COSI: Adding Constraints to the object-oriented paradigm

Gary Curtis, Mark Giuliano

Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218

I. Overview

Trans [1] is a Lisp system at the Space Telescope Science Institute (STScI) which is a key part of the proposal preparation ground system for the Hubble Space Telescope (HST). It was originally developed in the late ‘80s using a mixture of procedural, and blackboard architectures. While the original application met its requirements and performed well for a number of years, the increasing complexity of the system and its changing role meant that it eventually needed to be re engineered.

In developing a replacement we wanted a mechanism to manage the complex inter-dependencies between application objects in a dynamic system. To meet this need we developed the Constraint Sequencing Infrastructure (COSI) which supports a model of programming with methods as constraints. COSI automatically tracks dependencies between object state and constraints, manages the relationships between objects, and executes constraints when necessary to ensure that the internal state of the system is consistent.

COSI is implemented as a general purpose extension to CLOS via the Metaobject Protocol (MOP) [2]. It extends the object-oriented paradigm to add the notion of methods as constraints which can automatically rerun when their inputs change.

II. Motivation

Trans was originally designed as a batch system, taking as input a proposal specification and producing a detailed sequence of telescope activities that will achieve the specified observations. A proposal specification describes the observations that the user wants to perform in a relatively high level language and from this description, Trans produces a detailed schedule of spacecraft activities that will perform and support those observations. There is typically considerable flexibility in the details of this schedule and Trans attempts to find a solution that will make the most efficient use of the telescope.

Internally Trans used a scripted approach to flow of control with a predefined sequence of steps intended to reach a final solution. Each step in such a sequence depends upon the results of proceeding steps to provide their input. This is fine when the sequence of dependencies is relatively static, but it becomes extremely difficult to maintain when the dependencies change. If new requirements make a calculation dependent upon a later step in the sequence then the script must be rearranged and it proved extremely difficult to determine the correct order for the script due to the complexity of all the dependencies.

The nature of the HST, and the ground system that supports its operation, is one of constant change. Trans is constantly being updated and enhanced to adapt to changes on the spacecraft (both planned and unplanned) and to maintain and update the capabilities provided by the ground system. This environment of continual change made Trans difficult to maintain with its existing

architecture and we decided that we needed an architecture that would assist us in managing the dependencies in the system.

While Trans has historically been a batch system, its role is expanding and it will soon become a component of an interactive graphical system. This system will allow a proposal to be developed incrementally which will therefore require Trans to build its solutions incrementally. In addition, Trans must be able to make efficient updates its internal state in responses to changes to existing inputs.

The architecture that we developed solves both of these problems by automatically tracking the dependencies in the system and by rerunning code when necessary to maintain an internally consistent state.

III. Concept: Methods as Constraints

i) Constraint Sequencing

In order to address our concerns about the difficulty of managing code dependencies, and method sequencing within a complex dynamic system we developed the notion of "methods as constraints" which allows the system itself to manage these issues.

A constraint specifies some relationship between input and output parameters. In other words, it constrains its output to specific legal values based upon its input. Constraints are declarative and have an underlying propagation engine that applies the constraints to the current state of the system. If the inputs to a constraint change then the effect of that constraint must be propagated to its outputs, changing those outputs, which may in turn be inputs to other constraints.

Consider, for example, three values: X, Y and Z where constraint C1 implies Y = X + 10 and constraint C2 implies Z = Y + 5. If the value of X changes then C1 must be applied which will change the value of Y. As a result, C2 must be applied to change the value of Z. This propagation continues until the state is consistent.

In an object-oriented system such as Trans, many of the methods resemble constraints, but with no underlying propagation mechanism. Methods are used in such a system to create objects and to assign values to the slots of those objects and so on. The behavior of those methods depends upon the objects and slots to which they refer, but in order to keep the system consistent when its inputs change there must be explicit calls to methods that depend upon those input values.

In such a system it can be difficult to manage the dynamic nature of the objects and their interdependencies. Decisions can impact parameters already computed for objects which has a ripple effect requiring anything derived from that value to be recomputed. Once the system is required to be interactive, or to be able to search for better solutions then the need to respond to change is even more clear.

