REPRINT 9681

GENERAL ELECTRIC

GENERAL ELECTRIC COMPANY
CORPORATE RESEARCH AND DEVELOPMENT
P.O. Box 43, Schenectady, N.Y. 12301 U.S.A.

TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

D. Kapur, D.R. Musser, and A.A. Stepanov

TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

Deepak Kapur, David R. Musser, and Alexander A. Stepanov

General Electric Research and Development Center
Schenectady, New York 12345/USA

We will describe specification methods we are currently developing and illustrate their use on the workshop example of the communications network. The overall goal of our work is to develop formal notation and methods that allow conceptualization of problems and solutions at an appropriate level of abstraction. The kinds of problems we would like to be able to deal with range from the large software systems developed for commercial and government customers, to small "programs in products," i.e., microprocessor control programs whose reliability is of critical economic importance because they are duplicated in such large quantities. This implies that we must be able to deal with issues such as distributed computation, timing of real time control, and the requirements placed on programs by their hardware and application environments.

1. DESIGN PHILOSOPHY

The ability to reason abstractly, to see generality through the particular, and then to particularize the general, are very useful for the development of high quality software. A central problem with most current notations and methods of either specification or programming is that they do not allow enough use of abstraction. We are thus required to deal with too many details simultaneously. Most languages support a fixed set of abstraction mechanisms and ways to instantiate them. They do not provide general purpose mechanisms for abstracting a class of objects sharing common properties.

For example, most work on data abstraction has dealt only with the notion of abstracting away from implementation details. This is the important idea of expressing "what" operations are supposed to do, without getting into details of "how." But it is also often useful to

talk about a software module without knowing precisely "what" it does - that is, to abstract at the behavioral level also. For instance, it may be desirable to describe abstractions such as "process control system," "operating system," "transfer service," "electronic mail system," etc., without stating precisely what they do under all conditions.

We need to be able to use abstraction and specialization of concepts for purposes of organization and understanding, irrespective of the implementation question. The intention is to capture the general properties of an abstract notion without having to specify too much detail. When one wishes to describe a specialization of this abstract notion, only details particular to the specialization are described, while whatever is known about the abstract notion based on its general properties is also carried to the specialization.

It is also our goal that the notation and methods we develop ultimately be usable as a programming language/system as well as a specification language/system. This seems to be a different philosophy from most of the other participants in this workshop. In our view, specification and programming are not such distinct activities that they should be done in different frameworks. In many cases, the simplest and clearest way to express "what" is to be done is to say "how" it can be done, provided the "how" is expressed in terms of essential concepts rather than the irrelevant details required by most specification languages and all existing programming languages. Conversely, many of the tools that we are all finding useful for specification, particularly abstraction and specialization, should also be available to the programmer.

Having a unified framework for designing specifications and programming enables us to view the development of software as a stepwise and systematic refinement of higher level concepts into lower level machine (or programming system) supported primitives. In this way, writing specifications becomes an integral part of the programming activity and provides a way of recording higher level design decisions. This is in contrast to the view that writing specifications and programming are totally distinct tasks, performed in different frameworks and using different notations and styles - a scenario whose main consequence is that specification writing is regarded as burdensome activity.

In the proposed framework, a series of refinements constitute a set of design decisions leading to a program. It should be possible to backtrack to any stage of decision making and explore an alternate path of refinements. Different design decisions can thus be compared and evaluated.

A notion that seems to be particularly useful for achieving this kind of organization is that of "generic objects." A generic object is a notion for grouping objects sharing common syntactic structure and semantic properties. Informally, a generic object describes a whole class of non-isomorphic possibilities, unless a stage has been reached at which the described objects are fully defined. Adding properties to a generic object makes it more specific by narrowing the range of possibilities. Once we allow such genericity of objects, the way is open also to the development and use of "generic algorithms," which offer a way of organizing and extending our knowledge of algorithms and control.

We have begun developing a language called "Tecton" (Greek for "builder") for constructing generic objects and algorithms. Tecton will embody the above discussed design philosophy. It will provide a rich set of generic objects, and these objects will be built up and related by general description building constructs in Tecton. We will first discuss the constructs for building generic objects, then some of the different kinds of generic objects supported in Tecton. Later, we discuss how the communication network example can be done in Tecton using these mechanisms.

2. CONSTRUCTS IN TECTON

The proposed set of mechanisms in Tecton includes:

(i) create, to define a class of objects by describing their syntactic structure and a set of properties relating their various components.

(ii) refine, to define a class of objects by refining an existing class by adding a set of properties, potentially to the level of detail where the objects in the resulting class are narrowed down precisely to what is desired in an implementation.

