4.1 The Systematic Reuse of Software Assets

The traditional approach to software development originated with the implicit notion that every product would be unique with fixed requirements. In reality, most capabilities recur across many products and must be made to conform to differing and changing requirements for those products. This has led to the establishment of libraries of commonly needed capabilities in the form of previously developed software parts that may be beneficially used in building new products. Using such parts avoids the need to build equivalent parts from scratch as well as possibly needing specialized competence to do so. Furthermore, multiple uses of a part can induce investment in its improvement, including discovery and correction of residual flaws and inefficiencies, increasing the value of the part for both current and future uses.

[Most discussions of reuse focus narrowly on the reuse of existing software module implementations {see section 2.7}. However, every element of product model content offers potential for reuse in building similar products. Examples include content of environment, requirements, and design specifications, user training and documentation, and testing materials including scenarios and associated data, as well as hardware parts applied similarly in fabrication of physical devices.]

In building a new product, developers will reasonably consider starting with parts of a previously-built product. Such opportunistic reuse of past results can save effort if the parts to be reused were developed based on assumptions and tradeoffs that are appropriate for the new product. Existing parts can be used as-is or with modifications to fit the differing but more-or-less similar needs of a new product. When reusing a previously built part, the developer must ensure that the part—as-is or with necessary changes—is consistent with other elements of the product model and complete in accordance with project guidance.

As-Is Reuse of Existing Parts

Modern software-based products convey many complex capabilities that require substantial time and competence to build correctly. Furthermore, developers having the appropriate competence to build such capabilities are seldom readily available to every

product development effort that needs them. Conversely, many capabilities are routine to create even by less experienced developers and have been repeatedly redeveloped, often with only minor or incidental differences. As an alternative, the effort required to build software can be reduced when development practices support the effective use of previously built parts to provide many, either routine or complex, capabilities.

Effective reuse of existing parts can be opportunistic or planned. For opportunistic reuse, individual developers identify previously built parts that can be used in creating assigned product model content. For planned reuse, program management directs that elements of the product model be developed as reusable parts anticipating capabilities needed in future products and that those or that other previously developed parts be used in realizing current product elements. Parts developed for planned reuse are evaluated in light of both current and expected future uses.

For an existing part to be reusable without change, it must have been built under assumptions that are consistent with its intended new use, specifically conforming to all relevant product model specifications. Alternatively, aspects of the product model may need to be modified to allow as-is reuse of a given part. Such changes may be justified if the alterations do not diminish important capabilities or quality aspects of the product. (This reverses the traditional expectation that specifications dictate realizations; in this case, a realization may instead impose a constraint on its specification.) If relevant specifications cannot be changed, the existing part would have to be modified to be reused, possibly reducing the benefit of its reuse. In any case, any inconsistencies in the product model must be resolved for an existing part to be reused.

Using a Library of Previously Built Parts

Previously built parts may be held either in a program-maintained library of parts to be used in building its products or in a sanctioned but separately maintained library of parts developed elsewhere for such use. Any type of part can be retained in a library to be targeted for reuse.

A library-based approach to reuse has five steps:

1. Formulate selection criteria for a part that would provide content needed for a product model element
2. Based on the organization or search capabilities of a selected reuse library, identify candidate parts that satisfy defined criteria
3. Evaluate the fit of each identified candidate to the specified need to determine which candidates are an acceptable match, with or without changes, to the need
4. Select a suitable candidate part (modifying it for a better fit to current needs or building a new part if necessary) and integrate that part into the targeted product model element
5. For any new or modified part, optionally add it back to the library with assumptions, rationale, and guidance for it be reliably reused as-is or modified for other future uses

{address the standardization-customization perspective from 3.0?}

A library may offer multiple similar parts from different sources and any given part may exist in multiple versions, often as a modification of a prior version for an improved fit to differing needs. A best part for reuse is one that provides all needed capabilities without introducing extraneous content (which may require effort to properly remove). Choosing among a number of more or less similar parts or versions can be difficult and time-consuming unless the specific differences among them and the rationale for each are fully understood. Any part added to a library should have associated information regarding the assumptions and rationale that influenced its content, expressing how it differs from other similar parts. Still, additional work may be necessary to complete other content of the product model element in which a part is being wrapped.