Normally the dependencies between objects must be handled explicitly by the programmer, but by viewing methods as constraints we can build a propagation engine that will automatically track the dynamic dependencies between objects and constraints and which can therefore propagate the effect of a change throughout the system by rerunning constraints as necessary. This simplifies the programming task by making the code more declarative and self contained.

As a simple example from the HST domain, suppose that we define a class **exposure** with two slots: **filter**, and **duration**, and that there exists a mechanism for traversing a sequence of these exposures:

```
(defclass Exposure ()
  ((filter   :initform clear :reader Get-Filter)
   (duration                 :reader Get-Duration))
  (:metaclass Constraint-Sequencer))

(defconstraint Set-Duration ((self Exposure))
  (setf (slot-value self 'Duration)
    (case filter
      (filter1   10)
      (filter2   15)
      (filter3   20))))
```

The **Set-Duration** constraint is a method that can set the **duration** of an exposure to a value that is consistent with the **filter** for the exposure. If the **filter** for the exposure can change, either through a change to the input value, or through an internal decision by the system, then the **Set-Duration** method must be rerun to ensure that the **duration** of the exposure is correct. This method is actually defined to be a constraint and as a result the underlying constraint mechanism is responsible for detecting dependencies and propagating changes through the system. In this case the **Set-Duration** constraint for a specific instance of exposure is dependent upon the **filter** slot of that instance. If the **filter** slot should change then the infrastructure is responsible for rerunning the dependent constraints for that instance.

We believe that this provides a simpler, cleaner way of expressing the relationship between objects and object attributes. The notion of constraint makes the system more declarative by expressing the requirement upon the **duration** of an exposure in a way that is independent of any flow of control. The underlying propagation mechanism is responsible for detecting the dependency of the constraint on related objects and attributes, and for running the constraint when necessary to ensure that all objects have consistent state.

A common mechanism that is analogous to this approach is the spreadsheet. When a user constructs a spreadsheet he enters values into some of the cells, and formula into others. Those formula constrain the value of their associated cell, based upon the values in the cells to which they refer. If a change is made to an input value, or to a formula, then the change is propagated throughout the spreadsheet. Within this model of method as constraint, the constraints are like the formula of a spreadsheet and object slots are like the data cells.

ii) Object Management

Using the notion of method as constraint leads to a system which can automatically track the object state that code depends upon, and can rerun the code when that state changes. This implies, however, that constraints may run an arbitrary number of times. A key to containing propagation is to detect *change* so the system must recognize when a constraint changes some state and when it does not.

Recognizing changes to object slots is straightforward, but many constraints will also create new objects and in subsequent runs may create the same, additional, or fewer objects. When objects are instantiated by a subsequent run of a constraint it is desirable to have the instantiation return the same object rather than a new one.

This is achieved through an object creation mechanism that uses a unique key to identify the object to be created. The underlying mechanism maintains a table of known objects, associated with unique keys, and when a request is received for a specific object the system can determine if that object already exists. If so, then the existing object is returned, otherwise a new object is created. If a constraint reruns, therefore, and recreates the same object as the previous run then there will be no net change to the internal state.

Although constraints may recreate the same objects, it is also possible that they will not recreate objects that were created with the previous run. In this case the system should detect the change and remove the superfluous object(s) for the sake of efficiency.

Extending our HST example, let us suppose that longer exposures require additional calibration activities. Specifically, exposures that have a duration greater than 15 sec. require a calibration activity to immediately follow them:

```
(defconstraint Create-Required-Calibration ((self Exposure))
  (when (> (Get-Duration self) 15)
    (Set-Next self
      (ensure-instance
        'Calibration ()))))
```

In this case we have a new constraint that depends upon the duration of the exposure. As the duration changes this constraint will need to rerun to determine whether or not the calibration is required. Successive runs of this constraint may or may not create the calibration. The use of the **ensure-instance** form means that each time this code creates the calibration it is same instance that is returned, and if the code does not create the calibration then the object is "cleaned up".

