tac._Collection 'buf'" in str( dr.buf )

# get iterators
it1 = zip( dr.reg.keys(), dr.reg.values() )
it2 = dr.reg.items()
assert list( it1 ) == list( it2 )

# "x in arr" searches for an index, not a value
assert 1 in dr.reg

# for array keys that are not integer literals, 'in' throws IndexError for
# invalid keys
try:
    _ = 999 in dr.reg
except IndexError:
    pass

# array keys which use the integer literal syntax, return false on invalid keys
assert 1 in dr.buf
assert 999 not in dr.buf

# delete all the elements
dr.reg.clear()

```

Vector

A vector is a dynamically-growable collection of items stored contiguously in memory. It permits random access to its items, but growing and shrinking the collection is only permitted at the end (like a stack).

TAC

You can define a vector collection attribute using `vector` keyword.

For example,

```
item : vector T[];
```

defines a vector attribute `item` of any TAC type `T`.

Small vectors

If the size of a vector collection is known to be small then it's possible to use an optimized representation in which items are stored inline.

For example,

```
age : vector U32[] {  
    `smallVectorSize = 4;  
}
```

will use an inline representation for `age` as long as its size doesn't exceed four.

Small vectors otherwise behave no differently.

Limitations

Vector attributes don't support the `captureNotifyValues` mechanism.

C++ API

Linking requirements: `libTacMarco.so`

This is a summary of the generated C++ API for a vector attribute `item` of any TAC type `T`:

```

// Access the first element.
T const & itemFront() const;

// Access the last element.
T const & itemBack() const;

// Access the element at index i.
T const & item( U32 i ) const;

// Get the count of elements.
U32 items() const;

// Get the count of elements the vector can store without reallocation.
U32 itemCapacity() const;

// Increase the storage capacity.
void itemReserve( U32 capacity );

// Decrease the storage capacity.
void itemShrink();

// Add an element at the end.
void itemPush( T element );

// Remove the element at the end.
T itemPop();

// Delete all elements.
void itemPopAll();

// Update the element at index i.
void itemIs( U32 i, T element );

// Access the underlying container. Note that the non-const version is non-
// notifying!
Marco::VectorStack< T > const & itemColl() const;
Marco::VectorStack< T > & itemCollNonNotif();

// Get an iterator to the first element.
auto itemIteratorConst() const;
auto itemIterator();

// Get an iterator starting at the ith element.
auto itemIteratorConst( U32 i ) const;
auto itemIterator( U32 i );

```

Let's take an example of a TAC type containing a vector attribute named `agent`:

`Switch.tac`

```

Switch : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    `hasFactoryFunction;
    agent : vector Tac::String[];
}

```

Accessors

A vector attribute generates three different accessors: the front element, the back element, and the element at an index. Unlike other TAC collections, these accessors return a const-reference, rather than a copy, of the element¹.

An out-of-bounds access throws `Marco::OutOfRangeException`.

```
#include "Switch.h"

int main() {
    // Populate the vector
    Switch::Ptr sw = Tac::allocate< Switch >();
    sw->agentPush( "Fru" );
    sw->agentPush( "Sand" );

    // Front and Back
    Tac::String const & f = sw->agentFront();
    Tac::String const & b = sw->agentBack();
    assert( f == "Fru" && b == "Sand" );

    // Indexed
    assert( sw->agent( 0 ) == f );

    // Out of bounds access throws exception
    try {
        auto const & last [[maybe_unused]] = sw->agent( 2 );
        assert( false );
    } catch ( Marco::OutOfRangeException const & e ) {
        std::cout << e.what() << std::endl;
    }
}
```

Size and capacity

You can get the size and the capacity of the `agent` vector by calling `agents()` and `agentCapacity()` respectively.

You can increase the capacity by calling `agentReserve(U32 newCapacity)`. If `newCapacity` is greater than the current capacity, new storage is allocated. Otherwise, it's a no-op.

You can request for the removal of unused capacity by calling `agentShrink()`.

```
#include "Switch.h"

int main() {
    Switch::Ptr sw = Tac::allocate< Switch >();
    sw->agentPush( "Fru" );
    sw->agentPush( "Sand" );

    U32 size = sw->agents();
    U32 cap = sw->agentCapacity();
    assert( size == 2 );
    assert( cap >= 2 );

    sw->agentReserve( 16 );
    assert( sw->agentCapacity() >= 16 );
    sw->agentShrink();
    std::cout << sw->agentCapacity() << std::endl;
}
```

Mutators

A new element can be inserted or pushed only at the back via `agentPush`. An element can be deleted or popped only from the back via `agentPop`, which returns the popped element. You cannot insert or delete at a random index. An element at a given index can be updated in-place via `agentIs`. `agentPopAll` deletes all the elements.

```
#include "Switch.h"

int main() {
    Switch::Ptr sw = Tac::allocate< Switch >();

    // push elements at the back
    sw->agentPush( "Fru" );
    sw->agentPush( "Strata" );

    // update element at index 1
    sw->agentIs( 1, "Sand" );

    // pop removes and returns the last element
    Tac::String last = sw->agentPop();
    std::cout << last << std::endl;
    assert( sw->agents() == 1 );

    // popAll clears up the content
    sw->agentPopAll();
    assert( sw->agents() == 0 );
}
```

Underlying container

A vector attribute is represented by a data member of the `Marco::VectorStack` container type in the generated C++ class. You can get a reference to the data member container using the `Coll` accessors as shown below:

```
#include "Switch.h"

int main() {
    auto sw = Tac::allocate< Switch >();

    // Coll accessor with "NonNotif" suffix gives a non-const reference
    sw->agentCollNonNotif() = { "Fru", "Strata" };

    // Coll accessor without any suffix gives a const reference
    for ( auto const & a : sw->agentColl() ) {
        std::cout << a << std::endl;
    }
}
```

Note that the non-const version is explicitly named as "NonNotif" to remind that making changes in the vector through it won't generate any notification even if enclosing type supports notification. Be careful while using it.

Iterators

Vector attributes follow the tacc iterator protocol. Specifically, they have the following properties:

- Iteration order is well-defined and is based on the key type, which is a `U32` index in the case of vectors.
- Collections can be iterated starting from a given key or index. In other words, an iterator can be created from any arbitrary key.
- Iterators are always valid and are convertible to `bool`. If an iterator is `true` then it denotes an element in the collection. The `false` iterator is the end of the collection and should not be dereferenced.

These properties make them suitable for writing `codefs` to process large collections.

The following example shows how to use iterators in the tacc style:

```
#include "Switch.h"

int main() {
    Switch::Ptr sw = Tac::allocate< Switch >();
    sw->agentPush( "Fru" );
    sw->agentPush( "Strata" );

    // Iterate from the start to end
    for ( auto it = sw->agentIteratorConst(); it, ++it ) {
        std::cout << "index: " << it.key() << ", value: " << *it << std::endl;
    }

    // Iterator starting from a stored bookmark
    U32 bookmark = 1;
    if ( auto const it = sw->agentIteratorConst( bookmark ) ) {
        assert( *it == "Strata" );
    }
}
```

Using with the STL

Vector iterators also meet the `LegacyRandomAccessIterator` requirement, meaning they can be used easily with the C++ STL. You typically have to work with a pair of iterators -- one denoting the beginning of the iteration and one denoting one-past-the-end. There are no generated methods to get such pairs, but `Marco::VectorStack` provides conventional `(c)begin` and `(c)end` methods.

The following example demonstrates how to use a vector collection in a way that's compatible with the other containers in the standard library.

```
#include "Switch.h"

int main() {
    Switch::Ptr sw = Tac::allocate< Switch >();
    sw->agentPush( "Fru" );
    sw->agentPush( "Strata" );

    // Iterate from the start to end
    for ( auto const & a : sw->agentColl() ) {
        std::cout << a << std::endl;
    }

    auto const it = sw->agentColl().cbegin();
    auto const end = sw->agentColl().cend();
    if ( auto const found = std::find( it, end, "Strata" ) ) {
        std::cout << *found << std::endl;
    }
    assert( it + 2 == end );
}
```

It's important to be mindful of the operations on vector collections which invalidate iterators. The iterator invalidation rules differ from those for the `std::vector`.

The following table shows which iterators get invalidated by a given operation.

| Operation | Invalidated |
|--------------|-------------------------------------|
| agentPush | only <code>end()</code> |
| agentPop | denoting popped, <code>end()</code> |
| agentPopAll | all |
| agentIs | none |
| agentReserve | none |
| agentShrink | none |

An invalidated `end()` iterator above means it does not represent an "off-the-end" iterator anymore.

Dereferencing the `end()` or an invalidated iterator does not result in undefined behaviour. It throws `Marco::OutOfRangeException`.

Notification

Vector attributes defined in a type with notification support generate notifications when modified. An operation which results in a change notifies with an index where the change occurred:

- `vecAttrPush` notifies with the new element's index, i.e. `size - 1` after push
- `vecAttrPop` notifies with the popped index, i.e. `size` after pop
- `vecAttrs` notifies with the updated index, unless it's an idempotent operation

State machines (`Tac::Constrainer`s) can react to changes within a vector using the same collection reactor syntax as shown in the following example.

```

Switch : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    `hasFactoryFunction;
    agent : vector< Tac::String[]>;
}

SwitchSm : Tac::Type( sw ) : Tac::Constrainer {
    sw : in Switch::PtrConst;
    handleAgent : extern void( index : U32 );
    sw::agent[ index ] => handleAgent( index );
}

```

The index within square brackets binds to the notification index described above. The handler must compare the index against the size of the vector to distinguish between pop and push as shown below.

```

#include "SwitchSm.h"

void
SwitchSm::handleAgent( U32 index ) const {
    U32 const size = sw()->agents();
    if ( index < size ) {
        // case of push or update
        std::cout << "Push or update at " << index << std::endl;
        std::cout << "New element: " << sw()->agent( index ) << std::endl;
    } else {
        // case of pop
        std::cout << "Pop at " << index << std::endl;
    }
}

int main() {
    auto sw = Tac::allocate< Switch >();
    auto swSm = Tac::allocate< SwitchSm >( sw );
    sw->agentPush( "Fru" );
    sw->agentPush( "Sand" );
    sw->agentIs( 0, "Strata" );
    auto popped = sw->agentPop();
    std::cout << "Popped " << popped << std::endl;
}

```

Introspection API

In GenericIf, the `isVector` flag on the `TacAttr` identifies it as a vector.

The following table shows how the generic `TacAttr` API maps to the C++ API which is generated:

| TacAttr method | C++ |
|------------------|----------------|
| genericIfGet | agent(index) |
| genericIfFront | agentFront() |
| genericIfBack | agentBack() |
| genericIfMembers | agents() |

| TacAttr method | C++ |
|-----------------|----------------------|
| genericIfExist | index < agents() |
| genericIfSet | agents(index, val) |
| genericIfAdd | agentPush(val) |
| genericIfRemove | agentPop() |

Python API

The following script shows a few examples of how to interact with a vector within a C++ TAC object:

```
import Tac

sw = Tac.newInstance( "Vector::Switch" )

# push elements
sw.agent.push( "Fru" )
sw.agent.push( "Strata" )
sw.agent.push( "Launcher" )

# pop an element
print( sw.agent.pop() )

# update an element
sw.agent[ 1 ] = "Sand"

# access front and back
first = sw.agent.front()
last = sw.agent.back()
print( f"{first} {last}" )

# get lists
assert sw.agent.keys() == [ 0, 1 ]
assert sw.agent.values() == [ "Fru", "Sand" ]
assert sw.agent.items() == [ ( 0, "Fru" ), ( 1, "Sand" ) ]

# get iterators
it1 = zip( sw.agent.keys(), sw.agent.values() )
it2 = sw.agent.items()
assert list( it1 ) == list( it2 )

# "x in vec" searches for a value, not an index
assert "Sand" in sw.agent
assert 1 not in sw.agent

# delete all the elements
sw.agent.clear()
```

¹ `bool` is returned by value, not reference, since `std::vector< bool >` is specialized for space optimization.

Dynarray

A dynarray is a dynamically-growable collection of items stored contiguously in memory. Along with the random access to its items, it supports deleting any element within the array, but with a big caveat. Instead of deleting the entry and shifting the rest of the elements, it replaces the element with the default value representing a "deleted" slot. This does not work well for operations on array which typically expect contiguously stored elements, and often results in unexpected behaviour.

Dynarray is deprecated. Please use [Vector](#).

Templated Attribute Alias

A Templatized Attribute Alias introduces a new attribute name that is an alias for a number of other attributes, distinguished by some other type. This is useful for when one wants to write a templated utility function that has similar business logic for multiple types. The classical example for this is having two code paths: one for IPv4 and a second for IPv6. Each path needs to talk about a different attribute name, but the rest of the logic is agnostic to the actual types being discussed.

TAC

You can define a template alias for scalar attributes, or hashmaps, ordered maps, or sets as follows:

```
SomeObj : Tac::Type : Tac::Nominal {
    ipSrc : Arnet::IpAddr;
    ip6Src : Arnet::Ip6Addr;

    ipvXSrc : : templateAlias ipSrc< IPv4 > | ip6Src< IPv6 >;
    ipObject : Arnet::IpAddr[ U32 ];
    ip6Object : Arnet::Ip6Addr[ U64 ];

    ipvXObject : : templateAlias ipObject< IPv4 > | ip6Object< IPv6 >;
}
```

This creates a new attribute name `ipvXSrc` which may refer to `ipSrc` or `ip6Src`, depending on if the function call is specialized with the `IPv4` or `IPv6` type. Similarly a new attribute name `ipvXObject` which may refer to `ipObject` or `ip6Object`, depending on if the function call is specialized with the `IPv4` or `IPv6` type. You may specify as many attributes as you wish as long as each one is associated with a distinct template type. You cannot mix scalar and collection types, though you may mix hashmap, ordered maps, and sets within the same templateAlias.

C++

This is used from C++ by using the same accessor and mutator syntax as normal, but specialized with one of the types used in the declaration. In the above example, the accessor `ipvXSrc<>()` is created, as is the mutator `ipvXSrcIs<>()`. The accessor just turns around and calls `ipSrc()` if fully specialized with `IPv4`, or calls `ip6Src()` if fully specialized with `IPv6`. Same logic for the mutator. For scalar attributes which have operators declared ('+=;', for example), this would also generate 'Inc', 'Dec', 'Mul', 'Div', 'Mod', 'And', 'Or', 'Xor', as appropriate. For collections, this also includes the `ipvXObjectHas`, `ipvXObjectIterator`, `ipvXObjectIteratorConst`, `ipvXObjectHas`, `ipvXObjects`, `ipvXObjectDel`, and `ipvXObjectDelAll` methods.

For example:

```
template < typename Domain >
void doSomethingWithObj( SomeObj & obj ) {
    if ( obj.ipvXSrc< Domain >().isZero() ) {
        // Do something if the src IP is "zero"
    }

    for ( auto & e : obj.ipvXObjectsIteratorConst< Domain >() ) {
        // Do something with each element in either
        // ipObjects, or ip6Objects, depending on what
        // type Domain is. Which means that 'e' would
        // work out to be either Arnet::IpAddr or
        // Arnet::Ip6Addr, depending on what type
        // Domain is.
    }
}

void SomeSM::handleIpv4() {
    doSomethingWithObj< IPv4 >( objAttr() );
}

void SomeSM::handleIpv6() {
    doSomethingWithObj< IPv6 >( objAttr() );
}
```

Limitations

This syntax does not get exposed via genericIf, thus these aliases cannot be used from Python.

This section describes properties which can be defined on an attribute.

excludeFromComparison

When an attribute is marked with `excludeFromComparison`, comparisons on instances of the enclosing type behave as though the attribute doesn't exist.

A *comparison* may be equality, ordering, or computing an instance's hash (depending on the kind of the enclosing type).

TAC

In this model, `Record::a` and `Record::b` are considered during comparisons, but `Record::c` isn't.

```
Record : Tac::Type( a, b, c ) : Tac::Ordinal {
    `hasDefaultHash;
    a : U32;
    b : Tac::String;
    c : F64 {
        `excludeFromComparison;
    }
}
```

Example

In this C++ example, both instances of `Record` compare equal and have the same hash even though they have a different value for `Record::c`:

```
#include "Record.h"

int main() {
    // Note that the third attribute differs!
    Record r1( 10, "abc", 3.5 );
    Record r2( 10, "abc", 0.213 );

    assert( r1 == r2 );
    assert( r1.hash() == r2.hash() );
}
```

gnmiJsonString

[gNMI](#) is an open protocol for managing state on network devices. It is typically used to retrieve switch state for monitoring purposes and to modify switch configuration.

The AirStream package converts EOS-native state to/from the format used in gNMI. It does this transparently for all TAC types, without the authors of the types needing to do anything special.

You can add the `gnmiJsonString` property to an attribute to tell AirStream to not do its default conversion for that attribute, but instead to treat it as a scalar string containing a JSON value. You have to provide an implementation of `fooGnmiJsonString()` (where `foo` is the name of the attribute) to convert the attribute to a JSON-encoded string. Note that this is per attribute, not per type. It can be applied to both singleton and collection attributes.

There's no need to provide a conversion the other way around (from a JSON string to the attribute).

Example:

```
CanisLupusFamiliaris.tac
Dog : Tac::Type( name, bornAt ) : Tac::Nominal {
    name : Tac::String;
    bornAt : Tac::Time {
        `gnmiJsonString;
    }
}
Menagerie : Tac::Type() : Tac::PtrInterface {
    dog : Dog[ name ] {
        `gnmiJsonString;
    }
}
```

```
#include "CanisLupusFamiliaris.h"
#include <chrono>
#include <fmt/chrono.h>
#include <Marco/Fmt.h>

Tac::JsonString
Dog::bornAtGnmiJsonString() const {
    std::chrono::sys_days t{ bornAt().asDuration < std::chrono::days { } };
    return FormatStr( "{{ \"day\": {0:%d}, \"month\": {0:%m}, \"year\": {0:%Y} }}",
                      t );
}

Tac::JsonString
Menagerie::dogGnmiJsonString() const {
    Marco::String s = "[ ";
    for ( auto dog : dogIteratorConst() ) {
        if ( s.size() > 2 ) {
            FormatTo( s, ", " );
        }
        FormatTo( s,
                  " {{ \"name\": \"{}\", \"bornAt\": {} }}",
                  dog->name(),
                  dog->bornAtGnmiJsonString().value() );
    }
    FormatTo( s, " ]" );
    return s;
}

int
main() {
    using namespace std::chrono_literals;
    Menagerie::Ptr m = Menagerie::MenagerieIs();
    m->dogIs( Dog{ "misha", Tac::Time{ 1489233600s } } );
    m->dogIs( Dog{ "sasha", Tac::Time{ 1588507200s } } );
    std::cout << m->dogGnmiJsonString() << "\n";
}
```

gnmiScalarArray

gNMI is an open protocol for managing state on network devices. It is typically used to retrieve switch state for monitoring purposes and to modify switch configuration.

The AirStream package converts EOS-native state to/from the format used in gNMI. It does this transparently for all TAC types, by leveraging introspection data generated by the compiler.

In gNMI, there is a difference in how collections of vectors or sets can be sent. In AirStream by default vectors, queues, and sets are sent out as individual key value pairs. The key in these cases is just the internal integer index. However in some cases, this doesn't produce a desired YANG mapping of a leaf list. By default, AirStream sends a vector in gNMI as multiple values

```
/prefix/obj,    updates{    vec[vecKey=0]/vecValue  1,    vec[vecKey=1]/vecValue  2}
```

However, this doesn't produce a desired mapping to YANG because the keys are synthesized. (i.e. there is no explicit way to name a key in a vector)

A leaf list in YANG is just like a python list that doesn't have a key. In order to map to such a leaf list, the `gnmiScalarArray` property can be used. The property dictates that entire collection needs to be sent out whole shot in a single gNMI message without any keys as a simple array of scalars. `/prfix/obj/vec, uint[1, 2, 3, 4]`

Example:

```
ScalarArrayNom : Tac::Type() : Tac::Nominal {
    strVec : vector Tac::String[] {
        `gnmiScalarArray;
    }
    regularSet : set void[ U64 ] {
        `gnmiScalarArray;
    }
    ordSet : ordered set void[ S32 ] {
        `gnmiScalarArray;
    }
}

ScalarArrayEnt : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    strVec : vector Tac::String[] { // sent as atomic array of strings over gNMI
        `gnmiScalarArray;
    }
}
```

gnmiStreamAttr

gNMI is an open protocol for managing state on network devices. It is typically used to retrieve switch state for monitoring purposes and to modify switch configuration.

The AirStream package converts EOS-native state to/from the format used in gNMI. It does this transparently for all TAC types, by leveraging introspection data generated by the compiler.

The compiler implicitly tags all attributes of a type with `gnmiStreamAttr` provided that those types have corresponding storage (i.e. attributes that are not declared as `extern`). In some cases, it is necessary to control if a given attribute is streamed via gNMI or not. For instance, an attribute might be private to EOS and should not get exposed to external customers. In these cases, the `gnmiStreamAttr` property can be explicitly specified to disallow streaming of that attribute. In cases where external attributes or aliases have to be streamed out, the `gnmiStreamAttr` can be used to enable streaming.

Example:

```
EntWithNonGnmiAttrs : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    // gnmiStreamAttr=true implicitly by default
    someState : U32;

    // disable streaming explicitly
    noStreamState : U32 {
        `gnmiStreamAttr = false;
    }

    // enable streaming explicitly because extern attrs
    // are not streamed by default
    streamExternState : extern Tac::String {
        `gnmiStreamAttr = true;
    }

    // enable streaming for an alias
    aliasAttr : : someState {
        `gnmiStreamAttr = true;
    }
}
```

hasBarrierSemantics

This property is used to control the visibility of attribute changes when ScheduledAttrLog is enabled. See [go/scheduledattrlog](#) for more details. When ScheduledAttrLog is not enabled, attribute writes will always be seen in the order the writer makes them and this property has no effect. However, we recommend that if the ordering of updates is important for proper functioning that this property always be used in order to both capture the semantic intent and to insure proper functioning if ScheduledAttrLog is later enabled on this mount or connection.

When ScheduledAttrLog is active, an attribute marked with `hasBarrierSemantics` will always be sent last by the writing agent and Sysdb.

This holds true for all updates across entities made by that writer.

The purpose of this property is to allow a process that writes to multiple attributes in different entities to provide a coherent view to readers. All readers reacting to the marked attribute will find the other attributes in the same state as the writer saw when it wrote the marked attribute.

Readers that need this coherent view should react only to the attribute that has this property set; when a notification is received, all attributes of interest should be read before returning to the event loop.

This property can only be applied to the last attribute in an entity. This is enforced by the compiler. If multiple entities have attributes marked with `hasBarrierSemantics`, all unmarked attributes will be updated before any of the marked ones. Marked attributes will notify in the order they were modified.

Note carefully that if there is heavy traffic on a connection, especially mount traffic, quick multiple updates to an attribute marked with `hasBarrierSemantics` may be combined; all other updates to attributes from that writer will be combined as well. Algorithms using incrementing generation IDs, for example, must handle the case where a generation is skipped. This is done to maintain ordering semantics even under heavy load, following the eventual consistency model. Readers that need to see every update need to take this into account.

TAC

In the following Tac model, `Bar::done` has `hasBarrierSemantics` set.

