AgilPro Metamodel
vs
XPDL 1.0 Specification

By Steve Egbert
Bull HN Information Systems Inc.
steve.egbert@bull.com
Revision 1.0

This paper is an assessment of the compatibility between the AgilPro metamodel and XPDL for the purpose of identifying enhancements needed for the Eclipse JWT project. I have decided to initially approach this issue from the direction of importing an arbitrary XPDL into the AgilPro metamodel. To my mind, this is the quickest way of determining if the metamodel is complete enough to describe anything that can be described by XPDL.

I have also decided to initially focus on the XPDL 1.0 specification, with Bonita extensions. The XPDL 2.0 specification is massively more complicated than the 1.0 spec, and a thorough treatment of it would take considerably longer. Starting with the 1.0 spec will give us a basis to get some initial traction on the project, and if this document proves not suitable, we can take a different track more rapidly.

I am including the Bonita extensions because every XPDL implementation is expected to have vendor specific enhancements. The Bonita engine is the implementation that I am familiar with, so that is what I am able to document. Hopefully the issues that are highlighted accommodating the Bonita extensions will be sufficient to accommodate other vendor specific extensions.

There are two similar but distinct questions that might be asked:

1. Can everything that can be described by XPDL be modeled in AgilPro
2. Can everything that can be modeled in AgilPro be described with XPDL.

In this particular document I am specifically analyzing the first question. In the event that I should note items that have a bearing on the second question, I will mention them, but this document will not be an exhaustive analysis in this regard.

References:
The focus of this document is describing the information that will need to be available in the metamodel to be able construct any XPDL file. I have not exhaustively delineated every nuance of the XPDL Schema, and in some cases have summarized element hierarchies with a textual designation of the information that would be needed to construct them. This document should not be taken as a normative specification of XPDL

requirements. Any area where you find this document to be unclear or should it differ from the XPDL specs, the XPDL specs rule:

XPDL 1.0 Specification:
http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

XPDL 1.0 Schema:
http://www.wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd

Conventions:
Unfortunately, we have terminology overloading between AgilPro and XPDL. For example, the term Activity. An AgilPro Activity corresponds to an XPDL Workflow Process or an APDL ActivitySet, and an XPDL Activity corresponds to an AgilPro Action. Therefore, when there is need to differentiate between them, this document will use the namespace colon notation. (AgilPro:Action and XPDL:Action)

Data is handled somewhat differently in AgilPro and XPDL. In order to discuss this meaningfully, I am going to use the metaterm data item. This represents a piece of data at the level that corresponds to a single argument of an Application's function call:

myReturn = foo("bar",max); → myReturn,"bar", and max would be data items.

A data item is atomic from the perspective of the process being defined, in that it is the smallest unit of data that is directly manipulated by the process description. A data item may indeed have a rich internal structure, and that structure will be described in the definition of the data item, in sufficient detail that the workflow engine can construct a concrete object of that data item. However, the process description will not directly manipulate any of this internal structure.

General Model Mapping Discussion

One of the first things that jumps out at you when you begin to compare the XPDL specification with the AgilPro metamodel, is that most of the entities in XPDL have many more parameters than are currently present in AgilPro. A significant portion of these are not required to be present in a valid XPDL by the XPDL schema. This does not necessarily mean it is optional for us to support these parameters. A distinction should be made between being optional in the schema, and whether or nor the data is needed to execute a workflow process. A particular vendor implementation may need the information specified in some of these optional parameters, and so they would be required for that implementation. I think we will need to support all of them.

Another issue is how to handle vendor specific extensions. XPDL1.0 has an ExtendedAttributes element sprinkled at various strategic locations throughout the specification, within which custom extensions can be added. In addition to this mechanism, the XPDL2.0 schema allows additional namespace qualified attributes and

elements to be added pretty much wherever attributes and elements exist in the schema. This is intended to allow direct expansion of the schema by adding vendor specific attributes and elements directly into the existing structure without having to resort to embedding them in the awkward ExtendedAttributes element of XPDL 1.0.

In addition to vendor specific extensions, different workflow engine implementations might support the XPDL specification differently. For example, XPDL:DataType can define an almost unlimited number of data types. The Bonita engine only supports String and Enumeration (standard and with one vendor specific extension). It also uses the vendor specific extension “Hook” to call legacy procedures on the server, instead of XPDL:Application. I would expect other workflow engines to have their own, differing, restrictions.

We will obviously need to be able to specify to the tool the output language, (XPDL, BPEL, others). I suggest we also need to be able to specify the specific implementation. This way, the validation rules that we check against and the specific XPDL encoding can be properly tailored. I believe being able to tailor JWT to the target workflow engine implementation is a VERY important point, if we want our tool to replace the vendor specific tools as stated in our mission statement. The reason for the proliferation of vendor specific tools is, in my opinion, that they are tailored to the specific peculiarities of the corresponding execution environment. Thus, they are more efficient at generating a product that will run as intended the first time. If we only produce generic output, production users will naturally gravitate to a tool that will reliably alert them at design time to the things that their particular workflow engine will have problems with.

Michael Giroux of Bull has suggested we use Eclipse extension points to accommodate various vendor specific workflow engine plugins to our JWT plugin. We would have to provide an extension point in, at a minimum, our data model, our validation engine, and our workflow language export module, and probably other areas as well. A vendor specific plugin would extend each of these areas to be able to handle the vendor specific features. Perhaps the plugin would even entirely replace the validation engine, the export engine, or other tool sections, with its own. Different languages, such as BPEL could be accommodated this way through different plugins. We could supply, as standard, a generic XPDL plugin, as well as those that our partners desire, such as for Bonita. This would allow any vendor to develop a customization plugin for their specific implementation, without requiring us to modify the JWT core to accommodate them.

Another issue is certain XPDL elements that are not rigorously defined. For example, the XPDL1.0 Xpression element, which is used in conditions, is defined as type xsd:all, which means any combination of characters or elements.

We will need to decide whether we want to be able to import an XPDL file into our tool and get back out an identical file, or just an XPDL that has the same functionality. If we want an identical file back out, we will have to store all of the data in the incoming file in our model somewhere. For example, every XPDL element that is referenced by another has an ID, which must be unique over its scope and is used as the reference. If we want

the IDs to remain the same, we will have to store them somewhere in the data model. Alternatively, we can create a new ID for each such element when we export the XPDL file.

With respect to parameter passing to subflows and applications, XPDL defines FormalParameter lists and ActualParameter lists. The FormalParameters are similar to the argument and return specifications in a function definition. The actual parameters correspond to the values that are passed to the function when it is called.

int foo(string S1, int w) {….} --- These correspond to the formal parameters

int j = foo(“bar”,max); --- These correspond to the actual parameters

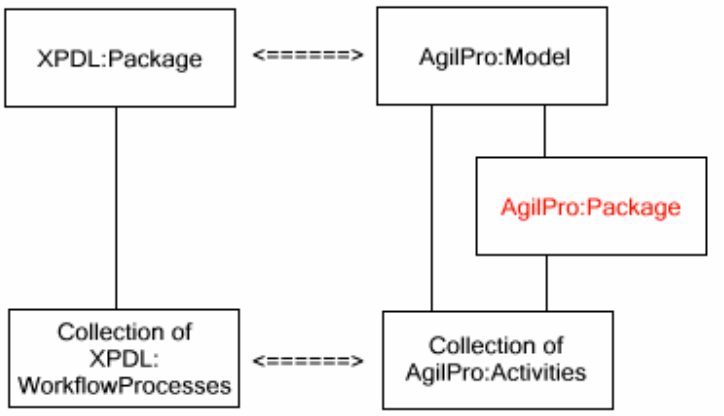
The actual parameters are modeled very well by AgilPro:Data.Parameter.
I think the area of describing formal parameters of an application needs some more thought. See XPDL:(Tool)Activity below for an example and further discussion.

