A Constraint-Driven Executable Model of Dynamic System Reconfiguration

D'Arcy Walsh
Bedarra Research Labs, Ottawa, Canada
Email: jdwalsh@acm.org

Francis Bordeleau
Zeligsoft and Carleton University, Ottawa, Canada
Email: francis@zeligsoft.com

Bran Selic
Carleton University, Ottawa, Canada
Email: selic@acm.org

***Abstract*—Dynamic system reconfiguration techniques are presented that can enable the systematic evolution of software systems due to unanticipated changes in specification or requirements. The methodological approach is based upon a domain analysis, which identifies a set of concepts that reflect the types of reconfigurations possible and the system integrity characteristics that must be maintained during such reconfigurations, a domain design, which is expressed using the Unified Modeling Language (UML) as a constraint-driven representation of the domain analysis, and a domain implementation, which uses a programming environment that supports explicit metaclass programming to realize an executable model of the analysis and design. It was learned that explicit metaclass programming can effectively be used to encode the constrained model, as a static representation, at the metalevel. With respect to dynamic reconfiguration, it was learned that a base-level object could be an instance of a property metaclass that is unique to that base-level object. Through a mixin mechanism, emergent run-time properties could be dynamically applied just to that object. The set of available mixins should also be adjusted dynamically. This is the subject of future work.**

***Index Terms*—Component-based systems, Dynamic reconfiguration, Feature modeling, Model-driven development, Service-oriented Architecture, Software evolution, System integrity, UML**

I. Introduction

Deployed software systems can be characterized as being ever more complex, ever more prevalent, and ever more critical in many ways to society as a whole. Examples include the trend toward mobile and distributed systems, embedded systems and embedded controlling devices, personal digital devices, and others. Since these systems serve increasingly useful roles in many application domains, it is important they are engineered for future change as the demands placed upon them vary over time.

However, predicting future user requirements or anticipating changing computing environment imposed requirements is extremely difficult and error prone, often resulting in over- or under-engineered solutions. The very nature of these systems (and by inference the requirements placed upon them) seems to continually grow in levels of complexity, interdependence, dynamism, and in other dimensions, and to continually outpace the state-of-the-art of software engineering capabilities required to respond to these driving concerns.

Adaptive computing through dynamic system reconfiguration techniques can enable the systematic evolution of software systems due to unanticipated changes in specification or requirements. The kinds of change are dynamic in the sense they have not necessarily been pre-programmed as part of the current capability of a deployed system but instead represent a realignment of the implementation of a system. Software engineering techniques that enable dynamic system reconfiguration are viewed as an important basis for building software systems that can adjust their structural and behavioral make-up in phase with their run-time contexts.

The overall contribution of the paper is a domain model that is useful at build time, and at run time, to enable the process of systematically changing software due to changes in software specification or software requirements. The specific contributions are the presentation of an executable model of dynamic system reconfiguration that is encoded using explicit metaclass programming techniques and an application-specific example of its instantiation.

First, a description is given of the computing paradigm that is the basis for this investigation. The domain of dynamic reconfiguration is then presented, followed by an example that illustrates the model. The main parts of the paper describe the executable model in detail, including executing an application-specific example. A summary, conclusion, and description of future work come at the end.

II. Background Overview

The component-based and service-oriented systems paradigm is adopted as a realization platform because

Based on "A Constrained Executable Model of Dynamic Reconfiguration", by D. Walsh, F. Bordelau, and B. Selic, which appeared in the Proceedings of the 40th Hawaii International Conference on System Sciences – 2007. Copyright 2007 IEEE.

© 2008 ACADEMY PUBLISHER

complex systems can be decomposed as software components that implement services, that are units of deployment, that can in principle be dynamically reconfigured as distinct entities (through intra-component dynamic reconfiguration) and/or as cooperative entities (through inter-component dynamic reconfiguration). A component can be realized in many ways including through metaobjects that define its runtime properties or as a batch-compiled entity of an embedded system.

A fundamental assumption is that a technical system specification may be expressed as a set of constraints [1]. A global property is a functional or non-functional constraint that requires global knowledge for conformance. A local property is a functional or non-functional constraint that requires local (i.e. component-level) knowledge for conformance. Examples of global properties include system level performance, availability, synchronization, distribution, security, or control style constraints. Examples of local properties include limits on the number of service invocations or service bindings, state element immutability, local performance, or local availability constraints.

Importantly, since this is change to a running system, the existing system specification may impose hard constraints that limit the degree to which the need for change can be addressed.

Dynamic system reconfiguration is addressed within the context shown in Figure 1. What follows is a description of each of the main elements that makeup this context in order to more clearly define what is in scope and what is out of scope for this investigation.

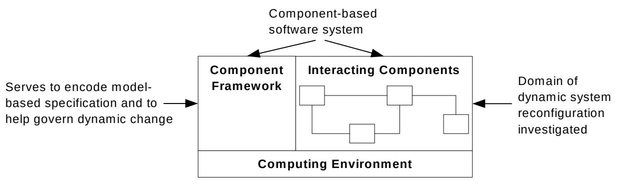

Component-based software system
Component Framework
Interacting Components
Serves to encode model-based specification and to help govern dynamic change
Domain of dynamic system reconfiguration investigated
Computing Environment

Figure 1. Context of Dynamic System Reconfiguration

A component-based software system comprises *Interacting Components* and a *Component Framework*. A *Computing Environment* is the run-time context of the system. A component-based software system is deployed and executes within this environment.

A *Component Framework* is the context for the instantiation of components and for the provision of services for coordinating those components that have been realized within the framework. The framework constrains how a component interacts with other components for the component to be an independent yet cooperative member of an overall system of components. More specifically, a *Component Framework* provides capabilities for their installation and initial configuration, loading and instantiation, configuration and connection to any required system artefacts, and for governing dynamic system reconfiguration.

Interacting Components form a system that implements required functional and non-functional properties through provided and required services (and associated service protocols). As a member of a greater system, a component is a unit of deployment whose encapsulated internal behaviour satisfies its external interaction obligations. Its internal behaviour may be implemented through the recursive composition of other components.

A component is defined to have a behavioural signature and a structural signature (See Figure 2). A component's behavioural signature is specific kinds of dependencies that determine its internal behaviour within a component-based system. The behavioural signature is composed of (i) *service to service protocol*, (ii) *operation to required service*, (iii) *operation to provided service*, (iv) *operation to operation*, (v) *operation to state element*, and (vi) *operation to composite component service* dependencies. A component's structural signature is specific kinds of dependencies that determine its external interactions within a component-based system. The structural signature is composed of (a) *component to component* and (b) *required service to provided service* dependencies.

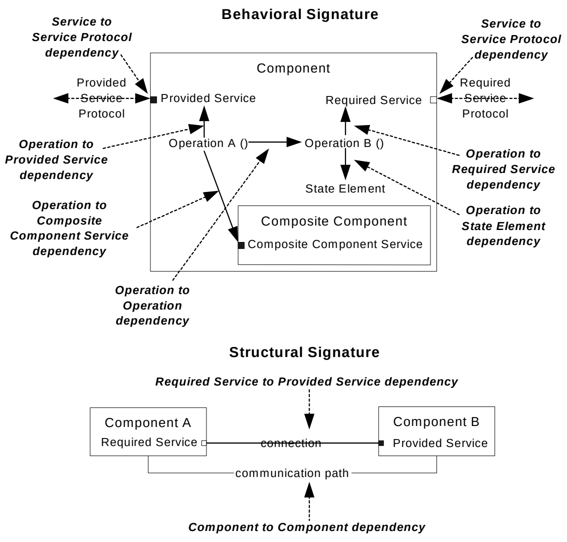

Behavioral Signature
Service to Service Protocol dependency
Service to Service Protocol dependency
Component
Provided Service Protocol
Provided Service
Required Service
Required Service Protocol
Operation to Provided Service dependency
Operation A ()
Operation B ()
Operation to Required Service dependency
State Element
Operation to Composite Component Service dependency
Composite Component
Composite Component Service
Operation to State Element dependency
Operation to Operation dependency
Structural Signature
Required Service to Provided Service dependency
Component A
Required Service
connection
Component B
Provided Service
communication path
Component to Component dependency

Figure 2. Behavioral and Structural Signatures of a Component

A component framework may change just as well as the configuration of components that it instantiated. This paper focuses on component-level dynamic reconfiguration. However, the model execution environment section provides an example, through simulation, of a component framework that does change to support the dynamic reconfiguration models that are presented.