A well-organized library will categorize parts according to the degree to which they offer similar capabilities. This helps reduce inadvertent duplication of equivalent parts and the effort needed to find a good fit to new needs. A library management goal would be to categorize and differentiate similar parts (addressed below regarding building parts for multiple and changing use). A program-maintained reuse library might well be

organized to mirror a common architectural structure for its products into which parts are intended to fit.

Modifying a Previously Built Part

Reuse works best if an existing asset is a close enough fit for a new use and can be used as-is, without change. Typically, however, an existing part will be only an approximate fit for a different purpose. Modifying a part that is a reasonably close fit to what is needed, to make it a satisfactory fit for reuse, may still reduce the initial effort versus building a new part from scratch. The challenge with reuse is how the relevance and fit of existing content to a purpose is to be determined and how that content can reasonably be modified to resolve any misfit to that purpose.

Different developers may solve a given problem differently but an experienced developer tends to produce similar solutions to similar problems. As with building a new part, modifying an existing part requires appropriate competence in relevant subject matter. In addition, it requires an understanding of the assumptions upon which the part was built and how it can be safely modified for a new use. A part's prior developer, in understanding the differences between the previous and current problem and the details of the prior solution, would be able to create a new solution as a modified copy of their previous solution, rather than being inclined to build a new solution from scratch.

For other than an asset's prior developer, a part having the potential for reuse can be identified due to inclusion in a repository having that purpose or by having been used in a product having similar needed content. This determination of reusability can be achieved conventionally, by categorization, search, or direct inspection of existing assets, or using a generative capability [discussed in 5.4], to analyze those assets for fit and any needed transformation. Implicitly, the derivation of a modified part creates a new candidate for future reuse.

Potential Issues in Reusing Existing Parts

Reusing existing parts, with or without modification, may impose constraints on a product's content (e.g., behavior and appearance). This may result in the customer

having to modify their expectations and business practices (possibly for the better) to suit the product rather than building the product to fit their exact needs. However, by being able to obtain the product faster or at lower cost, the customer may consider this to be an acceptable tradeoff.

Due care is required to mitigate the issues that can arise in reusing previously built parts, whether as-is or with changes:

- Typically, existing parts have been built narrowly tailored to fit specific needs. Such a part may not be easily changed to fit different needs. Building new content may be less effort and better quality than trying to modify an existing part. A proper fit to different needs may require adding, modifying, or even removing content. The effort to understand and change a part to fit different needs can reduce the benefits of reuse over developing a new part.
- The fit of an existing part and the effort to modify it as needed may not be easy to determine. Even if a part is free of defects based on past uses, a different use may violate the (possibly implicit) assumptions on which the part was built. Properly modifying a part—whether for reuse or just to meet changed needs—requires being aware of and adjusting for assumptions that the part's prior developer may have made concerning its content and expected use. Poorly considered changes may expose flaws in the unmodified portion of the part that show up only when it is subsequently used.
- Given new needs, there may be multiple existing parts that addressed similar needs. It can be difficult and time-consuming for a developer to determine which of those parts are a good fit for the new needs without an understanding of what particular needs determined each part and how those needs differed from the new needs. Even if prior and current needs are actually similar, different developers may have solved them differently or used terminology or organization that disguise the similarity among the resulting parts.
- Different existing parts that address similar needs may differ in breath or depth of content/capabilities, quality factors, assumptions, rationale, and tradeoffs that

must be considered in selecting a part for reuse, particularly if these aspects are not well documented. For example, a part built for use in one context may be understood differently if used as-is in another context, requiring changes just to keep the same meaning. A given part may be easier to modify for a different use than another similar part due to such differences.