```
taccExtension << "attrLog";

HasBarrierSemantics : Tac::Namespace {
    Foo : Tac::Type( name ) : Tac::Entity {
        a : U32;
        b : U32;
    }
    Bar : Tac::Type( name ) : Tac::Entity {
        x : U32;
        y : U32;
        done : U32 {
            `hasBarrierSemantics;
            `+=; // so we get Inc function.
        }
    }
}
```

The impact of this can be seen as follows. If the writer of Foo and Bar performs the following:

```
for( int i = 0; i < 100; i++ ) {
    foo->aIs(i);
}
foo->aIs(20);
foo->bIs(2);
bar->xIs(3);
bar->yIs(4);
foo->aIs(1);
bar->doneInc(); // increments done
```

and the reader has a notification routine watching `Bar::done`, the reader will see `foo->a()` equal 1, `foo->b()` equal to 2, `bar->x()` equal to 3 and `bar->y()` equal to 4 when the notification routine runs. Various values of these attributes may be read at other points in time depending on traffic, flowOuts, etc, but when the notification that `Bar::done` has changed is received, the values of all the attributes will be the same as they were in the writer when it called `doneInc()`.

hasDataMember

If an attribute is marked with `hasDataMember=false` then the attribute is not stored in-memory at run-time.

For most attributes `hasDataMember=true` but there are some exceptions:

- Aliases
- Bit aliases
- Constant expressions
- `extern` qualified attributes
- Static attributes

Practical use-cases

Attributes without data members don't have generated prototypes for accessors nor mutators, though these can be added with `hasExternalAccessor` and `hasExternalMutator`. So one can easily create a "virtual" (not in the C++ sense) attribute which does not correspond to anything stored in-memory, but still can return some computed value about the object.

Their use as constructor parameters are also permitted. They can come handy if the constructor should depend on the parameter, but the parameter itself doesn't need to be stored in-memory after the construction.

It is also possible to react on attributes with no data members. When the reaction is important, but the value is not, attributes without data members can save some run-time memory.

Example

The following example shows a type in which only the attributes `count` and `size` are kept in-memory and the accessor for `space` is a user provided function which calculates its its value on each invocation. This saves a couple of bytes of run-time memory at the cost of one additional multiplications on every access to `space`. This is a trade-off between memory usage and speed.

HasDataMember.tac

```
Config : Tac::Type( count, size ) : Tac::PtrInterface {
    count : U8;
    size : U8;

    // The total space is the product of the count and the size.
    space : U16 {
        `hasDataMember = false;
        `hasExternalAccessor;
    }
}
```

```
#include "HasDataMember.h"

U16
Config::space() const {
    return count() * size();
}

int main() {
    Config::Ptr conf = Tac::allocate< Config >( 2, 4 );
    assert( conf->space() == 8 );
}
```

hasExternalAccessor

When an attribute is marked with `hasExternalAccessor`, the compiler will not generate code for that attribute's accessor. A definition of the accessor method must be provided in a `.tin` or `.itin` file.

For example, in this model a `.tin` file must be included with a C++ definition for `RoutingConfig::priority()`:

```
RoutingConfig : Tac::Type() : Tac::PtrInterface {
    weight : F32;
    priority : F32 {
        `hasExternalAccessor;
    }
}
```

RoutingConfig.tac

This snippet shows both the C++ implementation (which would normally exist in `RoutingConfig.tin`) and also an example of using the externally-defined accessor:

```
#include "RoutingConfig.h"

F32
RoutingConfig::priority() const {
    return 999;
}

int main() {
    RoutingConfig::Ptr rc = Tac::allocate< RoutingConfig >();
    rc->weightIs( 2.0 );
    rc->priorityIs( 4.0 );
    assert( rc->priority() == 999 );
}
```

hasExternalMutator

When an attribute is marked with `hasExternalMutator`, the compiler will not generate code for that attribute's mutator. A definition of the mutation method must be provided in a `.tin` or `.itin` file.

For example, in this model a `.tin` file must be included with a C++ definition for `RoutingConfig::priorityIs()`:

```
RoutingConfig : Tac::Type() : Tac::PtrInterface {
    weight : F32;
    priority : F32 {
        `hasExternalMutator;
    }
}
```

RoutingConfig.tac

This snippet shows both the C++ implementation (which would normally exist in `RoutingConfig.tin`) and also an example of using the externally-defined mutator:

```
#include "RoutingConfig.h"

void
RoutingConfig::priorityIs( F32 val ) {
    priority_ = val * weight_;
}

int main() {
    RoutingConfig::Ptr rc = Tac::allocate< RoutingConfig >();
    rc->weightIs( 2.0 );
    rc->priorityIs( 4.0 );
    assert( rc->priority() == 8.0 );
}
```

hasExternalContains

When an external map is marked with `hasExternalContains`, the compiler will generate the declaration of the `Has` function for the collection. For example, an external collection named `foo` will get the generated membership check function named `fooHas`.

Such a method would not be provided by default by the compiler for an extern collection without the property. The user must provide the `Has` function definition in the C++ code.

For example, in this model a `.tin` file is included with a C++ definition for `Config::nameHas()`:

ExternalNameContains.tac

```
Config : Tac::Type() : Tac::PtrInterface {
    `hasFactoryFunction;
    name : extern Tac::String[ U32 ] {
        `hasExternalContains;
    }
}
```

This snippet shows an example of using the externally-defined `Has` in C++ and Python:

```
#include "ExternalNameContains.h"

bool
Config::nameHas( U32 i ) const {
    return i < 10;
}

void
extContainsExample( void ) {
    Config::Ptr c = Tac::allocate< Config >();
    for ( U32 i = 0; i < 10; ++i ) {
        assert( c->nameHas( i ) );
    }
    assert( !c->nameHas( 11 ) );
}

} // namespace HasExternalContains

int main() {
    Config::Ptr c = Tac::allocate< Config >();
    for ( U32 i = 0; i < 10; ++i ) {
        assert( c->nameHas( i ) );
    }
    assert( !c->nameHas( 11 ) );
}
```

HasExternalContains.py

```
Tac.dlopen( 'libTaccBook.so' )
Config = Tac.Type( 'HasExternalContains::Config' )
c = Config()
for i in range( 0, 10 ):
    assert i in c.name

assert not 11 in c.name
```

hasExternalDeleteAll

When an external map or an external set collection is marked with `hasExternalDeleteAll`, the compiler will generate the declaration of the `DelAll` function for the collection. For example, an external collection named `foo` will get the generated clear function named `fooDelAll`.

Such a method would not be provided by default by the compiler for an extern collection without the property. The user must provide the `DelAll` function definition in the C++ code.

For example, in this model a `.tin` file is included with a C++ definition for `Config::nameDelAll()`:

```
Config : Tac::Type() : Tac::PtrInterface {
    `hasFactoryFunction;
    name : extern Tac::String[ U32 ] {
        `hasExternalDeleteAll;
    }
}
```

ExternalNameDeleteAll.tac

This snippet shows both the C++ implementation (which would normally exist in `.tin`), and also an example of using the externally-defined `DelAll` in C++:

```
#include "ExternalNameDeleteAll.h"

bool delAllCalled = false;
void
Config::nameDelAll() {
    delAllCalled = true;
}

int main() {
    Config::Ptr c = Tac::allocate< Config >();
    assert( !delAllCalled );
    c->nameDelAll();
    assert( delAllCalled );
}
```

hasExternalSize

When an external map or an external set collection is marked with `hasExternalSize`, the compiler will generate the declaration of the size function for the collection. For example, an external collection named `foo` will get the generated size function named `foos`.

Such a method would not be provided by default by the compiler for an extern collection without the property. The user must provide the size function definition in the C++ code. When the extern collection is declared with an external iterator, the size function is necessary to access the extern collection from Python.

For example, in this model a `.tin` file is included with a C++ definition for `Config::names()`:

ExternalNameSize.tac

```
Config : Tac::Type() : Tac::PtrInterface {
    `hasFactoryFunction;
    name : extern Tac::String[ U32 ] {
        `iterator : extern;
        `hasExternalSize;
    }
}
```

ExternalNameSize.itin

```
class Config::NameIteratorConst {
public:
    NameIteratorConst() = default;
    NameIteratorConst( U32 ) {}
    Tac::String operator*() const { return Tac::format( "%d", counter ); }
    NameIteratorConst operator++() {
        counter++;
        return *this;
    }
    U32 key() const { return counter + 1; }
    bool isMemberOf( U32 off ) const { return off < 10; }
    operator bool() const { return counter != 10; }
    ~NameIteratorConst() = default;
    U32 counter = 0;
};
```

`} // namespace HasExternalSize`

This snippet shows both the C++ implementation (which would normally exist in `.tin`), the declaration of an external iterator and also an example of using the externally-defined size in C++ and Python:

```
#include "ExternalNameSize.h"

U32
Config::names() const {
    return 10;
}

int main() {
    Config::Ptr c = Tac::allocate< Config >();
    assert( c->names() == 10 );
    for ( auto it = c->nameIteratorConst(); !it; ++it ) {
        assert( it.key() == 1 );
        assert( *it == Tac::format( "%d", *it ) );
    }
}
```

HasExternalSize.py

```
Tac.dlopen( 'libTaccBook.so' )
Config = Tac.Type( 'HasExternalSize::Config' )
c = Config()
assert len( c.name ) == 10
assert list( c.name ) == [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

hasIdempotentMutator

This is a property that is specific to an attribute or collection within a `TacShark::Entity`.

By default, Shark collections or attributes are not idempotent in order to improve performance in the writer process: this can result in notifications received by Shark shadows if the writer repeatedly writes the same value.

The `hasIdempotentMutator` property can be used to make it so that repeated writes of the same value do not generate new notifications

```
// Copyright (c) 2025 Arista Networks, Inc. All rights reserved.  
// Arista Networks, Inc. Confidential and Proprietary.  
  
<<= TacModule( "TacShark::Entity" );  
  
SimpleKey : Tac::Type( value ) : Tac::Ordinal, U32 {  
    `hasDefaultHash;  
}  
  
SimpleValue : Tac::Type( key ) : Tac::Ordinal {  
    `hasDefaultHash;  
    key : SimpleKey;  
    data : U32;  
}  
  
SharkIdempotentEntity : Tac::Type( name ) : TacShark::Entity {  
    `hasFactoryFunction;  
    leaf : SimpleValue[ key ] {  
        `hasIdempotentMutator;  
    }  
}
```

You can find out more about Shark in [aid/7338](#).

hasInlinedAccessor

When `hasInlinedAccessor` or `hasInlinedAccessor=true` is set, calls to the accessor may be inlined by the C++ compiler.

A generated function that may be inlined must have its definition available to the caller. For most attributes, this means that the accessor definition is included in the generated header file.

If an `extern` attribute is marked with `hasInlinedAccessor`, then a C++ definition of the accessor must be provided in an `.itin` file.

The TAC model below shows some use cases of the `hasInlinedAccessor` property:

`Foo.tac`

```
Foo : Tac::Type : Tac::PtrInterface {
    // May be inlined by the C++ compiler
    a : U32 {
        `hasInlinedAccessor;
    }
    // Will not be inlined by the C++ compiler
    b : U32 {
        `hasInlinedAccessor = false;
    }
}
```

hasInlinedMutator

When `hasInlinedMutator` or `hasInlinedMutator=true` is set, calls to the mutator may be inlined by the C++ compiler.

A generated function that may be inlined must have its definition available to the caller. For most attributes, this means that the mutator definition is included in the generated header file.

If an `extern` attribute is marked with `hasInlinedMutator`, then a C++ definition of the mutator must be provided in an `.itin` file.

The TAC model below shows some use cases of the `hasInlinedMutator` property:

`Foo.tac`

```
Foo : Tac::Type : Tac::PtrInterface {
    // May be inlined by the C++ compiler
    a : U32 {
        `hasInlinedMutator;
    }
    // Will not be inlined by the C++ compiler
    b : U32 {
        `hasInlinedMutator = false;
    }
}
```

hasPublicDataMember

By default, an attribute's associated data member in the generated C++ class gets protected access. `hasPublicDataMember` changes the access of the marked attribute's data member to public instead.

TAC

Below, `Foo::y` has public access.

```
Foo : Tac::Type( x, y, z ) : Tac::PtrInterface {
    x : U32;
    y : U32 {
        `hasPublicDataMember;
    }
    z : U32;
}
```

Its data members are generated like so:

```
protected:
    U32 x_;
public:
    U32 y_;
protected:
    U32 z_;
```

Example

In the C++ example, `Foo::y` is modified and accessed directly.

```
auto foo = Foo::FooIs( 1, 2, 3 );
foo->y_ = 7;
assert( foo->y_ == 7 );
```

Restrictions

`hasPublicDataMember` cannot be used inside value types.

Further, the attributes with this property enabled:

- cannot be reacted on
- cannot have constraints defined on them
- cannot have `hasNotifyOnUpdate` enabled.

hasRawPtrAccessor

By default, accessors of a smart-pointer attribute return a copy of the smart-pointer. Using this property, you can change the default behaviour and return a raw pointer instead.

This property is useful when a smart pointer is required to own the object, but the object is commonly passed around as a raw pointer. Returning a raw pointer avoids forcing the caller to take ownership and increment a ref-count.

TAC

The property is false by default, and true if an explicit argument is not provided. The following example shows how to use it:

```
Node : Tac::Type() : Tac::PtrInterface {
    leftChild : Node::Ptr {
        `hasRawPtrAccessor;
    }

    rightChild : Node::PtrConst {
        `hasRawPtrAccessor;
    }

    children : Node::Ptr[ U32 ] {
        `hasRawPtrAccessor;
    }
}
```

Restrictions

hasRawPtrAccessor is only valid for smart-pointer attributes. For all other attributes, its presence is an error.

C++ API

For the **Node** class above, the following accessors are generated:

```
// Accessors for a smart pointer to a non-const object.
Node const * leftChild() const;
Node * leftChild();

// Accessor for a smart pointer to a const object.
Node const * rightChild() const;

// Accessors for a collection of smart pointers to non-const object.
Node const * children( U32 ) const;
Node * children( U32 );
```

isLoggedAttr

For objects that are attrLogged to Sysdb or to other agents, it is possible to specify if some particular attribute needs to be kept local to the agent. In other words, updates to attributes that are marked `isLoggedAttr=false` will not get reflected to the other side of the connection.

Example:

```
taccExtension << "attrLog";  
  
ISLoggedAttr : Tac::Namespace {  
  
    Speed : Tac::Type( speed ) : Tac::Nominal {  
        speed : U32;  
        localNote : Tac::String { // will not get sent to Sysdb  
            `isLoggedAttr = false;  
        }  
    }  
  
    Port : Tac::Type() : Tac::Entity {  
        speed : Speed;  
        duplex : bool;  
        localState : Tac::String { // will not get sent to Sysdb  
            `isLoggedAttr = false;  
        }  
    }  
}
```

isNetByteOrder

When an attribute is marked with `isNetByteOrder`, then the attribute is internally stored in **net byte order** instead of the default host byte order. See also [this concise description](#) about net vs host byte order.

TAC

In the following Tac model, attribute `Foo::anet` has `isNetByteOrder` set.

```
Foo : Tac::Type() : Tac::Nominal {
    // Stored in host byte order
    ahost : U32;
    // Stored in net byte order
    anet : U32 {
        `isNetByteOrder;
    }
}
```

Restrictions

`isNetByteOrder` cannot be used on an attribute inside a reference-counted type. The following model is not accepted:

```
Bar : Tac::Type() : Tac::Entity {
    // Stored in host byte order
    ahost : U32;
    // Stored in net byte order
    anet : U32 {
        `isNetByteOrder;
    }
}
```

The nature of `isNetByteOrder` makes it applicable only under certain conditions:

- Only plain fixed size integer attributes can be marked as `isNetByteOrder`: `U8`, `U16`, `U32`, `U64`, `S8`, `S16`, `S32` and `S64`.
- Non-fixed integer types (e.g. `short` or `size_t`) and of course all non-integer types (e.g. `float` or `Tac::String`) are not allowed.
- Attributes without data storage (e.g. an `extern U32` or an attribute marked with `hasDataMember=false`) cannot be combined with `isNetByteOrder`

Limitations

Currently it is silently ignored by the Golang backend.

Practical use-cases

The generated C++ code completely hides the fact that a specific attribute is stored in net byte order. For example, the usual accessor will return the *host byte order* representation of the attribute, making it indistinguishable from the vanilla equivalent.

Using net byte order storage makes sense only when business logic needs it, mainly when implementing network protocols that demand network byte storage for specific data fields.

Example

For a real use case, here is an excerpt from [RFC2210](#), describing data layout of "RSVP SENDER TSPEC Object":

| 31 | 24 23 | 16 15 | 8 7 | 0 |
|-----|---|-------|-------|---|
| | | | | |
| 1 | 0 (a) reserved | | 7 (b) | |
| 2 | 1 (c) 0 reserved | | 6 (d) | |
| 3 | 127 (e) 0 (f) | | 5 (g) | |
| 4 | Token Bucket Rate [r] (32-bit IEEE floating point number) | | | |
| 5 | Token Bucket Size [b] (32-bit IEEE floating point number) | | | |
| 6 | Peak Data Rate [p] (32-bit IEEE floating point number) | | | |
| 7 | Minimum Policed Unit [m] (32-bit integer) | | | |
| 8 | Maximum Packet Size [M] (32-bit integer) | | | |
| | | | | |
| (a) | - Message format version number (0) | | | |
| (b) | - Overall length (7 words not including header) | | | |
| (c) | - Service header, service number 1 (default/global information) | | | |
| (d) | - Length of service 1 data, 6 words not including header | | | |
| (e) | - Parameter ID, parameter 127 (Token_Bucket_TSpec) | | | |
| (f) | - Parameter 127 flags (none set) | | | |
| (g) | - Parameter 127 length, 5 words not including header | | | |

And here are the corresponding TAC attributes:

```
overallLength : U16 {
    `isNetByteOrder;
}
serviceHeader : U8;
reserved2 : U8;
requestDataLength : U16 {
    `isNetByteOrder;
}

tSpecId : U8;
tSpecFlags : U8;
tSpecLength : U16 {
    `isNetByteOrder;
}
// This should be a F32, however, F32 `isNetByteOrder is not supported. U32 are
// cast to/from F32 after the value is read or before the value is written.
tokenBucketRate : U32 {
    `isNetByteOrder;
}
// See comment above regarding F32/U32.
tokenBucketSize : U32 {
    `isNetByteOrder;
}
// See comment above regarding F32/U32.
peakDataRate : U32 {
    `isNetByteOrder;
}
minPolicedUnit : U32 {
    `isNetByteOrder;
}
maxPacketSize : U32 {
    `isNetByteOrder;
}
```

Notice how the lack of floating point net byte order attributes affects this example, although a workaround was not difficult.

local

Use the `local` property to mark an attribute local to the process in which it was created, as well as private to the object containing the attribute. The attribute is hidden from the outside world, restricting access and disabling data synchronization.

To restrict access, the compiler will generate `protected` accessors and mutators for a `local` attribute. Alternatively, it's also possible to only mark an attribute's assignment operator `local`. In this case, only the attribute's mutator is generated `protected`, effectively creating a read-only attribute.

Marking an attribute `local` is intended to also signal that only the owning process is interested in the attribute. Consequently, synchronization mechanisms (such as AttrLog and gNMI) are also disabled by default for `local` attributes and attributes with a `local` mutator. Note that currently, the `local` keyword affects `Tac::Entity` types and value types differently. Please refer to the [AttrLog](#) and [gNMI](#) sections for more information.

In case a `local` attribute needs to be synchronized across multiple processes, synchronization can be explicitly [re-enabled](#). This can be useful for a producer-consumer setup using an attribute where only the mutator is `local`. Data can then be produced by the owning process and consumed by external mounting processes.

TAC

The `local` property is not set by default. The following example shows how to use it:

```
Foo : Tac::Type() : Tac::Nominal {
    a : U32;
    b : local U32;
    c : U32 {
        `= : local;
    }
}
```

In the setup above,

- both the accessor and mutator are publicly visible (`a`),
- both the accessor and mutator are protected (`b`),
- the accessor is public and the mutator is protected (`c`).

Side-effects

AttrLog

In `Tac::Entity` types, `local` attributes and attributes with a `local` mutator have AttrLog synchronization disabled by default. This can be overwritten by explicitly marking the attribute with `isLoggedAttr=true`, e.g.,

```
c : U32 {
    `= : local;
    `isLoggedAttr = true;
}
```

Note that currently, this behavior is specific to attributes in `Tac::Entity` types. Marking an attribute or its mutator `local` in a `value type` will not disable its AttrLog synchronization.

gNMI

`local` attributes and attributes with a `local` mutator have gNMI streaming disabled by default. If only the mutator is marked with `local`, streaming can be re-enabled for the attribute by also marking it with `gnmiStreamAttr=true`, e.g.,

```
c : U32 {
    `= : local;
    `gnmiStreamAttr = true;
}
```

Note that unlike AttrLog synchronization, this behavior affects the attributes of both `value types` and `Tac::Entity` types.

GenericIf

For a `local` attribute or an attribute with a `local` mutator, the `readable` and `writable` flags reported by `GenericIf` respect the `protected` visibility of the underlying C++ type. Therefore, an attribute marked with `local` shows up as non-readable and non-writable in `GenericIf`, while an attribute with a `local` mutator is reported as readable but non-writable. Consequently, all features that rely on the procedural enumeration and modification of writable attributes will exclude `local` attributes and attributes with a `local` mutator, with features needing to read the state of an attribute also excluding fully `local` attributes.

Some commonly used features impacted by this are listed below.

Tac-Python bindings

A `local` attribute cannot be read or modified using the Python member access syntax, while an attribute with a `local` mutator can be read but not be modified. The following example demonstrates these in practice:

```

foo = Tac.newInstance( "Foo" )

# These both work; 'a' is a non-local attribute
foo.a = 10
print( foo.a )

# These both fail; 'b' is marked with `local`, so these both raise
# NotImplementedError with the following exception messages:
# 'Foo.b: mutator: Tac::OpNotSupportedException("mutator not supported for
# Foo::b")';
# 'Foo.b: Tac::OpNotSupportedException("accessor not supported for Foo::b")'
foo.b = 20
print( foo.b )

# The read works but the assignment fails; only the mutator of 'c' is marked
# with `local`, so the assignment will raise NotImplementedError with
# 'Foo.c: mutator: Tac::OpNotSupportedException("mutator not supported for
# Foo::c")'
foo.c = 10
print( foo.c )

```

This restriction can impact production Python code as well as Python tests.

EntityCopy

Configuration sessions (see [AID2009](#)) utilize EntityCopy to synchronize configuration entities. Consequently, configuration entity attributes marked with `local` won't be copied.

