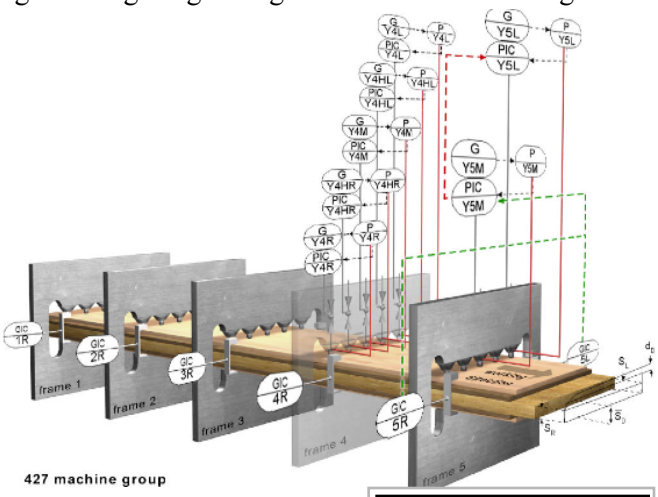
ICL Extensions for Distributed and Duplicated Unit Controls 7/18/02

This note summarizes proposals for ICL-programmed, array-like, duplicated controls for duplicated staged process units and elements. These programmed controls are intended to be supported together in a single computer or distributed among several computers, in either case, hierarchically arranged under a single plant-wide ICL Operation. In general, ICL has always been able to accommodate these capabilities; but the extensions will make programming many-stage applications more natural. The note starts with a motivating example and background discussion addressing related standard language mechanisms. It then develops comparative discussions of the corresponding ICL developments. It further details these ICL mechanisms, as previously specified, suggesting their limitations for the above goals, and proposing extensions. The concluding section summarizes each proposal in terms of its intended overall effect. Appendices detail the Activities and Tasks, as previously specified, provide several summary notes, and include a Small System Language listing example.

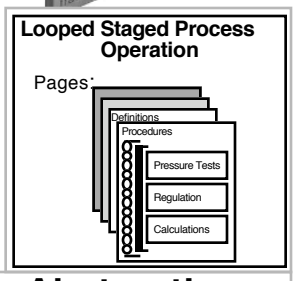
Caveats: The note discusses detailed issues designed to give ICL the long range integrated capability. Like the other specification documents, it is not a novice oriented promotion of ICL. The initial ICL users will all be novices or specialists in some dimension of conventional control practice, of the sort that ICL is trying to integrate. ICL will initially have to be taught in different ways from each of the resulting half a dozen simplified, specialist points of view, while still advocating the long-range integration. Even at this stage users are likely to find the more general ICL applications to be at least readable even if harder to program. Once a user has learned how to work the controls with which he is most familiar, he can expand his background to understand the larger view.



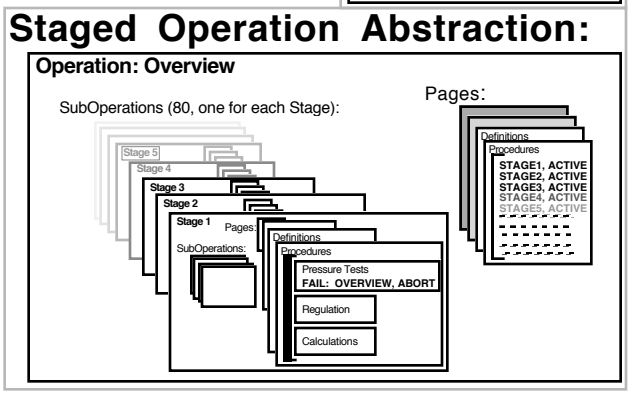
Motivating Problem

The motivating process example (which I will not pretend to fully understand) consists of eighty, identical, identically controlled stages. Roughly the controls for each stage include:

- Process pressure tests, failing which the whole application is shut down.
- Continuous Controls including:
  1. Conventional Regulatory Control.
  2. Included Supporting Continuous Statement Calculations.



As originally conceived the application was described in terms of an ICL-style looping based on process variable arrays and Idiom controls operating on those arrays.[1] ICL more naturally treats such a system as a set of (in this case eighty) identical Object-implementing Operations. Given that ICL already addresses this kind of application, the note will develop improved ways (particularly for many-stage applications) of supporting both points of view.



Such an application is most naturally implemented as a distributed system. In the large language, each identical Operation would naturally occur in its own machine, with all operating under a separate plant-wide Operation. While a simpler distribution results if each identical process stage is represented by its grouped Super-Variable, placed and controlled in an ICL Small System Language machine, the note will focus on the more elaborate Large System Language. There are several issues to be considered at some point.

- Expanding Lists to cover more array like behavior ap-

[1] The "Notes on the Original Motivating Problem Statement" in Appendix II addresses details of nesting Continuous Activities inside the Looping Activities, as a general example of Activity Bracket syntax. Should the Brackets be reversed?

plied more generally to all ICL Variable and Object (including Operation) structures.
- Expanding the role of the Looping Activity to address Control Idiom Structures and Distributed Controls.
- Supporting flexible specification of different distribution configurations of already programmed applications (allowing the same application to be reused in several differently distributed installations). The specification includes coverage of flexible messaging.
- Defining the programing and configuration mechanisms for operating Small System and Large System Languages together including the programming of duplicated copies of Small System Language systems.
- Supporting more flexible and appropriate process nomenclature for the duplicated stages. As proposed, the example depends on arraying the process variables. This will work under the array based proposals. But the Distributed Stage Operations undesirably restrict the user's process variable nomenclature. In this case, the Operations would probably be indexed within the notation and the variable names would be duplicated within each Operation (e.g.: STAGE.1.PYM, STAGE.2.PYM, etc. or STAGE[1].PYM, STAGE[2].PYM etc.). Functionally, and in language, these usages are effective but users might find these forms operationally unnatural. ICL already has a number of ways, including duplicated naming, for accommodating such a concern.

Background

The background discussion covers first the limits of traditional general computer language mechanisms, then relating these to the ICL advances and to the special requirements of duplicated and distributed controls. The material is drawn from throughout the prior specifications. The discussion depends on a number of ICL terms, outlined here but developed elsewhere as indicated. The note will focus on the function of the different ICL elements illustrating syntax but not defining it. Among the specific ICL terms referenced in the note:

- **Operations** are the collected Objects representing controls of process elements and the overall ICL entities. All ICL program elements are included in some Operation. The note develops the basic structure of the Operation without discussion of syntax.
- **Pages and subOperations** are the direct components of any Operation. Except for the subOperations, the (Operation, Definitions, Procedures, Details, Parameters, Summary, Idioms, Comments, Simulations) Pages are the separately listed expressions of the different kinds of program function, each Page formatted to best represent its type of program function.
- **Tasks and Activities** are the thread like bases of ICL procedural programming, related to the distributed control discussion and detailed in Appendix I.
- **Task and Operation Calls** correspond to traditional subroutine calls but take a number of forms addressing reuse of standard or process specific function. The note will discuss the uses and roles of Calls without addressing their implementation or syntax.
- **SuperVariables** are the ICL vehicle for defining variables as free form lists of Attributes supporting straightforward operational and programming tabular display under common sets of table headings.
- **Idioms,** defined elsewhere, are basic to ICL as the main operator form for representing regulatory controls in statements, either as built in or as user defined.
- **Blocks** represent a secondary regulatory control vehicle tying Idioms to traditional controls, and parameterizing either. The note discusses Block Calls implementing Block controls without detail definition.
- **Recipes** represent different Task modeled product and production strategies applicable to an established process. The associated Call mechanism must support different ingredient lists and equipment/Task assignments. The note discusses Recipe Call uses and roles without addressing implementation or syntax.
- **Footnotes** allow flexible computations in the form of computing statements in any ICL context.
- **The Small System (vs. the Large System) Language** is designed to represent small sensor/actuator dominated systems, based on a single listing compound (multivariable) SuperVariable, including embedded controls, rather than multiple multi-Page Operations. The simplified listing replaces the separately listed Pages by included Paragraphs as illustrated in the Appendix III figure.