(iii) abstract, to define a class of objects by generalizing an existing class of objects. The generalization can be specified by

generalizing a particular object(s) or class(es) of objects used in the description of the existing class or by forgetting some properties of the existing class.

(iv) instantiate, to identify commonalities among two classes of objects in order to transport the properties and algorithms of the class being instantiated to the other class.

(v) provide, to define new operations on a class of objects in terms of the existing ones.

(vi) inform, to add new properties to existing information about a class of objects.

(vii) implement, to give an algorithm for an already defined operations on a subclass of objects by making use of their specific properties.

(viii) represent, to relate a class of objects and its operations to another class of objects and associated operations as an aid to constructing implementations.

The create, refine and abstract constructs define new classes of objects. The abstract construct is roughly the inverse of the refine construct. The instantiate, provide, implement, and inform serve to include more knowledge about a class of objects, while represent is used to represent a class of objects in terms of another class. The instantiate construct records the information that a class of objects can be refined to another class; or, stated another way, a class of objects can be abstracted to another class. The latter could be obtained from the former by reversing the arguments to instantiate.

The use of these constructs is illustrated on structures (see below), a class of objects definable in Tecton, in [5]. (Except that abstract was not discussed and refine was called enrich in that paper.) In the next section, we will give examples of some of these constructs; their use is also illustrated in the discussion of the communication network example.

3. OBJECTS IN TECTON

We discuss four different types of objects in Tecton which we have found useful in describing different kinds of activities of a complex software system: structures, entities, events, and environments. Some of these types of objects have appeared previously in

data base query languages, simulation languages and functional programming languages, though not in as general a form as in Tecton.

3.1 Structures

A structure is a representation of a time-independent object. The language provides certain primitive structures, for instance "set", "multisets", and "sequences", from which new structures are built up step by step using constructs outlined in the previous section. Each new structure is specified as a collection of other structures and operations on these structures, which satisfy certain properties regarded as axioms; e.g.,

```
create semigroup(S:set; +: S+S -> S)
   with  x + (y + z) = (x + y) + z;

create monoid(S:semigroup; 0: -> S)
   with 0 + x = x;

refine monoid into abelian monoid
   with commutativity: x + y = y + x;
```

If for some reason monoid was defined directly from set, then semigroup could be created using the abstract construct by dropping the nullary operation 0 and the property that 0 is a left identity. Note that the above definition of semigroup does not define a specific semigroup, but a generic semigroup. Such a definition defines the semigroup structure type. This is different from defining a specific structure, such as natural numbers, which is called just a structure.[1]

We define an operator to be a secondary operation associated with a structure type, that is expressed in terms of the (primary) operations on structures, e.g., we can define an operator "reduction" on the structure type "sequences of monoid" by

```
provide sequences of monoid
   with reduction:
     x -> if x = null then 0
            else head(x) + reduction(tail(x)).
```

Such operators resemble those in APL [4] and in Backus's Functional

1. No distinction is made between structure and structure type syntactically in the examples discussed in [5], though the distinction is suggested in the discussion of the examples. The term generic structure is used there instead of structure type.

Programming Language [1]. They provide the same advantages of powerful, concise expression of computations, but differ in that they are defined with respect to a structure type - one to which they naturally belong. This is possible because Tecton permits description of structure types in terms of their properties.

In [5] an example is carried out of the development of several generic algorithms for sorting in terms of the reduction operator. The KWIC example can be done concisely in terms of a "closure" operator that gives a simple way of building the set of all rotations of a title. Operators such as reduction and closure should be a part of the standard vocabulary of both specifiers and programmers.

Structures types thus provide a method for abstracting a set of operations and properties which can be applied to different time-independent, unchangeable (also called immutable or constant) objects. They enable us to abstract general properties of data and express algorithms on that data in an abstract form. Structures are similar in flavor to immutable data types as discussed by Liskov et al [6] and Guttag [3], but they are more general and powerful. Structure types are similar to "theories" of Burstall and Goguen [2] and "sypes" of Nakajima et al [7], but we do not restrict ourselves to algebraic structures. Using structures and structure types, it is possible to define an abstract data type as well as a collection of abstract data types and associate operators with the collection as a whole.

3.2 Entities

An entity is an object that exists and changes in time. An entity type is a description of a collection of entities with common attributes and properties. For example, "parcel" and "message" are entity types in a transfer service and a mail service respectively, as will be discussed later.