Dynamic change may well lead to the need for further dynamic change (a system may not reach an acceptable equilibrium). Managing this kind of 'cycle of change' is considered out of the scope of this paper and to be future work. This investigation focuses on 'discrete change events' and the formulation of a model-based specification of such events.

Finally, in this context, a context-aware component-based software system is viewed to be a system that can continually dynamically reconfigure itself to stay in phase with its (changing) computing environment. This investigation describes an approach that enables context-aware computing. However, the latter is considered future work.

III. THE DOMAIN OF DYNAMIC RECONFIGURATION

This section presents a summary of a domain model of dynamic system reconfiguration first presented in [2] and more fully reported on in [3], which reviews significant related research work. The full analysis identifies and categorizes the various types of change that may be required, the relationship between those types, and the system integrity characteristics that need to be considered when such changes take place.

A. Summary of Domain Analysis of Dynamic System Reconfiguration

This subsection describes the common and variable causal flow of dynamic reconfiguration. The intent is to provide a description of the full spectrum of architectural, behavioural, and structural changes that could be exhibited by a dynamically reconfigurable component-based software system. The following nine *change groups* are identified: *Pure architectural change*; *pure behavioural change*; *behaviour-driven structural change*; *comprehensive behaviour-driven change*; *pure structural change*; *structure-driven behavioural change*; *comprehensive structure-driven change*; *pure behavioural and structural change*; and *comprehensive change*.

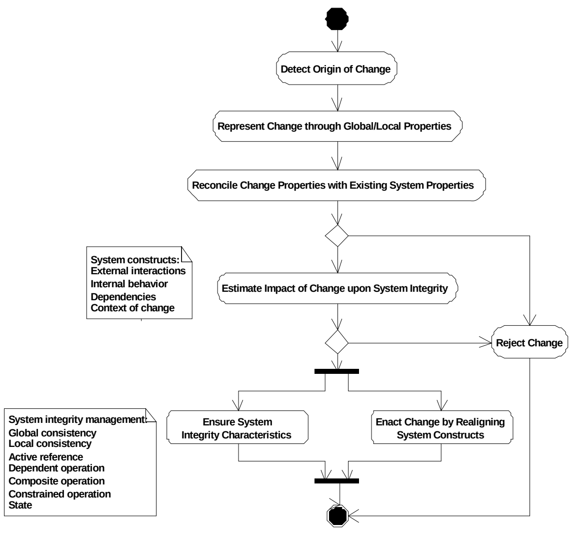
Each change group is composed of a set of related *change sequences* that describe how a particular change group unfolds. A change sequence combines up to six different *change types.*

Dynamic reconfiguration progresses according to the activity model shown in Figure 3. The decision points immediately after 'Reconcile Change Properties with Existing System Properties' indicates this reconciliation may lead to change rejection. The decision point immediately after 'Estimate Impact of Architectural Change upon System Integrity' indicates this impact analysis may lead to change rejection. The parallel fork indicates 'Enact Change by Realigning System Constructs' and 'Ensure System Integrity Characteristics' happens concurrently. The parallel join indicates a system has enacted dynamic change while attempting to ensure overall system consistency.

Given the interactions implied by Figure 3, Table 1 is a use case of the general context of dynamic reconfiguration. This use case describes the different levels of dynamism as alternatives based upon the change groups.

Pure architectural change is a change group that consists of a single change sequence involving a single change type: *architectural change*. In its purest form, this is a change to only global and/or local system *properties*. By implication, the existing behavioural and structural signatures of the system can accommodate the change and are unaffected.

Pure behavioural change is a change group that encompasses three change sequences, each driven by *architectural change*: (i) *protocol change* only, (ii) *protocol change* leading to *interface change*, or (iii) *protocol change* leading to *interface change* leading to *internal change*. By implication, *pure behavioural change* means that only the behavioural signature of a system is realigned.


Detect Origin of Change
Represent Change through Global/Local Properties
Reconcile Change Properties with Existing System Properties
System constructs:
External interactions
Internal behavior
Dependencies
Context of change
Estimate Impact of Change upon System Integrity
Reject Change
System integrity management:
Global consistency
Local consistency
Active reference
Dependent operation
Composite operation
Constrained operation
State
Ensure System Integrity Characteristics
Enact Change by Realigning System Constructs
Figure 3. Activity Model of General Context of Dynamic Reconfiguration

Behaviour-driven structural change is a change group that comprises two change sequences, each driven by *architectural change*: (i) *protocol change* leading to *topology change* or (ii) *protocol change* leading to *topology change* leading to *substitution*. By implication, *behaviour-driven structural change* means that, when the behavioural signature of a system is realigned, it causes the structural signature of the system to be realigned.

Comprehensive behaviour-driven change is a change group that is a combination of *pure behavioural change* and *behaviour-driven structural change*. By implication, *comprehensive behaviour-driven change* means that, when the behavioural signature of a system is realigned, it causes the structural signature of the system to be realigned. All combinations are possible as long as a behavioural change leads to a structural change.

Pure structural change is a change group that encompasses two change sequences that are each driven by *architectural change*: (i) *topology change* only or (ii) *topology change* leading to *substitution*. By implication, *pure structural change* means the structural signature, and only the structural signature, of a system is realigned.

Structure-driven behavioural change is a change group that comprises three different change sequences, each driven by *architectural change*: (i) *topology change* leading to *protocol change*, (ii) *topology change* leading to *protocol change* leading to *interface change*, or (iii) *topology change* leading to *protocol change* leading to *interface change* leading to *internal change*. By implication, *structure-driven behavioural change* means that, when the structural signature of a system is realigned, it causes the behavioural signature of the system to be realigned.

Use Case: General Context of Dynamic Reconfiguration
Description: This use case describes the overall casual flow of dynamically reconfiguring a component-based system.
Actors: System User and/or Processing Environment
Pre-Condition: Component-based system provides adequate capability to respond to run time change stimuli, characterize the nature of change required as global or local properties (including their reconciliation), enact generic types of change that may be required, and govern change in a manner to help ensure overall system consistency.
Triggering Events: *User-Driven Change* and/or *Computing Environment Imposed Change*
Sequence: Step 1. Sense *User-Driven* and/or *Computing Environment Imposed Change*. Step 2. Interpret and then represent the particular *Origin of Change* as *Global Properties* or *Local Properties* with an associated *Reconciliation Policy* (also reconcile with existing *System Properties*). Step 3. Determine what feasible change group (if any) satisfy the *Condition Change Criteria* and enact change (See Alternatives). Step 4. Realign *Dependencies* (if necessary) and therefore possibly *External Interactions* and *Internal Behavior*. Step 5. Validate preservation of applicable *System Integrity Characteristics* during change sequence.
Post-condition: Component-based system is dynamically reconfigured based upon a particular change groups' change sequence in a manner that helps to ensure overall system integrity.
Resulting Events: Further *User-Driven Change* and/or *Computing Environment Imposed Change*
Alternatives for Step 3: Alternative 1 – *Pure architectural change*; Alternative 2 – *Pure behavioral change*; Alternative 3 – *Behavior-driven structural change*; Alternative 4 – *Comprehensive behavior-driven change*; Alternative 5 – *Pure structural change*; Alternative 6 – *Structure-driven behavioral change*; Alternative 7 – *Comprehensive structure-driven change*; Alternative 8 – *Pure behavioral and structural change*; or Alternative 9 – *Comprehensive change*.

Table 1. Common and Variable Causal Flow of Dynamic Reconfiguration

Comprehensive structure-driven change is a change group that is a combination of pure structural change and structure-driven behavioural change. By implication, comprehensive structure-driven change means that, when the structural signature of a system is realigned, it causes the behavioural signature of the system to be realigned. All combinations are possible as long as a structural change leads to a behavioural change.