- If a part is modified to fit different needs, the modified part may be viewed as a new candidate for future reuse. Even if the new part closely resembles the existing part, any added, changed, or removed capability may make it a poor fit for use by current reusers of the existing part—they would each need to consider whether the changed part is a better or worse fit to their current needs. Repeated derivation of modified parts can result in a proliferation of increasingly dissimilar parts, requiring both more effort to select among them for reuse and duplication of effort to separately maintain them over time.
- Responsibility for maintenance of previously developed parts can be problematic unless there is clear "ownership" of each part that sanctions changes. If successor parts are separately maintained, they will not directly benefit from changes to the original part (e.g., to correct defects or improve behavior). Separately managed parts are likely to undergo uncoordinated changes that can cause them to diverge arbitrarily, in both essential and incidental ways, from other similar parts.
- A part to be reused may require use of other particular parts. For example, if a part is an element of a "framework" (a set of parts built to be used together), dependencies among them may make it impractical to use some of them without the others. However, using an entire parts ensemble could mean that some of its content may duplicate, or possibly conflict with, content of other product model elements. Reusing an existing part may also, for consistency, require modifying other related elements of the product model (e.g., developer documentation and verification materials).

Building Parts for Multiple and Changing Use

A conventional approach to reuse assumes that a part built for one purpose will either be useful for other similar purposes without change or can be changed for a different purpose with less effort than building its content from scratch. However, reuse as traditionally practiced—opportunistically after a part has been built for a given purpose—is not well suited to using an existing part to meet different needs. It can be difficult to determine both how the part needs to be changed to suit a different use and whether such changes will conflict with assumptions on which the part was built.

In one respect, a systematic approach for building reusable parts will not differ from what competent developers have always done—explore alternatives for building a part, to reduce risk and cost of having to change an inferior initial solution. For a part to be properly reusable, information and insights concerning how differing needs would lead to differences in part content are considered and retained rather than being discarded. In building a potentially reusable part, a developer will build the part in a way that makes likely changes easier to make when different needs arise.

Building a Reusable Part

Any previously built part can be reused, with or without change, for a new purpose but a "reusable" part is a part that has been built to be systematically reused—to be easily changed in specific ways to suit needs that differ in particular ways. A reusable part is not one-of-a-kind but in its formulation represents an envisioned set of similar/changed parts, all differing but alike in providing similar solutions to similar problems. Properly building a part for reuse means building it in a way that anticipates why and how it may need to be changed, resulting in the ability to more easily obtain an appropriately changed part. Nevertheless, with any approach, unforeseen changes can incur unpredictable cost. Without foresight on likely changes when building a part, all changes are implicitly assumed to be equally (un)likely so that the difficulty of making any particular change later is unpredictable.

Building a part to be reused in meeting the potential needs for changed or multiple, possibly not yet known, purposes requires a broader view of the content that the part will need to provide initially, over time, and for possible different uses. With such

broader scope, there are techniques for building a part that can be systematically changed to support other than its initial use. Each technique, to different degrees, reduces the difficulty and effort required to safely change a part to fit a different use or even to derive multiple customized versions of a part, with only limited additional effort over building a single-use part.

In any case, a part should be fully documented (e.g., as annotations in its content) as to assumptions that influenced its form and content and what changes can be safely made to the part without conflicting with those assumptions. If the developer has considered potential assumption-breaking changes, the implications of those on part content should be documented as well. Based on this information, future developers are able to determine whether and how to change part content so that it will be suitable for both current and new uses or if a new part is needed.

An alternative for software source code is to apply adaptation mechanisms, from simple preprocessing notations for changeable content with simple conditional text inclusion and substitution to full polymorphic (parametric or subtype) constructs that produce different behavior depending on the types of data being processed. These mechanisms provide a means for a part's developer to predetermine fragments that differ according to and consistent with the part's intended usage.

[object-oriented languages provide for classification (classes) and specialization (subclasses) of parts, in which tailoring can take the form of a choice among alternative subclasses determining differences in a product's behavior.]

A variation on this approach is to use integration- or installation-time mechanisms for selecting (e.g., from a source or runtime library) among alternative versions of a part (such as for differing quality criteria or computational platform).