Prior Language Conventions for Procedural Modules and Duplicated Structure

The traditional computer language program represents a sequence of computing steps operating on defined data structures and values. Whether those steps correspond consecutively to consecutive language symbols, or to some more elaborate mapping from those symbols, the computational execution is sequential. Such sequencing can include repeated (looping) execution of subsequences of the symbols, jumping between subsequences, and user definition of symbols to represent subsequences executable elsewhere in the overall sequence.

The original subroutine concept allowed the definition of a procedure to be executed from a simple call. But the calls were sequentially invoked and sequentially interpreted or executed before return to the

calling program. Such a mechanism requires nothing more than the existing control of the sequencing, except that it needs a general mechanism for storing the return address to the calling program and managing any call argument list. The storage of this information must be compatible with nesting of calls but does not need to support more than one call to a given subroutine at a time. Later developments allowed the subroutine to have its own internal variables, active only for the duration of the call.

All of this was adequate for a single sequential main program. Interrupt processing and time sharing with standard libraries finally required that several (reentrant) calls to the same library subroutine might be in the middle of their execution at the same time. Such a situation requires separate storage of the subroutine state data for each time shared main program. Over time there have been applications which were controlled by several parallel time shared programs, either independently or in coordination. There have also been (academic and commercial) languages which were able to represent such an application within a single language system. Generally these used a computational model based on activating separate processes or procedures in parallel.[2] This model is appropriate for simply implementing time related activities not intended to correspond to any particular real world perspective. In any case, when several subroutine calls to the same subroutine are simultaneously active all related temporary results must be stored in duplicate.

As a separate issue, traditional process control systems include PLCs and block based control data bases which implement active parallel continuous controls. Historically these gave rise to confusion as to whether a block was a subroutine, or an algorithm, or what. Conceptually a control block is a special kind of call on the control algorithm. ICL formalizes this concept with its Block Task Call.

The earliest mechanism for creating duplicated language elements was the array. More recently, data structures and Objects were defined as ways of expressing particular structures (originally of data and then including procedural dimensions). Both data structures and Objects can be conventionally arrayed. More specialized languages have provided Sets and Lists which can be made to behave as arrays but are usually related to different kinds of Set operations. Separately, other conventional uses of lists, not in a specialized language data type sense, but as argument lists and other syntactic entities, occur in many languages. The integration of list roles may be unique to ICL.

Some of the ICL usages anticipate issues addressed in more recent languages: The replacement of traditional indices with pointers integrated to the Lists anticipates the concerns of JAVA with pointers and the use of more restrictive forms (in JAVA's case: handles). The **NEXT** and **PREV** looping iteration commands directly anticipate similar JAVA usages. The use of selecting indices in a dot notation similar to data structure references is now being used in web application languages. The use of Lists instead of arrays corresponds to usages originating in LISP; it has not yet been used by another application language. The integration of the different roles of data entities, Objects, and syntactic lists is implicit in languages like LISP but less explicit there and not occurring in other languages.

The Relation of the ICL Activities, Tasks, and Operations to Prior Art

The ICL specification data is drawn from the Large ICL Specification document. ICL provides a notation for representing the several kinds of sequenced and parallel execution needed in process control applications, within a single integrated application program. In ICL, this parallelism is designed to model and extend traditionally parallel controls. There are three levels of program grouping: Activities, Tasks, and Operations. Appendix I details prior Activity/Task specification, with improved formal properties.

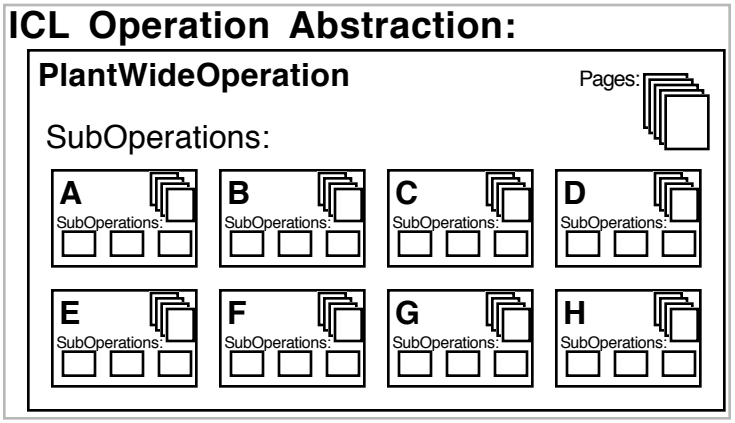
The Activity allows grouped statements and subActivities to be executed in sequence, in parallel, or in several other relative orders each involving the possibility of carrying on several of the included elements at the same time. Tasks are like Activities except that they are named and can be Called[3] by name; neither Activities nor Tasks have their own internal data for variables.[4] Operations represent controlled process units or elements with variables and data and with any included Tasks (and Activities). An Operation

[2] A more modern model considers a process as made up of simpler sequenced and parallel invoked “threads”.

[3] A Task (or Operation) Call acts as if the Call had been replaced by the corresponding Activity including any associated (usually temporary) suspension of the Calling Activity.

[4] In this respect, Activities are like threads. They are implemented as listed pcodes, but come in five types, supporting richer variations in internal sequenced or parallel execution.

may include an unnamed Activity on its Procedures Page, which can be called as a Call to the Operation, acting like a Task Call.



The Operation is the major ICL entity, containing all other entities, hierarchically grouped, both organizationally into (similarly structured) subOperations, and functionally into Pages of several types. Thus every Activity or Task is defined as part of some Operation's Procedures Page programming, and every process variable is defined as a SuperVariable on some Operation's Definitions Page.

When Tasks are Called they operate on the data (variables and values) in their argument lists and in including Operations (or other referenced Operations). Since the Calls may be made from parallel statements in other Tasks or Activities, nothing in the system prevents several Calls to the same Task from operating at the same time (even if with different argument lists). In addition, Tasks, unlike subroutines may continue indefinitely in real time. Improperly duplicated Calls can thus give rise to conflicting data accesses.

For this reason, a general Task is most simply written to be run only once to completion at a time (not in parallel with itself) to prevent conflicting simultaneous computations on variables referenced by it. When Tasks represent standardized control actions for their Operation's process unit this constraint follows naturally. In those cases where the duplicated execution is intended, a Task must usually be designed to operate only on its argument list variables. ICL includes a number of such special Task Call/Argument List mechanisms to support the execution of control Blocks, Recipes, and related control application needs. (Blocks model traditional standardized control modules and Recipes specify production strategies and variants.).

The Task also provides for an argument list key word **ALL** which creates an argument list with space for separate data for all of the Task's unaccounted for variable references[5]. The **ALL** allows general reentrancy. Otherwise, only careful use of multiple simultaneous executions of a given Task on its Operation variables should occur.

As indicated before, Operations can also be Called as Tasks. In addition, the Operation (or Task) can be activated by directly changing its operating states. But it particularly does not make sense to activate an Operation more than once simultaneously because its associated data is unique to the single execution process, and to the normal representation of the control state for a single process unit or element. Containing all the process data as well as controls, the Operation is even more definitely associated with a particular process element than the Task. However the Operation is designed to permit duplicated copies of itself (including its data), either under the same or different name, to allow the representation of controls of distinct similar process elements.[6] In this one sense, Operations support reentrant operation. Such a control system is treated as an Object with each separate process element being controlled by a separate Object instance, which includes its corresponding separate variables and parameters.