To avoid further terminology proliferation, I will adopt these terms, formal parameter, and actual parameter in my discussion.

XPDL:Package maps to **AgilPro:Model**

XPDL:Package is the root element of an XPDL file, which corresponds to the AgilPro:Model element.

NOTE: There is no XPDL concept corresponding to AgilPro:Package.



XPDL:Package(AgilPro:Model) contains a collection of XPDL:WorkflowProcesses(AgilPro:Activity). In XPDL, there is no intermediate container that can be a child of an XPDL:Package that holds a collection of XPDL:WorkflowProcesses. In AgilPro, a collection of AgilPro:Activities can be a child of either AgilPro:Model or the intermediate container AgilPro:Package, which itself can be a child of AgilPro:Model.

One apparent choice for an AgilPro Model that contained AgilPro:Packages would be to correspond each AgilPro:Package to an XPDL:Package. That would mean generating a separate XPDL file for each Package element, and putting an XPDL:ExternalPackage reference in the main XPDL file. This is possible, but it breaks the Eclipse paradigm of generating an XPDL output file that represents the AgilPro model file. In order to import it back into AgilPro from the XPDL, you would have to access and process all of the separate XPDL files referenced in the main file.

Another approach would be to rearrange the graph when we generate the XPDL so that all the AgilPro:Activities are directly subordinate to the AgilPro:Model. This may or may not be possible, depending on whether the parameters contributed by the AgilPro:Package can be re-distributed properly between the AgilPro:Model and AgilPro:Activities elements, without loss of meaning. With this approach, the XPDL would describe a workflow that would execute the same, but would produce a substantially different node topology if it was re-imported back into AgilPro. It should also be noted that "rearranging the graph" may prove to be a non-trivial task.

XPDL:WorkflowProcess maps to **AgilPro:Activity**

XPDL:Activity

There are several different types of an activity in XPDL:

- Basic - The process is performed manually
- Route – A "dummy" activity that does nothing, but whose built in split/join semantics are used to construct more complex transition cascading
- Block – An activity node that contains a subgraph of nodes and transitions. This is distinct from a subflow in that the subgraph is not in another XPDL:WorkflowProcess, but is wholly contained within the current XPDL:WorkflowProcess
- Subflow – An activity node that contains a reference to another XPDL:WorkflowProcess. This may be another WorkflowProcess in the same XPDL:Package, or it may be a WorkflowProcess located in an entirely different XPDL file which is referenced in an XPDL:ExternalPackage element.
- Tool – The process is performed by an application

Each different type is modeled in AgilPro in a different manner.

It should also be noted that only Subflow and Tool type activities support parameter assignment, with a correspondence between Actual and Formal Parameters.

XPDL:(Basic)Activity maps to AgilPro:Action + (AgilPro:Fork/ Join/Decision/Merge nodes)

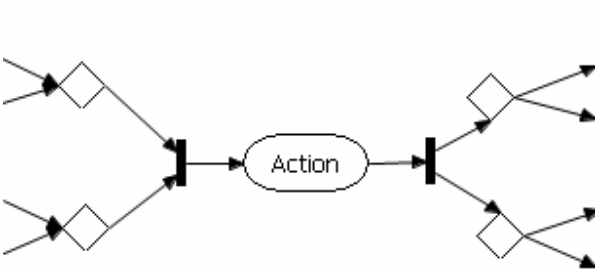
AgilPro:Actions may have only one incoming and one outgoing AgilPro:ActivityEdge. All XPDL:Activities may have multiple incoming and outgoing XPDL:Transitions. They have embedded joining and splitting semantics. This can be modeled in AgilPro by an association of AgilPro:Merge/Join and AgilPro:Decision/Fork nodes with the AgilPro:Action node, such as:

or

In addition, the embedded XPDL:Action splitting and joining contain semantics about the nature of the junction. The AgilPro:JoinNode and AgilPro:ForkNode have the AND senantics. The AgilPro:ForkNode and AgilPro:DecisionNode have the XOR semantics. To wit:

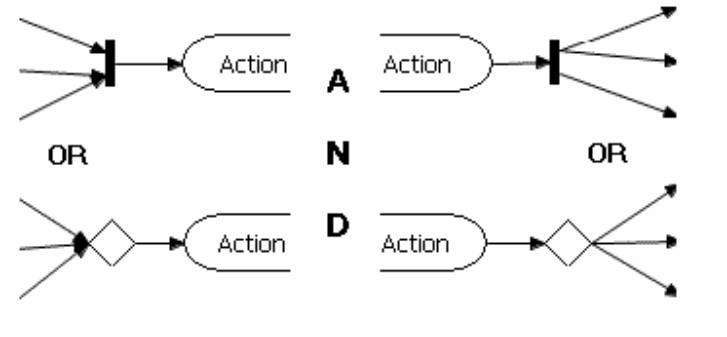
- AgilPro:JoinNode – the output edge is activated when all of the input edges have been activated.
- AgilPro:ForkNode – all of the output edges are activated when the input edge is activated, except for those edges that have an associated condition that is not satisfied
- AgilPro:MergeNode – the output edge is activated whenever one of the input edges is activated
- AgilPro:DecisionNode – when the incoming edge is activated, one of the outgoing edges is activated. The selection is based on the conditions attached to the outgoing edges. However, the XPDL semantics incorporate the concept of testing the conditions on the outgoing edges in a specific order until the first true condition is found. This aspect currently does not appear to be supported by these AgilPro constructs.

The XPDL schema allows multiple joining and splitting elements per XPDL:Activity as in:



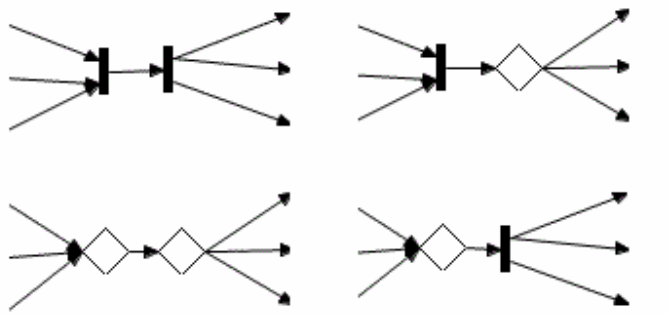
However, multiple joining elements are logically inconsistent, because there is no way to specify which transitions are associated which joining element. There is a way to associate outgoing transitions with the splitting elements, however language in other parts of the XPDL spec implies that an XPDL:Activity implements only one splitting element. It really does not matter for the purposes of this document, as long as we have elements that contain the splitting syntax (the AgilPro:DecisionNode and AgilPro:ForkNode), the model can handle either case.

Therefore I am adopting the following for the normal model of the XPDL:(Basic)Activity



XPDL:Route maps to AgilPro:Fork/Join/Decision/Merge nodes

An XPDL:Route is the same as a Basic Activity where the activity does nothing. It is provided so that the embedded transition joining and splitting semantics can be used to construct more complex cascading transition conditions. It would map to AgilPro like an XPDL:(Basic)Activity without the AgilPro:Action inside.

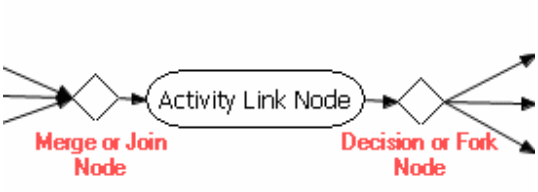


Note that in transforming from a network of interconnected AgilPro:Merge/Join/Decision/ForkNodes, each AgilPro node can be replaced with an XPDL:Route element, only half of which is actually used. This will directly produce a functionally equivalent XPDL network, however it is not the optimal graph. Constructing an optimal network of XPDL:Route elements from a complicated interconnected network of the AgilPro nodes would involve combining the AgilPro nodes in various combinations to produce the optimum network. This is a non trivial task.