Another variation on this approach is to build a part to be self-adaptive (i.e., behaving differently based on user direction or operational circumstances). With this variant, a single product can effectively emulate multiple different products (for operational flexibility at the cost of increased software complexity and increased processing and storage costs over multiple customized solutions). {This can be confused with normal

computational logic in which software behaves differently according to varying operational conditions or changes in operational mode (e.g., normal versus degraded capabilities).}

An advanced approach for reusable parts is the concept of *metaprogramming*—the development of software that generates other software. With this technique, individual parts are only implicitly expressed, with a means for deriving customized instances as needed. A metaprogramming representation can take a "prescriptive" or a "descriptive" form[1].

Early research in "automatic programming" took a prescriptive approach—writing software whose purpose was to generate other software based on a stylized specification of its intended functionality. This approach is a particularly effective technique for generating an initial result that can then be incrementally transformed into logically equivalent alternative realizations (e.g., to improve tradeoffs among quality attributes, for example transforming highly readable source code into highly performant object code). Related techniques can be used to optimize software source or compiler-generated object code, to inject instrumentation into code for purposes of evaluation or operational monitoring, to translate software from one source language into another, and to derive source code from object code. This may support reuse by allowing code to be customized to satisfy differing quality priorities among products, in effect deriving object code that exhibits different behavioral quality tradeoffs from a single source module, or for customization to different computational platforms.

The prescriptive approach is more flexible but more opaque than a descriptive approach —flexible in that it can effect more complex transformations of derived content with no definite limit on the ways in which instances can differ in form, opaque in that the exact form that a part will take is difficult to predict from its specification without a detailed understanding of the particular subject matter competence of a given prescriptive realization. The content that a prescriptive approach can derive can only be inferred

[1] The use of an alternative form of descriptive metaprogramming ("generative AI") for deriving content, along with the use of prescriptive metaprogramming for transformation, optimization, and instrumentation of derived content, is considered further in section 5.4.

based on the prescriptive mechanism used, including the meaning and form for specifying envisioned content, the nature and form of content that can be derived, and a definition of the transformations that can be progressively applied to the specified initial content to derive a final formulation of needed content.

A simpler but more limited approach to metaprogramming is an extension of the preprocessor approach. The descriptive approach to metaprogramming differs from a prescriptive approach in that it separates the representation of a set of similar parts from the mechanism used to derive instances of that set. This approach entails representing an envisioned set of similar instances in aggregate using a notation that distinguishes the common and differing form and content of instances. The notation has an associated mechanism (manual or automated) for deriving instances based on resolution by a developer of associated criteria that selectively reduces the set to a qualifying subset of instances, leading finally to determining a single instance.

The advantage of a descriptive approach is that the form, structure, and content of every derivable instance is easily seen in the notation; differences among instances are explicitly represented, with rationale implied in its differentiating criteria. This descriptive approach is a simple extrapolation from how parts are developed manually, mimicking the [source editing] actions a developer can take in building a part. Its limitation is that it cannot transform the essential structure of an instance, beyond alternatives as defined by the developer, relying instead on transformational post-processing as needed to refine the form or quality of the part.

MetaSynthesis—Generalizing from Metaprogramming

DsE was conceived based on the concept of a product family, representing an envisioned set of similar products in aggregate, as the only sound basis for systematic reuse. This approach can be applied to building customized instances/versions of all product model elements (including code, specifications, documentation, test scenarios, etc) and, by extension, to building customized whole-products.

With DsE, reuse entails conceiving an abstraction that encompasses a set of instances of similar needed content, specifying the criteria that distinguish among those instances,

and determining how content differs as needed for best fit to a purpose. The unifying abstraction is then the basis from which any encompassed instance can be obtained. Differing content among those instances corresponds to the diversity accommodated by the abstraction, expressed explicitly if the family is defined in an adaptable form or implicitly according to a generative mechanism's scope of relevant competence.

A family-based representation describes its instances by a unifying abstraction and associated criteria that are sufficient to uniquely distinguish among the instances. The derivation of products in DsE is based on a generalization of metaprogramming referred to as *metasynthesis*, recognizing that DsE targets the realization of complete products. ("Synthesis" is defined here as the composition of parts to construct a product; metasynthesis then is the exploration of mechanisms by which synthesis can be accomplished.)