Pure behavioural and structural change is a change group that is a combination of pure behavioural change and pure structural change. By implication, pure behavioural and structural change means that the behavioural signature and the structural signature of a system are realigned but one does not drive the other to change. All combinations are possible as long as behavioural change does not lead to structural change nor does structural change lead to behavioural change.

Comprehensive change is a change group that is a combination of *comprehensive behaviour-driven change* and *comprehensive structure-driven change*. By implication, *comprehensive change* means that the behavioural signature and the structural signature of a system are realigned as one drives the other to change. All combinations are possible as long as behavioural change leads to structural change and structural change leads to behavioural change.

B. Summary of Domain Design of Dynamic System Reconfiguration

Figure 4 shows a UML class model of the domain concepts of *Comprehensive change*. With respect to change properties reconciliation, the classes *System Properties* and *Change Properties* represent the current system properties and (new) change properties, respectively, as global and local constraints. Each grouping of properties is internally reconciled according to reconciliation policies. In addition, these groupings are reconciled with respect to each other. The overall reconciliation is modeled as a binary association linking *System Properties* and *Change Properties*. The reconciled change properties in turn determine which subsets of change types are required to perform the necessary system reconfiguration.

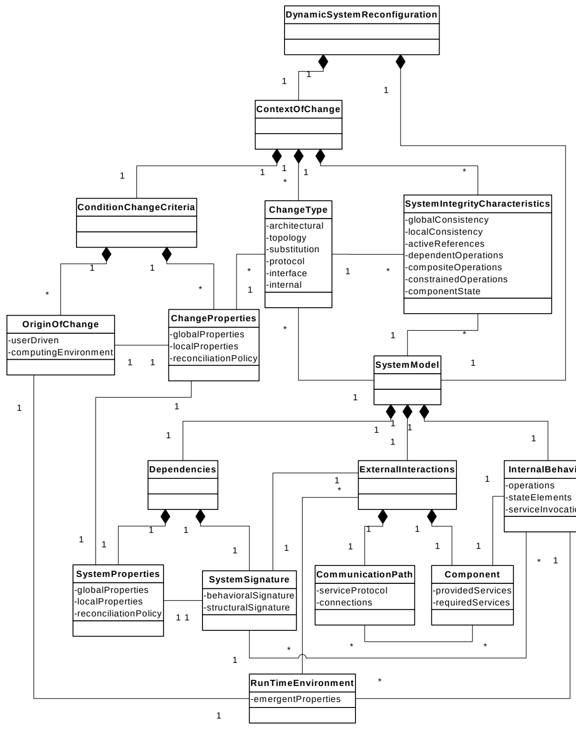


Figure 4. Class Model of Comprehensive Change

With respect to change enactment, the classes *Change Type, System Integrity Characteristics,* and *System Model* specify how the required change types, as constrained by system integrity characteristics, realign the system model. Depending upon the types of change required, the structural and/or behavioral signature of the system may have to be re-aligned, which, in turn, may impact its internal behavior or external interactions.

With respect to further dynamic reconfiguration, given the internal behavior or external interactions of a system have changed, the system may reach a failure state or require further change based upon emergent properties. This is represented by the group of classes *External Interactions*, *Internal Behavior*, *Fault Tolerance Mode*, and *Origin Of Change*. *External Interactions* and *Internal Behavior* are related to *Fault Tolerance Mode* upon failure (or emergent properties). *Fault Tolerance Mode* then drives *Origin Of Change* (which ultimately may be user driven or computing environment imposed) which in turn indicates needed change properties. This completes the cycle of system dynamic reconfiguration described by the domain model.

IV. Financial Analysis System Case STudy

The following is a review of a financial analysis system case study first presented in [4] and more fully reported on in [3]. The case study is an application-specific example of changing global and local properties leading to comprehensive change. The example describes the components and the dynamic interoperation of two initially decoupled financial systems that specialize in maintaining knowledge and providing predictions about a particular sector of the economy. System A's clients are concerned with shorter-term predictions. System B's clients are concerned with longer-term predictions.

The following are examples of global system properties:

- (GP1) The control style of each system (as depicted by use case maps);
- (GP2) The operations of different components provide needed behavior within limited time constraints (scenario analysis must not be invalidated by current financial conditions); and
- (GP3) The state elements of different components are updated in a synchronized fashion (present value analysis, cash flow projection, and scenario analysis reference data is synchronized).

The following are examples of local system properties:

- (LP1) A component has an upper bound on the number of threads that may be spawned in response to remote service requests;
- (LP2) A provided service has at most one required service bound to it and vice versa; and
- (LP3) The values of certain state elements may not change (Scenario Analysis reference data, once synchronized, remains immutable).

Figure 5 shows the original control style of System A. Use Case Maps (UCMs) [5] are used to illustrate the causal flow that is required of System A' s components to provide shorter-term predictions. For example, for timeliness reasons, cash flow projections and valuation assessment are done on-line.

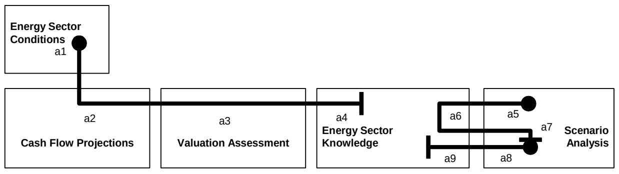


Figure 5. Original Control Style of System A

System A's responsibilities are: (a1) generate on-line financial conditions, (a2) provide cash flow projections, (a3) provide valuation assessment, (a4) update on-line financial conditions and update knowledge information about market sector, (a5) determine current market knowledge, (a6) current financial conditions and market knowledge, (a7) update preferred stock and common stock value predictions, (a8) provide knowledge information about market sector, and (a9) update knowledge information about market sector.

Figure 6 shows the original control style of System B. UCMs are used to illustrate the causal flow that is required of System B' s components to provide longer-term predictions. For example, for reasons of accuracy, cash flow projections and valuation assessment are done off-line on demand.

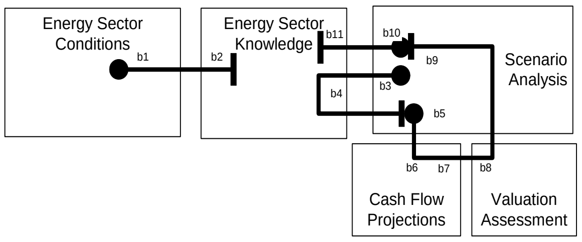


Figure 6. Original Control Style of System B

System B's responsibilities are: (b1) generate on-line financial conditions, (b2) update on-line financial conditions and update knowledge information about market sector, (b3) determine current knowledge about market, (b4) provide current financial conditions and knowledge about market, (b5) determine cash flow projections, (b6) provide cash flow projections, (b7) determine valuation assessment, (b8) provide valuation assessment, (b9) update preferred stock and bond value predictions, (b10) provide knowledge information about market sector, and (b11) update knowledge information about market sector.

The systems are dynamically reconfigured so that System A can leverage System B's preferred stock predictions. To do this, each system's architectural constraints are reconciled and changes are constrained to be backward compatible. System A is then able to provide improved analytic results for its respective clients based upon the new information that is available from System B.

Figure 7 shows the new control style of System A. A new UCM represents the causal flow linking the Scenario Analysis component of System A to the Scenario Analysis component of System B. This enables System A to use System B's longer-term predictions to validate its shorter-term predictions.

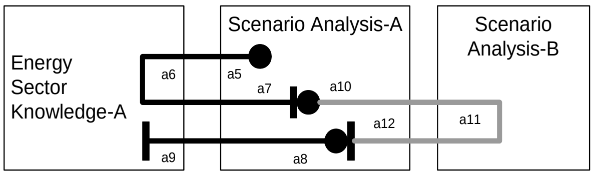


Figure 7. New Control Style of System A

The new responsibilities are: (a10) determine long-term predicted values, (a11) provide long-term predicted values, and (a12) validate short-term predictions using long-term predictions.