Acons

Acons also uses GenericIf to implement certain commands. Therefore, an attribute marked with `local` will not show up in the output of `ls`:

```
$ ls
$ /ar/Sysdb/foo is <entity('/ar/Sysdb/foo') of type Foo (non-const)>
a.          c.      fullName. name.
```

unless using the `--all` switch:

```
$ ls --all
$ /ar/Sysdb/foo is <entity('/ar/Sysdb/foo') of type Foo (non-const)>
a.          b.          c.      entity/      fullName.
name.        parent.    parentAttrName.
```

Additionally, trying to read (`local` accessor) or write (`local` mutator) the value of an attribute via the console will also raise the corresponding Python exceptions outlined above:

```

$ __.a = 10 # works
$ __.a      # works
$ __.b     = 20   # raises  NotImplementedError:  Foo.b:  mutator:
Tac::OpNotSupportedException("mutator not supported for Foo::b")
$ __.b      # raises  NotImplementedError:  Foo.b:
Tac::OpNotSupportedException("accessor not supported for Foo::b")
$ __.c     = 30   # raises  NotImplementedError:  Foo.c:  mutator:
Tac::OpNotSupportedException("mutator not supported for Foo::c")
$ __.c      # works

```

Finally, the output of `ls --long --all` will also display the value of a `local` attribute as `<unreadable>`:

```
$ ls --long --all
$ /ar/Sysdb/foo is <entity('/ar/Sysdb/foo') of type Foo (non-const)>
  a          : 10
  b          : <unreadable>
  c          : 0
  entity     : <non-iterable collection>
  fullName   : /ar/Sysdb/foo
  name       : foo
  parent     : <entity('/ar/Sysdb') of type Tac::Dir (non-
const)>
  parentAttrName : entityPtr
```

publishKeyWithNotification

`publishKeyWithNotification` is a property used to allow a notifying externally-keyed map collection to be used as the intermediate path of a `when` statement. Without it such cases are reported as an error by the compiler.

Under the cover it creates a class `FieldExtKeyCollection`, where `field` is the name of the publishing collection that is wrapping the key and value. This allows the collection to be made internally-keyed while the compiler generates the externally-keyed APIs that are called by the existing users of those collections.

Because the collection is now internally-keyed, the `when` statement generated code can now recover the key associated to the value type when reacting to more deeply nested attributes.

PublishKey.tac

```
Bar : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;

    var : U32;
}

Foo : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;

    bar : Bar::Ptr[ U32 ] {
        `publishKeyWithNotification;
    }
}

Sm : Tac::Type() : Tac::Constrainer {
    foo : in Foo::Ptr;
    handleVar : void( i : U32 ) {}
    when foo::bar[ i ]::var => handleVar( i );
}
```

Compound assignment operators (+=, -=, etc.)

tacc implements several compound assignment operators that allow for efficient operations on basic data types.

TAC

Here is an example of a model that makes use of several of these operators:

```
Record : Tac::Type() : Tac::PtrInterface {
    field1 : U32 {
        `+=;
    }
    field2 : U16 {
        `-=;
    }
    field3 : bool {
        `|=;
        `&=;
    }
}
```

Record.tac

The following table shows the syntax for supported operators, as well as the data types that tacc will automatically generate a body for:

| TAC Operator | Impl generated for ints? | Impl generated for floats? | Impl generated for bools? | Impl generated for pointers? |
|--------------|--------------------------|----------------------------|---------------------------|------------------------------|
| `+= | Yes | Yes | | Yes |
| `-= | Yes | Yes | | |
| `*= | Yes | Yes | | |
| `/= | Yes | Yes | | |
| `%=> | Yes | | | |
| ` = | Yes | | Yes | |
| `&= | Yes | | Yes | |
| `^= | Yes | | Yes | |

If one desires to create a compound assignment operator that isn't automatically generated, or provide some custom implementation (eg, to perform some custom error checking), it is possible with the `extern` keyword:

```
RecordCustom : Tac::Type() : Tac::PtrInterface {
    field1 : bool {
        `+= : extern;
    }
    field2 : U16 {
        `-= : extern;
    }
}
```

Record2.tac

This will generate the C++ function signature, but will allow you to create your own custom implementation (and will error if one is not provided).

C++

The following table maps the TAC operators to the C++ function signatures that are generated:

| TAC Operator | C++ Function Signature |
|--------------|------------------------|
| `+= | fooInc(T foo) |
| `-= | fooDec(T foo) |
| `*= | fooMul(T foo) |
| `/= | fooDiv(T foo) |
| `%=` | fooMod(T foo) |
| ` =` | fooOr(T foo) |
| `&=` | fooAnd(T foo) |
| `^=` | fooXor(T foo) |

Note that the first column represents the syntax in the `.tac` file, while the second gives the C++ generated signature where `foo` is the attribute name and `T` is the attribute type.

Looking at the first example, the `+=` operator for `field1` generates a function with the following signature:

```
U32
Record::field1Inc(U32 _field1);
```

The `field1Inc` function takes in one parameter, and increments `field1` by that amount. So, in the following code, if `condition` is true, `field1` will be incremented by 2.

```
if ( condition ) {
    field1Inc( 2 );
}
```

When statements

Agents in EOS perform their tasks by reacting to changes in the state of the switch. `when` statements allow agents to subscribe to state objects, and invoke the business logic if anything changes in the subscribed objects. This is conceptually similar to [the Observer design pattern](#).

Reacting to singleton attributes

The following example shows a simple example of a `when` statement. `InterfaceConfig` captures the state of a switch interface: whether it's enabled or not. The LED controller agent on a switch would run `LedSm`, which subscribes to `InterfaceConfig` and reacts to changes in `enabled`:

▼ WhenSimple.tac

```
// Identifier of a switch interface
IntfId : Tac::Type( value ) : Tac::Ordinal, U64 {
    `hasDefaultHash;
}

// State populated by the ConfigAgent based on CLI
InterfaceConfig : Tac::Type( intfId ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    intfId : IntfId;
    enabled : bool;
}

// State populated by the LED controller agent
LedStatus : Tac::Type() : Tac::PtrInterface {
    turnedOn : set void[ IntfId ];
}

// State machine run by the LED controller agent
LedSm : Tac::Type( intfConfig, ledStatus ) : Tac::Constrainer {
    intfConfig : in InterfaceConfig::PtrConst;
    ledStatus : LedStatus::Ptr;

    handleEnabled : extern invasive void();
    when intfConfig::enabled => handleEnabled();
}
```

The first part of the syntax is the `when` keyword followed by a "path". A path is a sequence of ""path elements" separated by the operator `::`. A path always starts with an attribute defined in the enclosing type's scope, called the "root" of the path, and refers to an attribute nested one or more levels deeper within the root. The final attribute is called the "leaf". In the second part of the syntax, an "action" is declared after the arrow operator `=>`. The action must be a function-call expression.

Like reactors, `when` statements require the root attribute to be marked as `in`.

The previous `when` statement declares that whenever the leaf, `enabled`, changes, call `handleEnabled`. More concretely, `handleEnabled` is called during the following events:

- `LedSm` is instantiated.
- `intfConfig` is updated. This can happen when `intfConfig` is mutable and the pointer is replaced via a call to `intfConfigIs`.
- `enabled` is updated.

The agent can take appropriate action in the user-implemented function `handleEnabled`. In the following example, it updates the collection of interfaces whose LED is turned ON:

▼ WhenSimple.cpp

```
#include "WhenSimple.h"
void
LedSm::handleEnabled() {
    const auto id = intfConfig()->intfId();
    if ( !intfConfig()->enabled() ) {
        fmt::print( "Turning off LED on interface {}\\n", id.value() );
        ledStatus()->turnedOnDel( id );
    } else {
        fmt::print( "Turning on LED on interface {}\\n", id.value() );
        ledStatus()->turnedOnIs( id );
    }
}

int main() {
    // LED agent would normally mount config and status objects from remote.
    const IntfId intfId{ 42 };
    const auto config = Tac::allocate< InterfaceConfig >( intfId );
    const auto status = Tac::allocate< LedStatus >();
    {
        // Once mount is complete, the SM will be started.
        fmt::print( "Starting the SM...\\n" );
        const auto ledSm = Tac::allocate< LedSm >( config, status );

        // ConfigAgent enables the interface, and the update reaches LED agent.
        fmt::print( "Interface enabled...\\n" );
        assert( !status->turnedOnHas( intfId ) );
        config->enabledIs( true );
        assert( status->turnedOnHas( intfId ) );

        // ConfigAgent disables the interface, and the update reaches LED agent.
        fmt::print( "Interface disabled...\\n" );
        config->enabledIs( false );
        assert( !status->turnedOnHas( intfId ) );

        fmt::print( "Shutting down the SM...\\n" );
    }
}
```

Reacting to collection attributes

The leaf of a path in a `when` statement is not restricted to a singleton attribute. It can also be a collection element as shown in the example below.

This time, `LedSm` subscribes to the `LinkStatus` populated by a Transceiver agent, and reacts to changes in the status of any interface:

▼ WhenCollection.tac

```
// Identifier of a switch interface
IntfId : Tac::Type( value ) : Tac::Ordinal, U64 {
    `hasDefaultHash;
}

// State populated by the Transceiver agent
LinkStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    Status : Tac::Enum {
        inProgress;
        connected;
        disconnected;
    }
    status : Status[ IntfId ];
}

// State populated by the LED controller agent
LedStatus : Tac::Type() : Tac::PtrInterface {
    Color : Tac::Enum {
        red;
        green;
        yellow;
        off;
    }
    color : Color[ IntfId ];
}

// State machine run by the LED controller agent
LedSm : Tac::Type( linkStatus, ledStatus ) : Tac::Constrainer {
    linkStatus : in LinkStatus::PtrConst;
    ledStatus : LedStatus::Ptr;

    handleLinkStatus : extern invasive void(
        IntfId : IntfId, status : LinkStatus::Status::Optional );
    when linkStatus::status[ intfId ] as currStatus =>
        handleLinkStatus( intfId, currStatus );
}
```

The previous `when` statement declares that, whenever an entry in `status` collection is modified, `handleLinkStatus` is called. More concretely, `handleLinkStatus` is called during the following events:

- For each `status` entry in `linkStatus` when `LedSm` is instantiated.
- For each `status` entry in old `linkStatus`, and then for each entry in new `linkStatus` whose key is not in the old collection, if `linkStatus` is updated. This can happen when

`linkStatus` is mutable and the pointer is replaced via a call to `linkStatusIs`.

- An entry in `status` is inserted or deleted.

Key binding

If a path element refers to a collection attribute, the user must provide an identifier within square brackets. In the previous example, `intfId` in `status[intfId]` is such an identifier.

This identifier refers to the index of the collection entry which got modified. The action can use the identifier while calling a function. The type of the identifier is inferred to be the same as the collection index type. The scope of the identifier is limited to the `when` statement which defines it.

When an action doesn't require the collection index, an identifier for the index can be omitted with `_`. For example,

▼ *WhenCollectionNoIndex.tac*

```

// Identifier of a switch interface
IntfId : Tac::Type( value ) : Tac::Ordinal, U64 {
    `hasDefaultHash;
}

// State populated by the Transceiver agent
LinkStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    Status : Tac::Enum {
        inProgress;
        connected;
        disconnected;
    }
    StatusIntf : Tac::Type( intfId ) : Tac::Nominal {
        intfId : IntfId;
        status : Status;
    }
    status : StatusIntf[ intfId ];
}

// State populated by the LED controller agent
LedStatus : Tac::Type() : Tac::PtrInterface {
    Color : Tac::Enum {
        red;
        green;
        yellow;
        off;
    }
    color : Color[ IntfId ];
}

// State machine run by the LED controller agent
LedSm : Tac::Type( linkStatus, ledStatus ) : Tac::Constrainer {
    linkStatus : in LinkStatus::PtrConst;
    ledStatus : LedStatus::Ptr;

    handleLinkStatus : extern invasive void(
        status : LinkStatus::StatusIntf::Optional );
        when linkStatus::status[ _ ] as currStatus => handleLinkStatus( currStatus
    );
}

```

▼ WhenCollectionNoIndex.cpp

```

#include "WhenCollectionNoIndex.h"
void
LedSm::handleLinkStatus(
    const Tac::Optional< LinkStatus::StatusIntf > & currStatus ) {
    auto newColor = LedStatus::off();
    if ( currStatus ) {
        switch ( currStatus->status() ) {
            case LinkStatus::inProgress():
                newColor = LedStatus::yellow();
                break;
            case LinkStatus::connected():
                newColor = LedStatus::green();
                break;
            case LinkStatus::disconnected():
                newColor = LedStatus::red();
                break;
        }
    }
    fmt::print( "LED is turned {}\n", valueToStrep( newColor ) );
    ledStatus()->colorIs( currStatus->intfId(), newColor );
}

int main() {
    // LED agent would normally mount these objects from remote.
    const auto linkStatus = Tac::allocate< LinkStatus >();
    const auto ledStatus = Tac::allocate< LedStatus >();
{
    // Once mount is complete, the SM will be started.
    fmt::print( "Starting the SM...\n" );
    const auto ledSm = Tac::allocate< LedSm >( linkStatus, ledStatus );

    // Transceiver agent updates the link status of an interface,
    // Updates reach LED agent.
    const IntfId intfId{ 42 };
    fmt::print( "Interface link in-progress...\n" );
    LinkStatus::StatusIntf statusIntf{ intfId };
    statusIntf.statusIs( LinkStatus::inProgress() );
    linkStatus->statusIs( statusIntf );
    assert( ledStatus->color( intfId ) == LedStatus::yellow() );

    fmt::print( "Interface link connected...\n" );
    statusIntf.statusIs( LinkStatus::connected() );
    linkStatus->statusIs( statusIntf );
    assert( ledStatus->color( intfId ) == LedStatus::green() );

    fmt::print( "Interface link disconnected...\n" );
    statusIntf.statusIs( LinkStatus::disconnected() );
    linkStatus->statusIs( statusIntf );
    assert( ledStatus->color( intfId ) == LedStatus::red() );

    fmt::print( "Shutting down the SM...\n" );
}
}

```

New value binding

While implementing the action, it's generally required to know the new value of the leaf. `[when]` statements allow passing this new value to the action.

For this syntax, the user provides an identifier after the keyword `as`, before the `=>` operator. This identifier refers to the new value of the leaf. If the leaf is a collection attribute, it refers to the value within the modified entry.

The type of the identifier is assigned as below:

- If the leaf attribute's type (the collection's value type if the leaf is a non-set collection attribute) is either a raw or a smart pointer, the result type is the same kind of pointer. If the parent of the leaf is a pointer to const, its constness is carried forward in the result.
- If the leaf attribute is a set collection, the result type is `bool` indicating insertion or deletion.
- If the leaf attribute is an instantiating attribute (singleton or collection), its type is treated as a smart pointer. The result type is determined using the same rule as above.
- For the remaining types -- Nominal, Ordinal, Enum, DenseEnumBoolSet, Union, and built-in types, the result type is an `Optional` wrapping the attribute's type (value type for a collection).

In the event when a leaf is deleted, including when one of its ancestors is deleted, there's no new value to process. In this case, the `Optional` is set to none, the `bool` is set to false, or the pointer is set to null, while invoking the action.

The scope of the new value identifier is limited to the `when` statement defining it.

The action in `LedSm` can use this new value to set the LED's color as shown below:

▼ *WhenCollection.cpp*

```

#include "WhenCollection.h"
void
LedSm::handleLinkStatus( IntfId intfId,
                        Tac::Optional< LinkStatus::Status > currStatus ) {
    auto newColor = LedStatus::off();
    if ( currStatus ) {
        switch ( *currStatus ) {
            case LinkStatus::inProgress():
                newColor = LedStatus::yellow();
                break;
            case LinkStatus::connected():
                newColor = LedStatus::green();
                break;
            case LinkStatus::disconnected():
                newColor = LedStatus::red();
                break;
        }
    }
    fmt::print( "LED is turned {}\n", valueToStrep( newColor ) );
    ledStatus()->colorIs( intfId, newColor );
}

int main() {
    // LED agent would normally mount these objects from remote.
    const auto linkStatus = Tac::allocate< LinkStatus >();
    const auto ledStatus = Tac::allocate< LedStatus >();
    {
        // Once mount is complete, the SM will be started.
        fmt::print( "Starting the SM...\n" );
        const auto ledSm = Tac::allocate< LedSm >( linkStatus, ledStatus );

        // Transceiver agent updates the link status of an interface,
        // Updates reach LED agent.
        const IntfId intfId{ 42 };
        fmt::print( "Interface link in-progress...\n" );
        linkStatus->statusIs( intfId, LinkStatus::inProgress() );
        assert( ledStatus->color( intfId ) == LedStatus::yellow() );

        fmt::print( "Interface link connected...\n" );
        linkStatus->statusIs( intfId, LinkStatus::connected() );
        assert( ledStatus->color( intfId ) == LedStatus::green() );

        fmt::print( "Interface link disconnected...\n" );
        linkStatus->statusIs( intfId, LinkStatus::disconnected() );
        assert( ledStatus->color( intfId ) == LedStatus::red() );

        fmt::print( "Shutting down the SM...\n" );
    }
}

```

It's recommended to use the new value binding for collection element leaves. It saves an extra collection lookup to fetch the new value in the action.

Old value binding

In some cases, the action requires the old value of the leaf to run its logic. The `when` statements allow passing the old value as well to the action.

For this syntax, a pair of identifiers is passed to the `as` operator, where the first name refers to the old value and the second name to the new value. If the leaf is a non-set collection attribute, the names refer to old and new values within the modified entry.

The old value cannot be bound if the leaf attribute is a set collection. A set collection leaf can only publish a single boolean value indicating insertion or deletion.

The typing and scoping of the old value is the same as the new value described in the previous section.

Updating the previous example to use both old and new values would look like this:

```
handleLinkStatus : extern void(  
    intfId : IntfId,  
    prev : LinkStatus::Status::Optional,  
    curr : LinkStatus::Status::Optional );  
when LinkStatus::status[ intfId ] as ( prev, curr ) =>  
    handleLinkStatus( intfId, prev, curr );
```

Nested paths

The leaf of a path in a `when` statement can be two or more levels deep.

Extending the previous example, the following example adds a level of hierarchy in the state, by pushing `LinkStatus` one level down under `LineCardStatus`. The `LedSm` subscribes to the link status of every linecard, and reacts to changes in the status of any interface of a given linecard.

▼ WhenNested.tac

```

// Identifier of a switch interface
IntfId : Tac::Type( value ) : Tac::Ordinal, U64 {
    `hasDefaultHash;
}

// State populated by the Transceiver agent
LinkStatus : Tac::Type( lineCard ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    Status : Tac::Enum {
        inProgress;
        connected;
        disconnected;
    }
    lineCard : U8;
    status : Status[ IntfId ];
}

// Config
LinkConfig : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    enabled : bool[ U64 ];
}

LineCardStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    linkStatus : LinkStatus[ lineCard ];
}

LineCardConfig : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    config : LinkConfig::Ptr[ U8 ] {
        `publishKeyWithNotification;
    }
}
// State machine run by the LED controller agent
LedSm : Tac::Type( lineCardStatus ) : Tac::Constrainer {
    lineCardStatus : in LineCardStatus::PtrConst;

    handleLinkStatus : extern invasive void(
        lineCard : U8,
        intfId : IntfId,
        prev : LinkStatus::Status::Optional,
        curr : LinkStatus::Status::Optional );

    when lineCardStatus::linkStatus[ lineCard ]::status[ intfId ]
        as ( prevStatus, currStatus ) =>
        handleLinkStatus( lineCard, intfId, prevStatus, currStatus );

    lineCardConfig : in LineCardConfig::PtrConst;

    handleLinkConfig : void( i : U8, c : U64, enabled : bool::Optional ) {}
    when lineCardConfig::config[ i ]::enabled[ c ] as enabled =>
        handleLinkConfig( i, c, enabled );
}

```

The `when` statement above passes four arguments to the action upon an event:

1. `lineCard` identifying which linecard's link status changed
2. `intfId` identifying which interface of the linecard changed

3. `prevStatus` capturing the leaf `Status` value before the event. The `Optional` is not set if the event is insertion of a new entry in `status`, i.e. a new interface is added in a given linecard. This includes insertion event for each interface for a new entry in `linkStatus`, i.e. if a new linecard with its `LinkStatus` is inserted.
4. `currStatus` capturing the leaf `Status` value after the event. The `Optional` is not set if the event is deletion of an entry within `status`, i.e. an interface is removed from a given linecard. This includes deletion event for each interface for a deleted entry in `linkStatus`, i.e. if a linecard with its `LinkStatus` is removed.

For the previous `when` statement, `handleLinkStatus` may be invoked during the following events, for relevant entries in `status`:

- `LedSm` is instantiated
- `lineCardStatus` is updated (in case it's mutable), for each nested `status` entry.
- An entry in `linkStatus` is inserted or deleted, for each entry nested in the associated `status` collection (If `status` is empty, nothing is invoked).
- An entry in `status` is inserted or deleted

The following implementation of the action shows some of these events:

▼ *WhenNested.cpp*

```

#include "WhenNested.h"
void
LedSm::handleLinkStatus( U8 lineCard,
                         IntfId intfId,
                         Tac::Optional< LinkStatus::Status > prevStatus,
                         Tac::Optional< LinkStatus::Status > currStatus ) {
    const auto prev = prevStatus ? valueToStrep( *prevStatus ) : "None";
    const auto curr = currStatus ? valueToStrep( *currStatus ) : "None";

    fmt::print( "For linecard {}, interface {}:\n", lineCard, intfId.value() );
    fmt::print( "Status changed from {} to {}.\n", prev, curr );
}

int main() {
    // LED agent would normally mount status object from remote.
    const auto status = Tac::allocate< LineCardStatus >();

    // Transceiver agent adds link status for linecard 8.
    const auto lc8Links = status->linkStatusIs( 8 );
    lc8Links->statusIs( IntfId{ 1 }, LinkStatus::inProgress() );
    lc8Links->statusIs( IntfId{ 2 }, LinkStatus::connected() );

    {
        // Mount is complete, LED agent starts the SM.
        fmt::print( "Starting the SM...\n" );
        const auto ledSm = Tac::allocate< LedSm >( status );

        // Transceiver agent updates interfaces on linecard 8.
        // Updates reaches the LED agent.
        fmt::print( "\nAdding interface 0...\n" );
        lc8Links->statusIs( IntfId{ 0 }, LinkStatus::connected() );
        fmt::print( "Updating interface 1...\n" );
        lc8Links->statusIs( IntfId{ 1 }, LinkStatus::connected() );
        fmt::print( "Removing interface 2...\n" );
        lc8Links->statusDel( IntfId{ 2 } );

        // A new linecard 9 is inserted and linecard 8 is removed.
        // Updates reaches the LED agent.
        fmt::print( "\nRemoving linecard 8...\n" );
        status->linkStatusDel( 8 );
        // Note, this does *not* trigger the handleLinkStatus function, as the
new
        // line card has no interface status objects yet
        fmt::print( "Inserting linecard 9...\n" );
        const auto lc9Links = status->linkStatusIs( 9 );
        // And now, when we add a status, we do get handleLinkStatus called
        fmt::print( "Adding status for interface 1 under linecard 9...\n" );
        lc9Links->statusIs( IntfId{ 1 }, LinkStatus::inProgress() );

        fmt::print( "\nShutting down the SM...\n" );
    }
}

```

More than 3-level nested paths

More than 3 levels of nesting in a path is supported only if any of the following condition is held:

- None of the intermediate elements or the root element is a collection. The leaf can be the collection,

- A collection is present as an intermediate or the root element, but its index is not used by the action,
- An intermediate or the root collection element, whose index is used by the action, meets all the following requirements:
 - The parent element must be an instantiating attribute,
 - The parent instantiating attribute must be defined within an Entity type,
 - The collection element's value type must be an Entity.