Building and Using Adaptable Components

For DsE, the concept of a "component" can be generalized as being a packaging of content of any type. Adaptable content is content that can be mechanically customized to potential different uses. As a realization of a descriptive metasynthesis technique, an adaptable component is a medium for representing and mechanically deriving instance parts that provide content customized according to explicitly specified criteria.[2]

The concept of an adaptable component originated as a realization of the concept [see Section 2.7] of a software component as a set of similar modules, each of which satisfies an associated component specification. Building an adaptable component is equivalent to building a set of similar parts for multiple and changing use [Figure 4.1-1].

An adaptable component is characterized by an *abstraction*—a unifying concept that expresses the ways in which instances are alike—and by *adaptability criteria* that is sufficient to distinguish any instance from the others[3]. Representing a set of instances with an adaptable component provides the means to standardize the realization of its

---

[2] Grady H Campbell Jr., *Adaptable Components,* ICSE'99, 18 May 1999. <www.domain-specific.com/PDFfiles/AC.pdf> => {a more complete presentation of this material with examples}

[3] Equivalent to the concept of a product family, as discussed in 1.2.

instances, eliminating incidental differences while still accommodating essential differences.

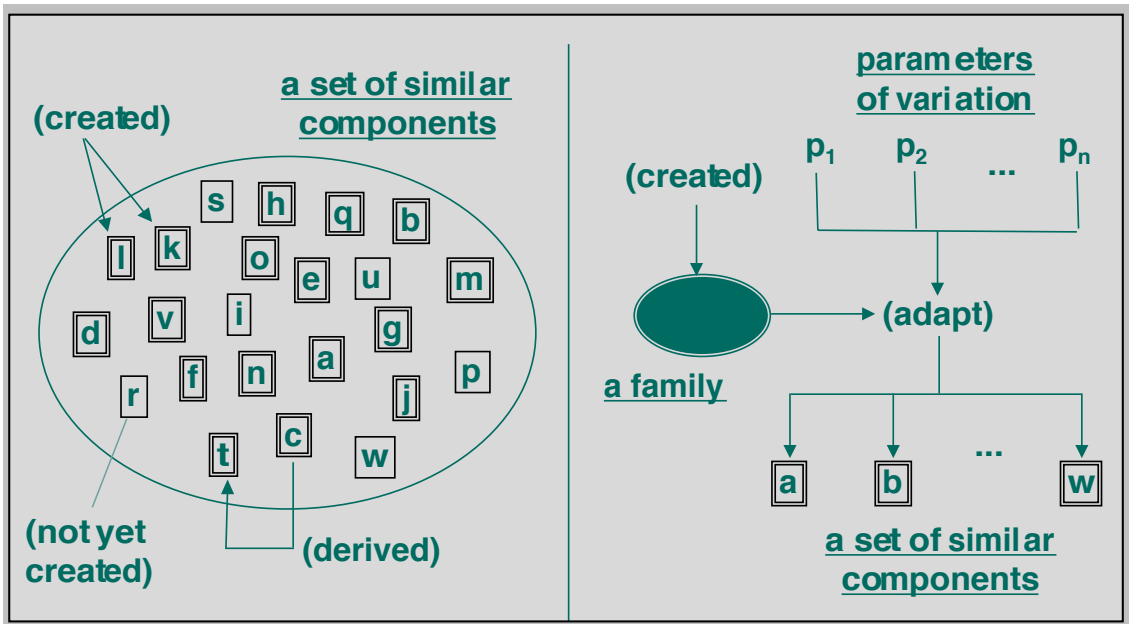

Figure 4.1-1. Two Views of a Set of Similar Parts

Adaptable components can be used to rationalize and compress the contents of a reuse library. Any set of parts that can be construed as instances of the same abstraction can be aggregated in the form of an equivalent adaptable component from which those and other similar instances can be derived as needed. The effect of modifying an adaptable component is to equivalently modify all derivable instances as appropriate, reducing library sustainment efforts. Reframing a library as a network of abstractions can substantially reduce the effort to obtain needed parts for comprehensive systematic reuse.

