Language Constructs and Support Systems for Distributed Computing

C.S. Ellis, J.A. Feldman, and J.E. Heliotis
Computer Science Department
University of Rochester
May 1982

1. Introduction

This paper describes programming constructs and system support functions that are intended to facilitate the programming of reliable distributed systems. The systems considered include very different kinds of computers communicating through a network. Such a heterogeneous network offers a number of advantages to designers of applications software. Different machines emphasize different capabilities and many problems naturally break down into subproblems that are best solved with specialized resources. There is clearly a need for programming tools that will allow users to exploit this kind of environment.

Distributed systems and application programs employing heterogeneous networks have been studied at Rochester for some time and many of our early insights were encapsulated in a proposed programming system, PLITS [Feldman79]. The general approach of expressing principles of distributed computing as primitives of a programming system has proved successful in the past and is continued in this paper.

The idea behind the PLITS effort has been to identify a small number of constructs that can be added to standard programming languages to facilitate message-based computing. These include self-contained and self-governing *modules*, shared symbolic *slotnames* as the basis for communication, asynchronous message passing, and *transaction keys*. The PLITS constructs do appear to provide a reasonable level of *programming* constructs for distributed computing, but do not provide any help for *task management* and this has turned out to be a major bottleneck. Our experience suggests that registration facilities and emergency message handling should have a more well-defined role in the programming system. More fundamentally, some additional structuring mechanism is needed to express associations between modules. For example, the user may view a problem as two loosely related subproblems each requiring several communicating modules. When the solution is written in PLITS (or other recent high level languges for distributed computing [Brinch Hansen78, Hoare78, Cook80]), a "flat" collection of loosely coupled objects results. It is very difficult to discern the relationships that may exist between some objects and not others. However, it is those relationships that greatly influence the effectiveness of various failure recovery strategies, debugging techniques, or other management policies. A hierarhical structure of modules will not suffice, because each module may be involved in several activities.

There appear to be very few research efforts aimed at providing structuring tools for programming large distributed applications with an emphasis on reliability. One major project is being carried out by Barbara Liskov et al. at MIT. This is in the form of a programming system named *Argus* [Liskov79, 81, 82]. The focus is on applications in which the maintenance of on-line distributed data is of primary importance. Consequently, atomic actions serve as the cornerstone of the language design. The implementation is based on a simplification of two-phase locking and a two-phase commit protocol, both concepts transferred from database research [Gray78]. The underlying system maintains data structures holding the recent action history. Although the content and organization of this data have not yet been presented, there are probably some similarities with the information to be registered in our approach. The notion of nested *Argus* actions is developed in some detail [Moss81]. The interactions between the parent action and its subactions with respect to locking and updating objects are specified. There are language constructs for explicitly starting and stopping top-level actions and subactions. Because of the emphasis on atomicity, the semantics available for dealing with functions that should not be recoverable (e.g. recording statistics) seem awkward.

The structuring unit is called a *guardian* and it can be viewed by the user as a logical node encapsulating various resources. Operations on the resources or objects managed by a guardian are called handlers, are invoked by remote procedure call, and are executed as subactions of the calling action. Each guardian resides at one physical machine. Recovery is on a per-guardian basis. After a crash, the stable objects are automatically restored to the values they had at the latest top-level *commit* involving their guardian and a process is started to restore volatile state. This programming system relies heavily on recovery that can be done automatically using atomic actions. It does not appear that the language provides any constructs to facilitate coding user-defined recovery steps. The guardian, as an abstraction of a reliable network machine, is not meant to capture the logical relationships that may span the network during a distributed computation.

Other related work on reliablity has been done at Newcastle upon Tyne [Randell75, Shrivastava78, Shrivastava81]. The most recent of the papers addresses problems of providing backward error recovery in a distributed system. It is based on an object-oriented multilevel model of computation and describes a design for object managers and for underlying low-level facilities that offers atomic actions as the central tool for dealing with node crashes and other exceptional conditions. What is proposed, in our terminology, is a particular scheme for registration of objects affected by a particular process (i.e. for each process, a table of recovery data and a remote worker list) and automatic notification to the object managers involved when a process establishes or discards a recovery point or when recovery to an existing recovery point is required. Recovery regions within a single process serve as the structuring mechanism. The worker list associated with a process identifies the agents created to represent that process at remote sites and is used in sending appropriate notifications to them. An action is committed once the outermost recovery point (of nested recovery regions) has been discarded. There is the notion of default exception handling; but for certain unrecoverable objects, such as those dealing with the external environment, state restoration must be explicitly done within exception handlers. This approach stresses support for atomicity. It does not address associations between processes or high-level language constructs. Earlier work focused on recovery in the multilevel model within a single machine and on recovery block structure in which there is spare code standing by to be used when the main code fails. Some aspects of both the MIT and Newcastle efforts are captured in our task management model but the emphasis is more general than atomicity and recovery.