▼ WhenDeepNesting.tac

```

MapEntry : extensible Tac::Type( seqno ) : Tac::Entity {
    `hasFactoryFunction;

    // sequence number
    seqno : U32;

    versionId : U32;
}

Map : extensible Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    `hasFactoryFunction;

    // mapEntry is an instantiating collection defined in an entity
    // it has a parent pointer.
    mapEntry : ordered MapEntry[ seqno ];
}

Config : extensible Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    `hasFactoryFunction;

    // routeMap is an instantiating collection defined in an entity
    // it has a parent pointer.
    routeMap : ordered Map[ name ];
}

ToExternalSm : Tac::Type( routeMapConfig ) : Tac::Constrainer {
    `hasFactoryFunction;

    routeMapConfig : in Config::PtrConst;
    currentVersionIdShadow : U32::Optional;

    handleMapEntry : extern invasive void(
        mapName : Tac::String, seqno : U32, currentVersionId : U32::Optional );
    when routeMapConfig::routeMap[ name ]::mapEntry[ seqno ]::versionId as curr
    =>
        handleMapEntry( name, seqno, curr );
}

```

Modifiers to modify default behavior

If the default behavior of `when` statements is not desirable, some aspects of it can be modified using modifiers. A modifier can be declared using a new keyword `with` after the value bindings, before the `=>` operator.

skipInitScan

By default, each `when` statement's action is invoked during the enclosing state machine's initialization. The `skipInitScan` modifier allows skipping the action of a given `when` statement during initialization.

▼ WhenSkipInitScan.tac

```
LineCardStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    linkStatus : Tac::String[ Tac::String ];
}

LedSm : Tac::Type( lineCardStatus ) : Tac::Constrainer {
    lineCardStatus : in LineCardStatus::Ptr;

    handleLinkStatus : extern invasive void(
        intf : Tac::String, status : Tac::String::Optional );
    when lineCardStatus::linkStatus[ intf ] as currStatus with skipInitScan =>
        handleLinkStatus( intf, currStatus );
}
```

The previous `when` statement declares that, `handleLinkStatus` should not be invoked when `LedSm` is instantiated.

▼ WhenSkipInitScan.cpp

```
#include "WhenSkipInitScan.h"
void
LedSm::handleLinkStatus( Tac::String const & intf,
                        Tac::Optional< Tac::String > const & currStatus ) {
    if ( currStatus ) {
        fmt::print( "Interface {} is updated to {}\n", intf, *currStatus );
    } else {
        fmt::print( "Interface {} is removed\n", intf );
    }
}

int main() {
    const auto lineCardStatus = Tac::allocate< LineCardStatus >();
    {
        fmt::print( "Populating the status...\n" );
        lineCardStatus->linkStatusIs( "eth0", "active" );
        lineCardStatus->linkStatusIs( "eth1", "inactive" );

        fmt::print( "Starting the SM...\n" );
        const auto ledSm = Tac::allocate< LedSm >( lineCardStatus );

        fmt::print( "Updating the status...\n" );
        lineCardStatus->linkStatusIs( "eth0", "inactive" );

        fmt::print( "Shutting down the SM...\n" );
    }
}
```

A good practice is to use the default behavior, and handle the state present when the state machine is initialized.

There are some exceptions where compiler-generated action calls are not desirable, such as:

- The initial state is handled in an ad hoc manner, and invoking the action on the initial state would be wasteful.
- The action function calls a virtual function, but the derived class function cannot be called from the base class constructor.

For such exceptional cases, `skipInitScan` can be useful.

skipOnEnclosingCreate

By default, each `when` statement's action is invoked during the enclosing attribute's creation. The `skipOnEnclosingCreate` modifier suppresses such reactor calls.

▼ WhenSkipOnEnclosingCreate.tac

```
LinkStatus : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    status : Tac::String;
}

LineCardStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    linkStatus : LinkStatus[ name ];
}

LedSm : Tac::Type( lineCardStatus ) : Tac::Constrainer {
    lineCardStatus : in LineCardStatus::Ptr;

    handleLinkStatus : extern invasive void(
        intf : Tac::String, status : Tac::String::Optional );

    when lineCardStatus::linkStatus[ intf ]::status as currStatus
        with skipOnEnclosingCreate => handleLinkStatus( intf, currStatus );
}
```

The `when` statement above declares that `handleLinkStatus` should not be invoked when a `linkStatus` is created.

▼ WhenSkipOnEnclosingCreate.cpp

```

#include "WhenSkipOnEnclosingCreate.h"
void
LedSm::handleLinkStatus( Tac::String const & intf,
                         Tac::Optional< Tac::String > const & currStatus ) {
    if ( currStatus ) {
        fmt::print( "Interface {} is updated to {}\n", intf, *currStatus );
    } else {
        fmt::print( "Interface {} is removed\n", intf );
    }
}

int main() {
    const auto lineCardStatus = Tac::allocate< LineCardStatus >();
    {
        fmt::print( "Populating the status...\n" );
        auto link = lineCardStatus->linkStatusIs( "eth0" );

        fmt::print( "Updating the status...\n" );
        link->statusIs( "active" );

        fmt::print( "Starting the SM...\n" );
        const auto ledSm = Tac::allocate< LedSm >( lineCardStatus );

        fmt::print( "Updating the status...\n" );
        link->statusIs( "inactive" );

        fmt::print( "Removing the status...\n" );
        lineCardStatus->linkStatusDel( "eth0" );

        fmt::print( "Shutting down the SM...\n" );
    }
}

```

The default behavior is usually desirable. But `skipOnEnclosingCreate` may be useful when the user wishes to implement asynchronous creation handling, or to handle changes in `linkStatus` via a different `when` statement.

skipOnEnclosingDelete

By default, each `when` statement's action is invoked during the enclosing attribute's deletion. The `skipOnEnclosingDelete` modifier suppresses such reactor calls.

▼ *WhenSkipOnEnclosingDelete.tac*

```

LinkStatus : Tac::Type( name ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    name : Tac::String;
    status : Tac::String;
}

LineCardStatus : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    linkStatus : LinkStatus[ name ];
}

LedSm : Tac::Type( lineCardStatus ) : Tac::Constrainer {
    lineCardStatus : in LineCardStatus::Ptr;

    handleLinkStatus : extern invasive void(
        intf : Tac::String, status : Tac::String::Optional );

    when lineCardStatus::linkStatus[ intf ]::status as currStatus
        with skipOnEnclosingDelete => handleLinkStatus( intf, currStatus );
}

```

The `when` statement above declares that `handleLinkStatus` should not be invoked when a `linkStatus` is deleted.

▼ WhenSkipOnEnclosingDelete.cpp

```

#include "WhenSkipOnEnclosingDelete.h"
void
LedSm::handleLinkStatus( Tac::String const & intf,
                        Tac::Optional< Tac::String > const & currStatus ) {
    if ( currStatus ) {
        fmt::print( "Interface {} is updated to {}\n", intf, *currStatus );
    } else {
        fmt::print( "Interface {} is removed\n", intf );
    }
}

int main() {
    const auto lineCardStatus = Tac::allocate< LineCardStatus >();
    {
        fmt::print( "Populating the status...\n" );
        auto link = lineCardStatus->linkStatusIs( "eth0" );

        fmt::print( "Updating the status...\n" );
        link->statusIs( "active" );

        fmt::print( "Starting the SM...\n" );
        const auto ledSm = Tac::allocate< LedSm >( lineCardStatus );

        fmt::print( "Updating the status...\n" );
        link->statusIs( "inactive" );

        fmt::print( "Removing the status...\n" );
        lineCardStatus->linkStatusDel( "eth0" );

        fmt::print( "Shutting down the SM...\n" );
    }
}

```

The default behavior is usually desirable. But `skipOnEnclosingDelete` may be useful when the user wishes to implement asynchronous deletion handling, or to handle changes in `linkStatus` via a different `when` statement.

Multiple paths

Execution order

If a state machine declares multiple `when` statements, the order of invocation of their actions is the same as the order in which `when` statements are declared in the state machine.

If `when` statements are defined in a type that is also declaring reactors using the legacy syntax the invocation of their actions will be last in the type's generated constructor.

In the following example, we have three `when` statements, one on local attribute `f`, one on its state `f::a` and one on `f::b`. We also have two legacy style reactors on `g` and one on `this` and `initialized`. The existing rules of execution apply for those legacy reactors mean that `this` will run first followed by the legacy reactors in the order of the attribute's declaration finishing with the execution of the `when` statements in the order of their declarations. Because `when f::b` is declared using `skipInitScan`, the action will not run in the construction of `Sm`. This means that for `Sm` we will run:

- `thisFunc`,
- `gInitiallyFunc`,
- `gaInitiallyFunc`,
- `initiallyFunc`,
- `handleLocalF`,
- `handleSubFA`.

▼ WhenOrder.tac

```

Foo : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;

    a : U32;
    b : U32;
}

Sm : Tac::Type( g, f ) : Tac::Constrainer {
    g : in Foo::Ptr;
    f : in Foo::Ptr;

    handleLocalF : void() {}
    handleSubFA : void() {}
    handleSubFB : void() {}
    when f => handleLocalF();
    when f::b => handleSubFB();
    when f::a => handleSubFA();

    gInitiallyFunc : invasive void() {}
    gaInitiallyFunc : invasive void() {}
    g => initially gInitiallyFunc();
    g::a => initially gaInitiallyFunc();

    initiallyFunc : invasive void() {}
    initialized : bool;
    initialized => initially initiallyFunc();

    thisFunc : invasive void() {}
    this => initially thisFunc();
}

```

Parent vs. child

In the following example, we have two `when` statements with one being the prefix of the other. `handleCard` is called before `handleActive` during initialization and during a call to `status` or `lcStatus`'s mutators.

▼ WhenParentChild.tac

```

LedSmParentAndChild : Tac::Type( status ) : Tac::Constrainer {
    status : in Status::PtrConst;

    handleCard : invasive void( lineCard : U8 ) {}
    handleActive : invasive void( lineCard : U8 ) {}

    when status::lcStatus[ lineCard ] => handleCard( lineCard );
    when status::lcStatus[ lineCard ]::active => handleActive( lineCard );
}

```

Siblings

In the following example, multiple `when` statements share a common prefix path. During `LedSm` initialization, the order of calls is the same as the order of their declaration in `LedSm` with `handleMode` being called first, followed by `handleActive` and finally `handleEnabled`.

Similarly, during initialization, or when an entry in `lcStatus` is inserted or deleted, `handleMode` is called before `handleActive`.

▼ WhenSiblings.tac

```
LineCardStatus : Tac::Type( lineCard ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    lineCard : U8;
    active : bool;
    mode : U32;
}

Status : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    lcStatus : LineCardStatus[ lineCard ];
}

Config : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    linecardEnabled : bool[ U8 ];
}

LedSm : Tac::Type( config, status ) : Tac::Constrainer {
    config : in Config::PtrConst;
    status : in Status::PtrConst;

    handleEnabled : invasive void( lineCard : U8 ) {}
    handleActive : invasive void( lineCard : U8 ) {}
    handleMode : invasive void( lineCard : U8 ) {}

    when status::lcStatus[ lineCard ]::mode => handleMode( lineCard );
    when status::lcStatus[ lineCard ]::active => handleActive( lineCard );
    when config::linecardEnabled[ lineCard ] => handleEnabled( lineCard );
}
```

Reacting to non-instantiating pointer attributes

If a path contains a non-instantiating pointer element, when the pointer is updated, the entire hierarchy under it gets updated in a single mutation call. This is different from an instantiating attribute, where it must be deleted first, recursively deleting the hierarchy under it, and then recreated with a brand new hierarchy. This section covers the behaviour of `when` statements when such a "pointer swap" occurs.

The events for a sub-attribute of a pointer are dispatched as follows:

- If the sub-attribute is a collection, events for the union of the keys in two collections are dispatched. First, each entry in the collection under previous pointer is dispatched with old value. If an entry with the same key exists in the collection under new pointer, its value is passed on as the new value during this dispatch. Finally, the entries in the new collection, which didn't exist in the old collection, are dispatched with the new value.
- If the sub-attribute is a singleton, a single event with old value from old pointer and new value from new pointer is dispatched.

▼ WhenPointer.tac

```
LineCardStatus : Tac::Type( lineCard ) : Tac::PtrInterface {
    `isNotifyingByDefault;
    lineCard : U8;
    connected : ordered bool[ U32 ];
}

Status : Tac::Type() : Tac::PtrInterface {
    `isNotifyingByDefault;
    // collection of pointers
    lcStatus : LineCardStatus::Ptr[ lineCard ];
}

LedSm : Tac::Type( status ) : Tac::Constrainer {
    status : in Status::PtrConst;
    handleConnected : extern invasive void(
        lineCard : U8, intfId : U32, prev : bool::Optional, curr : bool::Optional
    );
    when status::lcStatus[ lineCard ]::connected[ intfId ] as ( prev, curr ) =>
        handleConnected( lineCard, intfId, prev, curr );
}
```

In this example, `lcStatus` is now a collection of pointers, not an instantiating collection. The following snippet shows the behavior when a pointer is swapped:

▼ *WhenPointer.cpp*

```

#include "WhenPointer.h"
void
LedSm::handleConnected( U8 lineCard,
                        U32 intfId,
                        Tac::Optional< bool > prev,
                        Tac::Optional< bool > curr ) {
    const auto prevStr = prev ? fmt::format( "{}", *prev ) : "None";
    const auto currStr = curr ? fmt::format( "{}", *curr ) : "None";
    fmt::print( "For linecard {}, interface {}:\n", lineCard, intfId );
    fmt::print( "Status changed from {} to {}.\n", prevStr, currStr );
}

int main() {
    const auto status = Tac::allocate< Status >();
    const auto lcStatusOld = Tac::allocate< LineCardStatus >( 8 );
    lcStatusOld->connectedIs( 1, true );
    lcStatusOld->connectedIs( 2, false );
    lcStatusOld->connectedIs( 3, false );
    status->lcStatusIs( lcStatusOld );

    fmt::print( "Starting the SM...\n" );
    const auto ledSm = Tac::allocate< LedSm >( status );

    const auto lcStatusNew = Tac::allocate< LineCardStatus >( 8 );
    lcStatusNew->connectedIs( 2, true );
    lcStatusNew->connectedIs( 3, false );
    lcStatusNew->connectedIs( 4, true );

    fmt::print( "\nSwapping linecard 8 status...\n" );
    status->lcStatusIs( lcStatusNew );

    fmt::print( "\nShutting down the SM...\n" );
}

```

Reacting to Tac::Dir changes

`Tac::Dir` allows storing multiple entities of heterogeneous types. `when` statements can be used to react to changes in a subset of such entities under a `Tac::Dir`. The subset is identified based on the user provided type, essentially allowing filtering of dir entries based on the type.

In the following example, multiple `when` statements share the prefix `dir::entityPtr`. The first two expect an update when the collection element is of type `Intf` whereas the last two expect updates for elements of type `IntfStatus`. No updates are provided for elements which are of neither of those two types. The example also shows that in order to specify `IntfStatus` as the type being tracked by the `when` statement it needs to be declared as `entityPtr< IntfStatus >`, the following rules apply:

- The typename without `Ptr` or `RawPtr` wrappers are expected.
- `entityPtr< Base >` matches the entries of type `Derived` as well, where `Derived` is directly or indirectly derived from `Base`.

▼ WhenDir.tac

```

Intf : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    enabled : bool;
}
IntfStatus : Tac::Type( name ) : Tac::Entity {
    `allowsDirInstantiation;
    linkEstablished : bool;
}
Sm : Tac::Type( dir ) : Tac::Constrainer {
    dir : in Tac::Dir::PtrConst;
    handleIntf : void( name : Tac::String, intf : Intf::PtrConst ) {}
    handleIntfEnabled : void(
        name : Tac::String, prev : bool::Optional, curr : bool::Optional ) {}
    when dir::entityPtr< Intf >[ name ] as intf => handleIntf( name, intf );
    when dir::entityPtr< Intf >[ name ]::enabled as ( prev, curr ) =>
        handleIntfEnabled( name, prev, curr );
    handleIntfStatus : void( name : Tac::String, intf : IntfStatus::PtrConst )
{}
    handleIntfStatusLinkEstablished : void(
        name : Tac::String, prev : bool::Optional, curr : bool::Optional ) {}
    when dir::entityPtr< IntfStatus >[ name ] as intfStatus =>
        handleIntfStatus( name, intfStatus );
    when dir::entityPtr< IntfStatus >[ name ]::linkEstablished as ( prev, curr )
=>
        handleIntfStatusLinkEstablished( name, prev, curr );
}

```

This means that given an update to `entityPtr`:

- An element of type `Intf` is added or deleted `handleIntf` and `handleIntfEnabled` will be called,
- An element of type `IntfStatus` is added or deleted `handleIntfStatus` and `handleIntfStatusLinkEstablished` will be called.

If the `entityPtr` collection holds an element of another type than `Intf` and `IntfStatus` no reaction would be triggered with the previous state machine definition.

Limitations

- If an intermediate path element, i.e. neither root nor leaf, is a collection element, the collection may be internally-keyed, or an externally-keyed map marked with `publishKeyWithNotification`. For example, `linkStatus` and `linkConfig` collections in the [Nested paths](#) section.
- Stacks and Queues are not yet supported as a collection leaf.
- More than [3 levels of nesting in a path](#) is supported only in very specific situations.

This section describes properties which can be defined on a constraint.

isAppliedInDestructor

The `isAppliedInDestructor` property can be specified in order to execute a reactor when the enclosing object gets deleted:

```
MyType : Tac::Type() : Tac::PtrInterface {
    // ...
    handleDeleted : extern void();
    this -> {
        `isAppliedInDestructor;
        handleDeleted();
    }
}
```

In this example, `handleDeleted` is called when the reference count of the object drops to zero.

It is important to be aware that the body of a destructor is only executed when the underlying C++ object is actually destroyed. Some implications of that behaviour:

- If a btest obtains a reference to an object inside an instantiating container that is then removed from the container, the destructor would not be invoked. The btest is still keeping the object alive due to that reference.
- If the agent is crashing, destructors aren't invoked
- If one has some cleanup code invoked from the destructor, and one wants to test an unclean restart:
 - Agents normally get restarted by being killed and restarted
 - Such restarts don't get a chance to execute destructors
 - Dropping the last reference to an object in a btest will allow destructors to execute
 - The btest has to be carefully constructed to prevent that cleanup from happening

Mounts are the basic method for agents to share state with each other, usually via Sysdb or sometimes directly via an agent-to-agent mount. This chapter only covers attrlog-based mounts, and does not apply to shared memory mounts.

Managing mounts is critical to successful agent development in EOS.

Mount APIs

For an agent to mount a path from Sysdb, the agent must use four parts of the mount framework:

1. The `Sysdb::EntityManager` class
2. The `Sysdb::Mounter` class
3. The `Sysdb::MountGroup` class
4. One or more of the `doMount*` APIs

Used together, a basic example of mounting a path can look like this:

```
void
MyAgent::doInit() {
    Sysdb::MountGroup::Ptr mg = entityManager()->mountGroup( this );
    mg->doMountPath( "foo/bar" );
    mg->doClose( false, "", 0 );
}
```

```
def doInit( self, entityManager ):
    mg = entityManager.mountGroup()
    mg.mountPath( 'foo/bar' )
    mg.close()
```

Note that API and class names can differ slightly between C++ and Python interfaces. In this chapter, we use the C++ names by default and document both C++ and Python behavior.

Sysdb::EntityManager

`EntityManager` is the agent's main interface with the Sysdb mount framework. Internally, `EntityManager` does a lot of important work for the mount framework, but users only need to interact with two APIs for regular mounting use-cases.

getMountGroup()

`getMountGroup()` requests that the `EntityManager` create and return a new `MountGroup`.

```
MountGroup::Ptr
EntityManager::getMountGroup( Sysdb::Mounter::Ptr const & mounter );
```

The C++ API takes in a pointer to a `Sysdb::Mounter` as an argument. Since all C++ agents derive from `Agent::CAgent` and `Agent::CAgent` derives from `Sysdb::Mounter`, agents generally just pass `this` to `getMountGroup()`.

```
EntityManager.Sysdb.mountGroup( **kwargs ) -> EntityManager.MountGroup
```

The Python API takes in keyword arguments that will be passed to the `EntityManager.MountGroup` constructor. Two keyword arguments in particular are most likely to be useful for regular agent use-cases:

- `threadSafe=False`: Must be set to `True` if the agent is multithreaded.
- `mountFailureCallback=None`: Users may pass a callback function that will be called if the `MountGroup` fails. The function should accept a dictionary mapping failed paths to `FailedMount` objects.

The Python API notably does not require an argument of any class similar to `Sysdb::Mounter`, nor a reference to the agent class itself. Instead, the Python `MountGroup` takes an explicit callback in `doClose()`.

getEntity()

`getEntity()` takes a path to an entity that the agent has mounted and returns a pointer to that entity. The given path should be relative to the Sysdb root (i.e. the path should not start with `/ar/Sysdb`). Agents commonly use these APIs to fetch the entity once it has successfully been mounted in `doMountsComplete()`.

```
Tac::Entity::Ptr EntityManager::getEntity( Tac::String const & path ) const;
template< class T >
Tac::Ptr< T > EntityManager::getEntity( Tac::String const & path ) const;
Tac::Entity::Ptr EntityManager::safeGetEntity( Tac::String const & path ) const;
```

The C++ `getEntity()` API will assert if the path does not exist. `safeGetEntity()` does the same lookup, but instead returns a `nullptr` on failure. This assertion may seem extreme at first, but if an agent has successfully mounted a path and is not able to locate the mounted entity, then the agent is in an bad and unexpected state, so the safest thing to do is crash.

The templated version of `getEntity()` automatically casts the entity it looks up to the expected type, saving the user from having to explicitly do the cast.

```
EntityManager.Sysdb.entity( path, type='' ) -> Any
```

Although `entity()` is a valid way to fetch mounted entities, Python agents generally prefer the `EntityFuture` mechanism for its simplicity.

The Python `entity()` API raises a `NameError` if the path does not exist. `entity()` takes an optional `type` argument, which the user can set to a string representation of the entity's expected type (e.g. `entity('foo/bar', type='Foo::Bar')`). If the entity that `entity()` finds at the given path is of a different type than the `type` argument, `entity()` raises an `EntityManager.MountError` with more info. Although the `type` argument is not required, it helps handle TAC objects in Python since TAC objects are strongly typed but Python is not.

Regardless of the `type` argument, the entity that `entity()` returns will be of the full type of the entity at the given path. That is to say, given a path `foo/bar` of expected type `Foo::Bar`,

`entity()` will return a reference to an object of type `Foo::Bar`, not a generic `Tac::Entity`. This is the same behavior as the templated C++ `getEntity()`.

Sysdb::Mounter

`Mounter` is a simple wrapper around the `doMountsComplete()` callback, which the user should override. C++ agents indirectly derive from `Mounter`. In most cases, `this` is the only `Mounter` an agent will need.

A `Mounter` is associated with one or more `MountGroups`. When a `MountGroup` completes, it calls `doMountsComplete()` on its relevant `Mounter`.

Python agents do not need to use a `Mounter` or define a `doMountsComplete()`. See `getMountGroup()` and `doClose()` for details.

doMountsComplete()

```
virtual void Mounter::doMountsComplete( MountStatus mountStatus,
                                         EntityManager::Ptr const & em );
```

`doMountsComplete()` is a callback that gets called when an associated `MountGroup` completes. The function is passed in a pointer to the `EntityManager` that created the `MountGroup` and the status of the `MountGroup`, which has a value of either `MountStatus::mountSuccess` or `MountStatus::mountFailed`.

Agents generally should not survive failing a mount. Users should begin their `doMountsComplete()` with an assertion that `mountStatus == MountStatus::mountSuccess`.