System evolution happens as follows:

- Evolution is localized to the scenario analysis component of System A
- A communication path is established between the two systems
- System A's external interactions evolve to support new required service
- Internal behavior of System A's scenario analysis component evolves in place so that it can process the new information that is exchanged.

V. Executable Model

When a system's run-time properties are realized as instantiations of programmable metaclasses, these properties can be dynamically changed when their metaclasses change. This section describes the use of explicit metaclass programming techniques to encode certain aspects of the domain model [6]. The purpose of constructing an executable model is: (i) to gain a more direct understanding of the conceptual framework and (ii) to validate, through empirical proof and simulation, that the constrained model is applicable within application-specific contexts.

With this approach, metaobjects are employed to implement the *Component Framework* shown in Figure 1. The *Interacting Components* shown in Figure 1 are implemented by base-level objects, which are instances of the metaobjects. In this environment metaobjects themselves are run-time objects. As run-time objects they can interact and change their characteristics. When they change their characteristics they can change the run-time properties of base-level objects that are their instances.

Using the case study, a description is given of representing and then dynamically changing an application-specific global property leading to other kinds of change. The general approach is the following. The metaobjects of system and change properties encapsulate system signature 'fragments' that are implied by their instances. The current system signature of a system is the composition of the system signature 'fragments' of its global and local system properties. To change the signature of a system, the system signature 'fragments' of change properties are composed with the existing system signature 'fragments' of system properties. In this implementation, the global or local property metaobjects implement a standardized protocol for setting up a system signature 'fragment'. A particular fragment is expressed as a configuration of base-level objects that can be instantiated at the time change properties are reconciled with system properties.

A. MetaclassTalk as a Reflective Substrate

Bouraqadi-Saadani et al [6-7] describe the MetaclassTalk computing environment that provides direct support for meta object composition mechanisms to enable system evolution and adaptation. These composition mechanisms allow specific properties to be assigned to classes in order to allow the properties of their instances to dynamically change. They note that from an architectural viewpoint meta class composition allows a system to be organized into different levels of abstraction. An example is given showing the meta class composition "False + SoleInstance + Final" to create composed class properties that may then be instantiated as the sole false instance whose class may not be sub-classed.

They describe a reflective system development approach that utilizes safe and explicit metaclass programming techniques and an implementation environment that enables this that is known as MetaclassTalk. This paradigm is based upon meta-links, which causally connect base-level and meta-level objects, and meta-object cooperation, which is explicitly programmed. Examples of the kinds of relationships that may be implemented between base-level objects and meta-level objects include: (i) a single meta-object shared between instances of the same class; (ii) a single specific meta-object private to each base object; and (iii) many meta-objects shared among many base objects [7]. This is shown in Figure 8 below:

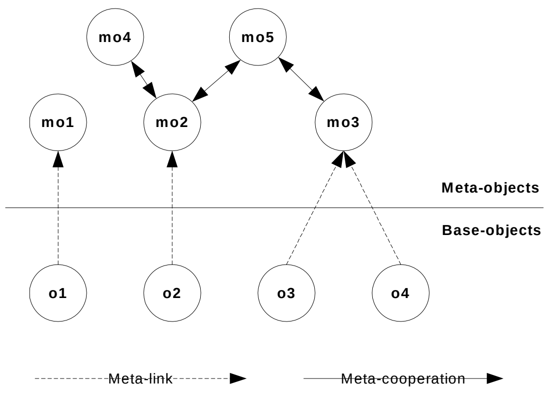


Figure 8. Base versus Meta-objects

Figure 9 shows how MetaclassTalk can be used as a reflective substrate of the domain model presented in Section III. The *System Model* and *Context of Change* classes, shown in Figure 4, are encoded as meta-objects that cooperate to enable the dynamic reconfiguration of base-level application-specific component interactions. The meta-link causally connecting the base-level to *Context of Change* represents emergent run-time characteristics that are ultimately manifested as change properties. *Context of Change* and the *System Model* cooperate at the meta-level to reconcile change properties with existing system properties and to realign the behavioral and/or structural profile of the system as required. The meta-link linking the base-level to the *System Model* represents the causal connection required to affect the realignment of the running system. This may in turn lead to new emergent run-time characteristics, leading to further dynamic reconfiguration, and so on.

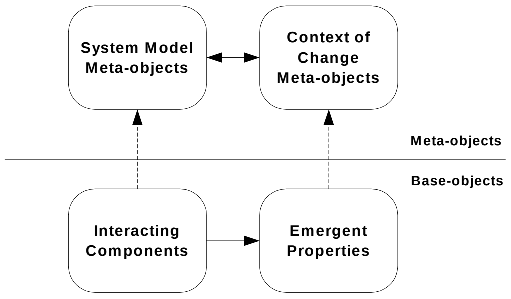


Figure 9. Using the reflective substrate

B. Using Explicit Metaclass Programming

This subsection illustrates how the domain design of comprehensive change is represented within the reflective substrate using explicit metaclass programming techniques. What follows is an explanation of how *Global Properties* are encoded.

Bouraqadi-Saadani [8] explains the use of *compatibility* and *property* metaclasses to ensure the overall compatibility of composed metaclasses that are explicitly programmed. This technique is employed when encoding the constrained model and therefore *compatibility* and *property* metaclasses are part of the encoding. Bouraqadi-Saadani [8] also implements meta-level support for mixins as a mechanism to augment *property* metaclasses. This technique is also used when encoding the model. As an alternative mechanism to mixins, the use of “traits” is under review Scharli et al [9]. Because Bouraqadi-Saadani [8] explicitly uses the term *property*, property named elements of the domain model are referenced as *constraints* below.

Figure 10 shows how global and local constraints are encoded. As discussed previously, a global constraint is a system characteristic that requires global knowledge for conformance. A local constraint is a system characteristic that requires local (i.e. component-level) knowledge for conformance. Examples of constraint characteristics that may apply, globally or locally, include state element immutability, cardinality (i.e. limits on the number of service invocations or service bindings, and so on), performance, synchronization, distribution, persistence, security, control style, and so on. These characteristics are represented as *Constraint Characteristics Meta Objects* in Figure 10. They are ‘mixedin’ when a global or local constraint is instantiated based upon a *mixin linearization list* that determines the order of composition.

The following is a short description of the metaobjects shown in Figure 10 that are required to encode a global property:

Global Constraint is defined as a concrete subclass of *Constraint*. It is an instance of *Global Constraint Property Meta Object*. When instantiated as a base-level object, a global constraint represents a particular global property of a component-based software system. Examples of global constraints were given in Section III.

Constraint is defined as an instance of *Constraint Property Meta Object*. It is the abstract superclass of *Global Constraint* and *Local Constraint*. It implements *constraintSpec*, which represents the specification of a particular global or local constraint that is instantiated by *Global Constraint* or *Local Constraint*. It also implements the creation method, *new: with Mixin Linearization List*, of *Global Constraint* and *Local Constraint*. The mixin linearization list determines the mixin properties that are composed when a global or local constraint is created.

Constraint Property Meta Object is defined as an instance of *Composite Class* (not shown). *Composite class* is part of the MetaclassTalk library. It is a class whose instances inherit from their superclass through a set of mixins. The inheritance relationship is via implicit subclass(es) of the ‘official superclass’ that are built by means of the mixin mechanism. *Constraint Property Meta Object* is defined as a subclass of *Constraint Compatibility Meta Object* with *Abstract* as a mixin. *Abstract* is part of the MetaclassTalk library of mixins. When it is added as a mixin to *Constraint Property Meta Class* mixins, *Constraint* becomes an abstract class.

Constraint Compatibility Meta Class is defined as a subclass and as an instance of *Standard Class* (not shown). *Standard Class* is part of the MetaclassTalk library. This is the root class of all MetaclassTalk explicit metaclasses. When subclassed, it enables the explicit definition of new kinds of (meta) classes. *Constraint Compatibility Meta Class* enables the compatibility model described in [8] among the inheritance hierarchies that underpin *Constraint*, *Global Constraint*, and *Local Constraint*.