iii) Relationship Management

Most of the relationships that we need to model in Trans are bidirectional and in order to manage the consistency of such bidirectional links in a constraint based programming model we developed the notion of relationships. Extending our HST example again, let us assume that calibrations should be linked to any exposure that they could be used to calibrate. While exposures that are greater than 15 sec. duration require a calibration, other exposures may use a subsequent calibration if one has already been created.

```
(defconstraint Calibrates-Relationship ((self Calibration))
  (let ((exposure (Find-Exposures-To-Calibrate self)))
    (Set-Relationship self 'Calibration-For exposures)))
```

Relationships are inherently bidirectional and have arbitrary cardinality. This means that relationships form a link between two sets of objects, either of which may be a list or a single object, and the creation of a link is seen by objects in both sets. In the example above, the constraint is creating a relationship called **calibration-for** from the instance of a calibration to all of the exposures that it can calibrate. The two directions of a relationship have different semantics and therefore they have different names with neither direction having any particular significance or precedence. In this case the reverse relationship might be called **calibrates** and as a result, each exposure in the relationship will now have its **calibrates** relationship pointing to the calibration object.

This mechanism goes a long way towards managing the internal consistency of a set of object relationships as the bidirectional links will be kept consistent. Within a constraint based model, however, subsequent runs of a constraint may set up a relationship in a different manner than a

prior run and in this case the system must maintain the consistency of the relationships as those relationships change.

Suppose in our example that an exposure in the middle of a long sequence now has its filter changed, causing it to become longer and therefore requiring a calibration. The constraints will insert that calibration into the sequence which would make the new calibration the correct calibration activity for some of the earlier exposures. When the **calibrates-relationships** constraint runs for the new calibration it will set the **calibration-for** relationship between that calibration and the suitable prior exposures. In order to maintain consistency in the relationships, the exposures which are now linked to the new calibration must be unlinked from the old one. As a result, the **calibrates** relationship for the early exposures points to the new calibration while the **calibration-for** relationship for the original calibration no longer points to those early exposures.

IV. COSI Implementation: Adding Constraints to CLOS

COSI is implemented using the CLOS Metaobject Protocol (MOP) which allowed us to build this capability as a direct extension to the standard object-oriented programming model rather than as a separate layer on top of that model. This is important as it enables the system to detect slot access and change through all of the built in functions, methods and macros.

The MOP provides developers with a mechanism to specialize the behavior of CLOS itself. This is possible because the elements of CLOS, including classes and methods, are themselves objects instantiated from classes that may be specialized and extended. Classes, for instances, are objects instantiated from a metaclass (i.e. a class that describes a class) and this allows a developer to subclass the default metaclass and to use the new metaclass for his application classes. The behavior of CLOS is implemented via methods on metaclasses, so new metaclasses can extend or specialize this behavior.

The components of COSI build on the capabilities of each other with the constraint sequencing requiring the object manager, and the object manager requiring the relationship manager. Our implementation of the relationship manager will be therefore be presented first, followed by the object manager and finally the constraint sequencer.

i) Relationship Manager

Relationships

Relationships are bidirectional links between objects whose integrity is automatically managed by the Relationship Manager. The role of the Relationship Manager is mainly to act as the repository of relationship data. A relationship is defined by the following attributes:

A name for each ‘direction’ of the relationship
A default value for each direction (i.e. nil or unbound).

There are two major components to the Relationship Manager, a new **relationship-class** metaclass, and the **relationship-manager** itself. The combination of these components manage the relationships between objects.

Relationship Class

A new metaclass is defined as a sub-class to **standard-class** called **relationship-class** which will

provide any class instantiated from this metaclass with the ability to use relationships with its instances:

```
(defclass Relationship-Class (Standard-Class)
  ())
```

Standard-class is the default metaclass for CLOS classes and by adding the **relationship-class** subclass we are creating a way to specialize the behavior of specific classes.

The **relationship-class** metaclass provides two things to the classes that are instantiated from it. Those classes all have a **relationships** slot which is initialized with a hash-table, and methods are defined for that implement the protocol used to cooperate in the management of relationships. The relationships hash-table provides a storage mechanism based upon keyed look-up by relationship name which means that classes do not have to specify which relationships they will participate in.