There are two distinct situations calling for such a representation: It may represent controls of several, similar, permanently installed and controlled, process units, usually under continuous process control. In this case, each copy (Instance) Operation is modeled (with a new Name) after an existing Operation

[5] Each referenced variable is defined by a Definitions Page SuperVariable (as are all argument variables) even when the working data is stored as a distinct instance value outside the SuperVariable, as with **ALL** Argument Lists, Blocks, and Recipes. In other normal Calls, argument variables are accessed and stored in the Call argument list variable's SuperVariable. All other variables referenced in the Task body are stored in their normal SuperVariable value Attributes within the including Operation (on its Definition Page).

[6] Separately named copies can also be edited, subject to modeling constraints and reflecting the fact that similar real processes often have minor design differences.

(whose existing Name is included in parentheses after the new Name). Or it may represent temporary (batch) production trains, under distinct batch Recipes, where the temporary assignment of units within the train may be defined through Recipe Arguments. In this case, the Operation's definition includes the number of copies contained in brackets after the Operation name (and a colon) on the Operation Page. The associated Instances will be assigned automatically as needed to any new Recipe Call.

As elaborated elsewhere, with the exception of the use of array-dimension-like bracketed copy numbers in List or Operation definitions as above, ICL generally replaces arrays with Lists, which are not generally designed to represent duplicated entities. ICL variables are uniquely named but can be duplicated as similar, but distinctly named, SuperVariables, under a common set of Attribute Headings.

Elements in a List can be selected in a number of ways. The major strategy relies on a pointer to a currently selected list element, incorporated directly into the List. The List name followed by a dot and nothing else returns that selected element. The pointer is advanced by **NEXT** and **PREV** iteration commands, and set by command or by assignments between distinctly named references to the same List. This usage imposes a tight typed discipline on the pointers. A secondary related (indexing) notation consists of the List name followed by a dot (period) and number or parenthesized index variable. This alternative to the conventional bracketed indices is used in other new languages.

Declarative vs. Procedural Structures

Standard languages sharply distinguish between declarative programming and procedural programming. In this context, declarations describe static application or data structures, whereas procedural statements define active computations. ICL makes the same distinctions except that the static declarations tend to describe structures which may be static, in the sense of representing fixed structures, but associated with continuing control computations. For example, process variables are defined as SuperVariables on a Definitions Page, each representing not only the variable structure, but its value, supporting Attributes, and the active processing of the sensed data or output activation. Parameters Pages also are declarative, including control Blocks or tables which contain the parameters and data values for control Blocks or Idioms. In this case, the declared values will change over time from their original programmed value.

In ICL, the procedural statements, representing conventionally active computations and calls, normally occur on the Procedures Page, grouped in different Tasks and Activities. As an exception, the Footnote concept allows procedural statements to incorporate general computations anywhere, including within the declarative SuperVariables. In addition to declarative but dynamic elements, ICL also contains procedural statements such as Idiom and Block calls which create semi-static structures with Block controls and Parameter Page tables and make them simultaneously declarative.[7] Nevertheless, ICL maintains the declarative/procedural distinction based on the essential character of the program element structure. These distinctions affect the later discussion.

The Requirements for Duplicated Controls in Distributed Systems

The note addresses two issues: duplication and distribution of controls. Duplication is the process of representing more than one similar control structure based on a single initial unit design, ideally allowing the copies to be separately named or accessed indexed under the name of the initial unit. ICL is designed to support duplication in the spirit of instantiating Object Classes where the Object class is represented by the initially designed control Operation, and any later instances of duplicated controls are represented by copies of that Operation, either with a new or shared and indexed name. The SuperVariable allows a similar structural duplication of process variable structures as separate lines under a single set of table headings, but it fails to support shared names.

Traditional languages support duplication through type and data structure definitions and through arraying which can be applied to individual variables or full data structures. The ICL thinking on arrays originally considered arrays as they are normally applied to multivariable controls. This usage generally expresses a control structure, represented by a matrix, which is applied to one or more sets of variables, each set represented by a vector. While such an approach is natural mathematically, it requires the separately named process variables, each having a distinct operational role, to be arbitrarily renamed and bun-

[7] The Block Call, particularly, combines the role of declaration and messaging (calling) of a specialized Object-like form.

dled under the single vector name.

Instead, ICL took the view that such systems were better addressed by grouping the separately named process variables into Lists, temporary to the control purpose, operated on by control List operators (generally Idioms), replacing the matrices. While these Lists can also be used to represent multiple similar Operations, ICL did not initially include all of the structure for this purpose. In such an application it is very natural to think of the different similar controls as numbered instances of the basic stage control structure, with their component variables to follow a similar numbering strategy.

Duplication of similar units does not necessarily assume distribution, but the implied existence of a large number of distinct controlled units makes distribution of the controls likely. ICL applies distributed controls on two levels: On the Large Language level the principal (hierarchically topmost) plant or other Operation may be distributed by distributing any subOperations. On the other hand, the Small Language allows the separate distribution of named compound SuperVariables (representing several variables with separate controls) from some Operation into a separately controlling computer.

Only these two cases are presumed: Large System Language distribution of Operations and Small System Language distribution of SuperVariables. The distribution is restricted to independent, named entities, augmented by messaging which controls lower level functions. This permits a controlled relation between any Tasks in a distributed Operation and the associated Variables, Parameters, and data. Such a restriction is natural both operationally and computationally.

ICL has taken the position that the implementation of the different non-sequential executions be formalized predictably, when integrated in single computers, by their simplest, most natural sequenced approximation. For example, initial execution of consecutive Statements and Activities in a Parallel Activity, in a single machine and sample time, is to be sequential as listed (rather than in some random implementation determined order).[8] This permits the user to predict the actual computational behavior, and, in those cases where it is appropriate to take advantage of the forced order,[9] to do so.

When Operations are distributed, or where communication between distributed Operations requires parallelism, no forced sequencing is assumed. Operations distributed in distinct computers proceed independently, each in its own internal order or as constrained by the timing required by any communicating elements. ICL assumes a modern high speed microprocessor implementation minimizing need for fussy optimized implementation scheduling (in normal process control applications).

Recent discussion emphasizes that distribution should normally be specified separately from the actual Operation programming. This allows the same application design to be applied more easily in separate installations where different distributions may turn out to be desirable. Thus a separate distribution definition tool is needed which is able to take the application program and a separate set of distribution declarations, and impose them. In such a case, there may be different communication configurations made, corresponding to the different distribution choices. When this is done it will be important to ensure that no inappropriate usages[10] of the single machine default orderings have been made, and that any communication delays introduced by the distribution do not have inappropriate effects.

Detailed ICL Task, Operation, and List Structures:

As Currently Specified, with Limitations and Proposed Extensions

Generally ICL has been designed to accommodate repeated control structures reused in both continuous staged processes and batch Recipes. Distribution has also been accommodated. The proposed extensions simplify systems with large numbers of duplicated elements.

Activities, Tasks, and Task Calls

An Activity is represented by a list of statements (and nested Activities) grouped (on the left) by a bracket whose shape distinguishes Sequential, Continuous, Parallel, Looping, and State Driven Activities

[8] Appendix I defines this concept as Constrained Parallelism.

[9] For example, the predictable sequenced implementation in place of randomized parallelism can avoid or minimize race conditions common in parallel implemented logic and tasking.