XPDL:BlockActivity maps to AgilPro:ActivityLinkNode (+ AgilPro:Fork/Join/Decision/Merge nodes)

An XPDL:BlockActivity is an activity that executes a self contained activity-transition map. This map is contained elsewhere in the **same XPDL file** (In an XPDL:ActivitySet)

NOTE: The AgilPro:ActivityLinkNode has not been released yet, so this section is tentative. The AgilPro mechanism of defining the nodes inside the ActivityLinkNode has also not been determined. I recommend opening an additional lower tab, which will contain the graph of the nodes inside the ActivityLinkNode. This mechanism would also facilitate handling ActivityLinkNodes nested inside other ActivityLinkNodes, which is allowed for the corresponding XPDL:(Route)Activity elements.



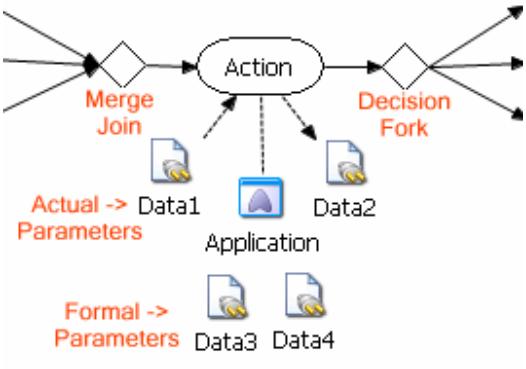
XPDL:(Subflow)Activity does not currently map to AgilPro

An XPDL:Subflow is rather like an XPDL:BlockActivity, except the elements inside the Subflow are described by a reference to a **different XPDL:WorkflowProcess** in the same or a different XPDL file. A Subflow also allows for the mapping of actual parameters to the formal parameters of the WorkflowProcess, whereas a (Block)Activity does not.

It should be noted that the XPDL:FormalParameters are defined in the XPDL:WorkflowProcess element that is referenced, which may be in another file. The ActualParameters are specified in the Subflow node element. This is going to make parameter mapping somewhat problematic. It is anticipated that parameter mapping will be handled in a similar manner to that described in the XPDL:(Tool)Activity, described below.

I would suggest that we should not try to diagram the inner elements of a (Subflow)Activity in this AgilPro editor. Rather, when the subflow element is opened for viewing or editing, if it is in the same file we should open another tab on the bottom, and if it is in a different file, we should create another instance of the AgilPro editor.

XPDL:(Tool)Activity maps to AgilPro:Action + Attached AgilPro:Application (+ AgilPro:Fork/Join/Decision/Merge nodes)



The AgilPro:Action maps to an XPDL:(Tool)Activity which references an XPDL:Application. The FormalParameters list of the XPDL:Application is derived from

AgilPro:Data.Parameters that are entered into the AgilPro:Application.InputParameters and OutputParameters lists. The XPDL:Activity.Tool.ActualParameters list is derived from AgilPro:Mapping elements in the AgilPro:Activity. The AgilPro:Data.Parameters that are used as actual parameters map to XPDL:DataFields. The AgilPro:Data.Parameter elements that represent the formal parameters of the application map to XPDL:FormalParameter elements.

I think this is confusing and unwieldy. My impression is that what makes it that way is dealing with the formal parameters of the application. You are describing the parameters of a function call, not actual data that is present in the workflow process. I think it might be useful to introduce something such as "argument", which would describe the calling and return argument specifications of the application (or subflow). This would allow AgilPro:Data.Parameter to always represent an actual instantiation of a piece of data.

Also, in XPDL, a FormalParameter can be of type "IN", "OUT", or "INOUT". There is currently no way to put the same AgilPro:Data.Parameter in both an AgilPro:Application.InputParameter and an AgilPro:Application.OutputParameter list.

In an XPDL:(Tool)Activity, multiple XPDL:Tool elements can be associated with one Activity. AgilPro can associate only one AgilPro:Application with an AgilPro:Action.

XPDL:Transition maps to AgilPro:ActivityEdge + AgilPro:Guard

An XPDL:Transition maps directly to an AgilPro:ActivityEdge. An XPDL:Transition may have an XPDL:Condition. I suggest we put the condition semantics, which consist of a type and an expression in the AgilPro:Guard element.

XPDL:Participant maps to AgilPro:Role

The XPDL:Participant performs the same function as AgilPro:Role. However, XPDL:Participant has a number of additional types of participation which we will need to support. Unfortunately, one of the XPDL:Participant types is named Role.

XPDL:Deadline does not map

An XPDL:Activity may contain an XPDL:Deadline, which specifies a duration or absolute time of the deadline of the activity, and specifies exception handling if the deadline is not met. There is currently no such entity in AgilPro.

XPDL:DataField

An XPDL:DataField is the only XPDL data item. They are what are what is mapped to the formal parameters of Application or a Subflow. They may be a simple basic type such as an integer or a string or they may be an arbitrarily elaborate composition of other data structures.

An AgilPro:Data.Parameter, is what is involved in argument mapping, and so is the closest analog of the XPDL:DataField. However, it does not stand alone, as an XPDL:DataField does, but must be a child of an AgilPro:Data element. In fact, the data

type of the parameter is a property of the parent AgilPro:Data element and not the AgilPro:Data.Parameter element. Therefore, the XPDL:DataField does not correspond directly to the AgilPro:Data.Parameter element, but rather to one within the context of its enclosing parent AgilPro:Data object.

This suggests the conceptual model of considering an AgilPro:Data element as a collection of data items (the children AgilPro:Data.Parameter elements), all of which are of the same type. However, in the AgilPro tool, in addition to adding parameters to an AgilPro:Data element by defining child Parameters, you can also select the combo box tool in the Eclipse properties view and add any other parameter that is defined under any other AgilPro:Data element, regardless of its type. This is inconsistent with the concept I proposed.

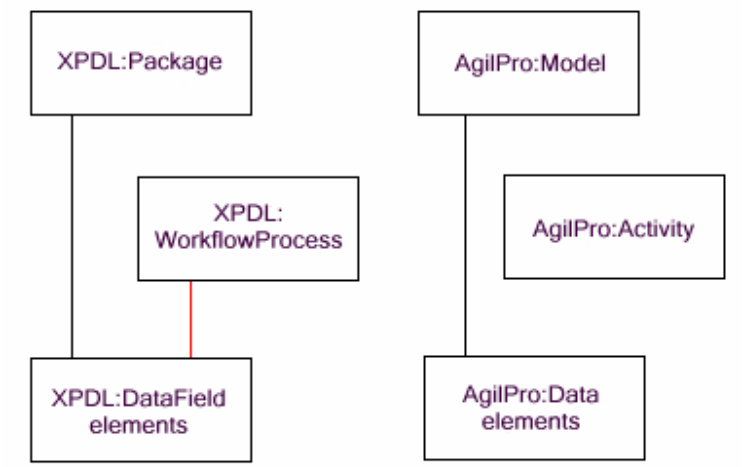
If AgilPro:Data is a collection of data items of the same type, this means that when we call an Application that requires several different types of arguments, we will have to have several AgilPro:Data elements clustered around the relevant AgilPro:Action. This is not an efficient use of scarce graphical real estate.

The AgilPro tool also has AgilPro:Integer and AgilPro:String elements located at the same level as the AgilPro:Data element. Presumably these are primitive type data items. I think it is somewhat confusing to have them here at a different level from the AgilPro:Data.Parameter date items, and handled specially.

I think the DataType property should be moved from the AgilPro:Data element to the AgilPro:Data.Property element. It is rightly a property of each individual data item anyway. This would make the AgilPro:Data element a true bundle of data items, which might then be of differing types. From a graphical design point of view this is a more useful entity than having to deal with a number of single-type bundles. This way, being able to add a Paramter defined in a different AgilPro:Data element makes perfect sense and is quite a useful feature.