The following describes a particular text-based notation for defining adaptable components—the *mtp* (metatext processor) notation— conceived as a realization of the descriptive form of metasynthesis. This notation can be used to represent the common and differing aspects of a family, along with an associated mechanism for deriving customized instances. An advantage of this notation is that it can be used not only for describing code but also any other product model information that can be expressed in

text[4]. This enables the representation of all elements (i.e., design, implementation, and verification) of the product component model.

This mtp notation can be supported by an mtp tool that can be used during product realization or installation to transform the resulting metaprogram into instances of customized singular content as needed for evaluation or deployment. Alternatively, an mtp tool can be used to transform metaconstructs into a computational form that can be processed during product use to act as a generalized adaptive product that can selectively emulate any of the encompassed set of similar instance products (or, for example, to exhibit differing behaviors depending on operational circumstances).

An mtp Notation

The mtp notation defines "normal" (i.e., *target*) text content in which *metatext* constructs are embedded [Figure 4.1-2a]. Normal text expresses the common content of instances whereas metatext constructs express how instances of the family differ in their content. Embedded within normal text, these constructs specify how that text is customized to produce a corresponding complete instance of target text.

[4] Section 4.3 describes how this text-based notation has been used as the basis for representing adaptable content expressed in a graphical form, associated with the *metasyn* tool family.

- Target text (common parts of content)
- Substitution: « p »
- Selective substitution:
  « p.q ? «with q» : «without q» »
- Repetitive substitution:
  « for i in p.pi «same but different due to «i».»»
- Metaprogram instantiation
- Metaprograms (variant parts of content)
  - Name, to identify the abstraction
  - Metaparameters, with which reusers control tailoring
  - Definition (target text containing metaConstructs), showing how to extract tailored instances

Figure 4.1-2a. Metatext Constructs

The primary metatext constructs, besides target text, are *metaprograms*, *metatypes*, *file-inclusions*, and *instantiations*. These constructs, in turn, contain fragments of target text that become elements of instantiated complete target text. Each metaprogram and metatype has an associated name that serves as a referenceable identifier of its definition. The content of an mtp file can be explicitly embedded in another mtp file using the *file-inclusion* construct. An optional "name space" identifier provided in a file-inclusion is used to qualify references to metaprogram constructs defined in the included file, avoiding naming conflicts with constructs defined in the including file.

A metaprogram has an associated name, metaparameters, and a definition. Each metaparameter has an associated name and a metatype. Each parameter can be designated as required or optional—an optional parameter can be included or omitted from an instantiation. The metatype of an optional parameter can be omitted, meaning its inclusion as an instantiation parameter can be detected but it has no associated text value.

There are four categories of metatype:

- A target metatype value is a literal string of target text; this may be extended with subtypes that syntactically limit instance content
- A tuple metatype is defined by a set of named metatypes, each element having a value of the specified type and designated as required or optional;
- A sequence metatype is defined by a single metatype, its value being a sequence of values of that type;
- A user-defined metatype serves as an abbreviated reference to an equivalent metatype.

A metaprogram definition specifies common target text in which instances of four metaprogram operators [Figure 4.1-2b] can be embedded to inject variable content; each of these operators is strictly "functional", simply replacing itself with its equivalent target text value within the text in which it is referenced:

- The common target text is content that all instances of the metaprogram share;
- A *substitution* inserts an associated target text value;
- A *selective substitution* inserts one of alternative target text values;
- A *repetitive substitution* inserts each of a sequence of (optionally separated) target text values;
- A *metaprogram instantiation* inserts the equivalent target text value of the referenced metaprogram's definition based on specified metaparameter values (showing a simple example of an instantiation of the metaprogram F defined in Figure 4.1-2a.)