[10] That is usages where distribution causes function dependent on the default sequencing to fail because the actual elements are distributed, no longer sequencing with the default order.

(defined and illustrated in Appendix I). A Task is a named Activity with argument lists and other associated connection mechanisms described below. As developed below, each Task is entirely defined in the Procedures Page of some Operation. Thus such Tasks and nested Activities constitute the programming of active computations in the Operation. As described in greater detail in the Appendix, each Activity type defines the corresponding order of execution of its statements or nested Activities. The initial restriction, that Operations are defined and executed entirely in a single machine (except for subOperations chosen for distribution in a separate machine), implies that the Statements and subActivities in a Task are all executed in a single machine.

Distribution in this sense includes the possibility that data accesses and Task or Operation Calls can be made from some remote Operation and machine using appropriate dot references or Scoping Prefix (with an **IN** keyword).[11] Data accesses will await the return of the necessary data before continuing execution (or could be pre-accessed to avoid any wait). But between distributed Operations we will now take the view that all Task or Operation Calls will act as if containing a **NEXT** continuation command which bypasses any suspension awaiting completed Task or Operation Call.[12] At the same time, the basic rule, that termination of an Activity terminates all included statements including Task (or Operation) Calls, still applies whether or not the Task or Operations are local or remotely distributed.

As indicated before, the execution of any parallelism in a Task is simulated by executing each paralleled piece consecutively within a given sample time. Since all (Parallel) Activities are carried out in a single machine, this rigorous sequenced simulation applies to all such Activities. The only other relevant form of parallelism occurs when separate Operations are initiated in parallel. When the initiation occurs in the same machine the corresponding execution is still predictably sequenced. But when the Operations are in separate machines the rigorous sequenced simulation no longer applies.

Tasks are Called in several different ways, with associated argument lists (which can also be expressed as control parameter Blocks or Recipes). Certain Task Calls include data connecting Statements defined in an associated Activity. The named type of Calls are: Simple (conventional) Calls, Block Calls, Recipe Calls, Idiom Calls and Path Calls. As indicated above, each of these will be entirely executed in a single machine (not distributed).

Limitations in the Above Prior Specifications. The basic limitation to the Task is as a subroutine. Apart from such essential structures as Block and Recipe Calls, a conventional subroutine may occasionally be needed. Normally such a case would limited to Sequential and Looping Activities. When implemented as above no reentrancy is needed.[13] The above **ALL** key word accommodates more complex reentrancy needs. None of this affects our duplicated and distributed applications.

The restriction of Activity computation to the defining Operation and machine limits distributed control, but a greater problem in duplicated controls is the combination of Looping with Idioms (or Blocks). An Idiom is both procedural and declarative. The conventional Looping Activity allows looping the same procedural control calculation through duplicated structures. But Idioms in intended duplicating loops also would require the corresponding looping of the control function and Parameter Table declarations. While one could generalize the existing Looping Activity, a better strategy would be to keep the existing structure simple and develop a new form of looping.

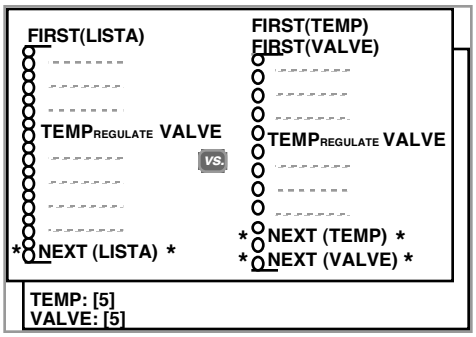
Suggested Accommodations. The existing strategy of distribution of Operations (and SuperVariables) only covers all needs; programmed controls internal to the Operations can impose any needed coordination. Any finer division would lose essential discipline. Originally we had considered allowing a special variant Looping Activity to include such a distribution capability. But the attendant coordination difficulties with other Activity capabilities make this inappropriate. We will restrict the extended form of Looping Activity, shown in the following figure, to supporting Idiom, Block, and Recipe Calls which combine procedural and declarative actions all in a single Operation. These Calls will not be available in the standard Looping Activity; any continuing statements or activities in such an Activity will be initi-

[11] A Scoping Prefix specifies some other Operation in which the following Statement or Activity is to be executed (for example: **IN FURNACE: FUELFLOW = 350**).

[12] Appendix II contains a note summarizing the coordination limitations applied to distributed Operations.

[13] One likely exception is Global Tasks operating in several subOperations.

ated only once under the Non-Terminal Exception (see Appendix I), and without control variable iteration.



In the figure, the left-hand (simple Looping Activity) includes the single Idiomatic control loop. The Activity does not permit List Idiom Statement variables to be affected by iteration commands during the Looping Activity (which would give rise to multiple declarations). Under the Non-Terminal Exception, the loop is initiated in the first pass through the loop and runs independently thereafter, terminating at the end of the Looping Activity. The right-hand (extended Looping Activity indicated by the separated Loops or circles in the Looping Bracket) generates a distinct Idiomatic control loop for each iteration indexing throughout the **TEMP** and **VALVE** List elements. It terminates on the final **TEMP** and **VALVE** List variable (the second **NEXT** termination is redundant).

In summary: The simple Looping Activity can only support a single set of internal declarations (in this case the Idiom Statement) with no indexed indexed variable Lists. The extended Looping Activity acts as if it were expanded out with every indexed item instantiated so that the whole acts as a Sequenced Activity. Extended Looping Activities would undergo compile time error checking that the iteration activity was consistent with this semantics. Perhaps the best way of thinking is that the Simple Looping Activity expresses a run time looping, and the Extended Looping Activity expresses a compile time expansion. There remains a question of looping a Continuous Activity to provide an overall continuous control. This is addressed in the "Notes on the Original Motivating Problem Statement" in the Appendix II.

Operations

The Operation is the most complete and powerful ICL language element (as in the earlier figure). It combines functioning and data, making it correspond to an Object and making it applicable to the modeling of process element controls. Everything else in an ICL application is a part of some Operation. Controls in Operations are activated by Operation Calls, or by direct assignment changes to the Operations operating States.[14]

Limitations in the Above Prior Specifications with Suggested Accommodations. The Operation provides the best approach to general distribution. However the prior specifications do not provide for the declaration of the multiple indexed Operation instances as separately referenced. Making the duplicated case, numbered bracket notation define an implicit Last (as repeated below under Lists) overcomes this problem by allowing the separate instances to be referenced by indexing as with arrays.

The analogous internal variables of different Operation instances will then share the same Name (within the distinct Scoping of the different Operations). The result is still a different nomenclature. However the ability of the SuperVariable to support aliased names permits any desired operational nomenclature.

Lists

Lists are defined, explicitly and implicitly, in a number of contexts. Lists are defined explicitly on the Definitions Page (below the SuperVariable tables) with a Naming Prefix (a Name followed by a colon) followed by a parenthesized, comma separated List of List element Names or references (e.g. **LIST: (ITEM1, ITEM2, ITEM3)**). The List can also be declared as a number in a bracket (e.g. **LIST: [5]**) indicating a corresponding number of unnamed elements, or as an open ended set of elements (by a bracket with at least two dots (periods as in **[..]**)). In this case the List elements can be modeled after some already declared element, indicated with its name in parenthesis directly after the List name in the declaration. The List may be prefixed by more than one consecutive Naming Prefix, setting up several Names for the single List. Each name is associated with in internal pointer set to select an intended List element under the various interation control commands (**FIRST**, **LAST**, **NEXT**, **PREV**).[15] An assignment between

[14] Operations (and Task Calls) have accessible States used to monitor and control their operation. Some are standardized and some may be defined by the user. (In the case of Task Calls the principal access is from within the body of the Task or through an argument list.)