An entity is characterized by a collection of attributes, which are functions from entities into structures. Examples of attributes of the entity type "message" are signature, creation date, sender, sending date, contents, etc. Attributes can have special properties, such as unique, immutability, required. For example, the signature attribute is immutable and required for every message. An entity type can have properties, which specify consistency relations among the

attributes of each entity. For example, the sending date of a message is always after its creation date.

Primitive updates on entities are create, which defines a new entity with specified attributes; destroy, which gets rid of entities with specified attributes; and change, which modifies attributes of existing entities. An entity type can also include a list of user defined updates, which have primitive updates as their building blocks.

An entity type can contain a list of triggers, which are functions on the attributes of the entity type. along with associated events. When a trigger changes its value, it activates an event. (Events are discussed below.)

3.2.1 Classes and Classifications

It is often necessary to dynamically group different entities based on their attributes or other properties. This is achieved using the class and classification mechanisms.

A class is a collection of entities of the same type. The class of all currently existing entities of some type is called the complete class of that entity type. Thus any class of entities of a certain type is a subset of the complete class of that type. Classes may be described by attributes or properties of attributes; e.g., the class of all messages with author "Jones," and the class of all messages received after "June 30, 1980" with author "Smith." Classes may themselves be regarded as entities; they are called class entities. A collection of class entities of the same type form another class; this latter class is called a classification of the type. For example, a classification of the message type can be based on the attribute author. One class of this classification is the class of messages that have author "Jones".

Classifications can have some special properties, such as being complete, which means that every entity of its type must belong to at least one class of the classification, or nonredundant, which means that every entity of its type belongs to at most one class of the classification. The classification on the message type based on author is complete but redundant, as a message can have more than one

author.

In addition to standard primitive updates of entities, there are two primitive updates particular to class entities: the update insert puts specified entities into the class, whereas delete gets rid of entities.

There is a primitive update particular to classifications, called transfer, which moves specified entities from one class in the classification into another. In case of a mail service, for example, transfer on a classification on the message type can be used to describe the event of sending a message from A to B. Every classification also has an associated binary relation, called a transfer relation, which specifies allowable transfers among classes in the classification. A transfer relation associated with a classification on the message type would describe who can send messages to whom.

3.2.2 Relations

A relation is a special kind of entity which establishes an association among entities of several types. Primitive updates on relations are link (which expands the relation by adding a tuple) and separate (which shrinks the relation by deleting a tuple). There are also special operations that build new relations, such as union, intersection, product, closure, etc.

3.3 Events

An event is a representation of when and how certain objects are changed. One can define a specific event or a generic event. A generic event is defined using the event type mechanism. The description of an event (specific or generic) includes an activation description, an action list, timing constraints, and an entity type which specifies attributes and properties of the event.

An activation description specifies how the event can be invoked: by another event, by time, or by a trigger.

An action list is a list of invocations of different events and updates on different entities that are caused by the event invocation.

These actions are executed in arbitrary order if a precedence of actions is not explicitly specified.

Timing constraints specify constraints on the duration of an event invocation, such as minimum and maximum duration, as well as global constraints on the duration of all event invocations, such as frequency distribution and time precision.

The inclusion of an entity type in the description of an event provides for the notion of instances of an event corresponding to different invocations of the event. This also allows for classes and classifications of events.

3.4 Environments

Environments are a mechanism for grouping together different kinds of objects such as structures, entities, events and (sub-) environments. It is thus essentially an organizational tool, recording the structural relationship among various objects. A specific environment as well as a generic environment can be defined in the same way other specific and generic objects can be defined in Tecton.

Environments also contain interfaces to other environments. An interface describe the conditions under which the objects of an environment may be used by other environment. These interfaces are built with the help of the privilege relation.

If an object belongs to an environment, i.e. the object is in the class, the environment has an unlimited privilege to use the object. It also has a privilege to give a privilege to use this object to other environments, including the privilege to give a privilege. The privilege relation is a ternary relation among the classification of environments, the entity of objects, and the class of parameters of operations. It specifies whether an environment can use an object as a particular parameter of an operation.

4. COMMUNICATION NETWORK EXAMPLE

The statement of the workshop problem does not explicitly deal with the question of what the network is to be used for. The presumption is, of course, that some sort of message system will be implemented with the aid of the network. We will begin at the level of the message system specification and refine it to the network level. Thus, rather than just making up a list of properties that the network should satisfy, the needs of the message system will place a natural set of requirements on the network. (We had already chosen to specify a message system, as a way of developing some of the features of the Tecton language, before receiving the statement of the workshop problems.)

