4. Automating Software Production

The two preceding chapters describe two related approaches to software production without assuming automation beyond the continued use of conventional tools. Although not a prerequisite to the described approaches, enhanced automation would enable significant advances in productivity and product quality with those approaches. This chapter describes a progression of additional automation for streamlining of software production, with an enhanced approach to component reuse, an environment for generating software products, and customized environments (i.e., factories) for rapidly manufacturing complete software-based products.

The automation discussed in this chapter builds on the concept of software reuse, which in conventional usage relies on access to libraries of previously built software components. A more flexible form of reuse is described here based on the notion of defining components to be adaptable for different uses. That formulation is a basis for building a product family from which customized products can be mechanically derived and revised over time. As more advanced forms of automation (surveyed in the next chapter) become practical, those may enhance or fill the role played by adaptable reuse in the automation of software-based product development.

The need for adaptability is based on a recognition that customers can have differing needs even for products serving similar purposes and that each customer's needs change over time. To account for this, a capability is needed to build products in a way that allows them to be progressively changed over time, whether to address a single or multiple customers' needs, without undue uncertainty or effort.

The point of software reuse is both to leverage expertise needed to properly build a given part and to avoid redundant work, not repeatedly building and verifying a selection of incidentally different parts when a single instance can be repeatedly used instead. Repeated use of a previously verified part in different products may also expose missing content that can be added or residual flaws that can be fixed, improving the quality of the part for all its uses.

The conventional approach to reuse is inefficient in that (1) an existing part built to satisfy specific needs is seldom an exact fit to different needs, requiring some degree of change; the potential benefits of reuse are lost if the "reused" part has to be modified anyway, not only initially but anytime thereafter when needs for that part change; (2) although the effort of building the part can be reduced, its being changed requires that it be evaluated as exhibiting expected behavior; (3) each time a new part is derived from an existing part or changed, a new potentially "reusable" part is created, resulting in a growing set of similar parts and making the selection of a part for future reuse more difficult. The practical result is that in many cases it is easier and faster to just create a new part from scratch.

This chapter describes three levels at which customized instances of a set of similar instances can be selected and derived:

- The Systematic Reuse of Software Components

  Systematic reuse is based on representing sets of similar components in the aggregate form of an adaptable component. Instantiating an adaptable component derives an instance component customized to have specific capabilities.

  an extension of the conventional practice of expressing behavior in a prescriptive notation that is mechanically translated into a native hardware notation

  Adaptable software is software that has been built to enable being mechanically customized to potential different uses. The source form of any software can be modified to change its behavior but software built to enable specific adaptability without having to directly modify its source form enables such changes to be made quickly and safely. The practice of creating a new instance as a copy that is then modified results in duplication of effort when changes in both are later needed; uncoordinated changes to the instances over time lead to the dilution of similarities and divergence lacking effort that would maintain common aspects to the greatest degree possible.

The complexity of software arises in the distinction between intended behavior and realized behavior. It is relatively straightforward, given a description of what behavior software should exhibit, to then create software that has some approximation of that behavior; it is far less easy, given realized software, to then state exactly what behavior it will in fact exhibit. Furthermore, many different software realizations can provide the same behavior; many others can provide similar behavior that will be better or worse depending on actual needs.

- An Abstraction-Based Environment

  An abstraction-based environment supports a descriptive notation for defining a generic product model for a needed product based on a lattice of supported abstractions. Each abstraction is represented as an adaptable component from which customized components are derived and combined to create a particular software realization.

- Domain-specific Environments for Producibility

  A domain-specific environment, pursuing the producibility vision, uses a constrained descriptive notation that reduces development efforts based on being limited to a specified product family. This increases the degree of commonality among derivable products, requiring resolution only of a lattice of decisions that discriminate among those products.