2. Activities and Objects

The one new concept added recently in our work is the notion of an *activity* that serves as a structuring tool for *objects*. Objects can be viewed in a traditional sense as abstractions of resources [Jones79] and can be realized by PLITS modules (among other things). An activity involves a collection of managed objects that are related to each other by their role in accomplishing some logical task. The activity can be defined as an abstraction of a set of actions which gives various objects some form of semantic coherence with respect to the user's problem. A single object may participate in several different activities. Thus activities impose an orthogonal structure on a computation crossing the physical boundaries of machine or module. An activity has some aspects in common with the notions of *job* (The principles of distributed job management were outlined in [Lantz80]. [Wulf81] described Hydra job objects.) and *transaction* (in the database sense, see [Gray79]). As an analogy from the 'real' world, consider the paint, brushes, walls, dropcloths, and human helpers as the objects involved in an activity like painting a room while protecting the floor. Some of the objects (e.g. the paint) will be dedicated to the painting activity but others may participate in other activities as well. For an example of an activity in a programming environment, consider compiling in a network where the compiler resides on a shared computer, the source file has been prepared and stored at a personal workstation and error messages are to be printed at a remote printer. This activity involves a number of server processes at different machines that may be shared among several activities. Notice that the names used for these activites (i.e. painting and compiling) are verbs describing actions.

One of the early goals of this work has been to develop a terminology for describing the actual or intended behavior of loosely coupled processes working together on a distributed job. For example, it would be useful to trace selectively all interprocess communication related to one activity. A number of other capabilities are made possible by having a structuring mechanism that captures the relationships among objects participating in a task as well as the potential interference within shared objects. Characterizing processes either as shared servers or as dedicated processes appears to be vital for understanding. A major motivation for this approach is the need to specify how a failure of one object affects the rest of the distributed computation. In many cases, one would like the computation to adjust to certain failures and continue, possibly with degraded performance. However, if essential objects are lost and there is no point in continuing, it should be possible to identify which other objects should be cleaned up as a result. When the application dictates that data at a crashed site should eventually be made consistent with related data at continuing sites, mechanisms for recovery must be available. Each of these scenarios can be formulated in terms of the activity-object framework.

One of the requirements we impose is that a reasonable degree of parallelism be achievable. In particular, a

multiplexed server process may at any point in time be dealing with a number of transactions in various stages of completion. It may set up a certain amount of internal state for each request it is processing. This presents the problem of how to selectively clean up this internal state when the server then receives notification that one of the requesting processes has aborted. Activity tags can provide a handle on managing internal state in a meaningful way.

One objective is to make the abstract activity structure visible in the design of a distributed computation. The intention is to provide language constructs for establishing an activity hierarchy and manipulating its components independent of how they may be distributed among objects and machines. The language should also hide from the user the functioning of the underlying support system and its registration tables. Registration of objects is automatic and based on activity affiliation. This contrasts with our previous registration schemes where processes explicitly expressed interest in each other and notifications for events that affected an entire task might have to work their way through a chain of dying processes [Lantz80].

The notion of activity is represented in part by the incorporation of an *activity tag* in messages and data structures. Each object "knows" a set of activity tags that includes the tag of the activity within which the object was created. These tags are the primary handle available to application code for exploiting the activity structure. If one could take a snapshot of an activity-based distributed system, the activities in existence could be discerned by scanning objects for activity tags. One of the uses of tags is to separate the internal data structures that a shared server process keeps for each activity it participates in. When an activity ends, tagged data allows the server to selectively clean up its state. Messages also carry an activity tag thus making selective suspension, tracing, and debugging easier to implement.