We will need to expand the AgilPro data types to support all the functionality of the XPDL:DataType. I think we should handle primitive data types in a uniform manner within this system, rather than having special case primitive data items such as AgilPro:Integer and AgilPro:String

Also, XPDL:DataFields can be defined as children of XPDL:Packages (AgilPro:Model), in which case their scope is all XPDL:WorkflowProcesses (AgilPro:Activity) defined in the file. In addition, XPDL:DataFields can be defined as children of an XPDL:WorkflowProcesses (AgilPro:Activity), in which case their scope is only that one WorkflowProcess. AgilPro:Data elements can only be defined as children of an AgilPro:Model (XPDL:Package) element.



XPDL:Participants (AgilPro:Role) and XPDL:Applications (AgilPro:Applications) also have the same scoping issues as discussed above for XPDL:DataFields.

XPDL:DataType

There is an AgilPro:Data.DataType parameter. However, in AgilPro, this is a simple string. The XPDL:DataType has semantics that is capable of describing, in detail, the structure of rich and complicated compound data objects, as well as simple primitive data types. At present, there is no way of modeling this complexity in AgilPro.

We might want to consider being able to define an AgilPro data type independently of its use within the parent object, especially for complex data types. This way we could refer to the relevant definition in each data item of that type, instead of re-specifying the complex definition each time. This would also allow us to modify a complex definition in one place, which would greatly enhance maintainability. AgilPro could have the primitive data types already predefined, so the user could just select a String, Integer, Float, etc data type from a data type drop down list in the Eclipse properties view. The user would have to explicitly specify a data type only if he wished to use a more complex data structure, and then these additional data types would also appear in the drop down list.

If we choose to adopt this suggestion, I recommend this AgilPro data type element correspond to the XPDL:TypeDeclaration element.

Bonita

Bull's editing tool, ProEd, uses the XPDL files that it handles as its sole persistence mechanism. Consequently, it employs a number of Extended Attributes for persisting tools settings, such as node size and position information. AgilPro will use a separate persistence mechanism, and so will have no need to embed any tool specific information in the XPDL that it produces. The down side of this is that you will not be able to

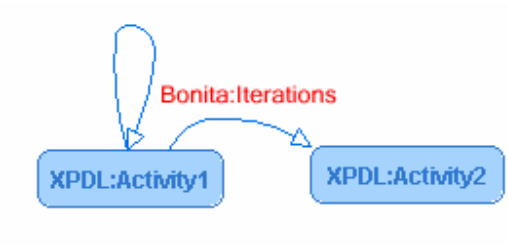
completely recreate an AgilPro project by importing an XPDL that has been produced by it. The ProEd tool specific Extended Attributes will not be covered in this document.

Bonita is capable of operating with a tool in a connected mode, which is, in fact, the preferred mode of operation. In the connected mode, the tool is able to open and store XPDL files directly in the server's XPDL repository. The tool is also able to communicate with the server and identify deployed resources, such as LDAP groups, Hooks, and Action Connectors. In fact, the Action Connector system requires that the tool be operating in the connected mode in order to produce the proper XPDL. AgilPro currently does not implement any of the facets of this concept.

Bonita:Iteration

In addition to XPDL:Transitions(AgilPro:ActivityEdges), Bonita supports Iterations. These are like transitions, but repeat until the associated condition is satisfied. Consequently, they must have an associated condition, and they can loop back to the same XPDL:Activity(AgilPro:Action) that they originate from. Their XPDL:ExtendedAttributes are located in the XPDL:Activity from which they originate.

Iterations could be modeled in AgilPro by adding a property to an AgilPro:ActivityEdge to select between transition and iteration. However, AgilPro:ActivityEdge does not support looping back to the same AgilPro:Action. It would also be useful if iterations were visually distinct from ordinary transitions.



Bonita:RollMapper

A Roll Mapper is a unit of work that allows rolls to be dynamically resolved at the time the workflow is **instantiated**. Roll Mappers are only applicable to participants of the Roll and OrganizationalUnit type.

The allowable RollMapper types are:

Participant Type	Allowable Roll Mapper Types
Role	Properties, LDAP, Custom
Organizational Unit	Custom

The Extended Attributes for a RollMapper appear in the XPDL:Participant to which they apply. There is currently no support for this in AgilPro

Bonita:PerformerAssignment

A Performer Assignment is a unit of work that allows the performer to be dynamically assigned at **execution time**. Bonita supports performer assignment based on a property or a called function. The ExtendedAttributes for PerformerAssignment appear in the XPDL:Activity to which the assignment applies. There is currently no support for this in AgilPro.

Bonita:ActivityProperties

XPDL allows the definition of XPDL:DataFields that have XPDL:Package scope and XPDL:WorkflowProcess scope. Bonita also supports defining a parameter at the XPDL:Activity level. This data item has the scope of that one activity, and can optionally have the scope extended to all activities that are "downstream" in the graph. That is, all activities that you can reach by starting at the indicated activity and following outgoing transitions.

It should be noted that Bonita only accesses the first XPDL:WorkflowProcess in the XPDL file, and so does not support XPDL:Package scope for XPDL:DataFields.

I have proposed AgilPro:Data.Parameter for the data item that corresponds to XPDL:DataField. AgilPro only supports creating AgilPro:Data items as children of AgilPro:Model, so it currently only supports the equivalent of XPDL:Package scope for data items, and does not support the XPDL:WorkflowProcess scope, or Bonita's XPDL:Activity scope.

The XPDL:DataField that defines an Activity Property is present in the XPDL:WorkflowProcess.DataFields list of the workflow process that contains the activity to which the property belongs. This data field will contain an empty ExtendedAttribute named "PropertyActivity" to identify the data field as an activity scoped data field rather than a workflow process scoped data field

The XPDL:Activity to which the property belongs will contain an ExtendedAttribute that references the data field described in the previous paragraph.

Bonita:DynamicEnumerations

Bonita only supports DataTypes of Basic.STRING and Basic.ENUMERATION. However Bonita splits enumeration into static enumeration and dynamic enumeration. Static enumeration is just the standard XPDL Basic.ENUMERATION data type. In a dynamic enumeration, the actual enumeration values are determined at execution time. Specific enumeration values can be specified in the XPDL, which will be shown in the tool as an aid for the designer, but which have no effect on the Bonita workflow engine.

Dynamic enumerations are identified by adding an empty Dynamic Extended Attribute to the affected DataField.

The complex data typing system of XPDL is not currently supported by AgilPro, which includes this extension.

Bonita:InitiatorRole

A Bonita Initiator Role is used to restrict the participants who may initiate a Workflow Process. The Extended Attribute for this appear in the XPDL:WorkflowProcess element to which they apply. AgilPro currently has no support for this concept.

Bonita:Subflow

The Bonita workflow engine can only access the first XPDL:WorkflowProcess in an XPDL file. Therefore, it cannot simply reference another process in the same XPDL, or one that has been included via an XPDL:ExternalPackage statement. Bonita also supports versioning, and must distinguish between different workflow processes whose identifying characteristics differ only by the version.

XPDL:(Subflow)Activities contain ExtendedAttributes to identify the WorkflowProcess that implements the subflow.

Subflow Activities, including this extension, are currently not supported by AgilPro.

Bonita:Hook

A Bonita:Hook is a method of calling a Java procedure on a server. It is considerably more versatile that an XPDL:Application call. Hooks are designed to facilitate interfacing substantially invested legacy transaction processing applications to a web-centric environment.

A hook is different than simply a java procedure that performs an XPDL:Activity. Hooks exist as a Java class residing on the server. Parameters are not passed to them via XPDL syntax, rather internal functions are available for them to access the workflow process data items themselves. They are written for a specific purpose and place in a workflow process, usually to package workflow process data items, use them to call a legacy routine, and post the return data back to the workflow process. This is why they are called hooks, they hook the Java web world into legacy applications. While this is the intended use of a hook, there is nothing to prevent it from containing the entire functionality to perform the XPDL:Activity in its own Java code, much as an Application.

