WHITE PAPER

Progress DataXtend®

COMMON MODELS IN SOA

Tackling the Data Integration Problem

Written by:
Dave Hollander
Mile High XML

PROGRESS SOFTWARE

TABLE OF CONTENTS

Enterprise service busses (ESBs) and service-oriented architecture (SOA) are the latest approaches to developing and integrating systems that are modular, flexible, and robust while reducing development costs and time to market. ESB and SOA are mature enough to have been successfully implemented and success stories abound. However, reports are also emerging of less than successful deployments citing complexity and unexpected costs as the causes[1]. These disappointing results highlight the limitations of SOA and ESB—limitations that many companies avoid by extending their development standards beyond those commonly understood to be part of the SOA and ESB to include a *common model* to simplify data integration.

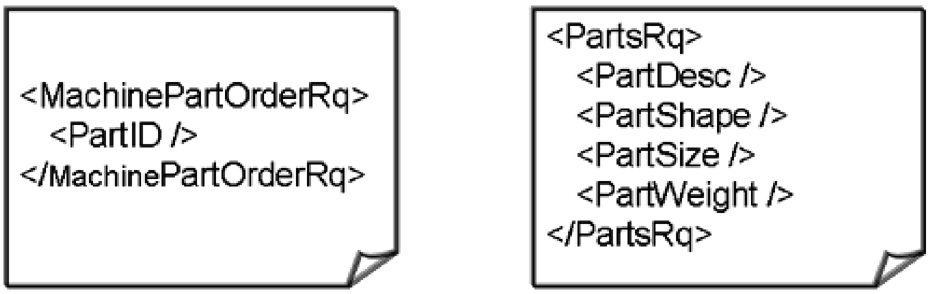
You're probably familiar with the use of common formats in integration. These common representations, or in some cases mutually agreed upon formats, are often found in B2B transactions. Common models (which have also been called common information models, common data models, unified data models, exchange data models, common data definitions or canonical models) include not just the common representation, but also the meaning of and relationships between logical entities—or the business context of the information. For example, common models often define not only which data elements comprise "customer," but what customer means, and how it relates to other entities such as mailing address, purchase order, and billing record. The information that describes the model, or the model's *metadata* is often described in a formal language such as the unified modeling language (UML).

To see how a common model can simplify SOA development, let's examine the standards at the foundation of SOA and ESB to better understand what they include and their limitations. SOA formalizes the principles of loose-coupling to achieve flexibility, independence and modularity. Key to this architecture is the *service interface* which formally defines how to interact with a service. Interfaces use *metadata* to define: input and output *documents* that contain the data the service uses, *operations* describing the message sequences, and *endpoints* describing the technical details needed to locate and access the service. While *service orientation* does not require the use of web service standards, over the past several years web service standards including XML, XML Schema[2], and Web Services Description Language[3] (WSDL) have increasingly been used as the format for service interface metadata.

1. Babcock, Charles "The SOA Gamble: One in Three Companies Are Disappointed, Our Survey Finds," Information Week, 8 Sept 2007. http://www.informationweek.com/news/software/soa/showArticle.jhtml?articleID=201804546
2. Henry S. Thompson et. al. "XML Schema Part 1: Structures, Second Edition; W3C Recommendation" 28 October 2004 http://www.w3.org/TR/xmlschema-1/
3. Roberto Chinnici et al. "Web Services Description Language (WSDL) 2.0; W3C Recommendation" 26 June 2007 http://www.w3.org/TR/wsdl20/

While the WSDL and XML Schema standards clearly define service interfaces documents, operations and endpoints, they do not describe everything needed to integrate services. They do not define how to move the documents between services, how to track the documents, or even how to interpret the documents. To illustrate the remaining needs, consider two different services for ordering machine parts: the first service accepts a simple message with just the part identification number. The second uses a more complicated message with a part description, physical characteristics such as shape, size and weight. Both conform to SOA principles and web service metadata standards, but are not interoperable. To use these services in an application you will also need to understand the part identification and descriptions, if the acknowledgement indicates success or failure, and how to reliably exchange messages with the service.

"The complexity that threatens many teams' success with SOA stems from reconciling the differences in the schemas used in service interface definitions and their implied semantics."



```
<MachinePartOrderRq>
   <PartID />
</MachinePartOrderRq>
```

```
<PartsRq>
   <PartDesc />
   <PartShape />
   <PartSize />
   <PartWeight />
</PartsRq>
```

Figure 1: Two machine part requests with different parts

Reliably moving documents between services is the primary job of an ESB. For simple service integration requirements, the endpoint can describe a transport standard such as HTTP. For more demanding applications, ESBs are deployed for reliable transport, tracking and auditing, integration with legacy systems, passing the message to multiple services, and for queues to provide temporary storage.

Understanding the documents being exchanged goes beyond what can be described in the schemas in the interfaces. Schemas describe the structure of the documents, which data fields they contain and how the fields are structured, but not the intent of the documents. Nor are the same schemas used for all services. *The complexity that threatens many teams' success with SOA stems from reconciling the differences in the schemas used in service interface definitions and their implied semantics.*

New metadata techniques and standards[4] are evolving for describing semantics, but these are not fully mature nor are they widely deployed. Successfully building large SOA applications requires resolving semantic differences using design processes. These processes create *data transforms* to convert one document format to another. The transforms are designed using tools to *define* mappings between fields in the source and target interfaces. These mappings are then expressed in XML Stylesheet Language

4. Semantic technologies include various ontology standards, RDF, OWL.

Transformation[5] (XSLT) or Java code and then deployed into the ESB as separate services. The use of transforms to reconcile syntactic differences in the documents allows each service to evolve with a greater degree of freedom and independence.

How transforms are typically developed is a significant source of complexity in large SOA projects. While the ESB and transforms break away from the complexity of point-to-point run-time integration they do not overcome the point-to-point dependency in the design process. Point-to-point design means that with each and every service added to the overall system developers must manually create transforms to each and every other service with which it interoperates. In the worse case, the complexity increases exponentially with each service added to the scope of the SOA project.

To create interoperable applications and services there has to be an agreement on the *meaning* of the exchanged data. That is, there has to be a shared, *common model* that describes the data in sufficient detail for all of the applications to understand the intent of the messages defined in their interfaces. The model may be informal, within the minds of the developers. This informal model is what developers must use when they use simple mapping tools as provided with many ESB systems. They must understand the intent of all of the fields in the interfaces and explicitly map each field to all the other fields in all the other interfaces. While this process may be acceptable in small projects, the complexity quickly grows as the scope of the SOA grows.