Conceptually, the activity structure is a tree of activities. A subactivity can be created whenever there is a subproblem which has different requirements with regard to failure handling, access to data, etc. A parent activity can respond to events affecting its children through a mix of automatic and user-defined actions. An important aspect of our model is the list of properties defined on activity-subactivity relationships and on (sub)activity-object relationships. The kinds of events that can be handled correspond to the properties assigned.

One aspect of activity-subactivity behavior can be characterized in terms of atomicity and inheritance (primarily, lock compatibility). In particular, activities may be *nonatomic*, *atomic*, or *subatomic*. It is anticipated that nonatomic activities will be appropriate for many applications such as on-going tasks. Atomic and subatomic are categories for activities whose effects should appear indivisibly. The offspring of an atomic (sub)activity must be classified as subatomic. The term "atomic" is somewhat misleading since the conventional meanings apply only when default event handling is relied upon. The general description of this nesting relationship is that the highest atomic ancestor of a subatomic subactivity is responsible for its actual commit. The default says this takes place when the top-level atomic activity itself commits. This default relationship is similar to the nested atomic actions of Liskov with respect to visibility and the true commit point. However, the user should be able to supply alternative actions that may lead the top-level atomic activity to prematurely commit some of its subactivities. In that case, the changes made by those subactivities become permanent regardless of a subsequent abort by their "atomic" ancestor.

There are also properties describing activity-object relationships attached to the objects involved. All objects participating in an atomic activity must have the atomic property; that is, the operations *commit activity* and *abort activity* are defined for the object so that it is able to respond to such requests. Atomic objects are provided with mechanisms for managing versions of selected data structures and default actions for synchronization and recoverability. Again we allow the designer to specify alternative actions (e.g., locking protocols tailored to the application) at the risk of not fitting the all-or-none semantics implied by the atomic property. Thus this property actually signifies conservative defaults and obedience of a particular protocol. Another property describing activity-object behavior is whether an object can be shared by more than one activity. Sharable objects must be able to respond to termination, suspension and resumption events for any activity that has arranged to share it. Nonsharable objects are restricted to participating in only one activity during their lifetime.

As an example of using this model to organize a distributed computation, consider the problem of providing a database system as a service for user applications. It might be structured in the following way: All of the modules involved with providing this service belong to (some level of) an on-going database activity. One of these modules may be a B-tree, offering search and update operations. This object would be sharable (by various user activities) and atomic (i.e. usable from within an atomic activity). Since it is desirable that an update operation have the "all or none" feature, an atomic subactivity would be created for the task of modifying the tree. In addition, to ensure repeatable reads from within an atomic activity, each search operation should also create an atomic subactivity (subatomic if the parent activity is itself atomic) for accessing the desired element. It may also be desirable to allow a reasonable degree of concurrency, possibly by exploiting specialized locking protocols as in [Ellis80]. This flexibility is possible within the framework of the model. The atomic property on the B-tree object says that it must respond to commit and abort activity events with the all-or-none semantics assumed, but it puts

no restrictions on the designer's internal implementation (such as requiring a two-phase locking protocol).

Implementation of the activity-object model assumes a division of labor among a number of components. The management functions can be classified along two dimensions: First, there is the distinction between what can be done automatically, independent of the particular problem being solved, and what requires application-specific policies. Previous work by others has emphasized the automatic without providing mechanisms for incorporating user-defined management. Second, there is the duality between activity management and object management. This leads naturally to a structure in which there are four kinds of components: user-provided processes for activity and object manipulation (*activity controllers* executing *controlling modules* and *objects* executing *process modules*, respectively) and system processes (*activity coordinators* and *object managers*). The user-defined modules must be able to respond effectively to a range of commands and notifications such as *suspend activity* or object death (with appropriate default actions supplied by the programming system). The underlying system processes must monitor the changing status of the activities or objects registered with them and react appropriately (e.g., sending notifications, modifying registration tables, creating or destroying processes). Specifically, the job of the activity coordinator is to keep track of the activity hierarchy, the status of each member (e.g., suspended, committed), and which objects are participating in each (sub)activity. It must service requests to change the activity structure (e.g., creating a subactivity), to register new objects within existing activities and to notify all the objects involved when activity-related events occur. The object manager is responsible for detecting the failure of managed objects and notifying affected activities (through their activity controller processes). This implies the need for a data structure that allows a mapping from objects to interested activity controllers and a specification of the information they expect to receive when the object dies. The object manager gets involved in the creation, destruction, crash recovery, and location of objects so that the appropriate registrations get made. All of this assumed structure has been specified in detail for an implementation based on UNIX-IPC [Rashid80] and prototypes for some parts have been completed [Hrechanyk81]. Of course, one would like to hide as much of this as possible from the user with appropriately chosen language constructs.

3. High Level Language and System Support

Given the assumption of cooperating proceses for activity and object management, the language requirements for the activity-object model become fairly simple. The focus of the language design is on constructs for specifying the application - dependent control actions. The basic requirements are tagged variables (and structures) and a clean interface into the underlying system. Our approach is to propose constructs that could be added to an existing language and would encourage users to think in terms of activities. The base language for this project is PLITS which itself is a set of constructs incorporated into various sequential languages. The examples later in this paper are written in a MESA-PLITS [Mitchell79] pseudocode with a preliminary set of activity - related extensions.

A tagged variable allows access to the desired data by using the activity tag as an index. The simplest use of tagged structures is to make the code of shared processes easier to read and write. Any data that is needed for every activity currently being served can be declared as a tagged variable and then individual instances can be allocated and deallocated as activities come and go. A construct for establishing the activity context of a section of code (e.g. WITH *activity tag* DO) can be used to suppress the explicit tag in variable references and message construction. The example developed in the next section offers an illustration of the convenience of tagged variables. Consider a process that represents a thermometer object whose temperature readings are used for various purposes (encoded as different activites). Since each activity may be interested in a different subset of readings (e.g. at regularly spaced time intervals or when the value lies in a "danger zone" of too hot or too cold), the shared process keeps a tagged record per activity describing the temperature readings of interest:

```
Conditions: TAGGED RECORD
            [upperbound, lowerbound: temperature,
             lastreport: time,
             delta t: INTEGER];
```

An extremely useful tagged variable for sharable objects would be

```
status: TAGGED {alive, suspended, aborted,
        committed}.
```

The simplest response in a shared process to an activity command would update the appropriate component of status. One can imagine high level contructs based on status such as

```
FOR EACH t: TAG SUCH THAT status[t] = alive DO
```

which would execute the following statement for all known tags whose status was alive or

```
RECEIVE-ACTIVE
```

which could ignore messages if the status of the activity tag carried with the message was not alive. *Locked* and *atomic* variables add more capabilities. For a locked data item, a lock and a waitqueue of tagged messages would automatically be provided along with primitives to set and release the locks. Similarly, atomic variables would provide the ability to do tentative-write, undo (abort), and update (commit) without the ability to explicitly unlock.

The extended control features were a major motivation for adding the notion of activity to the conceptual apparatus of PLITS. One would like a natural way of expressing the control dependencies of a coherent distributed task and invoking actions that translate into network-wide realizations of primitives like suspend, trace, or commit. Much of the required structure can be specified declaratively, but some explicit code by the user appears to be required.

The user-defined entities in the activity model for activities and objects are presented by two types of code modules, CONTROLLING MODULE and PROCESS MODULE, respectively. Processes controlling top-level activities, subactivities, and their remote agents execute CONTROLLING MODULES. The CONTROLLING MODULE supplies the information needed to start up the activity and to respond to subsequent events. The heading provides a name and properties such as whether the activity is to be ATOMIC or NONATOMIC. Each managed object initially participating in the activity can be declared with its type, any type-dependent arguments and associated notices for death or other events. Sharable objects that are created within this activity may be given an asserted name. Additional objects may be dynamically introduced into the activity. The local declarations must include procedure definitions for each distinct object notice. The body of the module should be limited to application-specific management functions and set up code such as

START *object* ON *codemodule* (*instantiation parameters*)

which starts up an object that has been created and registered in the activity associated with this CONTROLLING MODULE or

OPEN *activity tag* TO *asserted name* | *object id*

which locates a shared object, notifies that object about this activity, and completes the necessary registrations.

PROCESS MODULES are essentially extensions of PLITS modules. In particular, interprocess communication is by message-passing. The heading specifies the properties associated with the process executing the code: ATOMIC or NONATOMIC and SHARABLE or PRIVATE. The syntactic structure consists of the definition of public slotnames, declaration of local variables (the tagged ones are of interest here), the exception handlers for activity events, and finally the code body.