The intended focus of hooks is to support robust transaction processing. Therefore, hooks may be invoked on the occurrence of a number of different events in the execution of a single XPDL:Activity. For Example:

- When the activity is ready for execution.
- Just before the activity terminates.
- Just after the activity terminates.
- Just before the activity is canceled.

More than one hook may be attached to an XPDL:Activity, one hook for each possible event.

The hook Extended Attributes appear in the XPDL:Activity to which they apply.

An individual hook could be modeled by an AgilPro:Application. It already has the Java class. A parameter would be needed for specifying the hook event. However, there is no support in AgilPro for attaching more than one AgilPro:Application to an AgilPro:Action.

Bonita:ActionConnectors

The action connector system allows an activity to execute a simple, easily constructed Java class on the Workflow server without having to create a full-blown hook class. This facilitates the use of existing classes that are already able to perform some desired function, as well as accessing web services. Basically, what happens is the tool generates a wrapper script that encapsulates the simple java class, and allows it to function as a hook. This script performs the argument and return mapping in the workflow engine environment and calls the desired routine. It is based on a template that the tool retrieves from the server. This generated script is embedded in the XPDL within the Hook Extended Attribute.

This system cannot call just any class however. It must be properly named, and located in the correct location in the server's directory structure. It must also correspond to certain rules regarding parameter passing. As long as it meets these requirements, an action class is simply a java class that performs some useful function in the context of the Workflow server. Existing, useful classes can be reused without their having to have any knowledge that are being invoked from within the workflow engine, whereas a regular hook class must be specifically written for the workflow engine environment. Writing an action class that accesses a web service is fairly easy using WSDL2Java to create the stub classes necessary to access the web service. The action class may also make more than one function available to be called.

In addition, there is a user configurable properties file, located on the server, associated with each action class, which provides additional information to the tool. This allows it to enhance the user interface by providing such things as: a list of available functions, meaningful parameter names, parameter descriptions, and pre-defined lists of parameter value options.

A default template is provided by the server that handles the simple case of mapping parameters that are available to the XPDL:Activity to the call, calling the selected function, and posting back return values. For more complicated operations, custom templates may be provided for each action class, or group of action classes. The template is used as a scaffold by the tool in order to construct the action script that is passed back to the server in the XPDL. It contains a number of template tags, which are similar in nature to compiler directives, which tells the tool how to use the text of the template and the action connector information the user has input to construct the script.

The extended attributes for action connectors appear in the XPDL:Activity that they apply to. It is, in fact, the same elements as a regular hook, except it also specifies the action script.

In the Bull tool, ProEd, Action Connectors are only usable when the tool is operating in the connected mode. When the user wants to add an action connector to an activity, the tool queries the server and presents a list of all the action classes that are deployed. The user selects an action, and the tool retrieves and displays a list of the available functions. The user selects the desired function, and the tool retrieves and displays a list of the parameters of the selected function. The user maps workflow parameters available to the activity to the function's parameters. When the tool generates the output XPDL, it retrieves the action connector's template, and uses it to construct the action script for the XPDL.

A Bonita Action Connector could be mapped by an AgilPro:Application. AgilPro currently has no means of retrieving and displaying the available actions, functions, or parameters. It is possible, although not desirable, that these items would have to be known to the user a-priori, and input manually. However, the script template is a page or more of java code, which must correctly match the action class. It would be impractical to input this manually. It will need to be retrieved from the server. This means that AgilPro will need to provide an interface for logging in to the server, as well as internal links for the XPDL engine to be able to access the server. This is the most problematic area that I have encountered. These kinds of complicated issues would be less difficultly addressed if the vendor specific plugins were adopted for certain sections of the tool, as previously discussed.

Detailed XPDL Coverage Analysis of the XPDL Schema

Green Highlight will identify areas that are currently supported by the metamodel.

Yellow Highlight will identify those areas that are not supported by the metamodel, but are optional in the XPDL schema. This does not necessarily mean that it is optional for us to support them.

Red Highlight will identify areas that are not currently supported by the metamodel and are required by the XPDL schema or Bonita extensions. We will have to provide support for these items in some manner.

Pink Highlight will identify areas that are not currently supported by the metamodel and are required by the XPDL schema or Bonita extensions, but could possibly be derived from elsewhere than the metamodel.

Turquoise Highlight will identify areas that are not currently supported by the metamodel and are required by the XPDL schema or Bonita extensions, but can be derived, provided we do not need to re-export an identical XPDL (This is mainly the Id)

Grey Highlight is used for areas where I feel the current metamodel needs significant resolution, assessment, or work. Therefore, it is premature to assess detailed coverage at this time. My evaluation of these areas, and some recommendations, are presented in the corresponding part of the preceding section.

Please note that these colorations are my opinions, and it may very well be that there is an alternate way to model XPDL items with existing AgilPro elements which I have not yet thought of, which will indeed support XPDL items that I have marked as not supported. I eagerly seek such critiques.

Note: The XPDL spec says an Expression may be used wherever an element is of type xsd:string. These are shown as "String or Expression" below. In many cases, for example Author, using an expression instead of a string does not make much sense, even though it is permitted. However elsewhere it is quite valuable. This allows an ActualParameter to be a DataField, a literal constant, or a complex expression.

One approach would be to treat Expressions as Strings in AgilPro. The optional XPDL:RedefinableHeader.Script element defines the scripting language to be used for such expressions, and their syntax is not further defined in the schema. I have taken the approach in the discussion below that whenever an AgilPro string parameter corresponds to such an XPDL element, I have marked it in green to indicate the element is supported by the AgilPro metamodel. Should we choose not to handle Expressions as Strings, we will have to modify all such parameters.

Package

XPDL:Package corresponds to AgilPro:Model

- **ID** – A unique NMTOKEN – Required
- **Name** – String – Optional – corresponds to AgilPro:Model.Name parameter
- **Package Header** – Required
    - **XPDLVersion** – String or Expression- Required
      XPDL Specification version (NOT the version of the workflow definition)
      This can likely be derived from settings in the tool. We will probably need to select the XPDL flavor to generate, as different Vendor implementations will support different things.
    - **Vendor** – String or Expression - Required
      Origin of process definitions (NOT the vendor of the tool).
      also supposed to contain product name and product release number,
      so this should be settable per AgilPro:Model, perhaps with a default value as a global option.
    - **Created** – String or Expression – Required
      Date the Package was created.

- **Description** –String or Expression - Optional – Modeled by attaching an AgilPro:Comment
- **Documentation** – String or Expression - Optional
  OS Specific path to help/documentation file
- **PriorityUnit** –String or Expression– Optional
  Semantics of this element are User Defined
- **CostUnit** – String or Expression– Optional
  Simulation Data Units (typically currency)

- **Redefinable Header** - Optional
  Defines values that are valid throughout the package BUT may be redefined for a specific WorkflowProcess by another RedefinableHeader element in the WorkflowProcess element
  - **PublicationStatus** – Enumeration- Optional
    - "UNDER_REVISION"
    - "RELEASED"
    - "UNDER_TEST"
  - **Author** – String or Expression – Optional – handled by AgilPro:Model.Author
  - **Version** - String or Expression – Optional – handled by AgilPro:Model.Version
  - **Codepage** – String or Expression – Optional – codepage of text parts
  - **Countrykey** – String or Expression – Optional – ISO 3166
    country code number (3 digits) or
    country code (2 alpha characters)
  - **Responsibles** - Optional – Sequence of Strings or Expressions – link(s) to workflow participant who is responsible for, or supervisor of, this workflow definition. Default is initiating participant
- **Conformance Class** – Enumeration - Optional
  Only used in Package element
  - "FULL_BLOCKED" – requires proper nesting of split/join and loops
  - "LOOP_BLOCKED" – requires proper nesting of loops
  - "NON_BLOCKED" – network structure not restricted (default)