A more scalable design approach uses a formalized, shared common model for all developers to use to understand interface data requirements and to design mappings that reconcile the differences. With a common model, developers reconcile their interface to the common model once and tools create the transforms needed to integrate any pair of specific interfaces. The power of the common model approach is that it requires developers to map an interface only once and then compile as many different transforms as needed. A common model simplifies how transforms are created which provides a scaleable, semantically-aware design process for large scale SOA environments.

5. XSL Transformations (XSLT) Version 2.0; W3C Recommendation; 23 January 2007; Michael Kay; http://www.w3.org/TR/xslt20/

THE ROLE OF A COMMON MODEL: SIMPLIFYING THE INTEGRATION LANDSCAPE

Service-oriented Architecture describes a set of standards-based technologies and a design approach to create interoperable systems. SOA principles can be used to create services, aggregate services into composite applications, develop whole new applications, and to integrate existing applications. Whether building new applications or integrating existing ones, the goals are to create interoperable, sustainable, robust solutions; solutions that meet today's business needs while being flexible enough to meet future needs. The role of the common model is to simplify the SOA landscape and make it more practical to create and maintain.

"The role of the common model is to simplify the SOA landscape and make it more practical to create and maintain."

As SOA principles become more broadly applied, the SOA landscape becomes more diverse and heterogeneous, both from a technology and a semantic perspective. Accommodating a diverse landscape requires an integration-based architecture pattern where integration techniques are used to achieve a semantic consistency while allowing for and accepting broad-based heterogeneity. This pattern relies on transport and transformation services to enhance loose coupling between service interfaces. The common model's chief contribution to this pattern is to simplify the development of the transformation services to reconcile differences in the service document formats and semantics.

It might be tempting to use the common model in an application development architectural pattern, however, this approach does not scale. This pattern enforces homogeneity where a common model defines the internal and external data structures used by applications and services. The limitations of this approach are the significant commitment that it requires to achieve consensus before development can begin and a strong governance infrastructure to keep systems synchronized as enhancements are made. As it is scaled up, without an integration layer, this tightly-coupled approach creates complexity and cost overruns. For this reason, it is not practical to enforce or impose a single common model across all applications.

An integration-oriented approach allows us to treat applications and services as a heterogeneous collection of "black boxes," to ignore their inner workings, and concentrate on their public interfaces. The integration layer with its transport and transformation services provides interoperability. This approach provides developers a significant degree of independence to evolve their systems.

To fully understand the role of a common model, consider your SOA initiative from a distance, removed from the individual projects from which the SOA is constructed. We will not ignore the project level view, but just step back to consider the overall *integration landscape* and then return to how common models impact projects. The integration landscape includes all of the applications, databases, and data resources, as well as composite applications and any relevant trading partners and customer systems that need to interoperate within the context of the enterprise's business processes. Finally, the landscape includes the ESBs and other integration layer services specifically designed to facilitate service-to-service communication, message exchange, and reconciliation. In the integration pattern, the role of the common model is to simplify the development of the "shared data services."

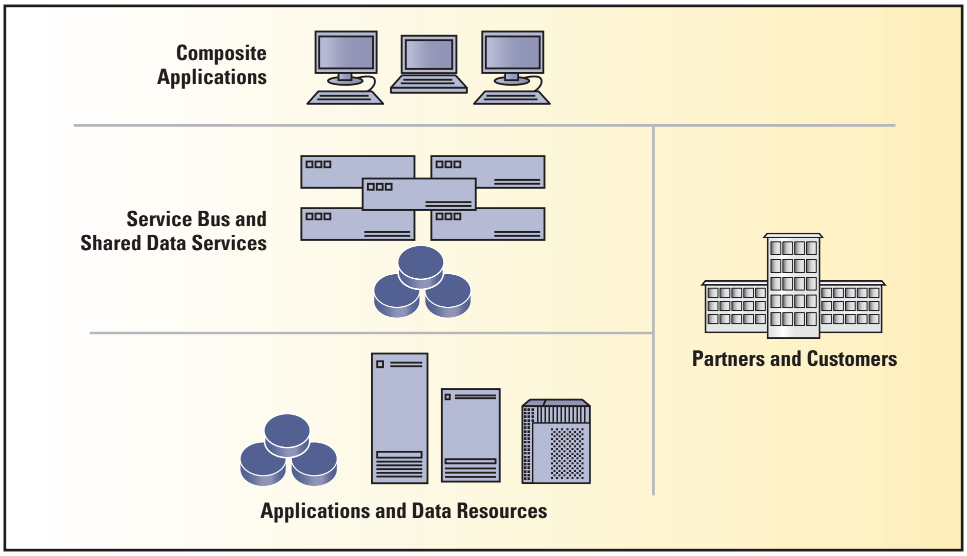

Composite Applications
Service Bus and Shared Data Services
Partners and Customers
Applications and Data Resources

Figure 2 Integration Landscape

A *metadata landscape* covers the same domain as the integration landscape but includes only the metadata exposed in service interfaces; metadata that describes the information flowing through the landscape. Without a common model in the metadata landscape we have different schemas representing each document in each service. Without the common model, each interface requires its own data integrity rules and its own mapping to each service it is integrated with.