While this capability could clearly have been written without resorting to the MOP, the fact that we needed to hook into the relationships in the same way that we hook into slot access, made this approach consistent with the needs of the constraint sequencer. The use of a protocol that delegates the work to the class of the object makes the relationship manager extensible via the MOP.

Define a Relationship

Relationships are defined using the **defrelationship** macro which includes the name of the relationship and the name for the reverse relationship e.g.:

```
(defrelationship forward-name reverse-name
  &key :forward-defaults-to-unbound
       :reverse-defaults-to-unbound))
```

The relationship manager maintains the table of defined relationships which provides the mapping between the two names. This is implemented via two entries into a hash table, one for each name with the name as the key. The value in this hash table is the reverse name and the default value of the relationship. The default value for a relationship may be either nil or unbound.

Create a Relationship

A relationship between objects is created with the **set-relationship** method which takes the objects that are related and the relationship name as arguments:

```
(set-relationship objects relationship objects)
```

Where *objects* may be a single object, or a list of objects. This method calls the relationship manager method **set-relationship-using-rm** with any single object parameter as a singleton list. The relationship manager is responsible for managing the bidirectional nature of the relationship and the many-to-many nature of any links:

```
(defmethod Set-Relationship-Using-RM
           ((self Relationship-Manager)
           objects relationship related-objects)
  (let ((reverse-relationship
          (Reverse-Relationship self relationship)))
    (dolist (object objects)
```

```
      (Set-Single-Relationship-Using-RM
        self object relationship related-objects))
    (dolist (relative related-objects)
      (Set-Single-Relationship-Using-RM
        self relative reverse-relationship objects))))
```

The relationship manager looks up the reverse relationship name and then iterates over each list of objects. For each object the method **set-single-relationship-using-rm** removes the object from any relationship it is currently in of that name, and then calls **set-relationship-using-class** on the class of the object to actually record the new relationship:

```
(defmethod Set-Relationship-Using-Class
           ((self Relationship-Class)
           object relationship related-objects)
  (setf (gethash relationship
          (Get-Relationship-Table object))
        related-objects))
```

Note that this is a method on the **relationship-class** metaclass. Instances of classes of this meta-class all have a relationship table that stores the relationships for the object.

Accessing Relationships

A relationship is accessed using **get-relationship** which takes the object and relationship name as arguments:

```
(get-relationship object relationship)
```

This method first calls **get-relationship-using-rm** which looks up the default value for the relationship:

```
(defmethod Get-Relationship-Using-RM
           ((self Relationship-Manager) object relationship)
  (with-slots (default-to-unbound-table) self
    (Get-Relationship-Using-Class
     (clos::std-instance-class object) object relationship
     (gethash relationship default-to-unbound-table))))
```

This method is responsible for looking up the default value for the relationship, for finding the class of the object and for calling **get-relationship-using-class** on that class. This method looks up the related objects from the **relationships** hash table in the object using the relationship name as key. If there is no entry in the hash table then the default behavior is used which either returns nil, or throws an unbound-slot condition:

```
(defmethod Get-Relationship-Using-Class
           ((self Relationship-Class)
           object relationship default-to-unbound)
  (multiple-value-bind (value boundp)
      (gethash relationship (Get-Relationship-Table object))
    (cond (boundp value)
          (default-to-unbound (error 'unbound-slot))
          (t nil))))
```

Note again that this is a method on the **relationship-class** metaclass.

ii) Object Manager

The object manager handles the creation and deletion of objects based upon a unique key within a specific defined context, also identified by a unique key. It also uses relationships to track which objects create other objects with a parent/child relationship.

Object Manager functionality is supported by a pair of macros: **ensuring-instances** and **ensure-instance**. The former is used to define the context for a set of objects created together, while the second is used to create a single object within that context.

Object creation context is necessary to define the scope within which an associated set of objects are created. If objects were previously instantiated within a context and during a subsequent run any of those objects are not recreated, then the system must clean up the objects. The most important clean up action is to remove such objects from any relationships in which they participate.