  NOTE: This setting affects how the validation engine should validate the graph
- **Script** – Optional
  Only used in Package element
  - **Type** – String - Required
    Identifies expression scripting language. If it is a standard scripting language, the following designations are recommended, but not required:
    - text/javascript
    - text/vbscript
    - text/tcl
    - text/ecmascript
    - text/xml
  - **Version** – String – Optional
    version of scripting language

    - **Grammar** – anyURI – Optional
      reference to grammar specification document. ie: XML, schema, DTD, or BNF
- **External Packages** – Optional – A list of references to XPDL:Packages that are located in a different XPDL file, possibly on another system. AgilPro currently has no mechanism for referencing elements in domains other than the one being described in the current instance of AgilPro.
- **Type Declarations –** Optional – A list containing all the TypeDeclarations that are used in the Package.
- **Participants –** a sequence of XPDL:Participant elements with Package scope. XPDL:Participant corresponds to AgilPro:Role.
- **Applications –** a sequence of XPDL:Application elements with Package scope. XPDL:Application corresponds to AgilPro:Application.
- **DataFields** – a sequence of XPDL:DataField elements with Package scope XPDL:DataField corresponds to AgilPro:Data.Parameter?
- **Workflow Processes** – a sequence of all the XPDL:WorkflowProcess elements in the package. XPDL:WorkflowProcess elements correspond to the AgilPro:Activitys that are directly under the AgilPro:Model
- **Extended Attributes**
  Vendor defined extensions

WorkflowProcess

An XPDL:WorkflowProcess corresponds to an AgilPro:Activity.

- **ID** – A unique NMTOKEN - Required
- **Name** – String – Optional - handled by AgilPro:Activity.Name
- **AccessLevel** – Enumeration – Optional
    - "PUBLIC"
    - "PRIVATE"
- **ProcessHeader** - Required
    - **DurationUnit** – Enumeration – Optional
        - "Y"
        - "M"
        - "D"
        - "h"
        - "m"
        - "s"

      Default duration unit for integer duration value that does not specify units
    - **Created** – String or Expression – Optional
      date the workflow process definition was created
    - **Description** – String or Expression– Optional – corresponds to attaching an AgilPro:Comment
    - **Priority** – String or Expression – Optional
      initial priority if the process. Assumed to be a non-negative integer, with higher numbers indicating higher priorities

    - **Limit** – String or Expression – Optional
      expected duration in duration units. What happens when the limit is reached is vendor specific. This is handled by AgilPro:Activity.TotalTime, however this is in seconds, and the XPDL:Duration's units are specified in the XPDL:ProcessHeader.DurationUnits property. AgilPro:TotalTime could also be corresponded to TimeEstimation.Duration below, but in XPDL:Activity I am corresponding it to Limit, so I am doing the same here for consistency.
    - **ValidFrom** – String or Expression– Optional – a date when the workflow process is valid – empty string means system date
    - **ValidTo** – String or Expression– Optional – a date at which the workflow process becomes invalid – empty string means valid forever
    - **TimeEstimation** – Optional
        - **WaitingTime** – String or Expression- Optional **–** time required to prepare for performance of the task in Duration Units
        - **WorkingTime** – String or Expression– Optional – time required by performer to perform the task
        - **Duration** – String or Expression– Optional – expected duration of the task in duration units
- **Redefinable Header** - Optional
  Defines values that are valid in this Workflow Process element. This may be a redefinition of the value from the XPDL:Package.RedefinableHeader. If a value is not specified here, it is inherited from the XPDL:Package.RedefinableHeader.
    - **PublicationStatus** – Enumeration- Optional
        - "UNDER_REVISION"
        - "RELEASED"
        - "UNDER_TEST"
    - **Author** – String or Expresion – Optional
    - **Version** - String or Expression- Optional
    - **Codepage** – String or Expression- Optional– codepage of text parts
    - **Countrykey** – String or Expression- Optional – ISO 3166
      country code number (3 digits) or
      country code (2 alpha characters)
    - **Responsibles** - Optional – Sequence of Strings or Expressions – link(s) to workflow participant who is responsible for, or supervisor of, this workflow definition. Default is initiating participant
- **FormalParamters –** Optional – Sequence of XPDL:FormalParemeters
  These are the input/output parameters of the workflow process. If not the WorkflowProcess is not used in a subflow, what this means is implementation specific.
- **DataFields –** Optional – Sequence of DataFields with WorkflowProcess scope. In addition, DataFields that were defined in the XPDL:Package are also available.
- **Participants –** Optional – Sequence of XPDL:Participant elements with Workflow Process scope. XPDL:Participant corresponds to AgilPro:Role
- **Applications –** Optional - a sequence of XPDL:Application elements with Workflow Process scope. XPDL:Application corresponds to AgilPro:Application.

- **ActivitySets** – Optional - Sequence of XPDL:ActivitySet elements. There is one of these elements for every XPDL:BlockActivity (AgilPro:ActivityLinkNode?) anywhere in this XPDL:WorkflowProcess, XPDL:ActivitySet elements contain lists of the elements in the corresponding XPDL:BlockActivity, and an Id to link them to the corresponding Block. These elements are XPDL:Activities (AgilPro:Actions) and XPDL:Transitions (AgilPro:ActivityEdges) Note: if we are going to derive a new Id every time we export an XPDL, we do not have to model an Activity Set. If we want to be able to import and export an identical XPDL, we will have to store the ID somewhere in the model.
- **Activities** – Optional - Sequence of XPDL:Activity elements that are direct children of this XPDL:Workflow Process. An XPDL:Activity corresponds to AgilPro:Action
- **Transitions** – Optional - Sequence of XPDL:Transition elements that are direct children of this XPDL:WorkflowProcess. XPDL:Transition corresponds to AgilPro:ActivityEdge.
- **ExtendedAttributes**
  - **Bonita:InitiatorRoll** - Optional
    - **Value –** roll type – Enumeration - Required
      - **"Custom"**
      - **"LDAP"**
    - **InitiatorName –** String - Required
      the name of the hook class or LDAP group.

Activity

XPDL:Activity is modeled by AgilPro depending on the type of Activity, see below. The following items apply to all XPDL:Activitues

- **Id** – xsd:NMTOKEN – Required
- **Name** String – Optional - handled by AgilPro:Activity.Name
- **Description** – String or Expression – Optional – corresponds to attaching an AgilPro:Comment
- **Limit** – String or Expression – Optional – expected duration in duration units. What happens when the limit is reached is vendor specific. Corresponds to AgilPro:Activity.TotalTime, however AgilPro:TotalTime is in seconds and XPDL:Limit is in XPDL:DurationUnits
- **Performer** – String or Expression– Optional – handled by association with AgilPro:Role, however this does not take into account allowing an expression so that the performer may be dynamically selected.
  Link to XPDL:Performer
- **StartMode** – Choice – Optional
  Depending on the XPDL Engine, it may be possible to infer the start mode from the performer
  - **Automatic** – empty element
    fully controlled by the workflow engine
  - **Manual** – empty element
    requires user interaction