The activity structure has a major influence on the exception handling strategy. Some of the activity related events that objects must respond to are "normal" rather than error conditions which should be dealt with as soon as possible and then execution of the process should resume where it was interrupted. In the current version of the language design, the asynchronous arrival of an activity-related message can raise an exception only at well-defined clean points. These can be characterized as calls of IPC or activity primitives excluding those which occur within designated critical regions such as the body of a notice catcher or the block of code after an explicit setting of activity context. Other approaches are being investigated.

4. An Example: Controlling a Solar Greenhouse

The application chosen to illustrate these ideas is that of monitoring and controlling an experimental solar greenhouse. Imagine that this is to be an ambitious effort in which all of the following demands have been made: The greenhouse is to be fully automated with a high degree of reliability. Data is to be collected for evaluating its heating capabilities. It is to serve as a testbed for various policies and for the use of AI techniques to manage heat storage, with an additional constraint that the greenhouse location cannot house a large general purpose computer. These diverse requirements point to a heterogeneous network environment. A physical configuration that matches the needs of the problem with appropriate hardware might involve a number of devices and sensors in the greenhouse (e.g. motors to open and close vents and insulating shades, fans, thermometers, pyranometers with simple interfaces assumed) and a user's terminal, all driven by a personal computer that is connected via a network to a file server/database computer where historical weather data for the region, current weather forecasts, and recorded measurements for the greenhouse are stored and to a "compute engine" for statistical analysis and generation of plans for achieving the heating goals. The files are available from a multiplexed server that is not designed by the user and is widely shared. The user views the problem in the following way: Operating the greenhouse is an on-going task that should continue even if various devices, computers, or communication links fail. Between the worst case (i.e. fully manual operation because the controlling computer is down) and the normal case, there should be a whole range of partially degraded performance. The vent and shade devices bring about external "irrevokable" actions (opening a vent loses hot air so that merely closing it again does not restore the original state) with time constraints that are important. Recording of measurements taken within the greenhouse is an example of file update that need not be part of an atomic action since an unfinished recording session is still valuable. The planning process must be brought up on the remote machine from the controlling site in the greenhouse and the file server must be located. Database manipulations by the planner may involve atomic transactions. It should be possible for the user to temporarily suspend the planning or the recording operations and take direct control of the greenhouse or to trace the interactions leading to a plan.

This problem embodies many of the important issues affecting distributed applications that activities are meant to resolve. To demonstrate the power of the activity-object model as a conceptual tool, we first describe the structuring of a solution in prose and later give samples of code modules that show how to express it in a programming system.

The first step is to identify the essential tasks and to understand what dependency relationships exist among them. Evidently, the most basic task is that of running the greenhouse continuously (i.e. issuing commands to operate vents, fans, shades, and so on). All of the other goals depend on this one function; therefore, this task is a candidate for the top-level activity. Because of its on-going nature and sensitivity to external changes, it is classified as nonatomic. The activity should continue as long as the greenhouse computer is up regardless of other failures in the network. It should also survive the breakdown of individual devices inside the greenhouse. A controlling module representing this activity executes on the local machine. It responds to the death of nonsessential devices by telling specific processes to revise their model of the greenhouse. This is indicated in the form of user-defined object notices (the default would have been to terminate the activity). In the particular network configuration assumed, a crash of the local computer leads to termination of the activity. Since manual operation of the greenhouse is required during the downtime and very different external conditions will likely be present when the computer eventually restarts, it is reasonable that this controlling module be neither replicated nor recoverable. The task of storing sensor measurements is not necessary for operating of the greenhouse and it may be desirable to exercise control over it separately; it could be a nonatomic subactivity. The planning function is independent of the recording subactivity and supplementary to the basic operation. It may be set up as a parallel second-level subactivity. Both of these subactivities have controlling modules with object notices that enable them to adapt to certain object failures. The files involved in the recording and planning subactivities belong to a distinct system activity that provides a file service. Thus, the solution can be organized as an activity with two subactivities that share objects of a distinct file service activity.