Object Creation Context

Ensuring-instances is the public interface to the object manager which defines the context for the creation of a group of objects. This context is important as it is used to determine when previously created objects must be deleted. Objects that are created on one pass through this context, but which are not recreated on the next pass are deleted.

An object creation context is defined by an object table and a parent object. **Ensuring-instances** is used to define the scope of that context and may be passed an optional key or parent. Either may be explicitly specified or left implicit. If no key is given then a unique key is constructed during macro expansion. If no parent is defined then the objects created in this context will have no parent:

```
(defmacro Ensuring-Instances ((&key key parent) &body body)
  `(apply #'values
    (unwind-protect
      (progn
        (Start-Object-Creation
          ,(or key `',(gensym "OM-Key")) ,parent)
        (multiple-value-list (progn ,@BODY)))
      (Stop-Object-Creation))))
```

The key is used to determine which object table applies in this context. The object manager maintains a hash table which stores all of the object tables, referenced by their associated key. Start-object-creation uses this key to create a new object creation context and push it onto a stack:

```
(defun Start-Object-Creation (key parent)
  (let* ((table (or (gethash key *OBJECT-TABLES*)
                    (make-hash-table :test #'equalp)))
         (current-parent (or parent (default-parent)))
         (context (make-instance 'object-creation-context
                    :current-object-table table
                    :current-parent       current-parent)))
    (push-object-creation-context context)
    (setf (gethash key *OBJECT-TABLES*) current-table)))
```

The object creation context contains a table of objects, and the parent for any new objects. **Stop-object-creation** is used to perform clean up of any objects that are no longer required, and to then pop the context off the stack.

Object Creation

Ensure-instance is similar to the CLOS make-instance in syntax and is intended to be its analog. The only addition is an optional key which should uniquely identify the occurrence of **ensure-instance** within the current object creation context. If no key is supplied then the macro creates a unique key, during macro expansion. This key will be adequate if the **ensure-instance** form will only created one object in any context but this is often not the case and a key will need to be specified. If the call is within any looping construct it will be used a number of times at each call will be expected to create a different object.

```
(defmacro Ensure-Instance (type (&key key) &rest initargs)
  `(Ensure-Object
    ,type
    ,(or key `',(gensym "EI-Key"))
    ,@initargs))
```

Ensure-object takes the given key and uses it as a reference into the object table associated with the current object creation context. If an object has previously been created with the key in this context then the object will be in the table and that object will be returned. If no object is found then a new instance is created and put into the table. In either case, any initialization parameters are applied to the instance which may, or may not, change slot values for an existing object:

```
(defun Ensure-Object (type key initargs)
  (let* ((table (Get-Table *CURRENT-CONTEXT*))
         (object (gethash key table)))
    (cond (object
           (change-class object type)
           (apply #'Initialize-Instance object initargs))
          (t
           (setq object
             (apply #'Make-Instance type initargs))))
    (setf (gethash key table) object)
    (add-new-object object)
    object))
```

The **add-new-object** call records which objects are created while a specific context is current which allows the system to clean up superfluous objects.

Object Deletion

Deletion means different things to different classes and we therefore defined a **delete-object** method which calls **delete-object-using-class** with the class of the object:

```
(defmethod Delete-Object ((self Standard-Object))
  (Delete-Object-Using-Class
    (clos::std-instance-class self) self))
```

The default implementation of **delete-object-using-class**, for **standard-class**, does very little, but is extended for classes of the **relationship-class** to remove the object from any relationships it

currently participates in.

iii) Constraint Sequencing

In order to support our constraint based programming model, COSI dynamically monitors the execution of constraints. It detects slot access and slot changes by specializing the associated methods for a new metaclass which is defined as a subclass of **relationship-class**:

```
(defclass Constraint-Sequencer (Relationship-Class)
  ())
```

This new metaclass is used to extend the default behavior of slot read and write methods and also to add two COSI specific slots to instances of classes of **constraint-sequencer**. These slots are called **dependencies** and **invocations** and they are used by COSI to record constraint invocations for an object, and to record which slots those invocations are dependent upon.