If `doMountsComplete()` is called with a successful `MountStatus`, then all of the paths in the `MountGroup` are fully synchronized with Sysdb. At this point, the agent can fetch pointers to these synchronized entities with `getEntity()` and start any SMs related to those entities.

Sysdb::MountGroup

`Sysdb::MountGroup` is entirely unrelated to the `MountGroup` keyword in mount profiles.

A `MountGroup` is a logical grouping of paths an agent wishes to mount. If any mount in a `MountGroup` fails, the mount framework declares the entire `MountGroup` as failed.

Most agents place all of their mounts in a single `MountGroup`, since agents never expect to fail their mounts. If an agent has multiple stages of mounts (i.e. mount some state, then mount more state based off of the first state), then the agent would use a new `MountGroup` for each stage.

Given a `MountGroup`, the agent adds paths to the `MountGroup` via one or more calls to the `doMount* APIs`, then closes the `MountGroup` and starts the mounts with `doClose()`.

Although the `doMount* APIs` are member functions of `MountGroup`, we document them in a separate section due to their importance.

`doClose()`

`doClose()` indicates that the agent has added all mount paths to the `MountGroup` and that the mounts may begin.

`doClose()` allows the user to request a blocking mount, where the entire thread blocks until the mounts complete. This can only be done in non-main threads of multithreaded agents or in tests that are doing mounts outside of the activity loop. Agents will generally use non-blocking mounts, where the mount framework notifies the agent of mount completion through a callback.

```
bool MountGroup::doClose( bool block, Tac::String const & msg, Tac::Seconds timeout );
```

All arguments to the C++ API are related solely to blocking mounts. `block` indicates whether or not this `MountGroup` should do blocking mounts, `msg` takes a description to print if the blocking mount takes longer than expected, and `timeout` takes a number of seconds after which the blocking mount will automatically fail.

If `block` is `false`, mounts do not print description messages and use the default timeout of 600 seconds.

`doClose()` returns `false` if a blocking mount got interrupted by a `SIGINT`. It returns `true` in all other cases.

```
EntityManager.MountGroup.close( callback=-1, blocking=False, msg="mounts to complete" ) -> None
```

Unlike the C++ API, the Python `close()` accepts an explicit callback function. The `MountGroup` will call this function with no arguments if the `MountGroup` succeeds. If the `MountGroup` fails, it will call the function supplied to `mountFailureCallback()` and raise an `EntityManager.MountError`.

One of `callback` and `blocking` must be set to a non-default value. The user must either specify a blocking mount, or otherwise provide a callback function from which they can be notified.

Like the C++ API, `msg` takes a description string for use with blocking mounts.

doMount* APIs

The `doMount*` APIs add mount paths to a `MountGroup`, requesting the mount framework to mount them. These APIs use the agent's `mount profiles` and `preinits` to reduce data duplication and allow for cleaner agent code. Any agent using the `doMount*` APIs must provide a mount profile.

All of the Python `doMount*` APIs immediately return one or more `EntityFuture`s. These objects are stand-ins for the entities that will be synchronized from Sysdb. Once the `MountGroup` completes, the `EntityFuture`s will have been resolved to the actual synchronized entity. Note that the Python APIs can also return `None` in certain cases. See [No-op paths](#) for details.

Although these APIs are a part of the `MountGroup` class, we document them separately since they are the main interface with the mount framework.

doMountPath()

```
void MountGroup::doMountPath( Tac::String const & path );
```

```
EntityManager.MountGroup.mountPath( path ) -> Tac.EntityFuture
```

`doMountPath()` takes a string representing a mount path and adds that path's mount info to the `MountGroup`.

The Python API returns an `EntityFuture` for the path.

Unless the agent needs to mount an individual path, prefer using one of the other `doMount*` APIs to mount multiple paths in a single line of agent code.

doMountGroup()

```
void MountGroup::doMountGroup( Tac::String const & mountGroupName );
```

```
EntityManager.MountGroup.mountGroup( mountGroupName ) -> dict[ str,
```

`doMountGroup()` adds paths from mount groups defined by the `MountGroup` keyword in the agent's mount profile to a `Sysdb::MountGroup`. There are no restrictions or implied relationships between the keyword and the class.

`doMountGroup()` takes the name of a `mount group` from the agent's mount profile and mounts every path that is a part of that mount group. Mount group names should be of the format

`<agent ID>::<mount group name>`, where the mount group name comes from the `MountGroup` line of the mount profile.

Consider the following mount profile:

```
agentName: FooAgent-%sliceId
foo/bar, Foo::Bar, r
MountGroup: mg1, 10
foo/slice/%sliceId, Foo::Bar, w
Include: Feature-include-*
```

Calling `doMountGroup()` with the mount group name `FooAgent::mg1` would mount `foo/slice/%sliceId` as well as any other paths from the `Include` line.

To mount `foo/bar` and any other paths not included in a mount group, the agent would call `doMountGroup()` with the special name `FooAgent::`.

The Python API returns a dictionary that maps mounted paths to `EntityFuture`s.

doMountProfile() and doMountProfileWithIncludes()

```
void MountGroup::doMountProfile( Tac::String const & profileFileName );
void MountGroup::doMountProfileWithIncludes( Tac::String const & profileFileName );

EntityManager.MountGroup.mountProfile( profileName, withIncludes=False ) ->
dict[ str, Tac.EntityFuture ]
```

`doMountProfile()` takes the name of a mount profile file and mounts all mount paths explicitly listed within.

`doMountProfileWithIncludes()` does the same and additionally mounts all paths listed in any `Include`d files.

Use extreme caution when using these APIs in profile files that have the `Condition` or `ConditionBlocker` keywords! These APIs attempt to *unconditionally* mount all requested paths. You **must** ensure that all conditions are met for every mount group in the profile. Attempting to use these APIs when there are unmet conditions will result in your agent hanging!

The profile name argument to both APIs accepts a trailing `glob`-like `*` wildcard. This allows users to mount multiple files in a single `doMountProfile()` call, making it easy to include related include files from several packages.

```
doMountProfile( "Foo-*" );
```

The above call will mount profiles with names such as `Foo-Bar`, `Foo-Bar-Baz`, and `Foo-`.

These APIs are particularly useful for plugins and libraries that may be used by multiple agents. Since such code would have an include mount profile that any users would `Include` in the agent mount profile, the plugin can make a single `doMountProfile()` call to mount all the paths necessary for the plugin. An agent can alternatively perform a single `doMountProfileWithIncludes()` call on the main mount profile if it definitely needs to mount all included paths as well.

The Python API returns a dictionary that maps mounted paths to `EntityFuture`s.

`doMountAll()`

```
void MountGroup::doMountAll( Tac::String const & agentName );
```

```
EntityManager.MountGroup.mountAll( agentName ) -> dict[ str, Tac.EntityFuture ]
```

`doMountAll()` takes an agent ID and mounts every path from every mount profile with that agent ID. Since `doMountAll()` mounts everything for an agent, it is an ideal choice for simple agents and tests. However, since the user doesn't get any control over which paths are mounted or when they are mounted, more complex agents might prefer other APIs.

The Python API returns a dictionary that maps mounted paths to `EntityFuture`s.

Caveats

Profile loading

To use the `doMount*` APIs, the agent must correctly load its mount profiles. For this to happen, the agent's agent name must be set correctly.

This is done automatically in production and in non-cohab btests, but cohab btests often directly instantiate an agent type or an individual SM.

If profile loading fails, the mount framework will throw a descriptive exception. The solution in these cases are often to correctly set the agent name in a `Tac.newInstance` call that instantiates an agent type, or else to use `Tac.setproctitle` to manually set the agent name.

No-op paths

Paths with the `D` mount flags should not be explicitly mounted. These paths use a `DirMounter` to automate the mount of the path. See [AID10766](#) for more details on `DirMounter`s.

Since the `doMount*` APIs intend to make mounting large groups of mount paths easy, the APIs treat certain `D` flag paths as no-ops, silently skipping the path and returning `None` for Python APIs for ease of use.

`DirMounter`s behave differently for [active and passive mounts](#), so `doMount *` must treat no-op paths differently for the two as well.

For active mounts, any path with a `D` flag is a no-op path.

For passive mounts, any path with a `D` flag and a `?` wildcard is a no-op path. Any other `D` flag path will continue to be mounted and will delay the completion of the mount group until the `DirMounter` has finished mounting the path.

Legacy `doMount()` calls

Before adding the `doMount*` APIs, the mount framework exposed a single `doMount()` API for adding a path to a `MountGroup`. This API requires duplicating mount info in agent code and can only mount a single path at a time. Because of this, the `doMount* APIs` are preferred.

Although legacy `doMount()` calls are pervasive in our code base, they are deprecated. Any new mounts should use the `doMount*` APIs.

```
void MountGroup::doMount( Mount const & item );  
Mount::Mount( Tac::String const & path, Tac::String const & type, Tac::String  
const & flags );
```

```
EntityManager.Sysdb.mount( path, typeName, mode='r' ) -> Tac.EntityFuture
```

The C++ `doMount()` takes in an object of type `Sysdb::Mount`, which is a simple wrapper around a mount path, type, and flags. There are no concerns about the lifetime of the `Mount` object. Agents usually construct the object inline with the `doMount()` call, for example:

```
mg->doMount( Sysdb::Mount( "foo/bar", "Foo::Bar", "r" ) );
```

The Python `mount()` API functions the same as the C++ `doMount()`, but doesn't require the intermediate `Mount` object.

Mount flags

Mount flags specify permissions and modify the behavior of mounts. Users list mount flags on a per-path basis in the mount profile.

```
a/b/c, Foo::Bar, wc
```

Mount flags also appear in the old-style mounting APIs (consider using the preferred [doMount*](#) APIs instead).

```
// Old style, not preferred
mg->doMount( Sysdb::Mount( "a/b/c", "Foo::Bar", "wc" ) )
```

```
# Old style, not preferred
mg.mount( 'a/b/c', 'Foo::Bar', 'wc' )
```

Note that the order of mount flags does not matter.

User-facing mount flags

| Flag | Meaning |
|------|--|
| w | Read or write |
| r | Read only |
| c | Create in Sysdb if missing |
| i | Recursively mount entire sub-tree under a Tac::Dir |
| f | Force write mount on standby SSO |
| S | Force coalescing of AttrLog updates |
| D | Mount using a DirMounter |
| d | Automatically delete in Sysdb on deconfiguration |
| o | Mount optionally if present in Sysdb |
| A | Use Golang compatible AttrLog |
| K | Close connection on overload |
| L | Use local backup |

Read or write (w)

Mounts the entity with read-write permissions.

Read only (r)

Mounts the entity with read-only permissions. Note that this flag gets converted internally. See [Internal mount flags](#) for more details.

Create in Sysdb (c)

If the entity does not exist on Sysdb, create it first. This also creates all missing intermediate `Tac::Dir`s as needed. It should be combined with the `r` or `w` flags.

This is useful to create entities without having to mount the parent as writable.

Note that the entity is created at mount time with active mounts, and on agent connection with passive mounts. This is very different from preinits, which create the entity at Sysdb startup.

Mount recursively (i)

Mounts the entity recursively. This is most useful on `Tac::Dir`s to mount the entire subtree under the path. The subtree is also actively synchronized, meaning that if a new entity gets created in the subtree on Sysdb, the new entity will be synchronized to the Agent.

Take, for example, a `Tac::Dir` at path `a/b`, with child entities `a/b/c` and `a/b/d`. If an agent mounts `a/b` with just the `r` flag, the agent will receive `a/b` and its `entryState` collection, which tells the agent child entities `a/b/c` and `a/b/d` exist on Sysdb, but the agent does not receive the child entities.

If the agent instead mounts `a/b` with the `ri` flags, it will get all of the entity data for `a/b`, `a/b/c`, and `a/b/d`. If a different agent later creates a new entity `a/b/d/e`, the agent will also receive that entity at the time that it is created.

Force write mount (f)

Forces `w`-flag mounts via a `locallyReadOnly` EntityManager to actually be mounted as writable. Without the `f` flag, `w`-flag mounts via a `locallyReadOnly` EntityManager are instead mounted read-only. The common instance of a `locallyReadOnly` EntityManager is for agents running on the standby supervisor under the SSO redundancy protocol.

Force scheduled AttrLog (S)

Enables forced scheduled AttrLog for the mount, causing AttrLog messages to be buffered and coalesced in FlowOuts, regardless of congestion. By default, the connection is closed on congestion. See [go/scheduledattrlog](#) for more details.

Mount using a DirMounter (D)

Indicates that a DirMounter will be handling the mounting for this path. With passive mounts, one is automatically created on Sysdb. With active mounts, the agent must instantiate one locally. See [AID10766](#) for more details.

Automatically delete (d)

Marks the path as an auto-deletion mount point. When the agent's runnability becomes false and it becomes deconfigured, the path will be deleted from Sysdb. Must be used with the **c** flag. See [AID11583](#) for more details.

Optional mount (O)

Marks the path as an optional mount. If the path does not exist on Sysdb, simply skip the mount instead of failing with an `entity not found` error. Passive mounts only.

Golang compatible AttrLog (A)

Enables extra AttrLog messages to allow golang agents to properly construct entities without using the C++ constructor.

Close on overload (K)

Indicates that the connection should always close on overload for this path, regardless of whether other mounts are currently in progress. Overload is when the AttrLog buffer is full and Scheduled AttrLog is not enabled for this path.

Use local backup (L)

Creates the entity in local backup, ignoring Sysdb entirely. The path must start with `/backup`. See [AID9556](#) or the [Local Mount chapter](#) for more details.

Connection flags

Connection flags modify the connection between the agent and Sysdb, affecting every mount. Users specify them with the `connectionFlags` keyword in a mount profile.

```
connectionFlags: AN
```

Notice that some of the connection flags are also valid as per-path mount flags.

| Flag | Meaning |
|------|--|
| S | Force coalescing of AttrLog updates |
| N | Coalesce AttrLog updates during congestion |

| Flag | Meaning |
|------|--------------------------------|
| A | Use Golang compatible AttrLog |
| E | Skip ephemeral entity deletion |

Use scheduled AttrLog on congestion (N)

Enables scheduled AttrLog for all mounts on the agent, causing AttrLog messages to be buffered and coalesced in FlowOuts once the socket buffer becomes full. By default, the connection is closed on congestion. See [go/scheduledattrlog](#) for more details.

No ephemeral deletion (E)

Skips ephemeral entity deletion logic in the mount framework. This is a medium-term fix for a known issue. See [BUG413383](#) for more details.

Internal mount flags

The mount framework uses these mount flags internally. Users generally should not have to deal with these flags, but we document them here to ease debugging since they may show up in traces.

| Flag | Meaning |
|------|--|
| t | Mark as root mount |
| p | Mark as passive mount |
| c | Convertible from read-only to read-write |
| n | Close connection on overload |
| e | Wait for Entity in Sysdb |
| y | Mark as config path |

Mark as root mount (t)

Marks the root mount, the first mount across a connection. In agents, this is generally `/ar/Sysdb`.

Mark as passive mount (p)

Indicates to the mount framework that the agent uses passive mounts. The mount framework will append this flag to the root mount if it does not find the `activeMount` keyword in the agent's mount profiles.

Convertible from read-only to read-write (C)

Mounts the path as convertible read-only. All `r`-flag mounts are internally converted into `C`-flag mounts to indicate that they are allowed to later become writable via a `w`-flag mount. Internally, the `r` flag is strictly read only and cannot ever become writable.

Close on overload (n)

Indicates that the connection should be closed on buffer overload for this path. If the AttrLog buffer is full and there is an update for this entity, the connection will be closed instead of adding that update to a FlowOut. Applied to all paths if scheduled AttrLog is not enabled.

Wait for Entity in Sysdb (e)

Causes the mount to wait until the Entity exists on Sysdb, instead of failing immediately. Used with active mounts only and enabled by default outside of btests and stests.

Mark as config (y)

Marks a path as a config path for ConfigCounter. See [AID11212](#) for more details.

Mount profiles

Mount profiles are plain-text files that specify which entities an agent would like to mount from Sysdb. Every agent that mounts from Sysdb must have at least one mount profile. Mount profiles provide many benefits, including:

- Buildtime cross-checks of mounting behavior between agents
- Runtime checks to ensure agents do not misbehave
- Reduced duplication of mount info via the `doMount*` APIs

Mount profiles are installed at `/usr/lib/SysdbMountProfiles` or `/usr/lib64/SysdbMountProfiles`, depending on the system architecture.

The mount framework reads mount profiles line-by-line, although users may split logical lines over several lines with backslashes. Each line should be one of the following:

1. A mount path
2. A line beginning with one of several keywords
3. A comment (marked by a `#`)
4. A blank line

Mount paths

To specify a entity that an agent wishes to mount, the user needs to supply:

1. The path in Sysdb, relative to the base path `/<sysname>/Sysdb`
2. The type of the entity at that path
3. The mount flags with which to mount the path (see [Mount flags](#))

The mount path should consist of upper- and lower-case ASCII characters and numbers, as well as hyphens (`-`), underscores (`_`), and slashes (`/`).

Mount paths can also contain variables called template keys (see `agentName`). These template keys may be user-defined from `agentName` or `alias` lines, or may be one of the following predefined template keys.

| Template key | Expands to |
|----------------------------|---|
| <code>%agentString</code> | Full agent name (including template keys) |
| <code>%cellId</code> | Cell ID for current supervisor |
| <code>%peerCellId</code> | Cell ID for peer supervisor |
| <code>%cellPath</code> | <code>cell/%cellId</code> |
| <code>%peerCellPath</code> | <code>cell/%peerCellId</code> |

The following contains a few examples of mount paths in a mount profile. Note that the spaces are optional, but help with readability.

```
a/b/c, Foo::Bar, wc
cell/%cellId/supervisor/specific/path, Bar::Baz, r
```

Template keys are case-insensitive. Therefore, the following examples will all expand to the exact same path:

```
cell/%cellId/path, Foo::Bar, r
```

Keywords

Lines beginning with one of the following keywords specify metadata about the mount profile and mounts within it. The colon (:) is used as a separating character between a keyword and the rest of the line, where necessary.

Keywords are case-insensitive. The capitalizations used here are simply the most common ones.

| Keyword | Summary |
|--|---|
| <code>agentName</code> | List agent name |
| <code>activeMount</code> | Mark agent as using active mounts |
| <code>Include</code> | Import another file inline |
| <code>MountGroup</code> | Logically group mounts |
| <code>Condition</code> and <code>ConditionBlocker</code> | Make a conditional mount group (passive mount only) |
| <code>alias</code> | Construct a new template key |
| <code>connectionFlags</code> | Modify connection to Sysdb |
| <code>rootFlags</code> | Modify root mount |
| <code>subAgent</code> | List managed agents |
| <code>noDefaultMountGroup</code> | Skip default mounts |

agentName

The `agentName` keyword specifies the name of the agents that this mount profile belongs to. The `agentName` must appear on the first non-blank, non-comment line of the mount profile. The lack of an `agentName` line indicates that the file is explicitly for including in other mount profiles (see `Include`).

Multiple mount profiles may list identical agent names or agent names that may match the same agent. In the case that a running agent matches several mount profiles, the mount framework will merge the info from the separate files.

In its most simple form, the agent name just lists the exact name that the agent specifies in its `LauncherPlugin`. If an agent's `LauncherPlugin` lists the agent's name as `FooAgent`, its mount profile might look like this:

```
agentName: FooAgent
```

Note that this name is case sensitive and must match exactly. For example, an agent with the name `FooAgentBar` would not match this mount profile.

Launcher may encode metadata about an agent in its name when it starts. One common example of this is for slice agents, where Launcher starts a new instance of the agent for each linecard. Each agent needs a unique name in this case, so Launcher appends the linecard to the agent name, producing an agent name like `FooAgent-Linecard1`. Notice that this agent name will not match the mount profile in the previous example since it is not an exact match.

Agents often wish to mount different paths from Sysdb depending on this metadata. To allow this, mount profiles can specify variables called template keys in the `agentName` line. Users specify template keys by preceding the variable name with the `%` symbol. The agent name must start with a regular, non-template-key string, called the agent ID. The rest of the agent name must consist of zero or more template keys, separated by hyphens (`-`). The agent ID is the only part of the agent name used to match the agent to its mount profile. The default template key `%agentString` always holds the full agent name, including any template key values.

If an agent name does not use any template keys, the mount framework looks for an exact match. Consider an agent with the name `FooAgent-Linecard1` with the example above. Although the agent ID `FooAgent` matches the profile, the profile does not have any template keys. The mount framework enforces an exact match, rejecting the mount profile.

```
agentName: FooAgent-%sliceId
```

In this example, the agent ID is `FooAgent` and the line defines a template key `%sliceId`, which will get its value from the agent's name when it starts. Consider three cases here:

1. The agent name is `FooAgent`. Although the agent ID matches the profile, the agent name does not include any template values (i.e. there is no `-` in the name). The mount framework looks for an exact match and rejects the mount profile.
2. The agent name is `FooAgent-Linecard1`. This will match this mount profile and `%sliceId` will have the value `Linecard1`.
3. The agent name is `FooAgent-Linecard1-OtherInfo`. This will match this mount profile and `%sliceId` will have the value `Linecard1`. Since there is no template key for `OtherInfo`, that part of the agent name is ignored.

This third case is sometimes undesirable, since it loses information. If the agent would like to capture any extra data, they can do so by appending the `$` symbol to the end of the last template key. This will place any remaining text from the agent name into the last template key.

```
agentName: FooAgent-%sliceId-%otherInfo$
```

If the agent name is `FooAgent-Linecard1-OtherInfo1-OtherInfo2`, then `%sliceId` has the value `Linecard1` and `%otherInfo` has the value `OtherInfo1-OtherInfo2`.

Template keys may be used in [mount paths](#), [aliases](#), and [conditions](#).

activeMount

The `activeMount` keyword indicates that the agent associated with this mount profile should use active mounts. Without this keyword, the agent defaults to passive mounts. Note that active mounts are preferred to passive mounts, so any new agent should enable active mounts via this keyword. See [Active vs. passive mounts](#) for more info about active mounts.

The `activeMount` keyword occupies an entire line. Although it does not matter where this keyword is in the mount profile, users generally place it on the line after the `agentName` by convention. This helps it be clear to readers that the agent specified by `agentName` is using active mounts.

```
agentName: FooAgent
activeMount
```

Note that, like other information in mount profiles, the `activeMount` keyword is merged across mount profiles for an agent. If an agent has several mount profiles, but the `activeMount` keyword is only present in one, the agent will still use active mounts. Although it is not necessary to have the `activeMount` keyword present in every mount profile of an agent that uses active mount, it is recommended for clarity.

Include

The `Include` keyword allows a mount profile to include the contents of a separate include file. An include file is a stripped-down mount profile that helps reduce data duplication. One common use case is when several agents use a common SM that operates on some Sysdb paths. Each of these agents can `Include` an include file for the paths that the SM needs. Include files may not contain any `agentName`, `Include`, or `MountGroup` keywords.

Consider package dependencies when using include files. The current dependency infra does not generate dependencies from `Include` lines. See [AID10](#) for information on package dependencies.

```
# FooAgentProfile
agentName: FooAgent

Include: OtherPackage-UsefulSm-include
```

```
# OtherPackage-UsefulSm-include
useful/path/1, Foo::Bar, r
useful/path/2, Foo::Bar, r
```

In this example, the contents of `OtherPackage-UsefulSm-include` are effectively inserted into `FooAgentProfile` at the `Include` line.