The actual work in an activity-based system is still being done by the objects much as it would be in a PLITS solution. Each physical device is controlled by a process which is the managed object representing that device in the activity world. There are two classes of devices in this example: sensors (e.g. thermometers) and motors (e.g. vents). Consider first a thermometer object as typical of the sensor processes. It has the activity-related properties of being sharable and nonatomic. Sensors participate in the top-level activity (where they were created) and in both subactivities. The thermometer object periodically sends temperature readings to the user process (associated with the top-level activity), to a recording process to be formatted and written on a file and to a planning process for use in determining future motor commands. Tagged data structures are used to separate the information devoted to one of the activities sharing the thermometer from that of another. We have already discussed the Conditions record. The nonatomic property indicates that this object does not understand the commit protocol or require the extra mechanisms for recovery. In general, the properties determine the set of activity notices (primarily messages from the activity coordinator) to which the object is expected to respond. For example, the *suspend activity* command can be handled in the thermometer by updating the tagged status variable and refraining from sending temperature readings to processes working on behalf of the suspended activity. A vent object is typical of motor device controllers. It too is sharable and nonatomic. The vent's job is to generate the proper signals to open or close the physical vent as specified by commands received from the user or the planner. It participates both in the operating activity and the planning activity. In response to the same *suspend activity* notice, the vent might update status and ignore any commands arriving with the suspended activity tag. Other objects needed in the solution include the recorder, the planner, the user process, and various file objects. The recorder is a private nonatomic participant in the recording subactivity. It receives messages containing sensor readings from all of the live sensor objects and constructs a record which it sends to the file server. The file server is a shared object owned by a system activity and participating in potentially many user activities. It should be capable of taking part in atomic activities (although the greenhouse example does not demand it) and so would have the atomic property. The planner is a private nonatomic member of the planning subactivity that communicates with sensors, some database servers, and the motors. Finally the user process, involved in the top-level activity, might serve as a command interpreter allowing the user to issue commands to the activity subsystem and to the motor devices.

Figure 1 summarizes the activities and some of the objects that might be set up. The top-level activity, denoted by Operating, deals with the basic operation of the greenhouse. It involves the processes (drawn as boxes) that control the sensors, the motors, and the user terminal. In addition, it spawns two subactivities, which are drawn as ovals and labelled Recording and Planning. Recording is concerned with data collection function. It shares the sensor processes with Operating and a file object with a separate server activity. Finally, the recorder process only participates in this subactivity and is therefore a private object. Planning deals with automatic operation of the greenhouse based on a planning process. This subactivity shares the sensors and the motors with Operating and a database object with the server activity.

Given this outline of a solution formulated using activities, one can now follow a concrete example of the management facilities. Suppose the user process died in such a way that it was detected by an object manager in the system. The object manager would send a notification to the appropriate controlling module where the user may have specified how to handle the situation (e.g. substituting a different process). In this case however, the default applies and the activity coordinator terminates the activity by recursively terminating both subactivities and sending an activity notice to each object registered in the top-level activity.

The greenhouse example could be further elaborated upon to illustrate atomicity. Suppose that each request to write a record sent by the recorder to the file server requires it to perform an atomic action. The shared server could accomplish this by creating an atomic subactivity within the context of the recording subactivity. Since the recording subactivity is nonatomic, the new subactivity is registered and managed by the activity coordinator as atomic rather than subatomic.

Figures 2 and 3 show representative components of the solution translated into our proposed activity-based language constructs. Figure 2 shows an application specific controlling module that could be used to establish the activity structure and respond to object failures. Figure 3 gives a sample of a process-module.