[15] These commands and their use in Looping Activities are described in the Appendix I.

List Names of the same List will have the affect of changing just the assigned pointer (since the rest of the List is defined as identical).

A List element can be referenced by its Name (if it has one), or by an index number in a dotted data structure like reference (e.g. LIST.2), or by an integer valued (as Real or Counts valued) variable (or computed value) in parentheses (LIST.(INDEX) or LIST.(INDEX1+INDEX2)). The List specification addresses a number of other dimensions which will not be addressed here. The additional discussion will address implicit definition of Lists and proposed extensions. Prior specifications (and earlier discussions) already include the possibility of numbered sets of duplicated Operations, expressed by following the Operation Name on the defining Operation Page by a colon and number in brackets (as with Lists). The specifications indicate a List as being allowed to include: named references (or Calls) of Operations, Tasks, Blocks or Recipes, and SuperVariables). For our purposes, when the List element is unnamed and itself a simple object stored in the List, not a reference, the allowed elements will appropriately include only data values.

Limitations in the Above Prior Specifications. Lists address all of the array requirements but are unnecessarily obscure for the kind of application duplications intended here. And for our purposes it should not be necessary to separate traditional matrix/vector and List concept. The above discussion already addresses the concept of duplicated, "dimensioned" sets of Operations or List elements. Prior ICL specification does not support a similar "dimensioned" SuperVariable. In each case we would like to explicitly consider such dimensioned elements to be Lists, whose elements can be operated on by all of the List operations.

Suggested Accommodations. We have already introduced a "dimensioned" List usage and indicated the similarly dimensioned Operations (now considered Lists of identically named and structured Operations, the List sharing the Operation Name). A similar bracketed dimension number added to a SuperVariable can be considered to call for a set of duplicated SuperVariables also treated collectively as a List with Name identical to the shared SuperVariable Name(s).

SuperVariables include a mechanism for fixing the Attribute value of an Attribute over the full definition table as shown with the **CONV** heading. This applies the **SQRT** Attribute to all SuperVariables in the table. We can define a bracketed dimensioning number (the **[3]** in the figure) as applying the Name **ARR** to the next three SuperVariables. We can apply this convention similarly duplicating any Attributes in consecutive SuperVariables, but distinguish the Name Attribute as defining an implicit List accepting all of the List syntax and functionality (making the **ARR** List of three SuperVariables).[16]

NAME	IN	CONV SQRT	VALUE
TRF	2		2.0
TRV	3		3.0
ARR[3]	4		4.0
	5		5.0
	6		6.0
TRQ	7		7.0

These two forms would support the normal List dotted structure indexing notation (e.g. ARR.2) but we can add a shorthand more sympathetic to conventional array usage by allowing the conventional array indexing bracketing notation to be used. Thus ARR[2] might stand for ARR.2, and ARR[INDEX] might stand for ARR.(INDEX) (and LIST[] for LIST.).

Small System Language Distribution under the Large System Language. Prior specification has not addressed the precise language usage for combining Large and Small Language systems. It is generally intended that the Small System Language have its configuring software in a machine separate from the working controller. This machine could also include a Large System Language system. Small System Language systems can be viewed as Global, accessible to all other operations by name. Alternatively they can be assigned as subordinate to one or other of the application Operations. This could be declared by listing all subordinated Small Systems by name on the Operation Page under the subOperations or alternatively on the Definitions Page below the SuperVariables and above the Lists.

As illustrated on the Appendix III figure, the first (consecutive) **NAME** Attribute(s) (before the States and State Attribute) in a Small System, compound SuperVariable, listing defines the Small System name

[16] This somewhat obscure expression has the flexibility to mix "array" and other declarations in the single table and allow arbitrary aliased List element Names.

and any aliases. Following the above convention, any one of these **NAME** Attributes can be followed by a bracketed number to express index-able duplicates of the whole source program. The resulting copies can be treated as Global as before or assigned with appropriate index to any one of the application Operations.

Messaging

Distribution requires the ability to communicate between the distributed Operations. The above discussion indicates that any access returning a value would suspend the continued operation until the necessary returned data was provided. The requirement might be supported in several ways. Most simply the data can be fetched from the remote Operation and returned. Better would be to allow the compiler to code a pre-access which provided the data without wait on need. Alternatively duplicated copies of such communicated data can be kept in both Operations, updated automatically each sample time. The next section discusses the methods for user choice of strategy. But there will likely be a need for transmitting requests for more complex messages.

One way of making inter-Operation reference explicit is to generalize certain ICL syntax forms to represent them. These would be recognized when referencing Operations in distinct machines. The **IN** Scoping Prefix is a natural general purpose vehicle for invoking complex messages in separate machines. Individual references to remote machine Operations can also be individually identified to generate distinct messages. Such references to remote machines may include direct references to remote superOperations.

The concept allows referencing and referenced Operations in the same machine to be compiled as a normal reference. But when the two Operations are in separate machines, the reference is identified with any indicated computations already pre-compiled in the appropriate machine (in the case of a static computation), or converted to a communicable pcode form and transmitted to where it is to be executed. The compilation or interpretation would set up the appropriate return message handling. As part of the processing, the user might be given options for handling compilation and interpretation.

Post Programming Configuration of Distributed Operations

Prior ICL specifications define the effects and limitations of distribution without defining any specific distributing syntax. This question involves two sets of issues: The user needs to declare of the intended deployment of Operations between machines. And each deployment involves different sets of reference implementing messages. For each message the user can set the desired messaging strategy. Because we would like to use the same application program in many distinct, differently deployed, applications and because the messaging strategies are likely to be vendor dependent, the deployment needs to be handled by software and notation separate from the main language system, with its own syntax.

Machine 1:
Plant Wide Operation

Machine 2:
A, B, C, D

Machine 3:
E, F, G, H

Global Connections:

A: TRE, WOL2	Strategy B
A: TRF, WOL3	Strategy B

Connections:

PlantWideOperation: IN A:	Strategy A
PlantWideOperation: IN B:	Strategy A
PlantWideOperation: IN C:	Strategy D
PlantWideOperation: IN E:	Strategy A
PlantWideOperation: IN F:	Strategy C

The proposal is to provide separate software allowing the user to declare the intended deployment of Operations to machines. The compiler would then identify and display those Scoping Prefixes and references which communicate between machines. The user would then declare any of those which called for a non-default implementation. This would require that the system vendor have defined and implemented a number of standard named strategies and that the different kinds of references be classified in terms of their fit to any particular strategies. Related strategies would differ in the way their efficiencies were optimized but not in their semantics.

The discussion of arrayed or multistage applications creates other kinds of distribution complications. Such an application could involve arrayed elements distributed among the machines. Any of the above items with bracketed “dimension” numbers might be defined at one level of Operation but actually be used distributed between the lower levels of subOperations (similarly any Global variables might actually be used mostly in subOperations). The configuration tool could be extended to declare the effective

distribution of data and Tasks in this way. Ideally the design of Operations should minimize the problem.

Summary of Proposed Extensions

In summary the note has introduced proposed extensions supporting more generalized controls duplication and distribution:

- Integrating all bracketed array-dimension-like item numbering as defining a form of List, accessible by indices, iteration controls, or temporary (Batch) production Train reentrancy. The resulting index-able Lists can include Operations, SuperVariables, and make all such entities accessible uniformly under any List mechanism. Two cases apply:
  - Where the numbering follows a name and represents a List with copies of a basic defined entity (Operation or List) taking that name, the bracketed number is preceded by a colon. The bracketed number then represents a List before its Name is assigned.
  - Where the numbering follows an Attribute together representing a set of repeated Attributes (and when the Attribute is a Name, a corresponding List of Named SuperVariables), there is no colon. The bracketed number then represents a duplication dimension to the Attribute and to any thus Named SuperVariable.