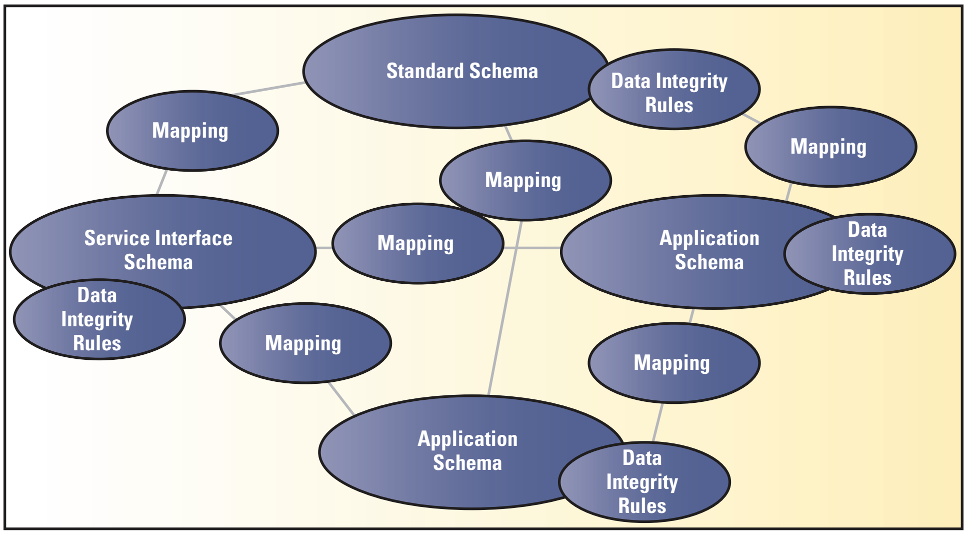


Figure 3: Point-to-Point Metadata Landscape

From a metadata landscape perspective, a common model dramatically reduces the number of mappings and overall complexity. A common model organizes all of our integration metadata into a shared context with a shared set of semantics. With a common model, each interface is mapped once and the data integrity rules are defined only once. The common model simplifies projects as well; instead of having to map to every other service with which it interacts, you can simply map the service to the common model.

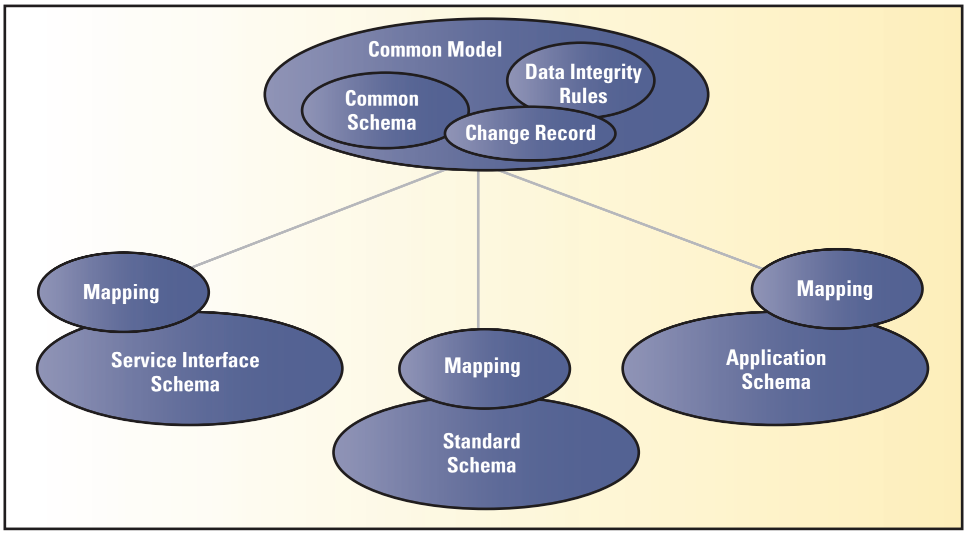


Figure 4: Metadata Landscape with a Common Model

In summary, *the primary role of a common model is to simplify the metadata landscape which in turn simplifies the integration landscape*. Adoption of a common model for use in the exchange of information in SOA greatly simplifies the metadata landscape which dramatically decreases the overall SOA complexity while preserving the flexibility of a loosely-coupled architecture.

DEVELOPING A COMMON MODEL

A common model will not simplify the metadata landscape and overall SOA development unless the development process and tools are designed to make practical use of the common model. The process described here meets these requirements. It begins with a one-time step of selecting, importing and customizing the initial common model. After that, the project processes consist of: 1) mapping service models, 2) deployment, and 3) maintenance. Separating the development and revision of the common model from the rest of the project tasks allows project teams to focus on a small portion of the landscape at a time and creates a natural checkpoint for good governance. As a practical matter, you will need to choose the right tools to help you design, deploy, and govern the data services based on the common model.

DEVELOPMENT REQUIREMENTS

To meet *enterprise requirements*, the process will need to be able to be executed across multiple concurrent projects in parallel. The process will have to have good impact analysis and reporting capabilities in order to understand and manage interdependencies between models and projects.

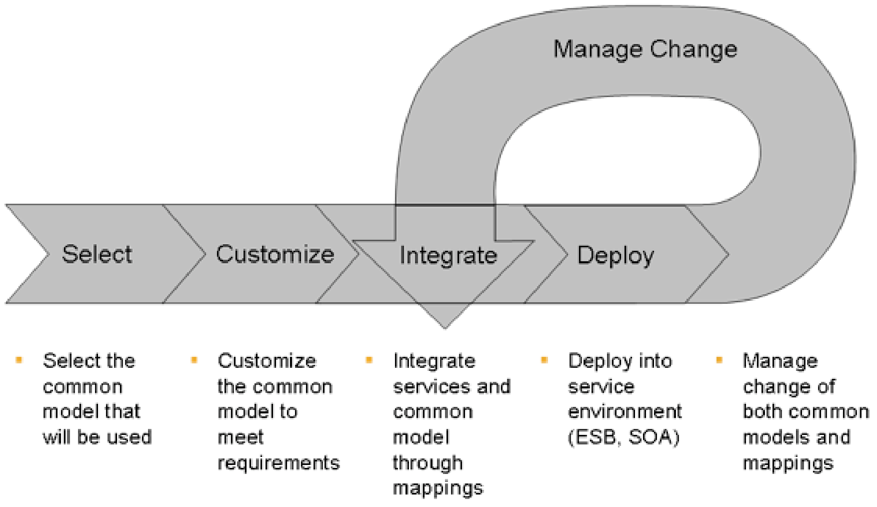

Manage Change
Select
Customize
Integrate
Deploy
Select the common model that will be used
Customize the common model to meet requirements
Integrate services and common model through mappings
Deploy into service environment (ESB, SOA)
Manage change of both common models and mappings

Figure 5: Integration Development Model

Adopting a common model means customizing it to meet the specific requirements of your landscape, however, in most companies the SOA integration landscape is rapidly changing and only partially understood. Notice that processes include a step to revise the common model to make it agile enough to support incremental development and deal with the rate of change each project.

"The lack of portability should be a major consideration when evaluating mapping solutions, especially those that are bundled with middleware or service bus software solutions."