Global Constraint Property Meta Object is defined as an instance of *Composite Class* to enable mixin property composition (not shown). It is defined as a subclass of *Global Constraint Compatibility Meta Object* to ensure it complies with the meta class compatibility model. *Class With Instance Mutable Meta Objects* is part of the MetaclassTalk library of mixins (not shown). When added as a mixin, the instances of *Global Constraint Property Meta Object* (i.e. *Global Constraint*) have their own metaObject, which can be changed over time.

Global Constraint Compatibility Meta Class is defined as an instance of *Standard Class* (not shown). As an explicit metaclass, it is defined as a subclass of *Constraint Compatibility Meta Object* to enable the compatibility model.

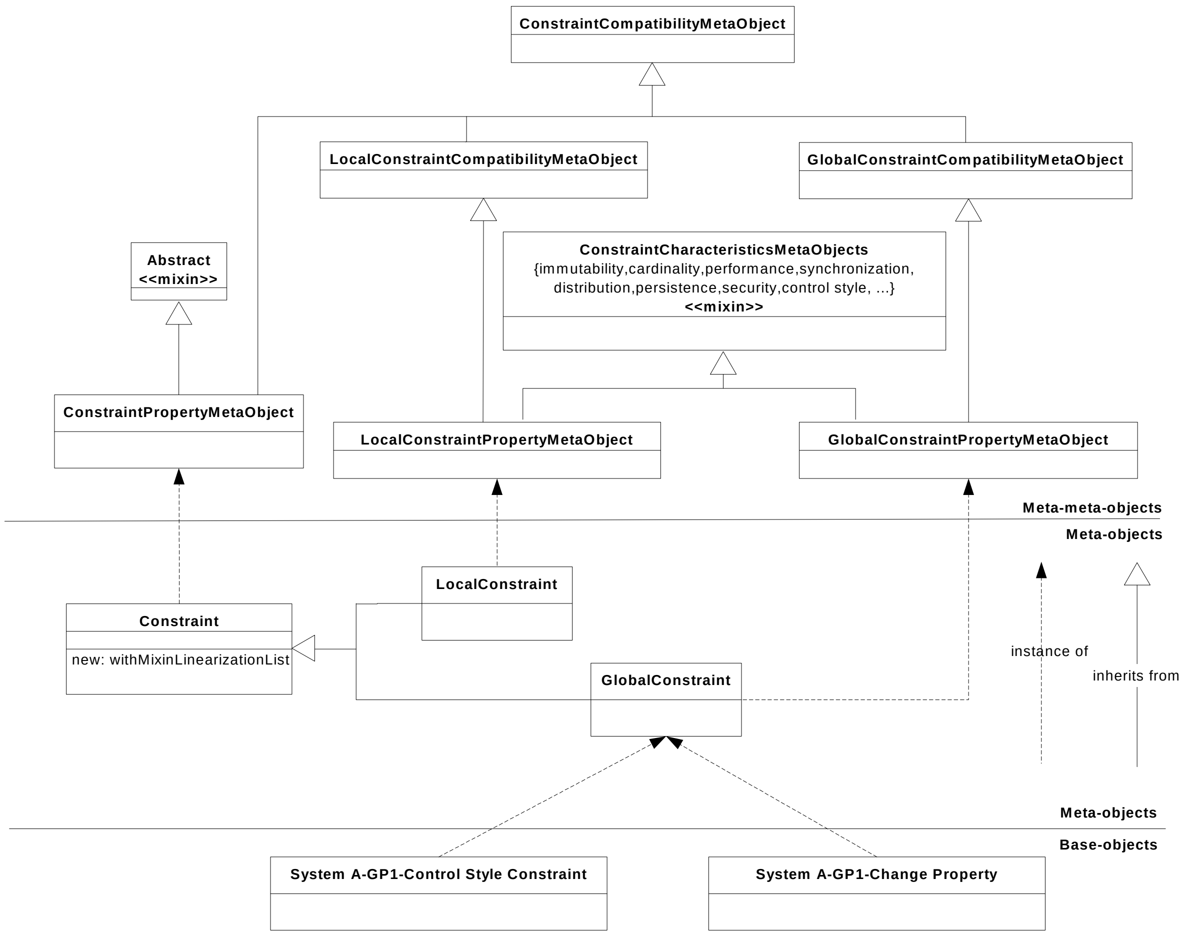

ConstraintCompatibilityMetaObject
LocalConstraintCompatibilityMetaObject
GlobalConstraintCompatibilityMetaObject
Abstract <<mixin>>
ConstraintCharacteristicsMetaObjects {immutability,cardinality,performance,synchronization, distribution,persistence,security,control style, ...} <<mixin>>
ConstraintPropertyMetaObject
LocalConstraintPropertyMetaObject
GlobalConstraintPropertyMetaObject
Meta-meta-objects
Meta-objects
LocalConstraint
Constraint
new: withMixinLinearizationList
GlobalConstraint
instance of
inherits from
Meta-objects
Base-objects
System A-GP1-Control Style Constraint
System A-GP1-Change Property

Figure 10. Encoding Global and Local Constraints

VI. Executing an Application-Specific Example

The requirement for System A to dynamically inter-operate with System B is an example of emergent properties as shown in Figure 9. In this example, the outcome is reflected by Figure 7 with a new UCM causally linking System A to System B. This subsection explains how the reflective substrate affects base-level application-specific dynamic reconfiguration, when a global property changes. The following is described: (i) how a global property is instantiated using the encoded model, (ii) dynamically reconfiguring the global property, (iii) how Architectural Change reconfigures the signature of a system, and (iv) how a reconfigured system signature leads to behavioral and structural change.

A. Representing a Global Property

This subsection illustrates how the reflective substrate, is used to create instances of GP1, which determines the control style of System A. Figure 10 shows the base-level and meta-level interactions required. *System A - GP-1-Control Style Constraint* and *System A - GP-1-Change Property* are shown as instances of *Global Constraint*. When *System A - GP-1-Control Style Constraint* and *System A - GP-1-Change Property* are created, their global constraint properties are augmented with an appropriate control style provided by *Control Style Characteristics MetaObjects*. *System A – GP1 – Control Style Constraint* conforms to the control style shown in Figure 5. *System A – GP1 – Change Property* conforms to the control style shown in Figure 7.

For example, when *System A - GP-1-Control Style Constraint* is created, the mixin linearization list drives the composition of specific control style metaobjects that constrain the configuration of System A’s components to ensure cash flow projections and valuation assessment are done on-line.

MetaclassTalk code fragments are shown below to illustrate how this is done. In the example, *gp1* is created to represent the global property *System A - GP-1-Control Style Constraint*. The specification of this property for System A is ‘On-Line Control’. When *gp1* is created, the metalink policy *Class With Instance Mutable Meta Objects* is applied that enables it to have a unique metaclass (this was reported as “a single specific meta-object private to each base object” earlier). In the case of *gp1* its particular properties are augmented with *System A On Line Control Style MetaObject* in support of its ‘On-Line Control’ specification:

```
gp1 := GlobalConstraint new.
gp1 constraintSpec: ‘Control Style of System '.
metaclass := MetaObject
subclass: #GP1AMetaClass
instanceVariableNames: ''
```

```
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Dynamic Reconfiguration-Interacting
Components-Instance Specific Property MetaObjects'
   metaclass: CompositeClass.
   metaclass addMixin:
ClassWithInstanceMutableMetaObjects.
   metaclass addMixin:
SystemAOnLineControlStyleMetaObject.
   metaclassInstance := metaclass new.
   gp1 metaObject: metaclassInstance.
```

When *GP1AMetaClass* is created, it is a metaObject specific to *gp1*. Its mixins include *System A On Line Control Style Meta Object:*

```
   MetaObject subclass: #GP1AMetaClass
         instanceVariableNames: ''
         classVariableNames: ''
         poolDictionaries: ''
         category: 'Dynamic Reconfiguration-Interacting
Components-Instance Specific Property MetaObjects'
         metaclass: CompositeClass.
   GP1AMetaClass mixins:
{ClassWithInstanceMutableMetaObjects.
SystemAOnLineControlStyleMetaObject}
```