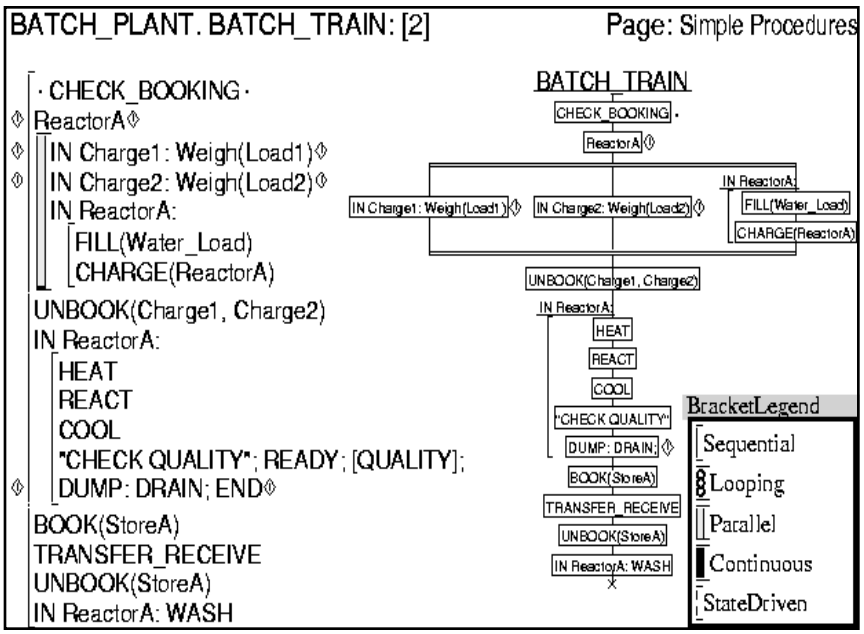
  The result, in either case, is a List, with iterated elements.
- Distinction of basic (Simple) Looping Activities which loops procedural computations as distinct from the Extended Looping Activity which can iterate through declarations (as long as no redundant declarations are made).
- Definition of an explicit set of Messaging capabilities based on normal reference and the more general use of the Scoping Prefix. This allows user optimized handling of messaging efficiency without generally affecting the control result.
- Provision of a separate Operations (and Task and data) distribution and messaging configuration tool.
- Definition of the Small System Language duplication syntax and the Small to Large System Language distribution syntax.
- (In Appendix I) refinement, generalization, and formalization of Activity definition exceptions.

Appendix I: Activities and Tasks

Introduction

As a general rule, computer language control structures address the related execution of their specific free form programmed components. ICL addresses four levels of control structure:

- Task/Activity[17] structures addressing the basic execution order of component statements or activities.
- The Prefix structures controlling conditional execution of associated statements or activities.
- The application specific Theme Statements.
- The Footnotes for flexibly associating general detailing computations.



This Appendix addresses the thinking behind the Activities, Tasks and Activity/Task Brackets.[18] Activities represent different views of execution of component activities:

- Sequential: the most basic ordering, inherent in simple computation. Sequential Activities execute their contained statements or Activities in order, terminating with the last one. **Sequential Activities model normal sequentially ordered, real time, manufacturing control.** But normal control also requires:
- Continuous control: the execution of all included statements and Activities continuously, which translates in a digital environment into the execution of each included statement (in sequential order) and Activity every sample time. **Continuous Activities model normal continuous feedback control, interlocking, and monitoring.**
- Parallel execution: the execution of all included statements and Activities, all starting at the "same time", each continuing on its own timing independently of the others. **Parallel Activities model the simultaneous processing of independent activities.**
- State Driven: more specialized, this recognizes that control equipment may be called to operate under several distinct modes (i.e.: Startup, Running, Shutdown) under externally imposed user commands. State Driven Activities do not terminate even if all active nested Activities have terminated (since a State change might reactivate a nested Activity); they must be terminated by explicit END commands or with the nesting Activity. **State Driven Activities model control which is to support a number of distinct mode or state dependent behaviors, whose time of switching may be operator managed and even sequenced or cycled. They are expected to operate over an extended span of time.**
- Looping: not normally basic to control but included for generality and because it applies to the processing of large sets of similar data. As described later Looping Activities are supported by several List related functions. **Looping Activities model normal traditional computer language looping; it can express cyclic operation.**

As shown in the figure, each Activity is associated with a notating Bracket whose shape is intended to be suggestive. Its definition is intended to address a simple class of control execution behavior simply. But necessary generalizations of the definitions may give rise to some complexities. In the normal situation the user should be able to ignore these by avoiding confusing cases. But the ICL design covers the most general definitions possible to allow both the basic use and intent, and any special requirements.

It will be noticed that several Activity types will be related, at least in their implementation. For example

[17] A Task is a named Activity which can act as a subroutine (sometimes with an argument list). However, it is not in fact a subroutine in the usual sense because it must operate under all kinds of parallel and duplicated conditions not suited to the normal subroutine stack/reentrancy mechanisms.

[18] However the relation between simple State Prefixes and State Driven Activities needs to be spelled out as well.

the included statements and activities in a Continuous Activity are inherently executed in parallel. They are also executed in a looping fashion with each iteration occurring in a separate sample time. There are a number of resulting redundancies in the expression of nested Activities:

- In a Continuous Activity, non-suspending Sequential Activities generally act as if Continuous (each of their included Statements executes every sample time).
- **NEXT** continuation commands[19] (defined later and in the earlier Large Language Specification) in a nesting Continuous or Parallel Activity do not have any effect (nested Activities do not suspend the nesting Activity anyway).
- Because of refinements described later, Sequential, Parallel, Looping, and Continuous Activities, which each contain only Continuous Activities and (non-terminal) Statements all act identically as nesting Continuous Activities. In the Intent documenting spirit of ICL, only a nesting Continuous Activity should be used when this is the intended effect.[20]

There are also some inherent incompatible usages which have been adjusted to better express useful practice. The immediate discussion improves and formalizes earlier Large System Language Specification discussions.

- Looping Activities are intended to reflect either pure looped computations or cycled production controls. Thus Looping Activities without an included END command or internal Real Time production activity or suspension (unlike similar Continuous Activities)[21] make no sense and are treated as Errors by the language compiler.
- Continuous or otherwise non-self-terminating Activities, without explicit termination or continuation commands,[22] nested in Sequential or Looping Activities would normally hang up the contained Activity execution. The language treats all such nested Activities as allowing the nesting Activity to continue operation in parallel unless this would allow it to terminate on its own.[23] These nested non-terminating Activities will still be terminated whenever some nesting Activity is terminated by some other means. For later convenience, this principal of continued bypassed execution of non-terminating Statements or Activities will be referred to as the Non-Terminal Exception.[24] One of the uses of this exception is in programming the staged application of continuous controls.
- Certain Statements and control usages (in particular Idioms) are intended to run continuously over some span of time (a number of consecutive sample times) even when separately restated in each of several consecutively executing Activities. These elements will be implemented in such a way that they are not aborted with one containing terminating Activity when they are also programmed to operate in the immediately following Activity (at any level of nesting). This mechanism allows the continuous control system to be systematically built up over a series of Activity calls in a way that naturally expresses the buildup as a sequencing of control initiations. It also allows any part of a control structure to continue to operate when it is included in each of several alternative elements in a switched structure. For later convenience this usage will be referred to as Consecutive Continuation.