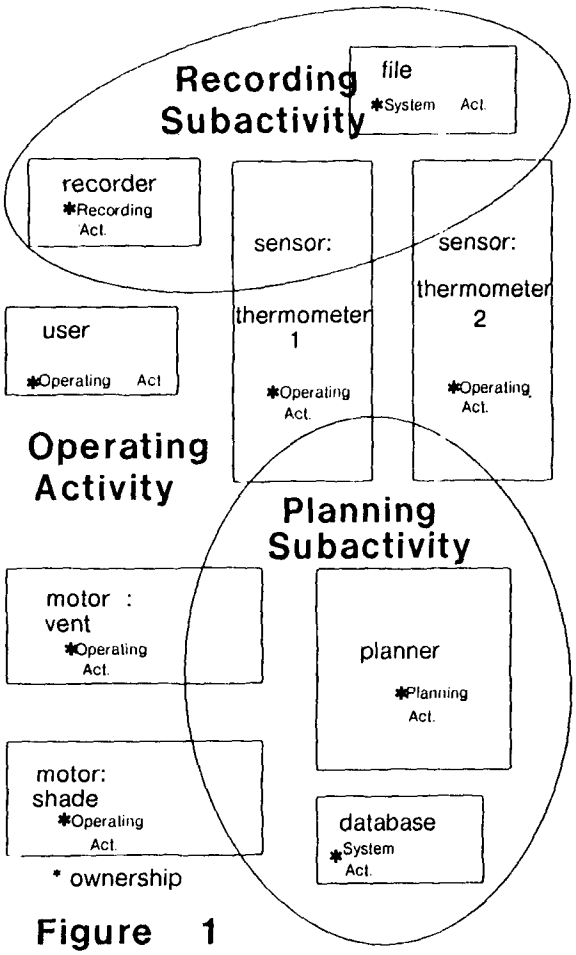

Figure 1

5. Summary

In this paper, we have proposed a new conceptual tool for organizing distributed software called an 'activity.' The structuring achieved by activities is orthogonal to that of objects. The model includes a set of properties that characterize the relationships among activities, their subactivities, and objects. Mechanisms for managing distributed tasks can be based on this notion of an activity hierarchy. We have outlined our current design for a programming system and a preliminary set of high level language constructs. Our approach differs from similar research efforts by providing additional flexibility for application-specific failure handling and extended control features. Finally, we have presented an example of what we consider a typical application.

Acknowledgements

This work was supported in part by NSF grant No. IST-8025761 and in part by NSF Grant No. MCS-8104008.

6. Bibliography

[Ball76] J.E. Ball, J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner,
"RIG: Rochester's Intelligent Gateway System Overview,"
IEEE Transactions on Software Engineering, vol. 2, No. 4, December 1976, 321-328.

[Brinch Hansen78] P. Brinch Hansen,
"Distributed Processes: A Concurrent Programming Concept,"
Communications of the ACM, 21, 11, November 1978, 934-941.

```
Greenhouse: CONTROLLING MODULE
      NONATOMIC =

PUBLIC
  -- The PLITS meaning of this keyword is intended
  type: DeviceName;
  which: Directions;

MANAGED
  user: PROCESS
    DEATH NOTICE Kill;

  ThermometerModule: TYPE = PROCESS
    TAGGED [ location: Directions ]
    DEATH NOTICE Reconfigure;

  thermometerSet: SET OF ThermometerModule;

  VentModule: TYPE = PROCESS
    TAGGED [ location: Directions ]
    DEATH NOTICE Reconfigure;
      -- Notice same death notice for
      -- therm's and vents

  ventSet: SET OF VentModule;

  recorderActivity: ACTIVITY;
    -- Death notice is left to be the default
  plannerActivity: ACTIVITY;
    -- Death notice is left to be the default

VAR
  p: PROCESS;

OBJECT NOTICES
  BEGIN
  Reconfigure (device: PROCESS) =>
    VAR
      msg: MESSAGE;
      a: ACTIVITY;
      i: INTEGER;
    BEGIN
    -- Construct a reconfiguration message and
    -- send it to intersted parties

    IF device IN thermometerSet
      THEN msg ← (type~thermometer),
             (which~thermometerSet.location[device])
        ELSE msg ← (type~vent),
             (which~ventSet.location[device])

      SEND msg TO user;
      SEND msg TO plannerActivity

      END;
    Kill (p: PROCESS) =>
      BEGIN
      notify vent motors to go into manual mode
      TERMINATE plannerActivity
      END;

    END;

BEGIN

  START PROCESS user ON UserTerm ( . . . );
  p ← NEW[ThermometerModule];
  PUT p IN thermometerSet;
  thermometerSet.location[p] ← south;
  START PROCESS p
    ON ThermometerCode ( HW-Address );
  ... and so on for other thermometers

  p ← NEW[VentModule];
  PUT p IN ventSet;
  ventSet.location[p] ← south;
  START PROCESS p
    ON VentCode ( HW-Address );
  ... and so on for other vents

  -- Activate the subactivities for the recorder and planner
  START ACTIVITY recorderActivity
    ON Recording ( themometerSet );
  START ACTIVITY plannerActivity
    ON Planning ( . . . )
END.
```

Figure 2

[Cook80] R. Cook
"*Mod-A Language for Distributed Programming,"
IEEE Trans. on Software Engineering, vol. SE-6, No. 6, November 1980, 563-571.