System A On Line Control Style Meta Object is a mixin with instance variable '*system Signature For System A On Line Control'*:

```
   Mixin named:
#SystemAOnLineControlStyleMetaObject
   instanceVariables:
'systemSignatureForSystemAOnLineControl '
   category: 'Dynamic Reconfiguration-Mixins'
```

System A On Line Control Style Meta Object implements a standardized protocol that is used to set up '*system Signature For System A On Line Control'*. It is an instance of *System Signature,* which represents the behavioral and structural dependencies reflected by Figure 5. Below is an example of part of the protocol implemented by *System A On Line Control Style Meta Object:*

```
   setUpConnectedComponentDependencies
   systemSignatureForSystemAOnLineControl
connectComponent: self
energySectorConditionsComponent  toComponent: self
cashFlowProjectionsComponent.
   systemSignatureForSystemAOnLineControl
connectComponent: self cashFlowProjectionsComponent
toComponent: self valuationAssessmentComponent.
   systemSignatureForSystemAOnLineControl
connectComponent: self valuationAssessmentComponent
toComponent: self energySectorKnowledgeComponent.
   systemSignatureForSystemAOnLineControl
connectComponent: self
energySectorKnowledgeComponent toComponent: self
scenarioAnalysisComponent.
```

The instance specific metaObject of *System A - GP-1-Change Property* also implements the standardized protocol. In its case, its *System Signature* represents the behavioral and structural dependencies reflected by Figure 7.

B. Dynamically Reconfiguring a Global Property

This subsection illustrates change properties affecting existing system properties, which subsequently drives the system signature of System A to change.

In Figure 10 emergent system properties are manifested as *System A – GP1 – Change Property*. *System A - GP-1-Control Style Constraint* represents an existing reconciled global constraint of System A before any change. *System A – GP1 – Change Property* is a global constraint that implies a change to the control style of System A as reflected by Figure 7. Its particular properties are augmented with *System B Interoperation MetaObject* which represents its augmentation of the 'On-Line Control' specification:

```
   newgp1a := GlobalConstraint new.
   newgp1a constraintSpec: 'Control Style of System '.
   metaclass := MetaObject
   subclass: #NewGP1AMetaClass
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Dynamic Reconfiguration-Interacting
Components-Instance Specific Property MetaObjects'
   metaclass: CompositeClass.
   metaclass addMixin:
ClassWithInstanceMutableMetaObjects.
   metaclass addMixin:
SystemBInteroperationMetaObject.
   metaclassInstance := metaclass new.
   newgp1a metaObject: metaclassInstance.
```

When *NewGP1AMetaClass* is created, it is a metaObject specific to *newgp1*. Its mixins include *System B Interoperation Meta Object:*

```
   MetaObject subclass: #NewGP1AMetaClass
         instanceVariableNames: ''
         classVariableNames: ''
         poolDictionaries: ''
         category: 'Dynamic Reconfiguration-Interacting
Components-Instance Specific Property MetaObjects'
         metaclass: CompositeClass.
   NewGP1AMetaClass mixins:
{ClassWithInstanceMutableMetaObjects.
SystemBInteroperationMetaObject}
```

To drive system change, the original metaclass properties of *System A - GP-1-Control Style Constraint* are dynamically augmented to reflect the new control style of System A by adding the metaclass properties of

System A – GP1 – Change Property as additional mixin properties of *System A - GP-1-Control Style Constraint:*

```
   gp1 metaObject class addAllMixins: (newgp1
metaObject class mixins).
```

This is the result after adding the mixins of newgp1's metaclass to the mixins of gp1's metaclass. The mixins of *GP1AMetaClass* now includes *System B Interoperation Meta Object:*

```
   MetaObject subclass: #GP1AMetaClass
           instanceVariableNames: ''
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Dynamic Reconfiguration-Interacting
Components-Instance Specific Property MetaObjects'
           metaclass: CompositeClass.
   GP1AMetaClass mixins:
{ClassWithInstanceMutableMetaObjects.
SystemAOnLineControlStyleMetaObject.
SystemBInteroperationMetaObject}
```

System A On Line Control Style Meta Object and *System B Interoperation Meta Object* each encapsulate a *System Signature* as described previously.

C. Architectural Change Reconfiguring the Signature of a System

Ultimately, a change to the global and local properties of System A leads to the reconfiguration of its system signature. In this implementation, Architectural Change regenerates the overall system signature of System A by composing the system signature 'fragments' of property metaobjects. This drives the behavioral and structural realignment of the system. This is shown in the code fragment below:

```
   Dependencies>>useSystemPropertiesToGenerateSyste
mSignature
   self systemProperties systemSignatures do: [ :aSysSig |
self systemSignature updateWith: aSysSig].
```

The definition of *SystemSignature* is shown below. Aligned with the domain model, it is composed of *StructuralSignature* and *BehavioralSignature*.

```
   Object subclass: #SystemSignature
           instanceVariableNames: 'structuralSignature
behavioralSignature '
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Dynamic Reconfiguration-Interacting
Components'
           metaclass: SystemSignaturePropertyMetaObject
```

The code fragment below shows how the existing system signature is regenerated with a system signature 'fragment':

```
   SystemSignature>>updateWith: aSystemSignature
   self behavioralSignature updateWith:
aSystemSignature behavioralSignature.
   self structuralSignature updateWith: aSystemSignature
structuralSignature.
```

The definition of *BehavioralSignature* is shown below. Aligned with the domain model (See Figure 2), it is composed of protocol, required service, provided service, operation, state element, and composite component dependencies.

```
   Object subclass: #BehavioralSignature
           instanceVariableNames: 'protocolDependencies
requiredServiceDependencies
providedServiceDependencies operationDependencies
stateElementDependencies
compositeComponentDependencies '
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Dynamic Reconfiguration-Interacting
Components'
           metaclass:
BehavioralSignaturePropertyMetaObject
```

When *Architectural Change* regenerates the *Behavioral Signature* of a system it updates the existing *Behavioral Signature* with a new *Behavioral Signature* that reflects any changes in behavioral dependencies:

```
   BehavioralSignature>>updateWith:
aBehavioralSignature
   self updateProtocolDependenciesWith:
aBehavioralSignature protocolDependencies.
   self updateRequiredServiceDependenciesWith:
aBehavioralSignature requiredServiceDependencies.
   self updateProvidedServiceDependenciesWith:
aBehavioralSignature providedServiceDependencies.
   self updateOperationDependenciesWith:
aBehavioralSignature operationDependencies.
   self updateStateElementDependenciesWith:
aBehavioralSignature stateElementDependencies.
   self updateCompositeComponentDependenciesWith:
aBehavioralSignature
compositeComponentDependencies.
```

The definition of *StructuralSignature* is shown below. Again, in phase with the domain model (See Figure 2), it is composed of connected component and connected service dependencies.

```
   Object subclass: #StructuralSignature
           instanceVariableNames:
'connectedComponentDependencies
connectedServiceDependencies '
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Dynamic Reconfiguration-Interacting
Components'
           metaclass:
StructuralSignaturePropertyMetaObject
```

When *Architectural Change* regenerates the *Structural Signature* of a system it updates the existing *Structural Signature* with a new *Structural Signature* that reflects any changes in structural dependencies:

```
   StructuralSignature>>updateWith:
aStructuralSignature
   self updateConnectedComponentDependenciesWith:
aStructuralSignature connectedComponentDependencies.
   self updateConnectedServiceDependenciesWith:
aStructuralSignature connectedServiceDependencies.
```

In the example up to now, the metaobject of *System A - GP-1-Control Style Constraint* has been augmented with the mixin properties of the metaobject of *System A – GP1 – Change Property*. Each mixin of *System A - GP-1-Control Style Constraint* encapsulates a system signature 'fragment' that Architectural Change now uses to regenerate the overall *System Signature* of System A.

In this case, it is *SystemBInteroperationMetaObject*, which encapsulates the structural dependencies that reflect a new communication path linking System A with System B, which leads to the Topology Change of System A.