- **FinishMode** – choice – Optional
  - **Automatic** – empty element
    fully controlled by the workflow engine
  - **Manual** – empty element
    requires user interaction
- **Priority** – String or Expression – Optional
  initial priority if the activity. Assumed to be a non-negative integer, with higher numbers indicating higher priorities.
- **Deadlines** – a zero or more of:
  - **Deadline –** Optional
    - **DeadlineCondition** – Required – xsd:anyType – an implementation dependent expression specifying the deadline time
    - **ExceptionName** – Required – xsd:anyType – Name of the exception that is raised.
    - **Execution** – Enumeration – Optional
      - "ASYNCHR" – The exception thread is initiated in parallel when the deadline is reached, and this activity proceeds until normal termination
      - "SYNCHR" – this activity terminates abnormally, and control passes to the exception thread
- **SimulationInformation** - Optional
  - **Cost –** String or Expression - Optional
  - **Instantiation** – Enumeration – Optional
    - "ONCE" – activity con be instantiated only once - Default
    - "MULTIPLE" – activity can be instantiated multiple times
  - **TimeEstimation** – Optional
    - **WaitingTime** – String or Expression- Optional **–** time required to prepare for performance of the task in Duration Units
    - **WorkingTime** – String or Expression– Optional – time required by performer to perform the task
    - **Duration** – String or Expression– Optional – expected duration of the task in duration units
- **Icon** – Optional – String or Expression - handled by AgilPro:Icon
  path and filename of Icon to represent the Activity
- **Documentation** – String or Expression – Optional
  path and filename to description of activity
- **TransitionRestrictions** – Optional – Sequence of any number (including 0) of:
  - **TransitionRestriction**
    - **Split** – corresponds to AgilPro:Decision/ForkNode
      - **Type** – Optional – Enumeration
        - **"AND"** – corresponds to AgilPro:ForkNode
        - **"XOR"** – corresponds to AgilPro:DecisionNode
      - **TransactionRefs** – Optional – Sequence of references to XPDL:Transction.Ids. If type is XOR, transaction conditions are evaluated in the order the transitions appear in this list until a condition is satisfied

NOTE: this concept of ordering of the outbound transitions is not currently supported in AgilPro

- **Join** – corresponds to AgilPro:Merge/JoinNode
  - **Type** – Optional – Enumeration
    - **"AND"** – corresonds to AgilPro:JoinNode
    - **"XOR"** - corresponds to AgilPro:MergeNode

XPDL:TransitionRestrictions will be handled by AgilPro:JoinNodes, AgilPro:ForkNodes, AgilPro:MergeNodes, and/or AgilPro:Decision nodes associated with the AgilPro:Action. See the discussion in the General Mapping section

ExtendedAttributes

- **Bonita:Iteration ExtendedAttribute**- Optional
  The Bonita:Iteration extended attribute appears on the XPDL:Activity that is the origin of the iteration.
  - **Value** – Required – String – the condition expression
  - **To** – Required – String – the NAME of the XPDL:Activity on which the iteration terminates.
- **Bonita:PerformerAssign ExtendedAttribute –** Optional – performer assignment
  - **Value –** Required – Enumeration
    - **"property"** – performer resolution based on property
    - **"callback"** - performer resolution based on procedure call
  - **Property –** Required iff property – String – NAME of property used to resolve the participant
  - **Callback –** Required iff callback – String – name of hook class that performs the performer resolution
- **Bonita:Property ExtendedAttribute** – Optional **–** one instance for each Activity Property that belongs to this activity
  - **Value** – Required – the Id of the XPDL:DataField that defines this property
  - **Propagated** – Optional – Enumeration – default is "No" indicates whether the property should be propagated to downstream activities.
    - "Yes"
    - "No"
- **Bonita:Hook ExtendedAttribute**
  used for simple Hooks and Action Connectors
  - **Value** – Required – String
    the hook class name
  - **EventName –** Required – Enumeration
    - "afterStart"
    - "beforeStart"
    - "afterTerminate"
    - "beforeTerminate"
    - "onCancel"
    - "onDeadline"

- "onTerminate"
- "onInstantiate"

- **HookScript –** Optional – String
  Used if this is an Action Connector

(Basic)Activities

XPDL:(Basic)Activities are modeled by an AgilPro:Action and associated AgilPro:Merge/Join/Decision/ForkNodes, without an associated AgilPro:Application. They also do not have associated AgilPro:Data elements

In addition to the items listed in the main Activity entry, BasicActivities also have:

- **Implementation –** Required
  - **No –** Required – empty element

This type of XPDL:Activity corresponds to an AgilPro:Action element that does not have an attached application and is not a Subflow.

(Route)Activities

XPDL:Route activities correspond to combinations of AgilPro:ForkNodes, AgilPro:JoinNodes, AgilPro:DecisionNodes, and/or AgilPro:MergeNodes, that are not associated with an AgilPro:Action.

In addition to the items listed in the main Activity entry, Route Activities also have :

- **Route –** Required – empty element

The presence of the Route element identifies this XPDL:Activity as a Route type activity.

(Block)Activities

XPDL:(Block)Activities are modeled by an AgilPro:ActivityLinkNode, which has yet to be defined, and associated AgilPro:Merge/Join/Decision/ForkNodes. They do not have associated AgilPro:Application or AgilPro:Data elements

In addition to the items listed in the main Activity entry, Block Activities also have :

- **BlockActivity** – Required – XPDL:BlockActivity
  - **BlockId –** Required – xsd:NMTOKEN
    This is the ID of the XPDL:Package's XPDL:ActivitySet that contains the elements inside the BlockActivity

Block activities are an activity node that represents a subgraph of other nodes that are contained in the same XPDL:WorkflowProcess

(Tool)Activities

Tool Activities are modeled by an AgilPro:Action with assocatied AgilPro:Application and AgilPro:Data elements, along with the standard associated AgilPro:Merge/Join/Decision/ForkNodes

In addition to the items listed in the main Activity entry, Tool Activities also have:

- **Tools** – one or more of:
  - **Tool** – Required – XPDL:Tool element
    - **Id** – Required – xsd:NMTOKEN
      identifies the application or procedure that is to be used. NOTE: this is different than most of the other Ids, which identify the element that they are in. This probably should have been IdRef. This value will be derived from the Id that gets assigned to the attached AgilPro:Application.
    - **Type** – Optional – Enumeration
      - **"APPLICATION"**
      - **"PROCEDURE"** – I do not understand how this differs from application, and I am not sure if it needs to be modeled differently from associating an AgilPro:Application
    - **ActualParameters** – Optional – Sequence of Strings or Expressions
      List of the invocation parameters that are passed to/from the application or procedure. This may be modeled by associating appropriate AgilPro:Data elements to the AgilPro:Action and using AgilPro:Mapping element children of the AgilPro:Action to specify the correspondence between the actual parameters and the formal parameters of the application
    - **Description** – Optional – String or Expression
      There currently is no place to attach an AgilPro:Comment that would end up here. Attaching to an AgilPro:Application would end up on the XPDL:Application
    - **ExtendedAttributes**

(Subflow)Activities

Subflow activities are not currently modeled in AgilPro.

In addition to the items listed in the main Activity entry, Subflow Activities also have :

- **ActualParameters** – Sequence of Strings or Expressions – Required
  list of the invocation parameters that will be passed to the subflow. Depending on how the subflow is modeled, this might be defined the same way it is in an XPDL:(Tool)Activity.
- **Id** – String – Required
  The identification of the Workflow Process to be performed. NOTE: this is different than most of the other Ids, which identify the element that they are

in. This probably should have been IdRef. This can reference another WorkflowProcess in the XPDL:Package in this XPDL file, or in any of the XPDL:Packages located in the external files referenced in the XPDL:ExternalPackage elements of this XPDL:Package.

- **Execution** – Enumeration – Optional
  - **"ASYNCHR" –** Output transitions are activated according to the embedded splitting semantics as soon as the subflow process is started
  - **"SYNCHR" –** Output transitions are activated according to the embedded splitting semantics when the subflow process completes
- **Bonita – Subflow ExtendedAttributes**
  The Bonita Engine also uses the following Extended Attributes in a Subflow Activity:
  - **SubflowVersion** – String – Required
    Since Bonita supports versioning, this identifies the proper version of the Subflow to use
  - **SubflowName** – String - Required

Transition

XPDL:Transition corresponds to AgilPro:ActivityEdge, and the optional XPDL:Condition corresponds to an AgilPro:Guard, with added condition parameters.