In the SOA integration approach *performance and operational* considerations are essential to success. Transforms can have significant performance and operational implications depending on the programming language used to define the transform, where the transform is executed, and how exceptions are handled.

Any computer language can be used for transforms, but in today's SOA the W3C XSLT and Java are the most common. In spite of the widely held belief that XSLT is more "standard" than Java, the reality is that XSLT was designed for stylesheets and requires extensions to be effective in data transforms. These extensions and the complexity of writing XSLT code reduce the interoperability and portability of these transforms. Whereas, as a general programming language, Java includes most of the features needed to transform business data. Because of this, Java transforms are more portable and will typically outperform XSLT transforms.

Where transforms are deployed in your landscape can also have significant impact on the overall performance of your infrastructure. Transforms can be both compute and memory intensive. If your landscape can only run transforms in one system, then that system will quickly become a bottleneck. Portable transforms that can be quickly redeployed on a different system or even broadly distributed are needed to avoid these bottlenecks. *The lack of portability should be a major consideration when evaluating mapping solutions, especially those that are bundled with middleware or service bus software solutions*.

The point-to-point approach, or the ability to develop both point-to-point and two-transform services, is also an important performance requirement. Transforms can be deployed either as point-to-point services or as two transforms which create an intermediate *common data format* (figure 6). This common data format, sometimes called a "canonical format" is a document that conforms to the common model. The common format is desirable in many integration patterns, for example the publish/subscribe (pub/sub) pattern where multiple services will be receiving the same document.

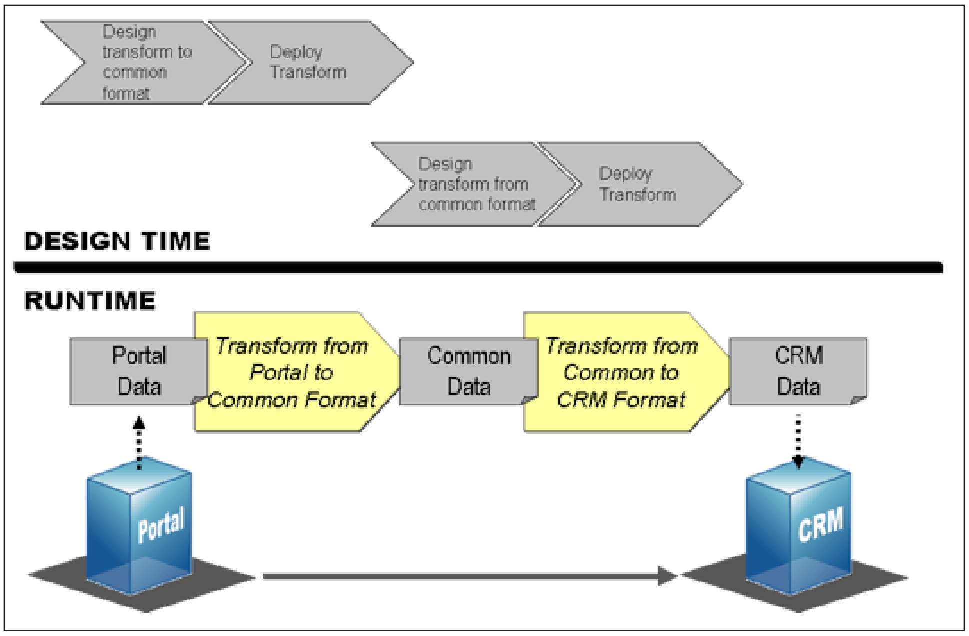

Design transform to common format
Deploy Transform
Design transform from common format
Deploy Transform
DESIGN TIME
RUNTIME
Portal Data
Transform from Portal to Common Format
Common Data
Transform from Common to CRM Format
CRM Data
Portal
CRM

Figure 6. Two Transform, Common Format Deployment

However, when applied broadly, the common format approach becomes unmanageable. As new applications and services are added to the architecture, extensions to the common message model are needed to meet their specific data requirements. As more applications or sources are added the size of this common message grows and grows becoming unwieldy to use and difficult to understand. As an alternative, some teams respond to new requirements by creating new versions of the common format and these versions proliferate creating even more complexity.

The common model approach avoids this complexity by developing the common model as a *design-time* standard. Changes are made to the model then *compiled* into transforms which are deployed into run-time systems. As the SOA grows and evolves, the existing run-time components can remain deployed, stable and unchanged while the design-time models change to accommodate the new requirements. *With the clear separation of design- and run-time, you can get the benefits of a common model without sacrificing the performance and operational benefits of point-to-point transforms.*

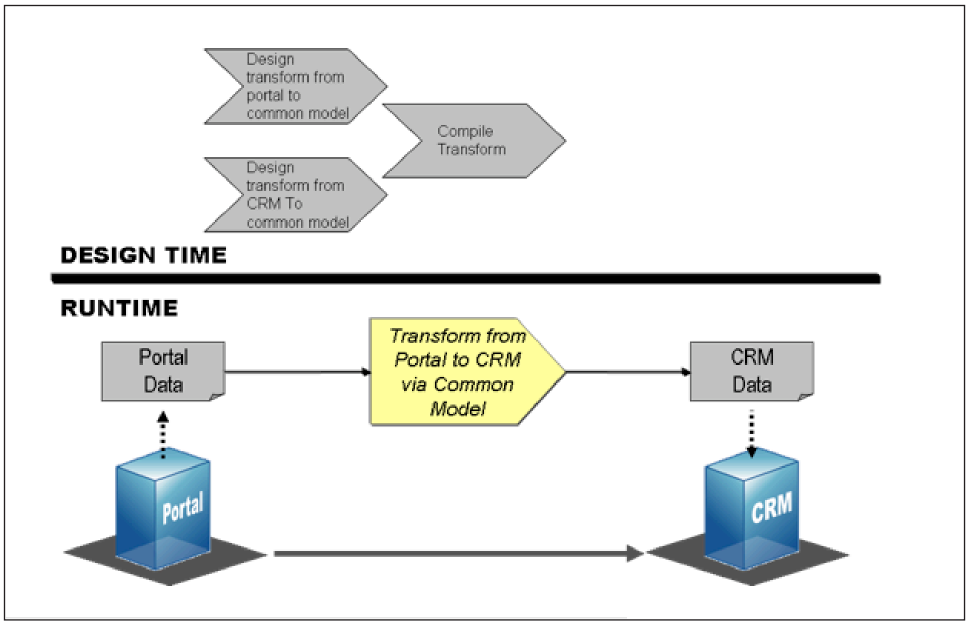
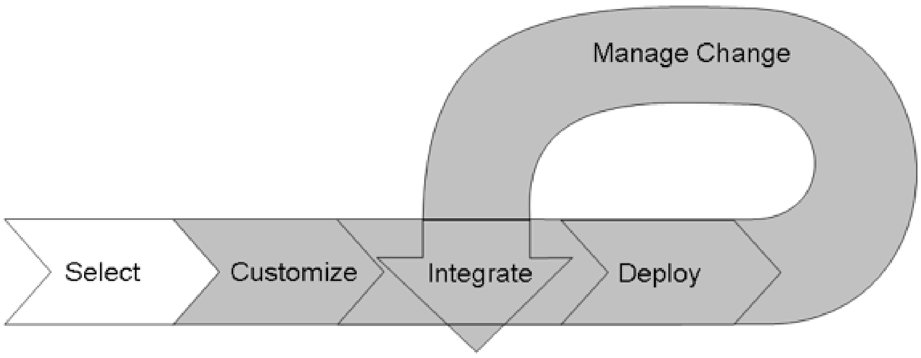