A key goal of our approach is to find ways of using abstraction to break up the treatment of complex systems into natural components that, when recombined, fully capture the essential properties of the system. (This division into components may be unrelated to decompositions that are made in implementations of the system.) In specifying a message system we have identified several main components. First, a message system provides a "transfer service" for messages, and below we shall concentrate on this part. Other components would be facilities for display and scanning, for composing and editing messages, and for filing and retrieving messages.

In specifying a transfer service, we first observe that we can view it more abstractly than just dealing with messages and passing them around in the way that users of a message system typically do. Instead of messages, our transfer service will deal with "parcels" and will serve "clients." In a message system, parcels will be specialized to messages and clients to users, but parcels could also be currency units and clients could be banks, for example, if the transfer service specification were used as part of the specification of an electronic funds transfer (EFT) service.

The question of who can send parcels to whom can also be dealt with more generally than just adopting the discipline of a message system. Both a message system and an EFT service have the same discipline, which might be called a "capitalistic" discipline - each client can transfer his own, and only his own, parcels to any other client. But a memory management subsystem of an operating system could also be viewed as a transfer service in which the parcels are

memory blocks, the clients are jobs, and the discipline of transfer is that there is a supervisor that has the privilege to transfer parcels from any client to another.

We first create a new type of environment, called "Transfer Service." and provide it with an event for sending parcels among clients.

```
create "Transfer Service"
  from environment type,
    parcel: entity type,
    clients: complete nonredundant classification of parcels.

provide Transfer Service with
  an event type Sending "Send P to C",
          where P: parcel, C: client,
          with action:
             transfer P into C.
             {will transfer only if the transfer relation
              of this classification allows it}
```

We ensure that each client can send parcels to every other client with

```
refine Transfer Service into "Complete Transfer Service" with
   transfer relation of clients is complete.
```

Now, to make Sending available in a Mail Service, we can

```
instantiate Complete Transfer Service in Mail Service as
    Mail Transfer Service with parcel=message,
                               clients=message bins of users.
```

Now we give a refinement of Transfer Service into a network.

```
create "Connected Static Network" from
    Transfer Service,
    a relation "C can route D to E", where C,D,E: clients
  with
    transfer relation of clients is
                    connected immutable relation;
    for every C,D there is an E such that C can route D to E;
    if C can route D to E then
```

```
        C is related by transfer relation of clients to E
          {making direct Send possible}
        and E is related by
           (closure of transfer relation of clients) to D.
               {that is, D is reachable from E}

    provide Connected Static Network
      with
        a functional relation "address" of parcel to client,
          initially the client such that parcel is in client;

        an event type "Start P on its way to C",
          where P: parcel, C: client,
          with action: change address of P to C;

        an event type "Forward P from C", where P:
          parcel, C: client,
          activated by trigger:
          P is in C and (address of P) is not C,
          with action:
               (Send P to E) for some E such that
                  C can route (address of P) to E;

        an event type Network Sending "Network Send P to C",
            where P: parcel, C: client;
           with actions:
            Start P on its way to C,
            then collect
                   Forward P from any client.
            {"collect" says that specified events constitute
            actions which are parts of this event}
```

The Network Sending event only actually performs the action of "Start P on its way to C"; it then just "observes" the forwarding events that occur because of activation by the indicated trigger. This is an abstract way of specifying a distributed implementation of an event.

Having defined the network, it is now possible to

```
    instantiate Complete Transfer Service
       in Connected Static Network
          by Sending = Network Sending.
```

with the result that this network refinement can be carried over to other instances of Complete Transfer Services, such as the Mail Service.

5. REFERENCES

1. Backus, J., "Can programming be liberated from the von Neumann Style? A functional style and its algebra of programs," CACM 21 (8), August 1978.

2. Burstall, R.M., Goguen, J.A., "Putting Theories Together to Make Specifications," Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August 1977.

3. Guttag, J.V., "Abstract Data Types and the Development of Data Structures," CACM 20 (6), pp. 396-404, June 1977.

4. Iverson, K.E., "Operators," TOPLAS 2 (1), October 1979.

5. Kapur, D., Musser, D.R., Stepanov, A.A., "Operators and Algebraic Structures," Proceedings of the Conference on Functional Programming Languages and Computer Architecture, New Hampshire, Oct. 1981.

6. Liskov, B.H., Snyder, A., Atkinson, R., Schaffert, C., "Abstraction Mechanisms in CLU," CACM 20 (8), pp. 564-576, Aug. 1977.

7. Nakajima, R., Nakahara, H., Honda, M., "Hierarchical Program Specification and Verification - A Many Sorted Logical Approach," preprint RIMS 256, November 1978.