- **Id** – Required – xsd:NMTOKEN
- **Name** – String – Optional – handled by AgilPro:Transition.Name
- **From –** Required – xsd:NMTOKEN
  Id of XPDL:Activity where the transition originates
- **To–** Required – xsd:NMTOKEN
  Id of XPDL:Activity where the transition terminates
- **Condition –** Optional
  Corresponds to adding AgilPro:Guard to AgilPro:ActivityEdge
  AgilPro:Guard will have to have condition parameters added
  - **Type –** Enumeration - Optional
    - "CONDITION" – transition taken if the condition is true
    - "OTHERWISE" – transition taken if none of the CONDITION conditions on other transitions are true
    - "EXCEPTION" – transition taken when an exception is raised if the condition is true
    - "DEFAULTEXCEPTION" - transition taken when an exception is raised if none of the EXCEPTION conditions on other transitions are true
  - **The Condition Expression –** 0 to unlimited number of XPDL:Xpression (xsd:any) elements. Basically, the contents of an XPDL:Condition element can be anything at all.
- **Description** – String or Expression– Optional – handled by attaching an AgilPro:Comment to the transition
- **ExtendedAttributes**

Participant

XSD:Participant corresponds to AgilPro:Role

- **Id** – Required – xsd:NMTOKEN
- **Name** – Optional – String – handled by AgilPro:Role.Name
- **ParticipantType** – Required – Enumeration
  - "RESOURCE_SET" – collection of resources
  - "RESOURCE" – specific resource
  - "ROLE" – function within an organization or a skill set
  - "ORGANIZATIONAL_UNIT" – department or other such unit
  - "HUMAN" – human interacting via a user interface
  - "SYSTEM" – automatic agent
- **Description** – Optional – String – handled by attaching an AgilPro:Comment
- **ExternalReference** – Optional – reference to an external definition of the participant
  - **xref** – Optional – xsd:NMTOKEN – element within the defining document that is to be used
  - **location** – Required – xsd:anyURI – URI of defining document
  - **namespace** – Optional – xsd:anyURI – specifies scope of xref?
- **ExtendedAttributes**
  - **Bonita:Mapper ExtendedAttribute** – Optional – implements RollMapper
    - **Value** – Required – Enumeration
      - "Ldap"
      - "Properties"
      - "Custom"
  - **Bonita:MapperClassName ExtendedAttribute** – Required if RollMapper is Custom
    - **Value** – Required – String – name of hook class that performs the roll resolution.

DataField

See the discussion in the XPDL:DataField part of the General Model Mapping Discussion section

- **Id** – Required - xsd:NMTOKEN
- **Name** – Optional – String –
- **IsArray** – Optional – Enumeration
  - "TRUE"
  - "FALSE"

  Indicates whether it is an array. I do not know how this differs from having an Array data type. Perhaps this is why the Array data type is deprecated.
- **DataType** – XPDL:DataType – Required

  Defines the type of this data item

- **InitialValue** – Optional – String or Expression
  The XPDL spec does not address how this is applied to complicated compound data types**.**
- **Length** – Optional – String or Expression
  Length of the data, I think of the array, if it is an array.
- **Description** – Optional – String or Expression
- **ExtendedAttributes**
  - **Bonita:PropertyActivity ExtendedAttribute –** Optional
    an empty element that identifies this DataField as an Activity Property
  - **Bonita:Dynamic ExstendedAttribute –** Optional - only applicable if the DataType is Basic.ENUMERATED, and indicates this is a Bonita Dynamic Enumeration.

DataType

See the discussion in the XPDL:DataType part of the General Model Mapping Discussion section
Note that since DataTypes can nest inside other DataTypes, and due to the inclusion of the SchemaType, a DataType element can be arbitrarily complex.
I believe the deprecated DataTypes are intended to be replaced with the SchemaType, with the structures they represented being described in XML Schema syntax instead.
Where I refer to members, this refers to an element that contains only a DataType specification.

- **Choice of one of the following:** - Required
  - **BasicType –** Enumeration
    the internal formats of these types are not defined by XPDL
    - "STRING"
      sequence of characters
    - "FLOAT"
      Maximum size not specified in XPDL
    - "INTEGER"
      optional sign plus digits. Maximum length not specified in XPDL
    - "REFERENCE" – deprecated in favor of ExternalReference
    - "DATETIME" – format not defined in XPDL
    - "BOOLEAN"
      Value may be "TRUE" or "FALSE"
    - "PERFORMER"
      has a value of a declared participant
  - **DeclaredType –**
    - **Id** – Required – xsd:IDREF - Reference to an XPDL:TypeDeclaration in the TypeDeclarations list of the XPDL:Package. This allows for defining a complex type once and reusing the definition in multiple places.
  - **SchemaType** –
    - a namespace qualified sequence of XML Schema elements that define the data

- **ExternalReference –** reference to an external definition of the DataType
  - **xref** – Optional – xsd:NMTOKEN – element within the defining document that is to be used
  - **location** – Required – xsd:anyURI – URI of defining document
  - **namespace** – Optional – xsd:anyURI – specifies scope of xref?
- **RecordType** – deprecated -
  An ordered fixed length sequence of members of the same or different data types that define a record
- **UnionType** – deprecated -
  a set of members of alternative data types, only one of which will be used for an actual instance of the data
- **EnumerationType** – deprecated – a sequence of xsd:NMTOKENS, which are the legal values
- **ArrayType** – deprecated -
  a fixed length set of data items of the same type
  - **DataType** – the data type of the Array - Required
  - **LowerIndex –** Required – xsd:NMTOKEN – lower bound of the array
  - **UpperIndex** – Required – xsd:NMTOKEN – upper bound of the array

  ListType – deprecated –
  a variable length set of data items of the same type
  - **DataType –** the data type of the list – Required

TypeDeclaration

TypeDeclaration lets a complex DataType be defined in one place and simply referenced wherever it is needed, instead of being re-specified.

This element might be derived by the XPDL export engine as an optimization, if it detects the same non-trivial data type being specified for more than one data item.
If we choose to allow an AgilPro:DataType to be defined independently of the parent object, this XPDL:TypeDeclaration would correspond to that definition. The coverage highlighting given here refers to this later usage.

- **Id** – Required – xsd:ID
- **Name** – Optional – String
- **DataType** – XPDL:DataType - Required
- **Description –** String – Optional – handled by attaching an AgilPro:Comment
- **ExtendedAttributes**

FormalParameter

APDL:FormalParameters will probably be modeled in AgilPro:Applications by the AgilPro:Application.Input/OutputParameters lists. It has not been determined how they will be modeled in a subflow

- **Id** – Required – xsd:NMTOKEN

- **Index** – Optional – xsd:NMTOKEN
- **Mode** Optional – Enumeration – default is IN
  - "IN"
  - "OUT"
  - "INOUT"
- **DataType** – Required – XPDL:DataType
- **Description** – Optional – String or Expression

Application

XPDL:Application coresponds to AgilPro:Application. Tools may be defined or just named. The real definition is not necessary and may be handled by an object manager, in order to support handling of the application in multi-platform environments.

- **Id** – Required – xsd:NMTOKEN
- **Name** – Optional – String – handled by AgilPro:Application.name
- **Description** – Optional – String or Expression – handled by attaching AgilPro:Comment
- **Choice of:** - Required
  - **FormalParameters** – Sequence of XPDL:FormalParameters, which describe the calling signature of the function call used to invoke this application. Probably modeled by AgilPro:Application.InputParameter and AgilPro:OutputParameter lists, although AgilPro does not currently support formal parameters of an application that are of "INOUT" type.
  - **ExternalReference** – Optional – reference to an external definition of the application
    - **xref** – Optional – xsd:NMTOKEN – element within the defining document that is to be used
    - **location** – Required – xsd:anyURI – URI of defining document
    - **namespace** – Optional – xsd:anyURI – specifies scope of xref?
- **ExtendedAttributes**

ExternalPackage

Not currently modeled in AgilPro.

- **href** – Optional – String
  Reference to another Process Model, possibly on another system
- **ExtendedAttribute**