As mentioned, the operating system calls are available in the form of Modula 2 procedures. They are imported from the modules of the operating system into the modules of the application and are used there. Thanks to this easy access to the services of the operating system, a program can be coded quickly and becomes very easy to read. The following diagram shows a section from an application module.

```
IMPLEMENTATION MODULE PressEX;
FROM SystemBase IMPORT SysRepSC;
FROM Request IMPORT ACCEPTREQUEST, REPLYREQUEST;
FROM IPCdataM IMPORT GETDATAFRAME;
FROM IPCbase IMPORT RegionTY, RenameTY, AccessRightSC;
FROM PressCD IMPORT PressGrid;
FROM PressCD IMPORT Sensor;
FROM PressPT IMPORT ComparePT;
END PressEX.
```

Fig. 16

Prospects

The breakdown of the services into simpler services and the finding of elementary, concrete services will be supported by a tool. The tool will mainly take over the handling of the specifications for the services and the monitoring of the relationships between them. The structures discussed such as family, service group, team and process are still being created today by the developer using operating system calls. The structures are known according to the Bottom Up part. The future tool will automatically generate the modules of a team and insert the appropriate operating system calls in them at the correct place.

During debugging the structures will be visible and testing will be possible at this high level.

---

A COMPLEXITY CONTROL STRATEGY FOR LARGE SYSTEMS MODELS

Paul T. Ward
Software Development Concepts
New York, NY

Paul Ward is a principal partner of Software Development Concepts, a New York City based firm providing consulting and training to organizations that develop real-time systems. He has more than 25 years experience as a systems developer, project leader, methods developer, and consultant. He is co-developer of the Ward-Mellor notation which is implemented in many real-time CASE products. Among his books, *Structured Development for Real-Time Systems* has become a standard text for software engineers.
Can Complexity be Managed with Object-Oriented Models?

Many of the concerns about the feasibility of object-oriented development center around the management of complex models for large systems.

Traditional functional modeling has been reasonably successful at complexity control because of its hierarchical, "divide and conquer" character.

However, there are also a number of complexity control strategies that can be used in conjunction with an object-oriented model.

Functional Decomposition Procedure

Traditional functional decomposition involves:

- Envisioning the system as a single function
- Breaking the function into a small group of subfunctions, interfaced with inputs/outputs or with state variables
- Breaking each subfunction into sub-subfunctions......

The process is complete when the lowest-level subfunctions can be described simply.
EXAMPLE: Several levels of decomposition represented as DFDs

Properties of a Functional Hierarchy

When a state variable is accessed by two or more higher-level functions, any contained lower-level function can potentially access the state variable.

Since the most important state variables are typically the highest-level ones in the hierarchy, they can be accessed by the largest number of functions.

These properties lead to a "ripple effect":

- A change to any low-level function can potentially effect state variables at high levels in the hierarchy.
- A change to a high-level state variable can potentially affect many low-level functions.
EXAMPLE: A structure vulnerable to the "ripple effect"

Influence of Object-Orientation on the Ripple Effect

Object-orientation encapsulates low-level functions (operations) with state variables (stored data within objects).
This means that:

- each state variable can only be accessed by the low-level functions within its own object, and
- each low-level function can only access the state variables within its own object,

thus reducing the ripple effect

Hierarchies using Composite Objects

A composite object can encapsulate

- data which is accessible by its own operations
- lower-level objects which contain their own encapsulated data and operations

This provides some aspects of a hierarchical structure while retaining the benefits of encapsulation.
EXAMPLE: A Composite Object

Multiple Concurrent State Machines

The qualitative aspects of an object's behavior can be described by a finite state machine.

EXAMPLE: States of a Gyroscope
A finite state machine may be imbedded in a data flow model to control its processes.

Such a data flow model represents both qualitative and quantitative aspects of the behavior of objects.

**EXAMPLE:** A Gyroscope object with an embedded state machine

In a typical real-time system, objects of a number of different classes may show behaviors that change qualitatively over time.

These behaviors also typically interact with one another; a state change in one object causes state changes in one or more other objects.

Thus, the collection of objects contains a collection of cooperating, concurrent state machines.
It has been proven that a collection of cooperating, concurrent state machines is exponentially smaller than a single state machine in describing a given behavior.

Although multiple cooperating state machines can be incorporated in models of various kinds, object-oriented models provide a natural framework for this kind of organization.

Separate Object-Oriented Models of Distinct Application Domains

One way to avoid the problem of large system models is to create multiple, relatively independent models rather than a single monolithic model.

The components of a single system can often be analyzed into two or more groups, each belonging to a separate application domain.

There is typically one dominant domain (the "real" application), with other domains providing resources to the dominant domain.
EXAMPLE: The objects in this model can be separated into "process control" objects and "data communication" objects.

Note that the message names to and from the "data communication" objects are inappropriately specialized. They should be called "message input", "message", and "message output."

Separating the domains results in models that are at a uniform level of abstraction and are easier to name.
A collection of models of distinct application domains can be represented by a directed graph with one node per model. The arcs point to domains that are used as resources by other domains.

The arcs are defined by associating them with mappings from the domain using the resource to the domain providing the resource.

EXAMPLE: The "Device Command" message in the Process Control domain maps to a "Message Input" at its origin and a "Message Output" at its destination in the Data Communications domain.

Summary

A complexity control strategy organized around

- Object Orientation
- Embedding of cooperating, concurrent state machines within the objects
- Separation of models based on distinct application domains

provides many opportunities for separation of concerns and has distinct advantages over a model organized around a functional decomposition hierarchy.