If an `Include` line is present under a `mount group`, then any included paths are part of that mount group.

```
# FooAgentProfile
agentName: FooAgent

MountGroup: mg1, 10
some/path, Foo::Bar, r
Include: OtherPackage-UsefulSm-include
```

In this case, mount group `mg1` includes `some/path`, `useful/path/1`, and `useful/path/2`.

The filename that `Include` takes supports `glob wildcards` anywhere in the path. This allows profiles to include all files named with a particular format, making it easy to include related include files from several packages.

```
# FooAgentProfile
agentName: FooAgent

Include: Foo*Deps-?
```

```
# FooBarDeps-1
dep/path/1, Foo::Bar, r
```

```
# FooDeps-2
dep/path/2, Foo::Bar, r
```

```
# FooDeps-Baz
dep/path/3, Foo::Bar, r
```

```
# Foo1Deps-
dep/path/4, Foo::Bar, r
```

In this example, `FooAgent` ultimately gets mount info for both `dep/path/1` and `dep/path/2`, but not `dep/path/3` or `dep/path/4` because a `?` matches exactly one character.

MountGroup

The `MountGroup` keyword allows users to group mount paths in a mount profile. The user can name these logical groups and organize their mounts. For passive mounts only, mount groups also play an important role in executing the mounts themselves.

The `MountGroup` keyword and the mount group it creates is entirely unrelated to `Sysdb::MountGroup`.

Users define a mount group by specifying a name and a priority. The priority must be an integer between 1 (highest) and 99 (lowest), and is only relevant for passive mounts. Multiple mount groups may share the same priority.

The mount group will contain all regular mount paths and `Included` mount paths below the `MountGroup` line until the mount profile ends or defines a new mount group.

```
agentName: FooAgent-%sliceId  
foo/bar, Foo::Bar, r  
MountGroup: mg1, 10  
bar/baz, Bar::Baz, r  
baz/bat, Baz::Bat, r
```

In this example, `bar/baz` and `baz/bat` are both part of the mount group `FooAgent::mg1`. Since the mount profile lists `foo/bar` before it defines any mount groups, `foo/bar` is included in the global mount group `FooAgent::`, which has the lowest priority of 99.

Notice that the mount group is named relative to the `agent ID`, not the full agent name. Agent code can use this name with the `doMount*` APIs to mount all paths in this mount group with a single function call.

Like other mount info, mount groups are merged across mount profiles. If multiple mount profiles define mount groups of the same name, the mount framework will combine the separate mount paths into a single mount group with the shared name.

For passive mounts, agents can use the priority to define the order in which Sysdb will send the mount groups to the agent, starting with mount groups of priority 1 and ending with those of priority 99. Sysdb may send mount groups of the same priority in an arbitrary order.

Condition and ConditionBlocker

`Condition` and `ConditionBlocker` are for passive mounts only. The mount framework will throw an exception if it finds either keyword in an active mount profile.

The `Condition` keyword modifies a `mount group` into a conditional mount group. Sysdb will not attempt to send the mounts in the mount group until the condition in the `Condition` line is true. `ConditionBlocker` is a variant of `Condition` that additionally prevents any mount groups of a lower priority from mounting until the condition in the `ConditionBlocker` line is true.

In the basic case, conditions compare a `template key` against a constant value or another template key. Sysdb evaluates these static conditions immediately upon agent connection. These simple conditions only support the `==` and `!=` operators.

```
agentName: FooAgent-%key1-%key2

MountGroup: mgFoo, 10
Condition: Foo == %key1
foo/path, Foo::Bar, r

MountGroup: mgNeq, 10
Condition: %key1 != %key2
not/equal, Foo::Bar, r
```

In this example, `foo/path` will only mount if the agent name starts with `FooAgent-Foo`. Similarly, an agent name of `FooAgent-Foo-Foo` will mean that `not/equal` will never be mounted. An agent name of `FooAgent-Foo-Bar` would allow both paths to be mounted.

Conditions can also compare a value or template key against an attribute of an entity in Sysdb. Conditions involving a Sysdb path support a variety of standard operations, as well as the `in` and `notin` operators for checking collection membership. Their general format is the following, shown with all supported operators.

```
Condition: val ( == | != | < | > | <= | >= | in | notin )
/%sysname/Sysdb/path/ent.attr
```

A few things to note about conditions with Sysdb paths:

- The Sysdb path must be on the RHS of the expression.
- The Sysdb path must be absolute, using the special `%sysname` template key.
- The attribute must be a builtin type or a collection thereof (e.g. integer, float, string, boolean, pointer, enum, etc.). Conditions do not support arbitrary nominals.

Users can combine conditions with the `and` operator. The mount framework evaluates conditions in the order the users defines them, allowing for short-circuiting.

By default, the mount framework considers any constant or template values to be strings. However, when comparing against an attribute of a Sysdb path, the mount framework attempts to coerce the string into the type of the attribute. For example, the mount framework will coerce the string `32` into an integer value when comparing against an integer attribute. Similarly, any string `foobar` will be coerced to an enum value if compared against an enum.

Aside from this general type coercion, the mount framework also recognizes a few special values for specific types.

| Keyword | Value | When compared against |
|--------------------|---------------|------------------------------------|
| <code>true</code> | Boolean true | Boolean attribute |
| <code>false</code> | Boolean false | Boolean attribute |
| <code>NULL</code> | Null pointer | Pointer attribute |
| <code>""</code> | Empty string | String attributes or template keys |

Only substituting special values when necessary allows users to also use the literal strings. For example, users may want to check if a string attribute is set to the literal string `true` or `NULL`.

The following is a more complete example of conditional mount groups.

```
agentName: FooAgent-%sliceId

MountGroup: mg1, 10
always/mount, Foo::Bar, r

MountGroup: mg2, 20
Condition: true == /%sysname/Sysdb/some/path.valBool and %sliceId in
/%sysname/Sysdb/slice.entryState
maybe/mount, Foo::Bar, r

MountGroup: mg3, 30
Condition: %sliceId == FixedSystem
fixed/mount, Foo::Bar, r
```

Consider this example of how mounting could happen with this mount profile.

1. The agent name is `FooAgent-FixedSystem`.
2. Evaluate which mount groups are mountable:
 1. `mg1` is always mountable
 2. Assume `some/path.valBool == false`. `mg2` is not mountable yet.
 3. `mg3` is mountable since `%sliceId` has the value `FixedSystem`.
3. Both `mg1` and `mg3` are currently mountable, mount them in order of their priorities. First mount `mg1` (priority 10), then `mg3` (priority 30)
4. Some time in the future, `some/path.valBool` becomes `true`
5. Check if `FixedSystem` is a child of `slice`. Assume it is, meaning `mg2` is now mountable.
6. Mount `mg2` in according to its priority. Since there are no other mountable mount groups, immediately mount `mg2`.

Notice that `mg3` got mounted before `mg2`, despite having a lower priority than `mg2`. Priority only assigns ordering between mountable mount groups. Since `mg2` was not mountable at the time that `mg3` was, `mg3` was allowed to be mounted first.

`ConditionBlocker` functions the same as `Condition`, except that it also prevents mount groups of lower priority from being mounted. This is useful if the agent needs to wait for a particular system state before it does anything. Checking for the `FruReady` entity to exist is a common use-case.

```

agentName: FooAgent

MountGroup: alwaysMount, 10
always/mounted, Foo::Bar, r

MountGroup: fruBlocker, 20
ConditionBlocker: FruReady in /%sysname/Sysdb/hardware/%cellPath.entryState

MountGroup: hardwareMounts1, 30
hardware/mount/1, Foo::Bar, r

MountGroup: hardwareMounts2, 40
hardware/mount/2, Foo::Bar, r

```

In this example, `FooAgent::alwaysMount` will always be mounted since its mount group has a higher priority than `FooAgent::fruBlocker`. On the other hand, `FooAgent::hardwareMounts1` and `FooAgent::hardwareMounts2` will both wait for the `FruReady` entity to exist as a child of `/<sysname>/Sysdb/hardware/cell/<cell ID>`. Notice that `FooAgent::fruBlocker` does not need any mount paths in it. It functions solely as a broad blocker for other mount groups.

alias

The `alias` keyword constructs new template keys from existing template keys or with a constant value. `alias` lines are evaluated in the order they are present in the mount profile. The new template key that an `alias` creates is only available in profile in which it's defined, as well as any included profiles (see [Include](#)).

`alias` creation supports four operators.

Assignment operator

The assignment operator simply assigns a new template key to the value on the RHS of the expression.

```

agentName: FooAgent-%genericSlice-%chip

alias: %slice=%genericSlice

```

In this example, `%slice` will expand to the same value as `%genericSlice`.

Comma operator

The comma operator concatenates two values with a hyphen separator. It can be repeated multiple times.

```

agentName: FooAgent-%genericSlice-%chip

alias: %slice=%genericSlice,%chip

```

Assume `%genericSlice` has the value `foo` and `%chip` has the value `1`. In this case, `%slice` would have the value `foo-1`. If `%chip` has no value (because the agent name was `FooAgent-foo`), then `%slice` would have a value of `foo`.

Tilde operator

If the RHS of the tilde operator is present in the LHS, then the tilde operator returns the value of the LHS. Otherwise, the tilde operator returns an empty string.

```
agentName: FooAgent-%genericSlice-%chip
```

```
alias: %slice=%genericSlice ~ %chip
```

Assume `%genericSlice` has the value `foobar` and `%chip` has the value `bar`. In this case, `%slice` would have the value `foobar`. However, if `%chip` instead has the value `baz`, `%slice` would expand to the empty string.

Ternary operator

The ternary operator functions like a C++ ternary operator. Given `a ? b : c`, the ternary operator returns `b` if `a` is not the empty string, otherwise returning `c`.

```
agentName: FooAgent-%genericSlice-%chip
```

```
alias: %slice=%chip ? %chip : %genericSlice
```

In this example, if `%chip` has a non-empty value, then `%slice` gets assigned the same value as `%chip`. Otherwise, `%slice` gets assigned the same value as `%genericSlice`.

connectionFlags

The `connectionFlags` keyword allows users to specify connection flags to modify the connection between the agent and Sysdb, affecting every mount. See [Connection flags](#) for more details on connection flags.

For example, including the following line in a mount profile will enable [scheduled AttrLog](#) on all mounts for this agent.

```
connectionFlags: N
```

rootFlags

Agents generally should not need to modify their root flags. If you are considering using this keyword, please contact `sysdb-dev` first.

The `rootFlags` keyword allows users to specify mount flags to use for the root mount. The root mount is the root of the entity tree that an agent mounts from Sysdb. In production, this is usually `/ar/Sysdb`. See [Mount flags](#) for more info on mount flags.

Agents mount the root with read-only permissions (`r` flag) by default. For example, including the following line in a mount profile will give the agent write permissions on the root.

```
rootFlags: w
```

subAgent

The `subAgent` keyword is exclusively for use in the `SuperServer` agent. If you are looking to run multiple agents in the same process, see [AID12178](#) instead.

The `subAgent` keyword indicates to the mount framework that the agent named in the `agentName` line is also running another agent, whose name the user specifies in the `subAgent` line. The mount framework loads that agent's mount profiles as well, ultimately merging the mount information with this agent's mount information.

noDefaultMountGroup

Agents should always mount the default paths. If you are considering using this keyword, please contact `sysdb-dev` first.

All agents mount a set of default paths from Sysdb (e.g. `agent/config`). Listing `noDefaultMountGroup` in a mount profile tells the mount framework not to include the default mount paths in the agent's mount info.

Adding a new mount profile

All (non-test) mount profiles reside in the `/src/<package>/SysdbMountProfiles` directory. If you are adding a new mount profile, create a file in that directory with contents specifying the agent's mounts. Mount profiles follow a general naming convention.

- Profiles for an agent should be named after the agent.
 - e.g. `FooAgent`'s mount profile should be named `FooAgent`.
- Profiles for a plugin should be of the format `<package>-<agent name>-plugin`.
 - e.g. The mount profile for `/src/StrataCentral/FruPlugin/StrataCentral.py` should be named `StrataCentral-Fru-plugin`.
- **Include files** should be of the format `<package>-<meta info>-include`.
 - e.g. `StrataCommon-FeatureAgent-include`.

Now add the new mount profile to the package's `BUILD.qb` by adding the following lines.

```
sysdb_mount_profiles(  
    # Notice: sources relative to ./SysdbMountProfiles  
    sources=[  
        <mount profile name>,  
    ],  
)
```

Finally, package the mount profile in an RPM by modifying `<package>.spec.ar`. Consider RPM dependencies when choosing an RPM. Ensure that the mount profile will be installed alongside the agent. It often makes sense to package mount profiles along with the agent binary itself. Users can specify a mount profile in the `<package>.spec.ar` file with the following.

```
%{_libdir}/SysdbMountProfiles/<mount profile name>
```

Preinit profiles

Preinit profiles are files containing the list of paths at which entities must be created in Sysdb. Preinit profiles are loaded on Sysdb boot. Every path in a mount profile must have an entry for the same path in a preinit profile.

Preinit profile format

Preinit profiles contain paths at which entities must be created in Sysdb.

The format of a preinit path is:

```
[CONFIG:] Path, TacType [, Attributes]
```

where [] indicates optional parameters.

Paths may also contain the template keys `%cellId` and `%peerCellId`, and their supersets `%cellPath` and `%peerCellPath`, which expand to `cell/%cellId` and `cell/%peerCellId` respectively. Intermediate paths are automatically created as mountable Tac::Dirs.

"**CONFIG:**" is used to identify the path as a config path (a path that participates in Config session and config replace operations).

Attributes are optional. Currently there are two supported attributes: `createOnStandby` and `createOnMount`.

Comment lines must begin with a `#`.

Sample Preinit Profile

```
eventMon/epochMarker,EventMon::EpochMarker
eventMon/status,EventMon::Status
eventMon/route6/status,EventMon::TableStatus
cell/%cellId/agent/commandRequest/config,Tac::Dir,createOnStandby

#Config paths begin here
CONFIG: eventMon/config, EventMon::Config
#I made up some of these paths and note no spaces between 'CONFIG' and ':'
```

createOnStandby attribute

The `createOnStandby` attribute indicates that the entities be **created for the standby supervisor** (if it doesn't exist) and mounted as writable between active and standby Sysdb. The `createOnStandby` attribute should be used if you mount a path with flag `f` in an agent - if the path on standby is meant to be read-only, this attribute is not required.

createOnMount attribute

`createOnMount` attribute indicates that the entity at the specified path should not be created now and instead created at a later time. In most cases, the entity is created at mount time rather than Sysdb startup time via a `c` flag in a mount profile. In other cases Agents, including Sysdb, use such entries for type information; their use will allow us to remove type information from both mount profiles and `doMount` calls in source code.

```
hardware/strata/nat/capability/config, StrataNat::PlatformConfig, createOnMount
```

These entries need to appear in files that are part of the same RPMs that contain the mount profiles with the mount to avoid dependency issues.

Tac::Dirs

Intermediate dirs need not be added to preinit profiles. However, all other paths that will ever be mounted (leaves of the tree), must be present in a preinit profile, including Tac::Dirs. The reason mountable Tac::Dirs need to be in the preinit profile is that, to be able to mount an entity, it must be present in its parent's "entry" collection.

All paths and intermediate paths in a preinit profile are considered as mount points.

Preinit profiles and dependency generation

Quote from [AID10](#) copied on July 27th, 2023:

If the file is a Sysdb mount profile (begins with `%{_libdir}/SysdbMountProfiles/`), if the corresponding path is in the global preinit mount path to RPMs mapping, and the number of RPMs is exactly one, add the RPM as the `Requires` for the current RPM. Paths inside conditional mountgroups are ignored (specifically, paths between a line starting with `Condition` and the next line starting with `MountGroup`).

Creating preinit profiles for new or existing packages

1. Create the preinit profiles in `/src/<package>/preinit/` with the format as explained above.
2. In `BUILD.qb`, add the following lines under the data targets section

```
sysdb_preinit_profiles(
    # Notice: sources relative to ./preinit
    sources=[
        'EventMon',
    ],
)
```

3. Add the files to **appropriate** rpm that needs the preinit profile in the `<package>.spec.ar` file

Maintaining the preinit profiles

The preinit profiles will need to be updated when additional entities need to be created in Sysdb or when entities move to different libraries.

Also, preinit profiles must be placed in the correct RPMs with regard to dependencies of the entity that needs to be created. There is no automatic dependency generation currently from the RPM containing the preinit profile to the RPM providing the type. If a package provides both the type and the preinit profile instantiating a path with that type, it is good practice to place both the preinit profile and the lib providing the type in the same RPM. The preinit profile could also be in an RPM downstream in the dependency tree.

There is a dependency generation mechanism currently between the preinit profile that provides a path and a mount profile that mounts the path. See: [AID10](#) for more details. Note that [AID10](#) also mentions that "the number of RPMs [containing the preinit] must be exactly one" for this dependency generation mechanism to work, so in scenarios where the same preinit path ends up in multiple RPMs, that defeats the automatic dependency generation mechanism. Note that `createOnMount` entries do not generate dependencies; they just communicate type information.

A tool is available to help cross-check both all the mount profiles and preinit files: `/usr/bin/tacMountProfileXCheck`. If not already present in your workspace, you can build and install this with `a4 make -p SysdbUtil`. This tool will detect mismatched types for the same path in mount profiles and preinit files, including implicitly declared intermediate `Tac:::Dirs` and wildcarded paths. In addition, it will detect missing or superfluous `createOnMount` entries. This script is run as part of an `EosImageTest` test during image creation scanning the files installed in `/usr/lib/SysdbMountProfiles` and `/usr/lib/preinit` directories. A more targeted description of the problems in various packages is generated when run against the source files like this:

```
/usr/bin/tacMountProfileXCheck -C
```

There are other options available; the help message (`-h`) contains additional explanatory text.

Loading specific preinit profiles in test

In some tests, you may want to specify which preinit profiles Sysdb loads, and subsequently which entities it creates (one example is in Launcher). In cohab tests where you directly instantiate `EntityManger.Local` as a stand-in for Sysdb, this can be done by passing a list of files and directories of preinits to load to the `preInitProfiles` argument.

```
# Paths can be to anywhere, /src is just used as an example
preinitProfiles = [
    '/src/<pkg>/test/testPreinits',
    '/src/<pkg>/test/extraPreinit1' ]
self.em_ = EntityManager.Local(
    sysname=self.sysname_,
    preInitProfiles=preinitProfiles)
```

In non-cohab tests, where you actually start Sysdb, preinits can be overridden through the `--preinitprofiles` option.

```
preinitProfiles = [
    '/src/<pkg>/test/testPreinits',
    '/src/<pkg>/test/extraPreinit1' ]
sysdbArgv = []
for path in preinitProfiles:
    sysdbArgv.append( f"--preinitprofiles={path}" )
Artest.startAgent(
    'Sysdb',
    self.sysname_,
    argv=sysdbArgv )
```

It is important to note that specifying preinit overrides will stop Sysdb from loading any other preinits, both any default preinits that Sysdb always loads and any preinits that are appropriate for your agent, like those you have in `/src/<pkg>/preinit`. If you intend to simply specify extra preinits to load for this test, you must also include the default preinit path as one of the overrides. The path is either `/usr/lib/preinit` or `/usr/lib64/preinit`, depending on the architecture. The easiest way to specify this in Python is using the `sysconfig` package.

```
import os
import sysconfig
defaultPreinits = os.path.join(
    sysconfig.get_config_var( 'LIBDIR' ),
    'preinit' )
preinitProfiles = [ '/path/to/extra/preinit', defaultPreinits ]
```

There is one implementation detail that's important for users to know. The preinit overrides are passed through the infra via the `SYSDB_EXTRA_PREINIT_PATHS` environment variable. Currently no tests use this variable and you should not try to set it yourself, otherwise it will be overwritten. The variable will be forcibly unset if you don't provide any preinit overrides.

FAQ

Is it a concern if the same path is listed in two preinit profiles?

No. The entities created in Sysdb are a union of all paths present in all preinit profiles. It may sometimes be necessary to list the same path in multiple profiles for dependency reasons, which is perfectly fine as long as the two entries have the same type and attribute (such as `createOnStandby`). An exception will be thrown if this is not the case.

How are these profiles related to the Sysdb mount profiles?

These profiles are complementary to Sysdb mount profiles. Preinit profiles create entities in Sysdb, while Sysdb mount profiles specify a particular agent's mounts.

References

- [AID8234](#) Sysdb Preinit Profiles

Active vs passive mount

An agent can either use active mounts or passive mounts. This is defined by the usage or omission of the `activeMount` keyword in the agent's mount profile (See [Mount profiles](#)). This is frequently shortened to describing an agent as active or passive.

We are trying to convert all agents to use active mounts so that we can deprecate the passive mount infrastructure in the future!

Active mounts

For an active agent, the agent must initiate the mount of any path it's interested in using a `doMount` API such as `doMountPath()` (See [doMount API](#)). An active agent is responsible for initiating the mounting of any path it needs.

Any active agent that has an associated mount profile operates in Auto mode. All flags/types in the old style `doMount()` calls are ignored since this info is automatically retrieved from the agent's mount profile. This is one reason why we prefer using the new `doMount` APIs which no longer have these redundant options (See [doMount API](#) for how to use the new `doMount` API). A test runs during build time to check the agent's mount profiles for syntax errors, as well as flag, path or type incompatibilities between different agents' mount profiles.

Active agents without a mount profile

Production agents should always be using a mount profile except for a very limited, narrow, specialized, hardcoded set of agents. Trying to create a production agent without a mount profile will also trigger an assertion! The following section should only be used by btests and stests where it's simpler to use a script to mount instead of an entire mount profile.

Sometimes an agent must be able to dynamically mount paths at runtime that are not present in a static mount profile. When there is no mount profile for an agent, the mount infra will use the flag and type info specified in `doMount()` calls. The enforcement for flag, path and type compatibilities is more basic than the build time check that runs for mount profiles. Agents without mount profiles should take precaution to avoid any conflicts with the types and flags of the paths they mount to avoid unexpected behavior.

Passive mounts

Passive mounts will be deprecated in the future. If your agent is using passive mounts,

please consider [converting](#) it to use active mounts. All new agents should use active mounts!

As soon as a passive agent connects to Sysdb, all the paths in the agent's mount profile are automatically mounted except for paths in mount groups with conditions that evaluate to false. These conditions are watched by Sysdb and the paths are mounted once they become true. This includes paths outside of a mount group and all mount groups without conditions. See [Condition and ConditionBlocker](#) for further semantics and information about conditional mount groups and their priorities. Passive agents still need to call `doMount()` to block until the mount completes and update bookkeeping data. Overall, passive mounts may seem like a more convenient approach for an agent but they have a few drawbacks:

- Passive agents often **over-mount** data that they do not actually need, leading to increased memory usage.
- Agents that connect to Sysdb and immediately perform a lengthy blocking task such as SDK initialization may find that they have trouble with `AttrLog` buffer overflows because all the non-conditional mount data arrives as soon as the agent connects.
- Mount group conditions don't allow for very much complexity compared to actively mounting in an agent's code.
 - What if you need to mount based on the state of a nominal or system memory usage?
 - What if you need dynamic mount group priorities or priorities that are more complex than an integer between 1 and 99?
 - What if you need to mount based on a gNMI Get request?
- Conditional mount groups move agent logic from the agent code into the mount profile, making it more difficult to understand agent behavior.