Defining Constraints

Constraints are defined using the **defconstraint** macro which has a similar syntax to **defmethod** but is restricted to a single parameter which must be typed:

```
(defmacro DefConstraint (name parameter &body body)
   `(defmethod ,name (,(first parameter))
     (push (get-invocation ,(caar parameter) ',name)
           *CALL-STACK*)
     (catch 'constraint-exit
       (handler-bind
           ((error #'Constraint-Condition-Handler))
         (Ensuring-Instances
           (:key (first *CALL-STACK*)
            :parent ,(caar parameter))
           (progn ,@body))))
     (pop *CALL-STACK*)
     nil))
```

This macro adds functionality around the body of the method itself. Prior to the execution of the body of the method it obtains an invocation for this constraint from the object itself, and pushes that invocation onto a stack. After the body of the method is complete this invocation is popped off of the stack.

The macro also adds a condition handler around the body of the method and sets up an object creation context for the constraint invocation.

Invocations

An invocation is the application of a constraint to a specific object. COSI has invocation instances that are created the first time a constraint is run for an object. The objects themselves are used to store their invocations in a hash table, indexed by constraint name. The **get-invocation** method is used to check for an entry in this table. If no invocation is found one is created and put into the table.

Condition Handler

Our constraint model required a slot value for ‘unknown’ when there was no suitable default value and we felt that leaving those slots unbound was a reasonable way to record that. Rather than use an ad-hoc mechanism for deferring constraints until the necessary data is available we use a condition handler to trap conditions of type unbound-slot that occur within a constraint. The purpose of the handler is simply to ignore the condition and exit the constraint normally. The fact that the current invocation depends upon a slot which happens to be unbound will have been recorded and if a value is ever assigned to the slot then the invocation will be queued to rerun.

Dependencies

An invocation is dependent upon a slot of the object if, during its execution, it accessed that slot. It is assumed that such an access effects the behavior of the constraint during that invocation and that the invocation should therefore be rerun if the slot changes. Each object has an associated hash table, indexed by slot name, which records invocations that are dependent upon that slot.

Creating Dependencies

Application level slot accesses are based upon the **slot-value** and **slot-boundp** functions which call the **slot-value-using-class** and **slot-boundp-using-class** methods on the class of the object. Access to relationships is through calls to **get-relationship-using-class** and **relationship-boundp-using-class** on the class of the object. By adding additional behavior to these methods we detect all slot and relationship accesses:

```
(defmethod clos:slot-value-using-class :before
           ((class Constraint-Sequencer) instance slot-name)
  (setq slot-name (clos:slot-definition-name slot-name))
  (Create-New-Dependency instance slot-name))
```

The :*before* methods that are added to these four methods all do the same thing. They are passed the object and the slot or relationship that is being accessed, and they create a new dependency between the slot and the current invocation on the stack. Two references are created, one from the object to the invocation in the objects dependency hash table, and another from the invocation to the object. The reference from the invocation to the object is used to clean up superfluous dependencies when an invocation is rerun.

Detecting Change

Application level slot modification is based upon the **(setf slot-value)** and **slot-makunbound** functions which call the **(setf slot-value-using-class)** and **slot-makunbound-using-class** methods on the class of the object. Access to relationships is through calls to **set-relationship-using-class** and **relationship-makunbound-using-class** on the class of the object. By adding additional behavior to these methods we detect all changes to the slots and relationships of an object:

```
(defmethod (setf clos:slot-value-using-class) :before
           (value (class Constraint-Sequencer)
            instance slot-name)
  (setq slot-name (clos:slot-definition-name slot-name))
  (unless (and (slot-boundp instance slot-name)
               (eq value (clos::slot-value--si instance
                                               slot-name)))
```

```
(Add-Invocations-To-Queue
  (gethash slot-name
    (clos::slot-value--si instance
                          'dependencies)))))
```

The :*before* methods that are added to these four methods all do the same thing. They are passed the object, the slot or relationship that is being modified and for two of them the new value for that slot/relationship. The :*before* methods compare the new value with the current value of the slot/relationship and if the new value represents a change then the method looks up the invocations that depend upon that slot/relationship, and adds those invocation to an invocation queue.