To do this, *SystemBInteroperationMetaObject* implements the following dependency:

```
   SystemBInteroperationMetaObject
>>setUpConnectedServiceDependencies
   SystemSignatureForSystemBInteroperation
connectRequiredServiceNamed: 'Short Term Predictions
Based On Long Term Predictions'
toProvidedServiceNamed: 'Short Term Predictions Based
On Long Term Predictions'.
```

SystemSignatureForSystemBInteroperation represents the *System Signature* 'fragment' of *SystemBInteroperationMetaObject*. The outcome of this metaobject setup operation is a *System Signature* 'fragment' with a connected service dependency linking a required service of System A to a provided service of System B:

```
   SystemSignature>>connectRequiredServiceNamed:
requiredName toProvidedServiceNamed: providedName
   | rs ps |
   rs := self behavioralSignature requiredServiceNamed:
requiredName.
   ps := self behavioralSignature providedServiceNamed:
providedName.
   self structuralSignature
connectedServiceDependencies at: rs put: ps.
```

When *Architectural Change* regenerates the overall *System Signature* of System A, the connected service dependency of *SystemBInteroperationMetaObject* is composed with the existing connected service dependencies of System A when System A's *Structural Signature* is updated (See *Structural Signature>>updateWith: aStructuralSignature* shown above).

D. A Reconfigured System Signature Leading to Topology Change

The regeneration of *System Signature* drives the behavioral and structural realignment of a system. A change to *Behavioral Signature* of *System Signature* leads to a behavioral realignment through *Protocol*, *Interface*, and/or *Internal Change*. A change to *Structural Signature* of *System Signature* leads to a structural realignment through *Topology Change* and/or *Substitution*.

In this implementation, the kind of (new) dependency added to *System Signature* determines what type of change needs to be applied. New protocol dependencies are manifested as a change to service protocols via *Protocol Change*. New required or provided service dependencies are manifested as a change to required or provided services via *Interface Change*. New operation, state element, or composite component dependencies are manifested as a change to operations, state elements, or composite components via *Internal Change*. New connected component dependencies are manifested as changes to components and communication paths via *Topology Change* or *Substitution*. New connected service dependencies are manifested as changes to connections via *Topology Change* or *Substitution.*

In this example, the topology of System A is changed by adding a connection between the *Scenario Analysis* component of System A and the *Scenario Analysis* component of System B. The new connected service dependency added to the *Structural Signature* of System A results in this *Topology Change*, which leads to *Protocol Change, Interface Change,* and *Internal Change*.

The *Scenario Analysis* component of System A is an instance of *Deployed Component*:

```
   Object subclass: #DeployedComponent
   instanceVariableNames: 'name dependencies
requiredServices providedServices communicationPaths
compositeComponents internalBehavior '
   classVariableNames: ''
poolDictionaries: ''
category: 'Dynamic Reconfiguration-Interacting
Components'
metaclass: ComponentPropertyMetaObject
```

Before any change the communication paths (and required services) of the *Scenario Analysis* component conform to the original control style of System A (See Figure 5):

```
   CommunicationPaths: a
Dictionary('EnergySectorKnowledge-ScenarioAnalysis'-
>a CommunicationPath )
   Required Services: a Dictionary('Knowledge
Representations of Scenario Predictions'->a
ProvidedService 'Scenario Predictions of Knowledge
Representations'->a ProvidedService )
```

Topology Change is manifested as a new communication path and connection added to the communication paths of the *Scenario Analysis* component. After *Topology Change*, the communication paths (and subsequently, after *Protocol* and *Interface Change*), the required services, conform to the new control style of System A (See Figure 7):

CommunicationPaths: *a Dictionary('EnergySectorKnowledge-ScenarioAnalysis'->a CommunicationPath 'ScenarioAnalysis-ScenarioAnalysis'->a CommunicationPath)*
Required Services: *a Dictionary('Knowledge Representations of Scenario Predictions'->a ProvidedService 'Scenario Predictions of Knowledge Representations'->a ProvidedService 'Short Term Predictions Based On Long Term Predictions'->a ProvidedService)*

VII. Summary, conclusion and future work

This paper explains what software evolution means in the context of software components that interact to implement services. Using domain analysis and design techniques, a domain model is defined of dynamic system reconfiguration due to user-driven or computing environment-imposed discrete change events. The domain model explicitly specifies the relationships that exist among changing global or local properties, changing behavioral or structural signatures, and intra- or inter-component change.

A. Summary

Based on the complete feature model of the domain analysis, Figure 11 summarizes the primary feature interactions of dynamic system reconfiguration.

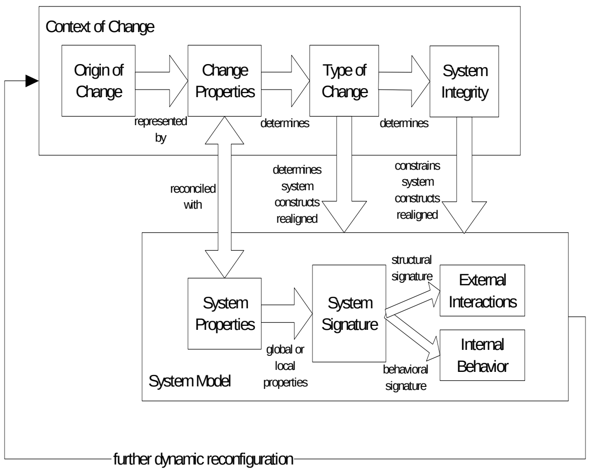


Figure 11. Primary Feature Interactions of Dynamic Reconfiguration

With respect to the common and variable causal flow presented in Section III, the process of dynamic reconfiguration progresses as follows:

- Sensing *User-Driven Change* and/or *Computing Environment Change*;
- Interpreting and then representing the particular *Origin of Change* as *Global Properties* or *Local Properties* with an associated *Reconciliation Policy* (this includes reconciliation with existing *System Properties*);
- Determining what feasible subsets of *Type of Change* (if any) satisfy the *Condition Change Criteria*;
- If necessary, realigning *Dependencies* and therefore possibly also *External Interactions* and *Internal Behavior*; and
- Ensuring *System Integrity Characteristics* when enacting change to help maintain overall system consistency.

The domain model of dynamic system reconfiguration is a useful conceptual framework for applying systematic techniques to engineer constrained software solutions within this open problem space. It is a significant artifact in the following roles:

In the role of a software architecture, it specifies a comprehensive dynamic reconfiguration capability in a manner that can be used to forward engineer dynamic systems;

In the role of a metamodel encoded by a component framework, it defines the steps of dynamic change that must be implemented by the framework and the 'plugin points' required for realization of the different aspects of the dynamic reconfiguration of a component-based software system; and

In the role of a reference model, it can be used to assess how dynamic an existing implementation is, the different levels of compliance of dynamic change, and how interoperable systems are at different levels of dynamism.

B. Conclusion

In the MetaclassTalk implementation, because executability is expressed as message passing among (pure) objects (where everything is a runtime object including classes), it became clear that a constraint could be represented by a metaobject which implemented a standardized protocol for creating a system signature 'fragment' specific to that constraint. It also became clear that system signature fragments could be composed as a system signature that represented the fragments of each global and local property and the overall signature of a system. This is not viewed to change the domain design per se but instead is considered to be a utilization of the model in a manner that only became apparent when the model was encoded with the model execution environment.

It was learned that the explicit metaclass programming facility of MetaclassTalk can effectively be used to encode the constrained model, as a static representation, at the metalevel. Base level objects could then be instantiated, based upon these static representations, as application-specific interactions. With respect to dynamic reconfiguration, it was learned that a base-level object could be an instance of a property metaclass that is unique to that base-level object. Through MetaclassTalk's mixin mechanism, emergent run-time properties could be dynamically composed that applied just to that object. The set of available mixins should also be adjusted dynamically. This was not addressed in this paper but is the subject of future work.

As a simulator of dynamic system reconfiguration, the MetaclassTalk programming environment is a build-time and run-time facility that is useful for validating that the domain design can be instantiated within an application-specific context. In this case, it is used to validate that the domain model is useful for dynamically reconfiguring a financial analysis system.