Converting from passive mount to active mount

1. Convert the mount profile under `/src/<package>/SysdbMountProfiles/<profile>` to have the `activeMount` keyword:

```
agentName: FireBreathingDragon
activeMount # Add this line
```

2. Directly call the `doMount` API to mount paths and mount groups when they're needed (See [doMount API](#) for examples). This includes converting any conditional mount groups to an SM, `if` statement during the agent's `doInit()`, or other agent logic that calls `doMountGroup()` at the appropriate time.
3. If you specifically have a `FruReady` conditionBlocker, replace it with an `include` for either `FruReadySlice` or `FruReadyCell` as appropriate. These includes will automatically induce the same `doInit()` waiting behavior as in the CAgent code.
4. If you have any paths with the `D` [mount flag](#) and/or your `doMount()` calls fail complaining about a missing DirMounter, see [AID10766](#). This is a necessary step whether or not you have converted to use the new `doMount` API, the `doMount` API just has an additional check for it in non-cohab tests.

5. Make sure your mount profile no longer has any **Condition** and **ConditionBlocker** statements since these are not allowed in an active mount profile and will trigger an assert during build time.

Tools

Managing mount profile files can become complex, particularly when agents include shared mount profile files. Some of the EOS agents mount more than 1300 paths; managing this manually can be difficult. There are some tools available in workspaces to help; these may be found in the Sysdb-devel rpm, which is installed by default in a workspace.

ShowProfile

Usage

```
showProfile [-p preinitdir] [-m mountprofiledir] -[jvh] agentString
```

| Option | Effect |
|--------|---|
| v | Verbose mode; print out other settings from profile files |
| h | Print help message |
| a | Merge flags as for active mount |
| j | Format output as JSON |
| p | Define a custom preinit directory |
| m | Define a custom mount profile directory |

Description

`showProfile`, given an agent string as argument, will load all the applicable profile files for that agent and will print each mount path that the agent can mount in the following space separated tabular form:

| field 1 | field 2 | field 3 | field 4 | fields 5-N |
|------------------|---------|---------|-------------|-------------------------|
| Mount group name | path | C++type | mount flags | file(s) containing path |

The `agentString` argument should be fully specified: `Fru` or `Strata-Linecard1`, for example.

Note that:

- `showProfile`'s operation is affected by the setting of the `A4PKG` environment variable.
- If the agent name contains templated values, the name of the template value chosen in the argument to `showProfile` doesn't matter; it only affects the expanded templated values seen in the output.
- Some paths are duplicated across multiple profile files; each file will listed for those paths.

whoMounts

Usage

```
whoMounts [-h] [-r] [-R REPLACEVARS] [-v] path [path ...]
```

Positional args:

| Positional | Description |
|------------|----------------------|
| path | Mount path to search |

Optional args:

| Option | Description |
|----------|--|
| h | show this help message and exit |
| r | Force rebuilding of cache |
| R | Replace variables with the provided value when generating the cache; forces cache rebuild. |
| v | Enable verbose output |

Example of **-R** usage:

Sample path from SandFabric mount profile:

```
agentName: SandFabric-%sliceId$  
...  
%cellPath/hardware/sand/slice/status/fe/%sliceId, Tac::Dir, fwci  
...
```

Without having built the cache with **-R**, if you just search the path directly, copying **%sliceId** as it appears in the mount profile path, there is no output:

```
% whoMounts %cellPath/hardware/sand/slice/status/fe/%sliceId
```

Instead you would have to search like so, making sure to substitute **%sliceId** from the path with however it was defined in the **agentName** (in this case, **%sliceId\$**), which can be frustrating and confusing:

```
% whoMounts %cellPath/hardware/sand/slice/status/fe/%sliceId$  
SandFabric-%sliceId$      cell/42/hardware/sand/slice/status/fe/%sliceId$      fiwc  
SandFabric
```

Preempt all this pain by using **-R** (note it will rebuild the cache, which takes a while):

```
% whoMounts -R 42 %cellPath/hardware/sand/slice/status/fe/42  
SandFabric-%sliceId$ cell/42/hardware/sand/slice/status/fe/42 fiwc SandFabric
```

After having built the cache with a `--replaceVars` value `X`, you don't need to specify the argument again on subsequent searches; note however you do need to substitute `X` for variables in your search paths that are not `%cellId` or `%cellPath`:

```
% whoMounts %cellPath/hardware/sand/slice/status/fe/$sliceId
% whoMounts %cellPath/hardware/sand/slice/status/fe/$sliceId$
% whoMounts %cellPath/hardware/sand/slice/status/fe/42
SandFabric-%sliceId$ cell/42/hardware/sand/slice/status/fe/42 fiwc SandFabric
```

Description

`whoMounts`, given one or more paths as arguments, will output the list of agents that mount these paths in the following space separated tabular form:

| field 1 | field 2 | field 3 | fields 4-N |
|------------|---------|-------------|--------------------------------------|
| Agent Name | path | mount flags | profile file(s) containing this path |

Note that:

- Even though a path appears in an agent's mount profile(s), it may never be mounted. For passive mount agents any requisite conditions may never be satisfied, and for active mount agents, they may never invoke the mount API to perform the mount.
- `whoMounts` works using all the installed profile files to find all possible agents mounting these paths; as a result it will ignore `A4PKG` environment variable if set.
- Any templated agent names will appear in the output with the templates unexpanded; e.g Strata-%slice, etc.
- Variables like `%cellId` or `%sliceId` are automatically expanded according to how `showProfile` does it: `%cellId` -> `42`, `%cellPath` -> `cell/42`; for the others, try using the `-R` arg, otherwise, good luck

If you get no matches, note:

- automatically expanded variables: `%cellId` -> `42`, `%cellPath` -> `cell/42`
- if your path includes variables besides those, they probably appear in `showProfile` output with BOTH the prefix `%` and suffix `$`, e.g. `%sliceId$`, even though it may be written in the mount profile with just the `%`.
- beware of aliases in the mount profiles
- if all else fails try using the `-R` argument

Local Backup mount

A path mounted with the Local Backup flag (`L`) will have two special properties:

1. The data stored there will be *private* to the agent that performed the mount. No data will end up in Sysdb. This would make it "Local".
2. The stored data will be present after agent restart. Even when the agent unmounts, quits or crashes, all previously stored state is promised to be recoverable. Data is only deleted when the agent overwrites it, or when Sysdb restarts. This contributes the "Backup" part - as it is mostly persistent.

Intended purpose

Local Backup is meant to be an easy solution for the problems which are associated with crashing agents. When an agent is restarted - for any reason - it starts with a clean state. The state an agent was sharing to Sysdb will still be present after mounting, except for data that was in the process of being written when the agent crashed. However, any local data kept by the agent will be lost. Although agents could use Sysdb to store their local state as well, the extra computational overhead or the exposure of the internal state can make it an unappealing option. Local Backup mounts are intended for exactly this purpose. The overhead is smaller and the state is kept private.

Note that the stored data is not encrypted, and thus confidential data must not be stored in Local Backup mounts. To this end, "Tac::Sensitive" types should not be written to Local Backup mounts. The names of the attributes and their state can be easily guessed just by looking at the contents of the files that Backup uses for the copies.

Consistency

Entities written to the persistent store are consistent and "transactional" at the attribute level. Should the agent crash while updating an attribute, on reboot and mount the attribute will either have the old value or the new value. It is guaranteed to never have a combination of the two.

This is useful if two values need to change at the same time. The transactional nature of attribute updates can be leveraged in the case when two values need to be updated together. Making both values part of a `Nominal` ensures that a restarting agent sees either both the old values or both the new ones - never a blend.

Usage

The TACC Framework considers a mount as a Local Backup mount if both:

1. The path starts with the `/backup` prefix.
2. The `L` mount flag is supplied.

Entities mounted with Local Backup must be declared with the `localBackupRoot` type property. Since Local Backup relies on S4Sync for serialization and notification support, all types written to backup must also be declared with `s4Sync`.

Local Backup mounts do not create objects in Sysdb, so should be omitted from mount profiles.

Example

```
MyNom : Tac::Type : Tac::Nominal {
    // MyNom will be written to backup, so must be S4Syncable.
    `s4Sync;
    a : U32;
}

MyType : Tac::Type( a ) : Tac::Entity {
    `localBackupRoot;
    `s4Sync;
    a : Tac::String;
    b : MyNom;
}
```

```
Tac::Ptr< Sysdb::MountGroup > mg = EntityManager->getMountGroup( 0 );
// note the "/backup" prefix and the 'L' flag
mg->doMount( Sysdb::Mount( "/backup/Example", "MyType", "L" ) );
```

Limitations

- Local Backup mounts - just as Sysdb mounts - create implicit copies of the data. This has an impact on the memory usage. Furthermore, not only the actual data is copied, but some bookkeeping information - some metadata - is also stored along with it.
- The persistent store won't survive a reboot. A system restart will purge all locally stored state for all the agents. There is no way to recover the data after a reboot.
- A Local Backup mount is always synchronous, and should only be done in a mount group by itself or with other Local Backup mounts.
- Local Backup currently offers no support for queues or sensitive types.

Clock notifyees

`ClockNotifiee` is a Python class that runs some code periodically. The handler will be executed when it is time to run, which is determined by setting `timeMin`. `ClockNotifiee` keeps a weak reference to the handler provided.

Here is a simple example of `ClockNotifiee` usage:

```
import Tac

def doneActivity( n ):
    print( 'Done activity', n )

clockNotifiee = Tac.ClockNotifiee( handler=lambda: doneActivity( 0 ) )

print( 'About to run activities' )
Tac.runActivities( 0.0 )
print( 'Done running activities' )
```

Note that just instantiating a `Tac.ClockNotifiee` object is enough to register the notifiee for later execution. There is no need to explicitly add the notifiee to a queue. Note that this is single-threaded code, so execution does not occur until the call to `Tac.runActivities()`, which passes control to the main tacc event loop. It is rarely necessary to have multiple threads in our Python code, and where you do use threads you should be careful to ensure that only one thread enters the event loop at a time.

The argument to `Tac.runActivities` is the number of seconds that should be spent in the event loop before returning control to the caller. In this case, executing for 0.0 seconds is a common idiom that means that any activities already due (or overdue) for execution should be executed, but the call will not block and wait for future activities.

You can also set a `ClockNotifiee` for execution in the future by setting the `timeMin` field on the `ClockNotifiee`:

```
import Tac

def doneActivity( n ):
    print( 'Done activity', n )

clockNotifiee1 = Tac.ClockNotifiee( handler=lambda: doneActivity( 1 ) )
clockNotifiee2 = Tac.ClockNotifiee( handler=lambda: doneActivity( 2 ) )
clockNotifiee2.timeMin = Tac.now() + 2

print( 'About to run activities' )
Tac.runActivities( 0.0 )
print( 'Done running activities' )

print( 'Running activities for 2 seconds' )
Tac.runActivities( 2 )
print( 'Done running activities' )
```

Tasks and the TaskScheduler

Tasks are an abstraction for running code asynchronously. They work by giving running code a "quantum", or unit of time, to run for, and then "requiring" they yield back to the tac activity loop. The "requirement" to yield is provided by the `taskShouldYield` function.

They are part of the cooperative multi-tasking structure of tacc based agents, but provide superior functionality to the raw tacc constructs like `Tac::Clock` and `Tac::FileDescriptor`, as well as other alternatives like `HTimerWheel`.

Tasks support design patterns like `CODEF` and `WAIT`, and can be used alongside a variety of helper code like:

- `Ark::taskWalkCollection`
- `Ark::processBacklogTask`
- `AttrLogTaskGroup`

The full documentation for this feature can be found at:

[Task Scheduler for Scalable and High Performance TACC / C++ Agents](#)

Other prior documentation can be found in a variety of locations:

Task based Codefs

Helper functions for simpler and faster backlog processing and yielding collection walks (Note this one is better used via the compiler builtins mentioned in Task based Codefs above)

On top of the tasks themselves, the TaskScheduler provides a scheduling system for prioritizing tasks, as well as debug tooling like CLI statistics.

Task flavors

Tasks generally come in 3 different flavors:

- Asynchronous but with no specific delay (code runs when possible, with no delay)
- Timer triggered (code runs after a specified delay)
- File descriptor triggered (read/write triggered)

These three generally map to:

- `Ark::Task`
- `Ark::TimerTask`
- `Ark::FileDescriptorTask`

Task

Tasks generally take 3 constructor arguments, a scheduler which should be passed into the state machine and shared throughout the agent, a priority, and unique task name (mainly used for identification in the CLI output). A common way to make a task name unique is via an identifier like intfld, vrfld, or so on. If you include a sequence number, pointer value, or something else that is "unbounded" and can result in a large number of unique names in case of flaps etc, then please also use taskStatsNames("NameWithSequenceEtcRemoved"); to avoid leaking task names.

A state machine that wants to defer processing (i.e. one with a codef) might look something like:

```
TaskSm : Tac::Type( scheduler ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    task : in Ark::Task = initially (
        scheduler, Ark::TaskPriority::normal, "MyTask" );

    handleTask : extern invasive void();

    // when the deferred processing is triggered, taskRun changes
    // We use skipOnEnclosingCreate to not trigger when the task is initialized
    when task::taskRun with skipOnEnclosingCreate => handleTask();
}
```

The task can be scheduled by calling the `scheduleTask` method.

TimerTask

A timer task is almost identical to a Task, but also provides ways to schedule with an explicit delay, in addition to the base `scheduleTask` (A Task is really a TimerTask that always uses a delay of zero).

The Sm is nearly identical:

```
TimerTaskSm : Tac::Type( scheduler ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    timerTask : in Ark::TimerTask = initially (
        scheduler, Ark::TaskPriority::normal, "MyTimerTask" );

    handleTask : extern invasive void();

    // when the deferred processing is triggered, taskRun changes
    // We use skipOnEnclosingCreate to not trigger when the task is initialized
    when timerTask::taskRun with skipOnEnclosingCreate => handleTask();
}
```

Primary methods:

- `scheduleTaskDelay` -- schedules the task so that it will run after the delay. If there is an existing scheduling, this is overwritten.
- `scheduleTaskLowestDelay` -- schedules the task so that it runs the sooner of the current delay, and the new delay.

FileDescriptorTask

For a file descriptor task, we schedule based on `read` or `writer` information of the associated file descriptor. If the file descriptor is a socket, for example, this could then be used to implement a simple HTTP server, or so on.

By default, a `FileDescriptorTask` automatically watches read, but does not watch write.

This can be controlled via the `taskwatchReadable` and `taskwatchWritable` attributes. (Note that `taskwatchWritable` will set itself to false after triggering -- read the class itself for more info).

The task otherwise provides two trigger attributes, `taskRunReadable` and `taskRunWritable`. Note that errors will generally also trigger `taskRunReadable`.

Example SM:

```
FdTaskSm : Tac::Type( scheduler, fd ) : Tac::Constrainer {
    scheduler : Ark::TaskScheduler::Ptr;
    fd : int;
    fdTask : in Ark::FileDescriptorTask = initially (
        scheduler, Ark::TaskPriority::normal, "MyTimerTask", fd );

    handleRead : extern invasive void();
    handleWrite : extern invasive void();

    // when the deferred processing is triggered, taskRun changes
    // We use skipOnEnclosingCreate to not trigger when the task is initialized
    when fdTask::taskRunReadable with skipOnEnclosingCreate => handleRead();
    when fdTask::taskRunWritable with skipOnEnclosingCreate => handleWrite();
}
```

TaskScheduler

The `TaskScheduler` provides a way to hierarchically prioritize different Tasks, and is otherwise too complicated to cover fully here.

We will instead discuss `TaskGroups` briefly, and the CLI integration.

TaskGroups

A `TaskGroup` is a `TaskScheduler` that lives hierarchically underneath a parent `TaskScheduler`. This means that all the Tasks in that group share a specific quantum from the parent scheduler, allowing a group of tasks to avoid starving the rest of the system.

The `TaskGroup` can also disable a group of tasks explicitly, which is most usefully shown via the [AttrLogTaskGroup](#).

This is a specific `TaskGroup` that will stop running tasks when the attrlog buffers get full, allowing an agent to stall noisy tasks until attrlog is processed.

TaskScheduler CLI

Agents have a builtin show command of the following form:

```
show agent <agent> task scheduler [hierarchical]
```

For example:

show agent IpRib task scheduler
TaskScheduler stats for IpRib-main

| Priority | RunnableCount | IdleCount | DoneCount | TotalRunCount |
|----------|---------------|-----------|-----------|---------------|
| critical | 0 | 0 | 0 | 0 |
| absolute | 0 | 0 | 0 | 0 |
| high | 0 | 0 | 0 | 0 |
| normal | 0 | 342 | 6 | 8567 |
| low | 0 | 11 | 0 | 23 |
| scavenge | 0 | 5 | 0 | 6 |

Maximum scheduler slip 0.050086 seconds

SP = Task status and priority % two characters:
 first T/t for active/idle timers, * for running task, blank or
 S for suspended tasks, X for tasks under suspended
 scheduler
 second priority and runnable status, upper case for runnable
 RunCount = Number of times task has run
 TotalSec = Total run time in seconds
 LastDelay = Scheduling delay last time task ran, in seconds
 MaxDelay = Maximum scheduling delay encountered, in seconds
 200% 500% = Number of times task used this percentage of quantum
 MaxMs = Maximum single run time in milliseconds
 Done = Number of times a task with this name has completed
 TotalWorkCount = Counter of total work units processed
 DecayAvgWorkCount = exponential decaying average of work units processed
 MaxWorkCount = high watermark of work units processed in one run
 LastWorkPending = Work units pending processing
 MaxWorkPending = high watermark of work units pending processing

| TaskName | SP | RunCount |
|--|-------|----------|
| TotalSec | | |
| Last/Max-Delay | | |
| 200% | | |
| 500% | | |
| MaxMs | | |
| Total/DecayA/Max-WorkCount | | |
| Last/Max-WorkPending | | |
| Smash-P=RN5B-route:DrainCallback | n | 409 |
| 0.309 0.000 0.000 0 0 50 | 10000 | 11.7 |
| - | 1883 | - |
| TaskGroup;SharedMem | n | 3051 |
| 0.242 0.000 0.022 0 0 2 | 8580 | 3.9 |
| 1 | 8 | 0 |
| SmashCrcsm-fd21:MessageHandler:fd:21:read | n | 782 |
| 0.022 0.000 0.007 0 0 15 | 784 | 1.0 |
| 0 | 2 | 0 |
| SmashCrcsm-fd22:MessageHandler:fd:22:read | n | 714 |
| 0.015 0.000 0.000 0 0 7 | 717 | 1.0 |
| 0 | 2 | 0 |
| SmashCrcsm-fd24:MessageHandler:fd:24:read | n | 799 |
| 0.011 0.000 0.022 0 0 1 | 806 | 1.0 |
| 0 | 2 | 0 |
| SmashCrcsm-fd20:MessageHandler:fd:20:read | n | 784 |
| 0.009 0.000 0.022 0 0 0 | 796 | 1.0 |
| 0 | 2 | 0 |
| SmashCrcsm-fd25:MessageHandler:fd:25:read | n | 800 |
| 0.008 0.000 0.007 0 0 0 | 801 | 1.0 |
| 0 | 2 | 0 |
| SmashCrcsm-fd26:MessageHandler:fd:26:read | n | 800 |
| 0.007 0.000 0.007 0 0 0 | 801 | 1.0 |
| 0 | 2 | 0 |
| SmashScsm-0x7f0f235ffb40:MessageHandler:fd:31:read | n | 410 |
| 0.006 0.000 0.000 0 0 0 | 410 | 1.0 |
| 0 | 1 | 0 |
| SmashRemountTimerMgr-5225 | Tn | 1 |
| 0.001 0.000 0.000 0 0 0 | 55 | 55.0 |
| 0 | 55 | 0 |
| Smash-o45w4C-via:DrainCallback | n | 1 |

| | | | | | | | | | | |
|--|-------|-------|---|---|---|--|-------|---------|-------|---|
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 1 | 1.0 | 1 | - |
| - | | | | | | | | | | |
| InotifyFd:read | | | | | | | | | n | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| SlabEmptyPageScanner | | | | | | | | Ts | | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 25 | 13.7 | 14 | - |
| - | | | | | | | | | | |
| Smash-VHrYfA-viaSet:DrainCallback | | | | | | | | n | 1 | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 1 | 1.0 | 1 | - |
| - | | | | | | | | | | |
| mallocTrimmer | | | | | | | | Ts | | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 32768 | 16384.0 | 16384 | - |
| - | | | | | | | | | | |
| NhLoopBackLog | | | | | | | | tl | | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 1 | 0.9 | 1 | - |
| - | | | | | | | | | | |
| SmashCrsn-fd29:MessageHandler:fd:29:read | | | | | | | | n | | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 2 | 1.0 | 1 | 0 |
| 0 | | | | | | | | | | |
| SlabPageMadvise | | | | | | | | Ts | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 30 | 30.0 | 30 | - |
| - | | | | | | | | | | |
| SmashCrsn-fd13:MessageHandler:fd:13:read | | | | | | | | n | | 2 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 2 | 1.0 | 1 | 0 |
| 0 | | | | | | | | | | |
| CompletedTaskCheckTask | | | | | | | | ts | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| ViaSet | | | | | | | | tn | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 1 | 1.0 | 1 | 0 |
| 0 | | | | | | | | | | |
| Shark-gBo_5A-viaMetricStatus:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| LoopingNh | | | | | | | | tn | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 0 | 0.0 | 0 | - |
| - | | | | | | | | | | |
| ViaSet:Route | | | | | | | | tn | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 1 | 1.0 | 1 | 0 |
| 0 | | | | | | | | | | |
| Shark-jmU5bC-viaMetricStatus:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| TaskGroup;RcfAetPreprocess | | | | | | | | n | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 2 | 2.0 | 2 | 0 |
| 0 | | | | | | | | | | |
| Shark-OoONJA-pending:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| Shark-LxNXFB-pending:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| Shark-hD5TBC-filteredRibStatus:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| Shark-4k793B-iprib:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| Shark-wyuDWB-colorToEndpoints:ShmallocLimboTimer-5225 | | | | | | | | Tl | | 3 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | - | - | - | - |
| - | | | | | | | | | | |
| TaskGroup;RcfSymbolUpdate | | | | | | | | n | | 1 |
| 0.000 | 0.000 | 0.000 | 0 | 0 | 0 | | 2 | 2.0 | 2 | 0 |
| 0 | | | | | | | | | | |

| TaskGroup;AttrLogTaskGroup | n |
|--------------------------------|---------|
| 0.000 0.000 0.000 0 0 | 2 2.0 2 |
| 0 | 1 |
| Skipped 133 tasks with no runs | 0 |

This can be extremely useful for debugging CPU issues, unexpected work, and so on.

Tracing

The EOS Framework provides a tracing mechanism to facilitate output of formatted debugging messages for agents and tests. Tracing output is not intended for customer consumption; syslog should be used for customer-directed messages.

Normally, all tracing output is disabled for agents. By default tracing output is directed to the stderr of the process; for agents the stdout is redirected to log files in `/var/log/agents/{agentName}-{pid}` by ProcMgr when the agent is started.