Propagation

When a change is detected which has dependent invocations, those invocations are placed in a queue. Invocations are queued rather than run immediately for efficiency. In order for the system to become internally consistent all of the invocations on the queue must be rerun through the propagation process.

Propagation of changes involves popping invocations from the queue and running those invocations:

```
(defun Propagate ()
  (do () ((Empty-P *INVOCATION-QUEUE*))
    (run (Pop-From-Queue *INVOCATION-QUEUE*))))
```

This process repeats until there are no invocations left on the queue at which point the model is consistent. Note that invocations may be added to the queue as constraints run due to changes made to the values of slots and relationships by prior invocations. This requires that we flag invocations when they are added to the queue, providing an efficient mechanism to avoiding duplicate invocations on the queue.

Querying the system

COSI delays propagation until as late as possible in order to reduce unnecessary constraint invocations. Specifically, propagation is delayed until it is required to satisfy a query of the system. In order to manage this process a defquery macro is provided for defining the external interface to the COSI application. This macro should be used when defining methods that require the system to be internally consistent as it adds a call to the propagation prior to running the body of the method.

V. Experience with COSI.

COSI was developed in response to specific issues with the re engineering of a key Hubble Space Telescope ground system. Development of the new system has progressed well and we currently feel that the infrastructure is functioning well.

There are, however, some limitations with our implementation of the constraint mechanism which do not significantly limit its utility, but which developers need to be aware of to make effective use of the system. The most significant of these is the fact that COSI only detects accesses and changes to the slots of objects, and as a result, modifications to other complex data structure such

as lists or arrays will not be detected as dependencies or changes. This requires care from the developers to ensure that changes to such structures are properly detected.

Currently propagation is all or nothing. When the system is queried by an external user it must propagate fully before responding to the query to be sure that its internal state is consistent. The forward propagation mechanism cannot tell whether some subset of the constraints would be sufficient to ensure the consistency of a particular value or not. This can mean that an initial query of the system appears slow as full propagation will occur at that time.

The constraint-oriented approach places additional semantics into the object model that developers must become comfortable with. Performance is something that must be considered from a slightly different perspective. Constraints should be kept as concise as possible to minimize the amount of work that will be done in response to a change. This encourages good programming style from developers who keep their constraints focused.

It is possible to write constraints with circular dependancies which introduces the possibility of unexpected infinite loops, but which also provides a powerful mechanism for converging to a solution. It is simple to detect the existence of circular dependencies, but it is not (in general) possible to distiguish between cases that will converge and those that will not. It is therefore left to individual developers to ensure that such loops will terminate.

The use of the MOP has important implications for performance. By specializing the methods that form the slot access protocol we make it harder for the system to optimize that access. In addition, there is simply more work to do when a slot is accessed or changed. While performance is a concern for us it is not our primary concern and the benefits of simplified maintenance outweigh the performance concerns. In addition, the system is efficient in determining what needs to be rerun when a slot is changed and the use of propagation defers those computations for as long as possible. So while the infrastructure is relatively inefficient at low levels due to the overhead of slot access, it is relatively efficient at high levels due to its ability to only rerun what is necessary, when it is necessary.

VI. Future Work

As our current project to re engineer Trans progresses we expect to continue refining and enhancing the COSI infrastructure. While this approach has been beneficial in simplifying the early stages of development we won't fully exercise the infrastructure until the later stages when we expect to see the full benefits from simplified maintenance.

References

1. "Transformation Reborn: A New Generation Expert System for Planning HST Operations", A. Gerb, in "Proceedings of the 1991 Goddard Conference on Space Applications of Artificial Intelligence", ed. J.L. Rash, NASA Conference Publication 3110 (Greenbelt: NASA), pp. 45-58, reprinted in Telematics and Informatics, 8, pp. 283-295.

2. "The Art of the Metaobject Protocol", Kiczales, des Rivieres and Bobrow. The MIT Press 1991. ISBN 0-262-61074-4