[Ellis80] C. Ellis,
"Concurrent Search & Insertion in 2-3 Trees,"
Acta Informatica, 14, 1980, 63-86.

[Feldman79] J.A. Feldman,
"High Level Programming for Distributed Computing,"
Communication of the ACM, 22, 6, June 1979, 353-368.

[Gray79] J. Gray
"Notes on Data Base Operating Systems,"
in *Operating Systems, An Advanced Course*, Springer Verlag, 1979.

[Hoare78] C.A.R. Hoare,
"Communicating Sequential Processes,"
Communications of the ACM, 21, 8, August 1978, 666-677.

[Jones79] A.K. Jones,
"The Object Model: A Conceptual Tool for Structuring Software,"
in *Operating Systems, an Advanced Course*, Springer Verlag, 1979.

```
ThermometerCode: PROCESS MODULE
    (line: HWAddress) SHARABLE NONATOMIC =

PUBLIC
  -- a slot for an activity name is automatically made public
  wantsData: PROCESS;

VAR
  sensormsg: message;
  conditions: TAGGED RECORD
      [ fields describing readings of interest ];
  status : TAGGED {alive, suspended};
  dataCollector: TAGGED PROCESS;
 tag: ACTIVITY TAG;

ACTIVITY NOTICES
  BEGIN
  OPEN (msg -- contains the "open activity" message ) =>
    BEGIN
    -- New instances of tagged types are automatically generated
    tag ← msg.activity;
    conditions[tag] ←
      [fields are filled from slots in msg];
    status[tag] ← alive;
    dataCollector[tag] ← msg.wantsData;
    REPLY (givesData~SELF) TO msg.wantsData
    END;
  TERMINATE (msg) =>
    BEGIN
    -- Instances of tagged types are automatically destroyed
    END;
  . . .
  END;

BEGIN

DO
  take temperature reading &
      construct sensormsg
  FOR EACH t: TAG
      SUCH THAT status[t] = alive DO
        IF conditions in conditions[t] are met
          THEN
            SEND sensormsg TO dataCollector[t]
  ENDLOOP
ENDLOOP

END.
```

Figure 3

[Lampson80] B.W. Lampson,
"Atomic Transactions,"
in *Lecture Notes for Advanced Course on Distributed Systems - Architecture and Implementation*, Institut fur Informatic Technische, Universitat Munchen, Munich, Germany, March 1980.

[Lantz80] K.A. Lantz,
"Uniform Interfaces for Distributed Systems,"
TR63, Computer Science Dept., University of Rochester, May 1980.

[Liskov79] B.H. Liskov,
"Primitives for Distributed Computing,"
Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, 30-42.

[Liskov81] B.H. Liskov,
"On Linguistic Support for Distributed Programs,"
Proceedings, Symp. on Reliability in Distr. Software & Database Systems, July 1981, 53-60.

[Liskov82] B.H. Liskov and R. Scheifler,
"Guardians and Actions: Linguistic Support for Robust Distributed Programs,"
Proceedings, Symp. on Principles of Programming Languages, Jan 1982, 7-19.

[Low80] J.R. Low,
"Name-Type-Value (NTV) Protocol Draft Proposal,"
TR73, Computer Science Dept, University of Rochester, July 1980.

[Moss81] J.E.B. Moss,
"Nested Transactions: An Approach to Reliable Distributed Computing,"
Ph.D. thesis, MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, Mass., 1981.

[Randell75] B. Randell,
"System Structure for Software Fault Tolerance,"
IEEE Transactions on Software Engineering, 1, 2, June 1975, 220-232.

[Rashid80] R.F. Rashid,
"An Inter-Process Communication Facility for UNIX,"
TR CMU-CS-80-124, Dept. of Computer Science, Carnegie-Mellon University, March 1980.

[Shrivastava78] S.K. Shrivastava and J.P. Banatre,
"Reliable Resource Allocation Between Unreliable Processes,"
IEEE Transactions on Software Engineering, 4, 3, May 1978, 230-240.

[Shrivastava81] S.K. Shrivastava,
"Structuring Distributed Systems for Recoverability and Crash Resistance,"
IEEE Trans. on Software Engineering, vol. SE-7, 4, July 1981, 436-447.

[Wulf81] W.A. Wulf, R. Levin, and S.P. Harbison,
HYDRA/C.MMP: An Experimental Computer System,
McGraw-Hill, 1981.