Design transform from portal to common model
Compile Transform
Design transform from CRM To common model
DESIGN TIME
RUNTIME
Portal Data
Transform from Portal to CRM via Common Model
CRM Data
Portal
CRM

Figure 7. Compiled Transform Deployment

In the common model approach, point-to-point transforms are compiled from service to common model mappings (figure 7). These transforms avoid the performance overhead of executing two transforms for each exchange which can become prohibitive for documents large enough to use most of the allocated memory. These point-to-point transforms are ideal for the SOA landscape where the *request/reply* exchange pattern is common. In request/reply data from one service is passed directly to the service that requested the data. Service reuse and scale are achieved by having multiple services directly request and transform the data they need.

GETTING STARTED: SELECTING THE BASIS FOR A COMMON MODEL



The development process begins with selecting then customizing the common model for its role as the center of the metadata landscape. After this is done, the project level tasks of importing service models and mapping them to the common model can proceed in parallel. However, even selecting a standard as the starting point for a common model can be complicated and contentious. For example, should teams select the industry-standard model[6] which best reflects their business process or should they use their current vendors' 'standard' implementation which might be more compatible with their current applications, technologies, and architecture?

In most industries there are several industry and vendor standards to choose from. Often an industry standard is considered because you may already have applications and services that communicate using the standard. Schemas developed by industry organizations are a great starting point because they encapsulate a significant amount of knowledge derived from participants with broad industry experience. Attempting to replicate this effort in-house would be costly, time consuming, and likely result in an inferior starting point.

Whether you select a starting schema using the criteria outlined below or the choice has already been made, the initial schema is unlikely to be complete enough to fulfill your enterprise-specific requirements. *Selecting an initial schema is important but* ***not*** *critical to success in deploying a common model*. Far more critical than the initial schema is the customization and enrichment needed to create a full common model complete with constraints and other semantic consistency rules. This enrichment process helps insulate your efforts from limitations of the initial schema.

Technical Quality is often the first and only criteria considered in the selection of a common model, but there are other important criteria for evaluation. Technical quality describes how well the data structures are captured in the technical metadata format being used. For example, are the XSD schemas well constructed, easily read and understood, consistent, and carefully crafted? Is the metadata technically compatible with your existing metadata standards and tools?

Standards Organizations can collapse because of issues within the standards organization. Don't overlook issues with the standards body simply because your partners or employees are involved. Factors to consider include: jurisdiction, adoption rates, industry alignment, openness, and financial stability. While, ideally, only standards from an organization that is credible and authoritative in its subject domain should be used as a starting point, a common model can be derived from any industry standard.

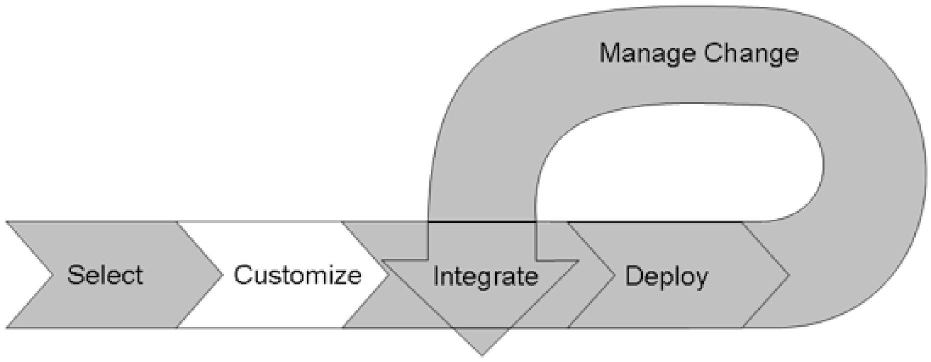
Business Alignment describes the degree to which the standard captures semantics relevant to your business and industry. Strangely, this is often a secondary

6. For more information see: Gilpin, Mike "Canonical Information Modeling Is Key To Many Information-As-A-Service and SOA Strategies" Forrester Research, 15 Nov 2007.

consideration—if there is poor alignment with your business, the standard will require extensive modifications to be useful. To assess business alignment, extract the dictionary terms and relationships defined in the standard and compare with those from your landscape. While no standard will ever capture all of the nuances important to running your business, many do a good job of establishing effective definitions for information and processes. This is not surprising for standards were created by experts in their fields working collaboratively within the standards organization where their depth of knowledge has been tempered by peer review and open discussion.

These criteria can not only help you choose an initial schema, they can also help you scope your common model development efforts. A selection that fits well with all criteria will require minimal customization. But, far more common will be a selection with compromises. Fortunately, with the right tools, any standards-based common model can be enriched or augmented to meet your enterprise standards and interoperability requirements. This extended common model, placed in the heart of your *exchange model*, can be used to tie together one or more standards. This means each group can make choices that fit best with their operating model without having to sacrifice applications and services data interoperability on the enterprise level.

CUSTOMIZING THE COMMON MODEL



The models created from an initial schema will need to be customized to be attuned to your business processes and semantics. Customization includes adding constraints, renaming, organizing and annotating elements to match your company's vocabulary and to make them easier to understand.

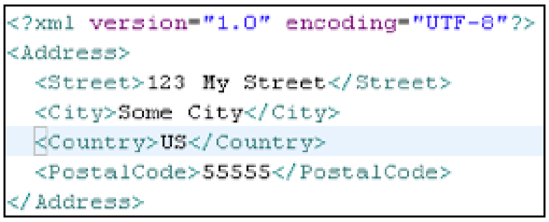
Organization and Annotations make the model more understandable. Annotations can be very valuable; they can describe the meaning of the element as well as complex rules and exception handing. Simply preserving descriptions from the data architects that present the assumptions that were made when the model was created can help inform integration teams as to how best to use the model.

Often annotations are defined in external documentation or spreadsheets and maintained out-of-band. These external text and spreadsheet files have to be maintained independently of the models themselves. This extra documentation is time consuming to produce, seldom referenced, and care must be taken to assure the documentation is kept up-to-date. This can lead to poor quality data when estimating the impact of future changes, or new projects—increasing the implementation risk on the project.

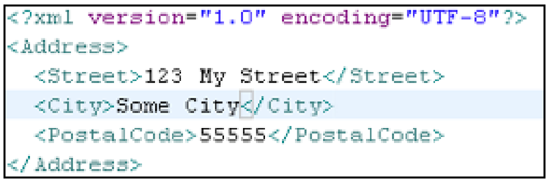
One way to mitigate this risk is to preserve the annotations in the exchange model, in-band with the development process. Annotations maintained within the exchange model are more likely to be current, and accurate. *By including the annotations in-band, or integrating documentation with the development process, we ensure the most current information is at hand for the development team. This cuts down on misinterpretation and misuse of the model, reducing the implementation risk of the project.*

Default values allow you to define what value to use for a data field when that field is not contained in the actual XML data.

For example, while a standard may be broadly defined to be used internationally, your systems may assume that a location is in the United States unless the data indicates otherwise. This can be captured in your models by setting the default for *country* in an address to "US." This would make the two XML documents at right have the same meaning even though the country is left out of the second.



```
<?xml version="1.0" encoding="UTF-8"?>
<Address>
  <Street>123 My Street</Street>
  <City>Some City</City>
  <Country>US</Country>
  <PostalCode>55555</PostalCode>
</Address>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<Address>
  <Street>123 My Street</Street>
  <City>Some City</City>
  <PostalCode>55555</PostalCode>
</Address>
```

Default values are very useful in a common mapping challenge: how to map an optional source field to a required target field. With the default value defined, mapping systems should be able to automatically create the transformation rules for these mappings.

Constraints are rules that limit the value domain of the data field or group. The models in your landscape will want to leverage all the constraints defined in the schema metadata and to add additional constraints that are specific to your data and system requirements. These constraints make the "service contract" nature of an interface clearer about what data is required to conform to the specification. Formalized constraints allow you to automate validating the data against the model to detect errors. A data file that conforms to the set of constraints defined in a schema is said to be "valid against that schema."

Ideally, the common model should fully describe a range of data that is valid for every element in the documents described in the service interfaces. However, most schemas from standards organizations have constraints. Adding constraints to the standard schema help give the schema "teeth," it narrows the range of XML documents that conform to the schema, ideally matching those that the service can work with. *Constraints strengthen a schema from a "suggestion" to a "standard."*

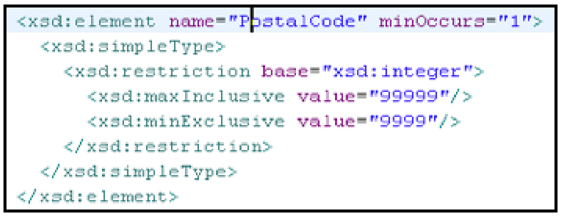
"To improve interoperability and the ability to detect data that does not conform to your requirements, your models should make all elements used and expected by your applications required."

You can always add constraints to the schemas describing documents that output from your services without risking interoperability with other systems that are implementing that standard. Adding constraints helps you ensure data will conform to both the standard and your own data requirements. Note that if you have to remove constraints, you must map the standard as another service to describe how to handle data that is now considered invalid.

In XML Schema, constraints can be placed on simple data elements (fields) using datatypes and facets. Constraints can also be placed on complex types (groups) using structure declarations and occurrence values. Like XML Schemas, constraints can be placed on attributes (fields) and classes (groups) in UML models. In this paper, XML Schema constraints will be illustrated. However, the format of your common model and the tools you use to implement it may handle constraints differently.

Simple Constraints specify that a specific field or group is required. In XML Schema, an element with the minOccur attribute set to greater than zero indicates it is required. For example, you can make a single postal code element in an address required by setting its minOccur and maxOccur value to one, as shown here.

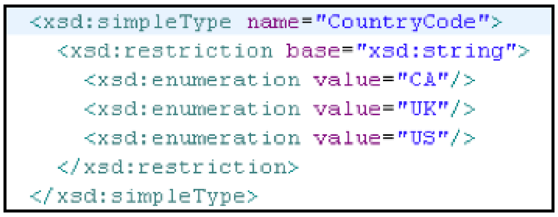
Datatypes are another means to constrain a schema. XML datatypes use *facets* to describe how atomic data elements are expected to be represented. Facets are often used to limit string lengths, such as 20 characters in a name. In the example at right, facets are used to constrain the valid values of a PostalCode to be within 10000 and 99999.



```
<xsd:element name="PostalCode" minOccurs="1">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minExclusive value="9999"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

To improve interoperability and the ability to detect data that does not conform to your requirements, your models should make all elements used and expected by your applications required. Since most standards define few required elements or datatype facets you will have to add these simple constraints.

Enumerations constrain a data field to contain only values from a specific set of values. For example, a *CountryCode* data element can be constrained to be only one of the values provided in the list of enumeration values.

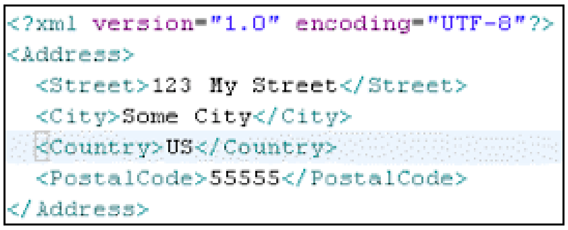


```
<xsd:simpleType name="CountryCode">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="CA"/>
    <xsd:enumeration value="UK"/>
    <xsd:enumeration value="US"/>
  </xsd:restriction>
</xsd:simpleType>
```

Often, standards leave enumerations out or make them "open," or even incomplete. Without enumerations, you will have to define rules to determine if "US" and "USA" are valid country codes and if they are the same code. Additionally, those rules have to be communicated in documentation accompanying the schema metadata.

Using enumerations within the model's schema helps make it clear what values are valid for a data field that is limited to a set of valid values. To make this customization to your models you will need to be able to create enumerations for some fields, or delete values for enumerations that are not valid in your landscape. Care should be taken in adding values to existing enumeration lists as this is extending not restricting the model and can cause interoperability problems.

Structural constraints describe the organization of data fields into groups. For example, the address shown here would be defined to be a sequence of fields, starting with the street and ending with PostalCode. Structural constraints typically should be adopted from the starting schema without modification.

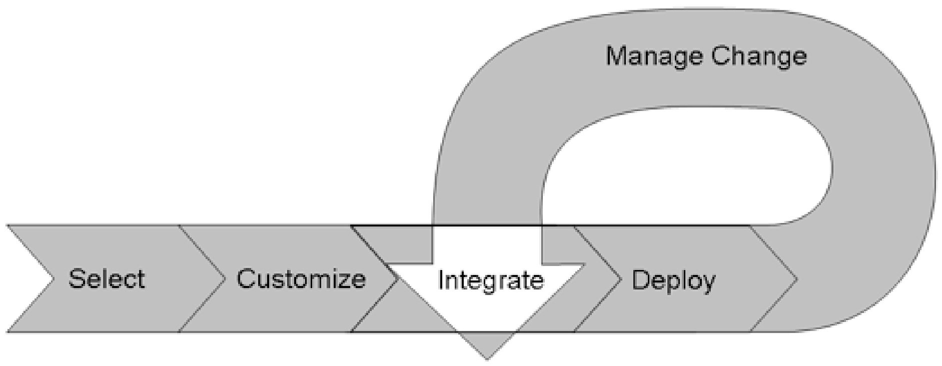