As a framework, the simulator implements 'plugin points' for realizing the general context of dynamic reconfiguration. In doing so, it provides the macro-level ordering of operations and the contextual information required at each step for a more sophisticated implementation. Generally, there is a need for a facility that allows backing out and restoring the configuration of the system if dynamic change is rejected.

The reconciliation of change properties with (existing) system properties is represented as a distinct action. When properties are expressed as constraints this reconciliation could be computed through a constraint solver (such as the Alloy Analyzer [10] or a different implementation of a similar capability). Importantly, depending upon the context, there may not be a solution. In this case the change may be rejected or the change constraints re-expressed in a manner that enables a resolution. This issue is not addressed in this paper but is the subject of future work.

The current implementation of the reconciliation of change properties with system properties is based on name matching (to associate a change property with a system property), with the change property taking precedence. A more sophisticated implementation would provide a constraint solver-like capability to determine whether multiple constraints were compatible.

Given that change constraints can be reconciled with (existing) system constraints, the UML models represented estimating the impact of Architectural Change upon system integrity as a distinct action. Computing this estimate is viewed to be an open problem and to be highly dynamic in nature since it is a function of the (changing) state of the system at the time the estimate is computed. Importantly, depending upon the context, a system may change significantly during the time it takes to perform this computation to such a degree that the estimate is no longer valid. This issue is not addressed in this paper but is the subject of future work.

This implementation simply accepts the cost of enacting reconfiguration when estimating the impact of change. A more sophisticated implementation would assess this cost taking into account previous results (through heuristics), the current configuration (through dataflow dependency analysis), and future results (through predictive simulation).

Given that the system will proceed with the change, the system model constructs relevant to the change type must change and each system integrity characteristic relevant to the change type must be assured. The UML models represent this in a “don’t care” order. Depending upon the context, a particular ordering may be preferred (for feasibility, efficiency, or other reasons). This issue is not addressed in this paper but is the subject of future work.

The current implementation of the 'don't care' ordering of the application of a change type and its integrity characteristics is arbitrary with the change made first. The implementation assumes integrity characteristics can be assured. A more sophisticated implementation would ensure system integrity including assessing what was the optimal order of application based on current conditions. For example, in the case of Substitution, if possible the change could be made during a time window when there were not any active references. As another example, in the case of *Internal Change*, when a new operation is deployed to replace an existing operation, both operations could co-exist for a limited time period to ensure continuity of any active dependencies on the old operation.

Finally, as discussed, a change to system dependencies is manifested as realigning external interactions or internal behavior through an appropriate type of change. Multiple global or local properties of one change can be represented as multiple kinds of dependency when updating the signature of a system. A more sophisticated implementation would establish an optimal change strategy when there is more than one way to dynamically reconfigure a system to achieve the same end result.

C. Future Work

The following is future work to be undertaken:

- Determining, through simulation, statistics, heuristics or other means, under what conditions a system undergoes adaptation and how continuous system evolution is interpreted as discrete change events that can be represented as global or local properties, including the different forms through which system constraints are expressed;
- Investigating the use of a SAT (satisfiability) solver for reconciling change properties with system properties including the reformulation of change constraints in a manner that ensures there is a resolution;
- Investigating dataflow dependency analysis (and like) techniques for assessing what the impact will be when a particular change is made including the problem that a system may change significantly during the time it takes to perform this computation to such a degree that the estimate is no longer valid;
- Investigating the preferred ordering of the application of change types and associated system integrity characteristics (for feasibility, efficiency, or other reasons);
- Investigating the cross-coupling effects that can occur because of the additive composition of change group models, including investigating

giving precedence to certain kinds of change and the serialization of the application of change types across change groups while maintaining global consistency;

- Investigating different levels of control in the context of decentralized software evolution (in support of load-balancing, change rejection and roll-back, fault-tolerance, state splitting and merging, and the system's life cycle); and
- Refining the domain model and the simulator (including their further validation and automatic transformation, dynamically adjusting the set of mixins, and the use of 'traits') when implementing this capability in an industrial context.

A key aspect enabling adaptive computing is the level of dynamism supported by the computing environment in which a system is running. To be fully dynamic, emergent runtime properties must influence not just what is computed but how a system computes what is computed. Open-ended facilities that enable a system to extend itself, based upon self-representation, are viewed as mandatory characteristics of any system that can adapt to emergent properties.

Using the domain model, the greater intent is the provision of a set of guiding principles, and an associated suite of techniques, which together help to ensure that a system, or a family of systems, can better adjust in a systematic way to dynamically changing run-time environments.

REFERENCES

[1] Marriott, K. and P.J. Stuckey, *Programming with Constraints: An Introduction*. 1998, Cambridge, Mass.: MIT Press.

[2] Walsh, D., F. Bordeleau, and B. Selic. "A Domain Model for Dynamic System Reconfiguration", in Proceedings of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems. 2005. Montego Bay, Jamaica: Springer.

[3] Walsh, D., F. Bordeleau, and B. Selic, *Domain analysis of dynamic system reconfiguration.* Published Online First in Software and System Modeling, DOI: 10.1007/s10270-006-0038-4, Springer-Verlag, 2006.

[4] Walsh, D., F. Bordeleau, and B. Selic. "Change Types of Dynamic System Reconfiguration", in Proceedings of 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). 2006. Potsdam, Germany: IEEE.

[5] Buhr, R.J.A. and R.S. Casselman, *Use Case Maps for Object-Oriented Systems*. 1996, New York, NY: Prentice Hall.

[6] Bouraqadi-Saadani, N., T. Ledoux, and F. Rivard, "Safe Metaclass Programming", in Proceedings of Object-Oriented Programming Systems, Languages, and Applications. 1998. Vancouver, B.C.: ACM Press.

[7] Bouraqadi-Saadani, N. and T. Ledoux, "Supporting AOP Using Reflection", in *Aspect-Oriented Software Development*. 2005, Addison-Wesley: New York, New York. p. 261-279.

[8] Bouraqadi-Saadani, N., "Metaclass Composition Using Mix-in-Based Inheritance" in Proceedings of 11th Smalltalk ESUG Conference. 2003. Bled, Slovenia.

[9] Scharli, N., et al. "Traits: Composable units of behavior", in Proceedings of *ECOOP 2003*. 2003: LNCS 2743, Springer Verlag.

[10] Jackson, D., *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. 2002, MIT Lab for Computer Science: Cambridge, Mass. p. 1-58.

D'Arcy Walsh received his Honours B.A. in 1981 from Queen's University in Kingston, Ontario. He received his B.C.S. in 1989, M.C.S. in 1994, and Ph.D. of Computer Science in 2007 from Carleton University in Ottawa, Ontario.

His research interests include software engineering methods and techniques that support the development and deployment of dynamic systems, including dynamic languages, dynamic reconfiguration, context-aware systems, and autonomic and autonomous systems research work.

Francis Bordeleau holds a B.Sc. Mathematics from University of Montreal, a B.Sc.A. Computer Science from Université du Québec à Hull, and a Master of Computer Science and a Ph.D. Electrical Engineering from Carleton University.

He is the Founder, President and CEO of Zeligsoft, a leading provider of market-specific embedded software development tools that enable the development of component-based systems. Francis is also an Adjunct Professor at Carleton University in Ottawa, Canada. Francis has over 13 years experience managing, researching, teaching and defining in the domain of Model Driven Development (MDD), software engineering, component-based technologies and Software Defined Radio (SDR) systems. He has worked, consulted and collaborated with numerous companies.

Bran Selic received his Dipl.Ing degree in 1972 and his Mag.Ing degree in 1974, both from the University of Belgrade in Yugoslavia.

He is an IBM Distinguished Engineer at IBM Canada and an adjunct professor of computer science at Carleton University in Ottawa. At IBM, he is a member of the CTO team, responsible for defining the strategic direction for Rational's software tool products. Bran has over 30 years of experience in designing and implementing large-scale industrial software systems and has pioneered the application of model-driven development methods in real-time applications. He is currently chair of the OMG team responsible for the UML 2 standard.