As with Idioms and other aspects of ICL, the Activities are designed to represent application Intent, even if in a very generalized sense. Redundant computational capabilities are appropriate when this allows less ambiguous expression of Intent. However the different Activity expressions should be used to properly express the associated Intent.

Sequential Activities

This Activity causes all of its nested Statements and Activities to be executed consecutively (in sequence), each to completion before the next, until one of these Activities suspends. Any execution suspended is then picked up in the next sample time until the whole Activity terminates. The Non-Terminal Exception men-

[19] **NEXT** continuation command without argument list; not the **NEXT** List iteration command/function.

[20] Ideally, the substitution of a Continuous Bracket would be made automatically. However the ambiguity could also result from an earlier edit or a later intended change. For this reason, only a warning indication will be provided.

[21] The original specification documents considered executing successive Loop Activity iterations in distinct sample times, making them more similar to Continuous Activities. Current thinking eliminates this at the risk of introducing unintended hang-ups. The intent is to make more compatible use of Lists when nested in Continuous Activities.

[22] The language includes explicit commands (**END** and **NEXT**) for terminating an Activity or allowing nesting activities to continue in parallel.

[23] In other words, a final non-terminating Statement or Activity still makes the main Activity non-terminating.

[24] Several formalizing definitions are emphasized in this note. Their (first) defining Appendix I occurrence (and other document occurrences) will be underlined, in addition to being capitalized according to the usual ICL convention for terms which take specialized ICL meaning.

tioned before applies: If any but the last nested Activity or Statement lacks a programmed termination condition, it is initiated without suspending the continued execution of the remaining statements or Activities as if including a **NEXT** continuation command. The Activity terminates after the last Statement is completed (or, as is the case with any Activity, by an included END command or the termination of a nesting Activity).

Parallel Activities

This Activity causes all of its nested Statements and Activities to be executed in "parallel". True parallel execution presents a number of linguistic problems: the relative timing of statements in Activities executed in parallel is ambiguous. Accordingly ICL disambiguates the parallelism, imposing a specific order: Each nested Activity element is executed consecutively to completion or suspension within the given sample time, according to its natural rule.[25] Each later sample time causes the consecutively resumed execution of each included currently suspended Statement or Activity in its same initiation order. The Parallel Activity terminates (terminating any still-active, indirectly-nested Activities) when all directly nested Statements or Activities terminate (or by explicit command).[26] For later convenience we will refer to this systematically interpreted digital simulation of parallelism as Constrained Parallelism.

Constrained Parallelism corresponds to the practical control need for production control parallelism: The only imposed sequential behavior is applied to pure computations needing constrained unambiguous relative order. Real Time productive activities associated with waiting suspensions, are effectively run in parallel with each other and with the totality of the sequenced computations.[27] In replacing fast controls traditionally implemented in parallel logic this distinction must be born in mind. Computers are not naturally parallel operating devices. However, once understood, the constraints simplify the implementation of these controls, eliminating internal race conditions and other traditional anomalies.

Continuous Activities

This Activity causes all of its nested Statements and Activities to be executed under Constrained Parallelism (as above). On succeeding sample times, each nested Statement or Activity is resumed or repeated. Stated alternatively, this effectively combines parallel execution of each element with each element being repeated or looped[28] each sample time.

Looping Activities

This Activity causes all of its nested Statements and Activities to be executed consecutively in sequence, each to completion as with a Sequential Activity, unless one of these Statements or Activities suspends the whole Activity by suspending itself. On completing the last nested element, the execution repeats itself starting with the first nested element. The execution never terminates as a whole except by **END** or similar command or termination of the nesting Activity. Absence of some terminating condition (or suspension) constitutes an error; **this would hang up the system**. The Non-Terminal Exception applies, but the associated non-terminated Statement or Activity is initiated only the first time around the loop. In ICL, unlike conventional languages, initiating and terminating are not built into the Loop structure but are programmed explicitly in standard ICL statements.

State Driven Activities

A State Driven Activity consists of its dashed bracket, grouping a set of (generally) State Prefixed Statements and Activities.[29] In the above sense these Statements and Activities are processed under Constrained Parallelism. Those statements whose Prefix States correspond to the States active in their application environment,[30] will be initiated and executed normally. Any inactive Statements whose Prefix States do not cor-

[25] This coordination is not imposed between simultaneously active Activities carried out in separate control processors in a distributed system where the isolation is likely to ensure that the ambiguity will not affect the actual application semantics.

[26] Thus, if the Parallel Activity contains a non-terminating statement or Activity it will prevent the Parallel Activity from terminating by itself (for instance requiring an **END** command or containing terminating Activity to terminate it).

[27] The resulting unambiguous, constrained, sequential computation of internal States and variables can be used deliberately. When not desired, the programmer can ensure that all variables are independent between computations and achieve results equivalent to true parallelism.

[28] Looped as if each statement were included in its own once through for each sample time looping activity and all were included consecutively in a Parallel Activity.

[29] A State Prefix is a State Name followed by a colon expressing the conditional execution of the Statement or Activity listed directly after the Prefix.

[30] These States will normally be System States of the including Operation but the State Driven Activity can also be preceded by a Local State Environment Declaration (followed by a grouping semicolon, as defined elsewhere) which lists variables whose States define the State Driven Activity's States.

respond to currently (and immediately previously) active States are unaffected. But an active State Prefixed Statement or Activity, whose State was active at the previous sample time but is no longer active, must be terminated under any active controls internal[31] to it.

As with any terminating activity, this means that any internally nested (at whatever nesting level) ABORT State Prefixed Statement or Activity is carried out to completion as part of the termination of its nesting Activity. The State Driven Activity will only be terminated by an explicit END command or a nesting terminating Activity.[32]

The State Driven Activity is also set up to represent mode dependent action where the control may involve a natural sequencing of the mode States or a cycling though some set of these States. The Activity may itself include commands that change the State, in lieu of external or operator action, either setting it to a changed desired State or (with the NEXTSTATE command) to the next State Prefix State according to the listed order. Changing the State to correspond to the initial State Prefix can be used to impose cyclic operation. And, of course, the basic Constrained Parallelism allows parallelism; note that more than one State and State Prefix may be active at the same time, with several distinct mode changes overlapping in time.

The State Prefixes in State Driven Activities vs. Simple State Prefixes

State Prefixed Statements and Activities represent conditionally executed controls or computations. A Prefixed Statement or Activity is executed according to its own rule once the associated State conditions have been met. If not immediately nested in a State Driven Activity, it terminates only under its own rule or if its nesting Activity terminates. In contrast, a Prefixed Statement or Activity, immediately nested in a State Driven Activity, also terminates whenever the associated State conditions cease to be met.

A State Prefix by itself defines a condition for initiating the Prefixed Statement or Activity. Within a State Driven Activity, the State Prefix represents a condition for the initiation and continued operation of the Statement or Activity. The first case is simple conditional execution. The second has to do with mode switching of controls operating over a considerable span of time.