```
<?xml version="1.0" encoding="UTF-8"?>
<Address>
  <Street>123 My Street</Street>
  <City>Some City</City>
  <Country>US</Country>
  <PostalCode>55555</PostalCode>
</Address>
```

Semantic consistency rules describe complex constraints. In general, complex constraints define an element to be constrained and a different element whose value must be evaluated to determine the constraint. For example, a complex rule could declare that if one data field has a specific value then another field must be constrained to contain values in a specific range—if an address has a country value of "US" then there must be a valid five digit postal code. In addition, there may be a rule to define what happens when data is found to be invalid.

Complex "co-variant" constraints can not be described using XML Schemas but they are possible with some common modeling tools. *Complex rules included in an exchange model should focus on data integrity and not business logic*. This helps maintain a clear distinction between the specification of input and output documents that the service can work with from the logic that should be applied within the service itself.

INTEGRATION PROJECTS USING THE COMMON MODEL



Projects to integrate new functionality or applications into the SOA landscape are greatly simplified by using the common model. The integration effort is just a few simple steps that leverage the existing common model and simplify the application or service development effort. These steps are: 1) import the service metadata, 2) map the service documents to the common model, then 3) create, test and deploy the transforms into the integration data services.

The "map once, deploy many" nature of this architectural approach allows developers to concentrate most of their effort on business logic instead of all the details of the services it interacts with.

IMPORTING SERVICE METADATA

The first task is importing the metadata for the service. Here we read in the metadata regardless of format and allow it to be reorganized to be easily understood. While this task can be done manually using analysis tools and a simple XML Schema editor, such a process is time consuming and error prone. Full interface schemas typically have hundreds or thousands of unique datatypes, each of which will contain dozens of data elements for which the order, spelling and capitalization must all be faithfully reproduced. The only realistic and scaleable solution is to use a tool to convert the import formats, understand the file structures being used and normalize the style and structures to a common form.

Whether performed manually or automatically, the import step must simplify and normalize all of the differences to provide a common format which can be used in the later tasks. Specifically:

- The import process must be able to use the specific metadata format used by standards, application adapters and service interfaces which can include: XML Schemas, XML DTDs, Web Service WSDL files, and UML XMI files.
- The process must be able to collect all of the various files included or imported into the schema and represent them as a single model.
- The process must be able to rationalize the various styles found in the metadata.

MAPPING SERVICE METADATA

Service models represent the application and service interface documents. These models are created from the metadata that defines the exchanged messages and extended with mapping information. As with common models, the modeling process begins with importing the schemas, but with much less customization because the *service models* should closely represent the service or application they represent.

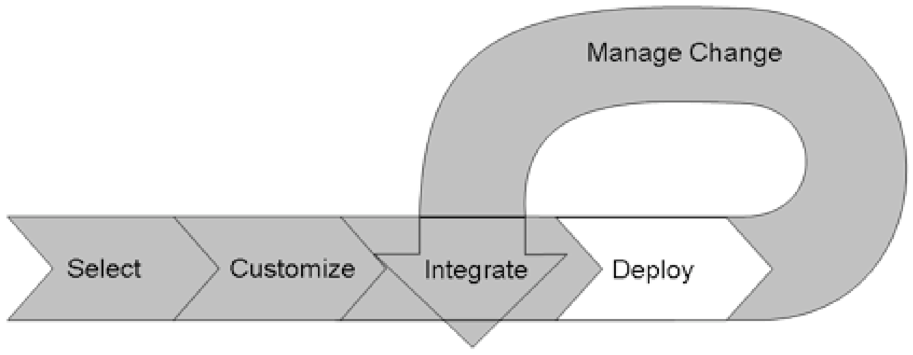
Once the metadata is imported and rationalized, its fields and groups are be mapped to the common model. Data conversion rules are added to the mappings if needed to define how to convert data from one datatype to another. These mappings identify which data in the service format relates to which data in the target format by defining field-to-field and group-to-group relationships.

A high quality mapping tool is essential to successfully deploying a common model. While there are fewer mappings using the common model than traditional point-to-point integration, the mappings play a central role and must be easy to develop and maintain.

While there are many mapping tools on the market, the importance of mapping to this approach makes features such as ease of use, group level mappings, embedded interactive testing, and the ability to manage test data sets essential. Mapping needs and requirements specific to using a common-model based architecture include:

- Support for conditional logic to express rules describing complex relationships between source and common data formats *and their reverse relationships*. The reverse relationships are particularly important to allow the system to calculate source-to-target mappings from the mappings to the common format.
- Embedded data integrity rules within the transforms to simplify the deployment and improve performance.
- Embedding the mappings within the exchange model to assure they are kept in sync.

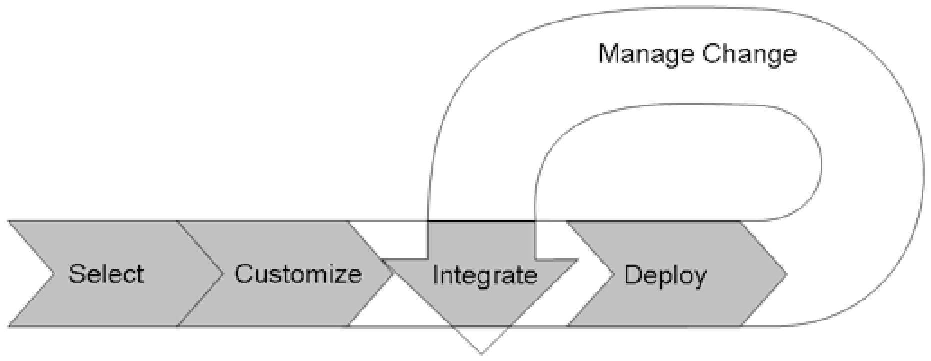
DEPLOYMENT



After the mapping process, the transforms needed for production are created as a result of the deployment process. In this phase we test, compile and then package for the target service environment. Finally, the transforms are inserted at the appropriate place in the business process flow.

For this approach to be successful the common model must be used to directly compile run-time components. This direct binding to run-time systems saves time and effort, prevents errors, and assures that the model will be kept accurate throughout the development process. Without this assurance, the model is no longer authoritative and will quickly become neglected and obsolete.

MAINTENANCE



Change is inevitable throughout the integration landscape. Services, applications, standards and even the integration systems all evolve. This change is driven by threats or opportunities in the market that the enterprise must respond to. The common model approach to SOA helps give IT the agility to execute on their new strategies and tactics by refactoring process, applications, infrastructure and data. It helps create a flexible IT infrastructure that can change rapidly, at a low cost, with a low impact to the SOA landscape.

"Effective, affordable and responsive maintenance processes are a significant benefit of the common model approach."

Effective, affordable and responsive maintenance processes are a significant benefit of the common model approach. They provide the ability to assess and scope the impact of change and quickly and efficiently deploy updated metadata and transforms to affect those changes.

The common model and surrounding exchange model provide all the details needed to understand and analyze the impact of change. The data about the magnitude of the work can drive governance processes, budgets, timelines and even cost-benefit discussions—after all, not all change is good especially in light of other competing initiatives for time, money and resources. The governance team can focus on how to manage the change rather than consume time and money on the invasive tasks of discovery and research.

With data at hand that details the impact of changes, issues are identified and development activity assigned. The development activity includes creating mappings to meet the needs of the new or modified services. These mappings are tested then deployed.

Effective maintenance can heavily leverage technical capabilities that are often overlooked or undervalued. These features include:

> A *unified metadata format* for all services enables more complete analysis and minimizes developer availability and training issues.

> *Reports* to understand and review new schemas and models to be added to the landscape.

> *Difference analysis* that highlight changes to schemas.

> *Impact analysis* to understand the interrelationships between the various source models and their maps between the models.

> *Change management tools* that allow analysts to accept/reject changes to the models in much the same way as editors use "Track Changes" in Microsoft® Word.

> Interactive, easy to use testing tools applied in-line with the development process that use test data maintained in the exchange model create higher quality components, save significant time, and minimize the impact on the existing production environment.

CONCLUSION

Mainframes, client-server and now SOA continue a trend in computer architecture—over time systems are being designed with smaller, more modular components. Service orientation creates systems by integrating large numbers of services defined by their interface metadata. Successfully deployed service oriented architecture integrates services into affordable networked applications that are modular, flexible and robust.

This is made practical by integration technology that leverages web service standards that define how services formally describe their interfaces. With standardization come scaleable, affordable technologies like the Enterprise Service Bus (ESB). While ESBs can connect a large number of services, they do not provide a scaleable approach to reconciling the difference between services. For the SOA methodology to successfully scale, it must extend to include reconciling semantic differences between services.

The common-model driven approach is a pragmatic and cost effective solution to reconciling these semantic differences. The approach focuses on minimizing developer efforts while creating programs that can be effectively deployed in today's most demanding environments. Today's structural transforms designed with common models and deployed in the integration layer will continue to be needed long into the future to reconcile structural differences prior to semantic processing. Not only is the approach practical for today's needs, it helps organizations prepare for future advances in semantic technologies by establishing a shared semantic service in the infrastructure.

AUTHOR'S BIO

Dave Hollander has been teaching computers how to help humans communicate for over 20 years. He is a technology strategist and architect who has been on the leading edge of many key information technologies. Dave's career has concentrated on information management, blending new standards and technologies with corporate opportunities and needs.

Since its inception Dave has had a leading role in the development of XML. He is a co-inventor, member of the first XML Working Group, co-chaired the XML Schemas Working Group, and co-chaired in the Web Services Architecture Working Group. Dave co-authored the W3C Standard, "Namespaces in XML," co-authored a book titled XML Applications published by Wrox Press, and was technical editor for XML Schemas from Morgan-Kauffmann. Dave has also contributed to a variety of eCommerce standards: MISMO, OAGI, RosettaNet, OBI and the ECO Framework.

Focused on business oriented information management, Dave's corporate achievements include founding and chief architect of www.hp.com, leading the development of Contivo's semantic modeling technology, deployment of an internal HP-wide search engine and development a CD-ROM publishing system for all of HP's technical manuals. Dave has been on the advisory board for Cohera, XMology, OSM, and Noonetime and held senior technology positions at Contivo, CommerceNet, Hewlett Packard and Bell Laboratories.

Dave can be contacted at dmh@mhxml.com.

Worldwide Headquarters
Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
www.progress.com

For regional international office locations and contact information, please refer to www.progress.com/worldwide