A typical default line of trace output looks like this:

```
2023-10-24 12:01:13.042506 3927 EventHistory 1 Writer::eventIs() some output here
```

Some things to note:

- The time is formatted as yyyy-mm-dd hh:mm:ss; this makes the time numerically sortable so that log files from different agents may be sorted together to understand their interactions.
- The first non-time stamp number (3927 in the example above) after the time stamp is the tid (same as pid for single-threaded processes) of the tracing process.
- The facility name (here EventHistory) comes next; this is how different sets of trace points are enabled and disabled as a group.
- The last portion of the fixed format output (the 1 in the example above) represents the tracing level; usually, higher levels are used for more detail. There are 10 trace levels, 0 through 9. The uses to which these levels are put varies with different agents.
- The remainder of the line is free format.

Controlling tracing

Tracing can be enabled for tests using environment variables, and for agents can be enabled or disabled using the Cli once the agent is running normally; this takes effect without restarting the agent. Tracing agent startup is either done by hard-coding the desired tracing settings into a debug version of the agent, or by modifying the agent command line to set the environment variables with `/bin/ProcMgrModifyCmdline` and restarting the agent. See below for how to trace during EOS bootup.

Disabled traces have very little performance impact, particularly in C++.

Environment variables that affect tracing

TRACE

The `TRACE` environment variable enables tracing. Each desired facility and level can be specified in comma separated clauses. For example,

```
export TRACE=foo/0,bar/125,baz*/0,bang/2-5,boom/*
```

enables tracing for

- Facility "foo", level 0
- Facility "bar", levels 1, 2 and 5
- All facilities starting with baz, level 0.
- Facility "bang", levels 2 through 5.
- Facility "boom", all trace levels.

Setting `TRACE` to / will usually generate far too much output to be useful.

`TRACEFILE`

It is sometimes difficult to capture the stderr, especially in some isolation cases. In this case, trace output can be re-directed by setting `TRACEFILE` to point to the desired output file name. Note that if the file path is not writable, the program will fail.

`TRACEFORMAT`

It is possible to modify the default trace output shown above using the `TRACEFORMAT` environment variable. See [aid86](#) for details. This should not be set except by end users.

Using the Cli to control tracing

Note that the Cli-controlled tracing is not applied until the agent has largely completed its mounts, since each agents gets the desired traces information from Sysdb mounts. If it is desireable to trace agent behavior before it has completed its mounts, `/bin/ProcMgrModifyCmdline` must be used to set the `TRACE` environment variable.

In the following table, the word `agent` in the command syntax is the name of any running agent on the system, and `facility` is the name of a trace facility.

| Command syntax | Mode | Description |
|--|--------|---|
| trace <code>agent</code> setting TRACE | config | Set trace facility/levels in the TRACE environment variable format. |
| no trace <code>agent</code> setting | config | Clear all trace settings for the agent. |
| [no] trace <code>agent</code> enable <code>facility</code> (levels level+ all) | config | Enables or disables one or more trace |

| Command syntax | Mode | Description |
|---|--------|--|
| | | levels for a given facility. |
| trace agent filename filename | config | Enables tracing to the specified file. |
| no trace agent filename [filename] | config | Disables tracing to file, and re-enables tracing to stderr. |
| show trace agent | enable | Shows trace settings for an agent. |
| trace monitor agent | enable | Outputs trace from the specified agent to the current CLI session. Press Ctrl-C to stop. |

Tracing during EOS boot

It is sometime necessary to trace one or more agents startup during EOS boot. The easiest way of doing this is to create a `/mnt/flash/rc.eos` file that invokes `/bin/ProcMgrModifyCmdline` once for each agent; this script will be run early on before ProcMgr starts so that all enabled agent traces will be captured from startup in their respective log files. No error messages are captured or displayed, so testing the script before rebooting is suggested. The resultant changes will appear in `/var/run/ProcMgrModifyCmdline`.

A sample script that enables tracing in Fru including startup might thus be:

```
#!/bin/bash
# the line above is necessary.
#
/bin/ProcMgrModifyCmdline add Fru environ "TRACE=Fru/*"
```

Writing tracing statements

Trace statements can be written in Python and C++.

For both language, you'll need a trace handle, which explicitly or implicitly names the facility associated with the trace statements.

Tracing in Python

```

#!/usr/bin/env python3
# Copyright (c) 2024 Arista Networks, Inc. All rights reserved.
# Arista Networks, Inc. Confidential and Proprietary.
import Tracing

# If TRACE is set to TracingExample/0-2, you'll see output from this code
# as the default name of the trace facility is the basename of the file.
traceHandle = Tracing.defaultTraceHandle()
# Alternatively, we could set the trace handle like this:
# defaultTraceHandle = Tracing.Handle( 'MyTraceFacility' )

t0 = traceHandle.trace0
t1 = traceHandle.trace1
t2 = traceHandle.trace2
t3 = traceHandle.trace3

t0( "level 0 would normally be used for fundamental messages" )
t1( "level 1 could offer a more detailed view" )

# We can use a decorator to automatically trace entrance and exit
# to a routine.

@Tracing.traceEntryExit( level=2 )
def debugRoutine():
    return "it works!"

# We can also check if a tracing level is enabled if we need to conditionally
# run code outside a trace statement itself. This is a good idea if your
# arguments to the trace function are expensive to compute, since they're
# always evaluated even if the trace is disabled.

if traceHandle.enabled( 2 ):
    debugRoutine() # call debugging code
else:
    print( " tracing at level 2 is not enabled" )

# Example of why the enabled function is useful
def tattle():
    print( "tattle was called" )

t0( "this string won't appear if level 3 is not enabled, but the " +
    f"the output from {tattle()} will always appear even if tracing is
disabled" )

# For more fun with Python tracing, read /src/tacc/Fwk/Python/lib/Tracing.py

```

Tracing in C++

There are two tracing APIs for C++; they both use the same underlying mechanisms, and both rely on variadic macros to skip argument evaluation if the specified tracing level is not enabled.

The first example uses the legacy APIs; these rely on C++'s iostream mechanism, so are not particularly efficient.

```
#include <Tac/Tracing.h>
#include <stdlib.h>
// Select a facility name for trace statements in this lexical scope
DEFAULT_TRACE_HANDLE("Example");
class TracingExamples {
public:
    static void example1();
};

void
TracingExamples::example1() {
    // A simple trace statement.
    TRACE0( __PRETTY_FUNCTION__ << "example of tracing at level 0 from pid "
            << getpid() );
    // Demonstrate how to check if tracing is enabled...
    if ( defaultTraceHandle.enabled( 1 ) ) {
        fprintf( stderr, " Level 1 is enabled \n" );
    }
}
int
main() {
    // Force-enable tracing for our example
    setenv( "TRACE", "Example/*", 1 );
    Tac::TraceFacilityMan::Ptr tfm =
        Tac::Entity::singleton< Tac::TraceFacilityMan >();
    tfm->doReadEnvironment();
    TracingExamples::example1();
    exit( 0 );
}
```

The second C++ tracing example uses the newer FMTRACE mechanisms; these avoid the inefficiencies and often awkward syntax of streamio. Instead, this framework uses the C++20 std::format syntax. There are lots of additional features for formating available with this newer tracing implementation; see [AID9560](#) for more details.

```

#include <TacMarco/FmtraceDebug.h>
#include <stdlib.h>
// Select a facility name for trace statements in this lexical scope
DEFAULT_TRACE_HANDLE("Example");
class TracingExamples {
    // Need FMT_CLASS macro since this is not a `TAC` generated class;
    // see AID9560 for details.
    FMT_CLASS( "TracingExamples" );
public:
    static void example2();
};

void
TracingExamples::example2() {
    // A simple trace statement.
    FMTRACE0( "example of tracing at level 0 from pid {} ", getpid() );
    // Demonstrate how to check if tracing is enabled...
    if ( defaultTraceHandle.enabled( 1 ) ) {
        fprintf( stderr, " Level 1 is enabled \n" );
    }
}
int
main() {
    // Force-enable tracing for our example
    setenv( "TRACE", "Example/*", 1 );
    Tac::TraceFacilityMan::Ptr tfm =
        Tac::Entity::singleton< Tac::TraceFacilityMan >();
    tfm->doReadEnvironment();
    TracingExamples::example2();
    exit( 0 );
}

```

Writing useful trace statements

Tracing is used throughout EOS to help developers in multiple ways, including:

- Tracing helps developers understand the code via reading, similarly to comments.
- Examining tracing output allow developers to follow the flow of execution during debugging.

Good tracing tells a story about what is happening to the agent or test. If parts are left out, or transitions are not noted, the developer who is not familiar with the code (or the 'story' being told) will be confused.

For agents, tracing is generally disabled in normal execution, except for perhaps the most basic state: "mounts complete", for example. It is somewhat awkward to enable tracing for agents during startup, however, since any tracing enabled through the Cli doesn't take effect until the agent is warm; developers are left to modify ProcMgr command lines or Launcher profiles to address this. Enabling some tracing by default during startup eases debugging as a result, and since agents are not starting repeatedly (hopefully, anyway), we need not worry about excessive log file growth.

For tests, the practice varies. Since test output is cached and then discarded for successful tests, but included when the test has failed, the writer prefers to enable tracing in tests by default so that build logs with failing tests are useful in debugging. This is particularly important for intermittent failures, since they are sometimes difficult to reproduce outside the Abuild

environment. It is most useful if the test can have additional tracing readily selected via the **TRACE** environment variable to permit developers to easily expand the scope of their investigations into test failures.

One easy way of doing this is to use

`Marco::Platform::setupMarcoDebug(const char * traces)` to set the desired test tracing. This function will add the specified trace facilities and levels to any already specified in the environment.

Tac::Dir

WIP - This chapter is still being developed

`Tac::Dir` is a fundamental TACC framework type that is used to represent a filesystem-style directory. This chapter refers to the type itself as `Tac::Dir` and objects of the type as "Dirs".

Functions of a Tac::Dir

Dirs are useful for containing and organizing `Tac::Entity` objects and essential for performing mounts. Dirs are mountable and can be nested/contained in other Dirs. Dirs allow the creation of an object tree structure and also allow mounting some or all parts of the tree unlike any other object deriving from `Tac::Entity`. Dirs also contain special synchronization mechanisms and interact with the rest of the TACC framework to achieve all of its functions.

Structure of a Tac::Dir

A `Tac::Dir` type contains just two user-facing collection attributes - `entityPtr` and `entryState`.

entityPtr

The `entityPtr` collection represents the set of *synchronized*¹ entities contained within the `Tac::Dir`. The `entityPtr` collection can be accessed like any collection defined in the Tac language. i.e:

- `entityPtr("name")` to lookup an entity in the collection
- `entityPtrIterator()` to create an iterator beginning from the start of the collection etc.

In addition to the regular collection accessors, a template accessor function `entityPtr<T>("name")` is also available.

Note that deletion of entities from this collection is not allowed. To delete entities, please refer to [deleteEntity\(\)](#)

Python support for `entityPtr` collection is limited. See [Python support for Tac::Dir](#)

entryState

The `entryState` collection is less useful for most applications. It is a collection of `Tac::Dir::Entry` objects which contains the `name`, `typeName` and other Dir implementation details for each entity that is contained within the Dir. Note that, `entryState` is a superset of the

`entityPtr` collection. i.e: it contains all `Tac::Dir::Entry` associated with both synchronized¹ and unsynchronized entities.

The `entryState` collection can be accessed like any collection using accessors and iterators.

Python support for `entryState` collection is limited.

APIs for interacting with `Tac::Dir`

The main user-facing `Tac::Dir` APIs are:

| API | Function |
|----------------------------------|----------------------|
| <code>createEntity()</code> | Create an entity |
| <code>entityPtr()</code> | Lookup an entity |
| <code>deleteEntity()</code> | Delete an entity |
| <code>deleteAllEntities()</code> | Deletes all entities |

An advanced set of APIs for more fine grain control of `Tac::Dir` are:

| API | Function |
|---------------------------|---|
| <code>entryState()</code> | Lookup the <code>entryState</code> collection |
| <code>syncEntity()</code> | Synchronize an entity ¹ |

`createEntity()`

The `createEntity` API is used to create an entity with a Dir. Its two forms are:

```
myDir->createEntity( "Foo::MyType", "myEntityName" );
```

and

```
myDir->createEntity< Foo::MyType >( "myEntityName" );
```

Both forms are useful in different places. The template form offers better typing support whereas the non-template form allows for reducing unnecessary type dependencies.

Return type: *** WIP ***

Behavior:

- performs no action if an entity is already present with the same `name` and `typeName`.
- throws `NameInUseException` if an entity is already present with the same `name` but a different `typeName`.
- throws `TypeError` if the `Tac::Type` associated with `typeName` is not found.

deleteEntity()

The deleteEntity API removes an entity from the Dir.

```
myDir->deleteEntity( "myEntityName" );
```

Return type: *** WIP ***

Behavior:

- throws no errors if an entity is not present.
- performs no action if an entity is not present in the `entryState` collection.

deleteAllEntities()

The deleteAllEntities API removes all entities from the Dir.

```
myDir->deleteAllEntities();
```

Return type: `void`

syncEntity()

The syncEntity API synchronizes¹ a specific unsynchronized entity that already exists under the `Tac::Dir`. (i.e: it must be in the `entryState` collection)

```
myDir->syncEntity( "myNewEntity" );
```

Return type: `void`

Behavior:

- performs no action if an entity is not present in the `entryState` collection.
- throws no errors if an entity is not present in the `entryState` collection.
- performs no action if the entity is already synchronized.¹

Role of `Tac::Dir` in Mounting

WIP

Python support for `Tac::Dir`

WIP

Dirs have special behavior in python. Dirs appear as collections and can be iterated over as collections. The underlying `entityPtr` collection is reflected into the `Tac::Dir` as a collection. i.e: All synchronized¹ entities within a Dir are members of the `Tac::Dir` collection, can be iterated over, show up in `items()`, `keys()`, etc.

Entities within Dirs can also be accessed like collection elements:

```
>>> Tac.root[ 'myDir' ][ 'mySubDir' ]  
entity('/myDir/mySubDir')
```

¹ When an entity is mounted from a peer (remote process), the underlying mechanism creates a local object (on the heap) that is of the same type as the remote object, and ensures every attribute of that object is the same as it is on the peer process. This mechanism is referred to here as "synchronization". Once complete, the object is considered "synchronized".

Tac::Range

`Tac::Range<T>` expresses the finite set of allowed integer-like values of type `T`, which must be known at compile-time.

The set of values is defined based on its lower and upper bounds.

Applicable TAC types automatically work with `Tac::Range`.

TAC

Integral user-defined TAC types with statically-known bounds support `Tac::Range`. Two such types are non-invasive [enumerations](#) and split-inheritance types.

Range.tac

```
Shape : Tac::Enum {
    circle;
    rectangle;
    triangle;
}

Rank : Tac::Type( value ) : Tac::Ordinal, U8 {
    value {
        `range = 1 .. 10;
    }
}
```

C++ API

```
namespace Tac {

template < typename T >
class Range;

}
```

Minimum and maximum

The minimum and maximum values in the range are available at compile-time:

```
#include "Range.h"

int main() {
    using ShapeRange = Tac::Range< Shape >;
    assert( ShapeRange::min() == circle() );
    assert( ShapeRange::max() == triangle() );

    using RankRange = Tac::Range< Rank >;
    assert( RankRange::min() == Rank( 1 ) );
    assert( RankRange::max() == Rank( 10 ) );

    // The bounds are available at compile-time.
    using FirstShape = std::integral_constant< Shape, ShapeRange::min() >;
    assert( FirstShape() == circle() );
}
```

Iterating through a range

`Tac::Range` has forward- and reverse-iterators compatible with the C++ standard library.

```
#include "Range.h"

int main() {
    std::cout << "The shapes are:\n";
    for ( Shape shape : Tac::Range< Shape >() ) {
        std::cout << shape << '\n';
    }

    // Print the highest ranks first.
    std::cout << "\nThe ranks are:\n";
    Tac::Range< Rank > ranks;

    for ( auto iter = ranks.rbegin(); iter != ranks.rend(); ++iter ) {
        std::cout << *iter << '\n';
    }
}
```

Other documentation

Older documentation for tacc was scattered around lots of different places.

We intend for this book to be the only necessary source for information about tacc.

However, it will take time to consolidate the content from those older documents into the book. Until we finish, this section will include links to other useful documentation.

FAQ

[AID/49](#) is an older overview of tacc, structured as a series of questions and answers.

Some of the information is outdated but it's worth reading.

How to contribute

Our goal for this book is to be a comprehensive and helpful resource for all things tacc.

Your help is very welcome, and valuable!

You may [suggest an improvement](#) or help [developing the book](#).

Suggesting improvements

One of the easiest ways to help is to help us identify improvements to the book itself.

- Did you find anything incorrect, or missing?
- Does a topic need more explanation?
- Is an example unclear, or would more examples be helpful?

Contacting us

Please reach out to us in one of the following ways, or [file a bug](#):

- post in the "Tacc" category on [Discourse](#)
- Message the "tacc Support" GHC space
- email tacc-dev@

Filing a bug

Bug reports help us to track issues so that they don't get forgotten.

All bugs should be filed against the [TaccBook](#) package.

Bugs may be filed for any issue with the book, big or small. Issues could be:

- Missing chapters
- Unclear content
- Spelling and grammatical errors
- Problems with the supporting infrastructure (like the snippet system)

Developing the book

The source code of the book is hosted in the `TaccBook` package in Perforce. You need an `a4c` container to modify and build the package.

Building the book

The book source is composed of two kinds of files:

- Markdown files for the book's content (with the `.md` extension)
- Code files in various languages

In your `a4c` container, run the following command to build the book:

```
$ a ws mk -p TaccBook
```

As part of the build, TAC and C++ sources are compiled into a shared library. This step makes sure that the example code stays up-to-date with compiler and framework changes.

The build also invokes the `mdbook` tool to render the Markdown files to HTML. Apart from the Markdown syntax, this step also validates the links pointing to different parts of the book.

Running tests

The package also contains C++ and Python tests, which can be run using the following command:

```
$ a ws mk -p TaccBook check
```

Viewing and sharing the rendered HTML

The previous build step produces HTML pages under the build directory, inside `/bld/TaccBook/mdbook-out/html`.

To share your local version of the book (for example, as part of a code review) copy the generated HTML files to the `public_html` directory:

```
$ /src/TaccBook/Book/src/publish-preview.sh
```

Then, the book will be published to `http://usercontent/~$USER/taccbook-<subdir>` if you have CHD enabled, and `http://<your-user-server-name>/~<user-name>/taccbook-<subdir>` regardless of CHD. `<subdir>` is defined by the name of the current p4 project or git topic, depending on container type.

Observing the website for incremental changes

From within the `/src/TaccBook` directory, build and serve the HTML files:

```
$ mdbook serve ./Book -d /bld/TaccBook/mdbook-out --hostname $HOSTNAME
```

The server will run in the foreground and print the IP address and port it's listening on. You can view the generated website in your browser.

If you keep the `serve` instance running, it will watch the book source code for changes, automatically rebuild the book, and refresh the browser.

Writing conventions

This chapter describes conventions that authors should follow when adding new material to the book.

Semantic line-breaks

All narrative content in the book is written with [semantic line-breaks](#).

That is, a "hard" new-line is inserted in the Markdown file after punctuation like commas or periods, or at the natural break-points between ideas.

The reason for doing this is that it makes it easier to review changes to the content: most tooling is line-oriented.

For example,

```
It is very important to eat vegetables.  
There are so many to choose from,  
and with an enormous variety of different shapes and colors!
```

Markdown formatting

Sections

A top-level section title is written like this:

```
# My important thesis #
```

and a sub-section title is written like this

```
## Introduction ##
```

The number of `#` characters corresponds to deeper sub-sections.

Note that `#` characters trail the section name as well.

Section capitalization

Section titles are written in "sentence case", in the same way that one would capitalize a sentence.

That is, prefer

```
# On the importance of writing tacc #
```

to

On the Importance of Writing tacc

Quoting in section titles

If text would normally be back-quoted then it should also be quoted in a section title.

For example,

Understanding the `Tac::Dir` type

Unordered lists

- My first list item is very important.
It's quite long
- My second list item is a little bit shorter

Ordered lists

The list item numbers are automatically incremented appropriately when the Markdown document is rendered to HTML.

1. Mix the ingredients
1. Put the pan in the oven
1. It's time for cake!

How to write snippets

This chapter describes how to write and structure snippets while contributing to the book.

In general, code shown in snippets should be compiled and/or executed during a build of the TaccBook package. This is one way of ensuring the book doesn't get outdated.

A snippet is defined in Markdown like this:

```
```py
def square(n):
 return n * n
```

```

This is a Python snippet. If the language is indicated then the book will syntax-highlight the snippet automatically.

These are the languages supported in this book:

| Label | Language |
|-------|----------|
| py | Python |
| tac | TAC |
| cpp | C++ |

Including files in snippets

Snippets are most often *extracted* from source files that exist alongside the Markdown file for a chapter.

The `#include` directive injects the contents of a named file into a snippet.

For example, given this directory containing a chapter describing "Thing"

```
Section/
  Thing.cpp
  Thing.md
  Thing.py
```

then

```
```cpp
{{#include Thing.cpp}}
```

```

will include the entire contents of `Thing.cpp` in the snippet and syntax-highlight it.

Extracting parts of files with anchors

Frequently, a single file is divided into several snippets using "anchors".

An anchor is given a name and indicated in a file like

```
class Component;

// ANCHOR: square_function
inline int square( int x ) {
    return x * x;
}
// ANCHOR_END: square_function

double sqrt( double );
```

The snippet

```
```cpp
{{#include Thing.cpp:square_function}}
```
```

will be rendered as

```
inline int square( int x ) {
    return x * x;
}
```

Interactive snippets

C++ and TAC snippets may be made interactive.

A TAC snippet may be uploaded to [Code Explorer](#). An upload button is rendered when you specify the `action=upload` option in the code fence info string as shown below:

```
```tac, action=upload
Person : Tac::Type(name, age) : Tac::Nominal {
 name : Tac::String;
 age : U32;
}
```
```

A C++ snippet may be uploaded to Code Explorer the same way.

A C++ snippet may also include the code generated by a TAC snippet, use generated types in a function, and upload the complete code on Code Explorer. Furthermore, if the snippet has a `main` function, it may be executed on Code Explorer.

To link a C++ snippet with a TAC snippet, first you need to provide an identifier to the TAC snippet using the `id` option. A custom header filename is used as an identifier in this book as shown in the example below:

```
```tac, action=upload, id=Person.h, label=Person.tac
Person : Tac::Type(name, age) : Tac::Nominal {
 name : Tac::String;
 age : U32;
}
```

```

The `label` is optional. It gets printed at the top of the snippet in the rendered book.

With this setup, a C++ snippet may include the generated code from "Person.h" as shown below:

```
```cpp, action='run,upload'
#include "Person.h"

void greet(Person const& person) {
 std::cout << "Hello, " << person.name;
}

void main() {
 Person joe("Joe", 30);
 greet(joe);
}
```

```

Note how the `action` is provided two comma separated values within quotes. `run` causes a play button to be rendered which executes the main function in Code Explorer. `upload` causes an upload button to be rendered.

The interactive features of a snippet can be disabled by marking it with `ignore`:

```
```cpp,ignore
int square(double);
```

```

Exercising snippets

In addition to being compiled, it's important that snippets are actually executed as part of a build of TaccBook.

The easiest way to do this is to invoke snippet code in breadth tests.

For example, the models defined in `Book/src/Attribute/Collection/Vector.md` are exercised in examples defined in `Vector.tin`. Those examples are executed in `test/AttributeTest.cpp`.

Python source files may be executed directly as breadth tests, as in `Book/src/Attribute/Collection/Vector.py`.