Developing an Adaptable Component

Having identified the need for a set of similar parts, representing them in aggregate—as an adaptable component—is similar to building a singular instance part. This approach works for building any type of text-based content such as software parts, product specification content, user document content, or testing materials.

```
« program F ( p1: text, p2: (p3?:text*, p4?:text) ) «
        "some common text «p1»; «
        p2.p3
            ? «with custom text: «for p in p2.p3 with «, » «p»»»
            : p2.p4 ? «plus alternate text: «p2.p4»»
            »"
»»

« F ( p1: «in all work products», p2: (p3:(«abc», «def», «ghi»)) )
----------------------------------------------------------
   =>
"some common text in all work products; with custom text: abc, def, ghi"
```

Figure 4.1-2b Simple Metaprogam Example

1. Capture content of one or more prototypical instances, each in the form of a base target text construct.
2. Identify a set of decisions that are sufficient to distinguish among these and other potential derivable instances.
3. Embed base target text within a top-level metaprogram, parameterized with decisions to represent how content is to be customized.
4. Build subordinate metaprograms to organize the top-level metaprogram into a coherent structure.
5. Derive the prototypical and 2-3 other instance parts using the metaprogram to verify that the adaptable component correctly expresses their content.
6. Refine and extend the metaprogram and its supported decisions to express the currently needed set of similar parts.
7. Extend the metaprogram further to support other potential future needs for a complete set of derivable instances as envisioned.

8. Generate prior and new instances to verify, revise, and extend the metaprogram, as needed.

Verifying an Adaptable Component

The verification of an adaptable component starts with the derivation of sample instances of content based on representative instantiation parameters. In instantiating a part, the metaprogram processor will detect any errors in usage of metanotation syntax. If the target metatype has been extended with subtypes, defining the correct form of target constructs, the processor can also detect syntactic errors in target content. A post-derivation evaluation (manual or automated) can be used to verify that derived instances are correctly rendered (e.g., that a code part conforms to language syntax and project conventions).

A review by developers of the adaptable component and associated derived instances can be used to inspect for quality criteria or to expose other more complex errors. Instances of the expressed family are examined to determine whether their content is properly formed, given particular instantiation parameters.

For content that is to be further processed (e.g., software source text), such processing may reveal errors in that content resulting from errors in the form of the adaptable component. This is extended to testing of derived software parts by integrating with related parts to build an executable for evaluating aspects of associated behavior. Other derived parts are further verified in their usage within product model elements, which may reveal flaws in the originating adaptable component.

If any formal methods are applicable to instances of a particular family, these methods can be applied in verifying properties of derived instances. These methods may be adapted to the verification of an adaptable component as a whole {as discussed in section 5.2} to establish that all instances, of the family or of a distinguished subfamily, satisfy particular properties of interest.

The Economics of Building and Using Adaptable Components

A systematic approach to reuse presents the alternative of relying on either a built (i.e., extensional) set of customized instances of each type of parts needed or an adaptable

component (i.e., an intensional set) from which customized instances can be mechanically derived as needed. Either approach can be justified depending on the extent of variety entailed in envisioned reuse.

Relying on a built set of parts is viable if only few instances are needed and those instances lack sufficient common content to justify an adaptable form. With only few instances having clearly delineated differences, a developer will be able to choose the best instance with only limited effort.

{see discussion of DsE program economics and metrics in sec 3.1}

Relying on an adaptable component requires a clear understanding of why multiple differing instances may be needed (i.e., why a single instance will not suffice for all potential uses) and the decisions a developer needs to make to obtain a customized instance (or due to uncertainties, a small number of candidates to comparatively evaluate). The logic of this approach is that building an adaptable component should cost less than building three customized instances. The case is easily made when the need exists for more instances, not only due to the cost of separately building each one but also due to the cost to developers of having to repeatedly locate and choose among many similar instances and possibly modify an existing instance to better fit current needs.

Recognition of the need for differing instances may develop over time as needs are better understood. An initial small number of crafted instances can at some point be unified into an adaptable component that will suffice for deriving both those instances and others that differ. The differences can be a reflection of future changes in a single use or the emergence of multiple differing uses.

The small added cost of building an adaptable component is mitigated by the reality that conventional development tends to require exploring alternative realizations of a part, all but one of which is then discarded. With an adaptable component, all feasible explored alternatives can be retained in a unified form for potential future consideration with small additional effort when needs change or new uses emerge.