Activity Control Commands (including NEXT and PREV Iteration Functions)

ICL includes a number of commands and functions to control the termination and continuation of containing Activities or Tasks:

- **END**, this command terminates one or more levels of nesting Activities with the highest level of termination being marked by paired diamonds or asterisks the leftmost to the left of the corresponding Bracket, as illustrated above.
- **HOLD**, acts as a last Statement in a Sequential Activity to prevent its end of Activity termination. Its purpose is to prevent the abrupt or arbitrary termination of nested control Statements or Activities. The Activity can still be terminated by an **END** command or by the termination of a nesting Activity.
- **NEXT** (as a command without function argument), is marked by dots (like the diamonds and asterisks of the **END** command) but causes the associated Tasks to operate (after initiation) independently of the nesting Activity, which then continues with the next statement (similarly to the Non-Terminal Exception).
- **NEXTSTATE**, is a special command usage within the State Driven Activity used to advance the associated State to the next associated State Prefix State and execute the associated program. It allows an automatic sequencing of these operating States under operator control.
- **NEXT** (as a function with a List Argument), acts on a List of elements selecting the next element in the List. In iterative applications, typically in Looping Activities, the **NEXT** function is displayed like an **END** command complete with associated diamonds and asterisks. It terminates the associated Activities whenever there are no more elements to select. (NEXT also supports more complex actions.) **NEXT** is naturally associated with a **FIRST** List function which selects the corresponding first List element to start the iteration (as in the figure).

- **PREV**, acts on a List of elements selecting the previous element in the List. In iterative applications, typically in Looping Activities, the **PREV** function is displayed like an **END** command (or **NEXT** function) complete with associated diamonds and asterisks. It terminates the associ-

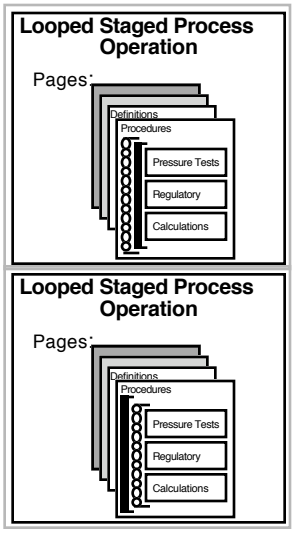
[31] That is, for example, by a direct **ABORT** action or a response to an **ABORT** State Prefixed Statement inside the Activity.

[32] This is necessary because an otherwise apparently inactive Activity might reactivate under a change in one of its Prefix States.

ated Activities whenever there are no more elements to select. (**PREV** also supports more complex actions.) **PREV** is naturally associated with a **LAST** List function which selects the corresponding last List element to start the iteration back through the List (reversing the behavior shown in the figure).

Appendix II: Summary Notes

Notes on the Original Motivating Problem Statement



The original motivating problem was defined in terms of nested Continuous and Looping Activities assuming the validity of the arraying of the Staged variables. The example provides a good illustration of some of the complications of such a nesting (with the Looping Activity in either Simple or Extended form). On the face of it the top figure (corresponding to the original attempts of ICL programming of the application; the iteration and termination commands are not visible in the figures) can be interpreted as carrying out the iteration of the successively indexed controls. This works because the Non-Terminal Exception causes each control to go on in parallel once it is activated.[33]

One might wonder if the whole control terminated once the last iteration was carried out completing the loop iteration. But the program is saved because the Non-Terminal Exception does not apply to the last nested statement or Activity.

One can then examine bottom figure and ask what applies here. At first glance this might be taken as the result of expanding all the iterations out and carrying out he resulting total continuous controls as desired. But does the whole loop terminate at the end of the Looping Activity. The Consecutive Continuation principle prevents this from happening to any internal Idioms, and the remaining computational controls are repeated with each Continuous repetition of the whole Activity. Thus this design should normally work.

However both of the programs involve complications that would more naturally be avoided. **This is another argument for using the duplicated Operations approach**. The use of the Non-Terminal Exception and Consecutive Continuation should really be applied in simpler cases where they avoid complication by preserving natural usages. On the other hand, if the Extended Looping Activity was used, representing a compile time expansion (or its equivalent), this would bypass he Consecutive Continuation and truly express the desired result most simply.

Notes on Distribution and Coordination

ICL expects the form of parallelism implemented in the Parallel Activity to be tightly coordinated. More complicated parallelism resulting when Tasks or Operations are activated within a single machine will also be subject to a single coordinated interpretation. But when the parallelism is a consequence of activating Tasks or Operations in separate machines, this fixed coordination is surrendered. Similarly Statements or Calls which would normally suspend continued operation of their calling Activity for their completion if executed in a single machine surrender that coordination when distributed. The reduced coordination, in each case, corresponds to the idealized position of parallelism. Reviewing, the affected language areas are:

- Parallel Operations in separate machines. In this case the idealized parallel Operations would be independent of each other. Deeper coordination breaches this ideal, rarely benefits, and complicates the design. When necessary, coordination can be forced by accessing special coordination States/variables. Similarly coordination can be made optional as part of the function of the distribution configuration tool.
- Statements and Called Tasks operating in parallel in separate machines. As before the uncoordinated parallelism is the normal case which can be overcome by access to coordination variables or made optional in distribution configuration tool.
- Statements and Called Tasks, which do not return data, activated from a common Task, but operate in parallel in separate machines. In this case the single machine execution would wait completion of Statement or Called Task before continuing on in the Calling Activity. The distributed case assumes eventual completion of the Statement or Call continuing the Calling Activity as if an internal NEXT command had been included in the Called Task. In other words, a single machine "run to completion is converted to a distributed operation of "start action and continue on with what you were doing before". The ideal here is based on the assumption that the Statement or Called Task will terminate anyway so that little loss of coordination normally occurs. Explicit coordination is again possible if really needed.

The goal in each case is to provide systematic, predictably interpreted parallelism in a single machine application, and yield to "real" randomly coordinated parallelism otherwise. But in any case:

◇ A Task or Statement always aborts if nesting Activity aborts, whether or not the nesting and nested elements are in the same machine.

[33] The Exception is imposed to prevent the hang up of the Looping Activity which would otherwise take place after the start of the first Continuous Activity execution in the loop.

Appendix III: Example Extended Small System Language Listing

Key Names: T101 T102 T104 F101 V101 COOKTEMP BIAS VENTTIME HEATTIME COOKTIME COOLTIME

NAME: ICL_BLOCK [10]

States: _

NAME	NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T101	Temperature_1	ECB1	_	0.0	250.0	°F	_	_	_	10.0
T104	TemperatureAv	_*[1]	_	0.0	250.0	°F	_	_	_	10.0

NAME	NAME	OUT	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
T102	Temperature_2	T102.SET	_	0.0	250.0	°F	_	_	_	10.0

NAME	NAME	IN	CONV	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
F101	OutletFlow1	ECB2	SQRT	_	0.0	250.0	GPM	_	_	_	10.0

NAME	OUT	VALUE	MIN	MAX	UNITS	HI	LO
V101	ECB3	_	0	100	%	90.0	5.0

NAME	VALUE	UNITS
COOKTEMP	200	°F
BIAS	7	°F

NAME	TIME	UNITS
VENTTIME	10	MIN
HEATTIME	20	MIN
COOKTIME	80	MIN
COOLTIME	15	MIN

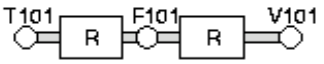
FOOTNOTES:

*[1] T104 = (T101 + T102) / 2;

FILTERS/COMPENSATIONS:

LOOPS:

T101 REGULATE[1] F101 REGULATE[2] V101;



	NAME	STATES	PB	INT	DER
[1]	T101	AUTO; +	100.0 %	0.2 MIN	0 MIN
[2]	F101	AUTO; +	100.0 %	0.1 MIN	0 MIN

THEME STATEMENTS:

RAMP T101 *START* FOR VENTTIME†[1]
HEAT TO COOKTEMP + BIAS, IN HEATTIME MIN
COOK AT COOKTEMP, FOR COOKTIME MIN
RAMP T102.VALUE *COOL* TO 0*[2], IN COOLTIME;

†[1] T102.VALUE = COOKTEMP;
*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2;

PROCEDURES: