Report Number: WUCS-82-5

1982-02-01

On Reducing Ambiguities in Methodology Definitions

Gruia-Catalin Roman

The paper describes a proposal for a methodology definition language and illustrates it on a variant of a well-known methodology, top-down program design. The language describes: (1) the structure of and the relations between various products of the design process (they are referred to here as configuration items and may include such things as system requirements, program specifications, module design, code, etc.); (2) changes in the state of these configuration items and consistency constraints between their states; and (3) the sequencing of design activities permitted by the methodology. A separate section addresses the way in which backtracking due to... **Read complete abstract on page 2.**

On Reducing Ambiguities in Methodology Definitions

Gruia-Catalin Roman

Complete Abstract:

The paper describes a proposal for a methodology definition language and illustrates it on a variant of a well-known methodology, top-down program design. The language describes: (1) the structure of and the relations between various products of the design process (they are referred to here as configuration items and may include such things as system requirements, program specifications, module design, code, etc.); (2) changes in the state of these configuration items and consistency constraints between their states; and (3) the sequencing of design activities permitted by the methodology. A separate section addresses the way in which backtracking due to design errors is treated in the language.

ON REDUCING AMBIGUITIES IN METHODOLOGY DEFINITIONS

Gruia-Catalin Roman

WUCS-82-5

February 1982

Department of Computer Science
Washington University
St. Louis, Missouri 63130

As appeared in Proceedings of the 1982 Conference on Trends and Applications, May 1982, pp. 47-54.

ON REDUCING AMBIGUITIES IN METHODOLOGY DEFINITIONS

GRUIA-CATALIN ROMAN

DEPARTMENT OF COMPUTER SCIENCE
WASHINGTON UNIVERSITY
SAINT LOUIS, MISSOURI 63130

ABSTRACT

The paper describes a proposal for a methodology definition language and illustrates it on a variant of a well-known methodology, top-down program design. The language describes: (1) the structure of and the relations between various products of the design process (they are referred to here as configuration items and may include such things as system requirements, program specifications, module design, code, etc.); (2) changes in the state of these configuration items and consistency constraints between their states; and (3) the sequencing of design activities permitted by the methodology. A separate section addresses the way in which backtracking due to design errors is treated in the language.

Acknowledgments: This work was partially supported by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284. H. N. Shaw experimented with the use of the language and provided valuable suggestions. The contributions of W. E. Ball, W. D. Gillett and M. J. Stucki in discussing and reviewing aspects of the language are also acknowledged.

INTRODUCTION

Efforts to enhance the preciseness of design methodology descriptions have been following two main directions. One avenue being pursued involves attempts to describe formally the nature of the design specifications and the logical and performance evaluation criteria used in determining the acceptability of proposed designs [1]. The second direction being observed, primarily in the data processing area [2], is characterized by efforts to treat methodologies as well-defined algorithms. While the notion of reducing design to a mechanical procedure may be subject to debate, its contribution to promoting precise methodology definitions is beyond dispute. Unfortunately, despite the trend toward better defined methodologies, their presentation continues to be almost exclusively informal and, as a consequence of this fact, is plagued by ambiguities.

The work being reported here is based on the premise that specification languages have an important role to play in the generation of unambiguous methodology definitions which, in turn, affect the way in which configuration control and project planning will be carried out in the computer-aided design systems of the future. Precise methodology definitions hold the promise for better communication among designers and also the key to increasing the designer's capacity to study, understand, evaluate, and compare one methodology against another. Furthermore, methodology specifications could be used by configuration control tools to enforce the correct use of the methodology on a given project and could be used by project management tools to develop accurate schedules and resource allocation plans.

These opportunities are only now beginning to be explored and no similar efforts have been yet reported in the open literature, to the best of our knowledge. This paper reports the results of an investigation whose three major objectives, while falling short of the stated goals, represent a necessary stepping stone toward them. The first objective is to identify a way of describing the structure of and the relations between various products of the design process. They are referred to here as configuration items and may include such things as system requirements, program specifications, module design, code, etc. The second objective is the ability to capture changes in the state of these configuration items and to define consistency constraints between their states. The third major objective is to be able to prescribe the sequencing of design activities permitted by the methodology in question.

The next four sections of the paper describe a proposal for a methodology definition language and illustrate it on a variant of a well-known methodology, top-down program design. A separate section is dedicated to discussing the way in which the issue of backtracking due to design errors is addressed in this paper. The concluding section summarizes the experience to date with the methodology definition proposal and some of the difficulties encountered.

METHODOLOGY DEFINITION APPROACH

The methodology definition consists of three parts: the definition of the configuration items, the definition of consistency constraints, and the sequencing of design activities. Syntactically, the definition assumes the following appearance with the keywords being capitalized. (See also Figure 1.)

```
METHODOLOGY methodology-name.

CONFIGURATION ITEMS.
...

CONSISTENCY CONSTRAINTS.
...

sequencing of design activities

MEND.
```

The configuration items represent entities being generated and used in the design process, e.g., documentation, programs, hardware components, etc. The exact configuration items one may include in the definition of the methodology depends upon the nature of the methodology and upon the granularity of its description. Program modules, for instance, may be relevant for a program design methodology, but they may not appear in a software system design methodology which treats programs as the lowest level entities of interest to the designer. Aside from the identification of configuration items, the methodology definition needs to include the structural relations between these items. Considering the case of the program modules again, they are grouped in a hierarchical structure to form a program. Moreover, the program, in turn, has a program specification (another configuration item) and perhaps a program design document. All this information has to be stated in the configuration items definition. This is accomplished through the use of a method borrowed from Hoare's treatment of recursive data structures [3]. The tree structure of most documents naturally led to this particular selection which, in turn, suggested a LISP-like notation for defining the structural and state invariants over the configuration items [4].

The consistency constraints over the configuration items originate in the design rules prescribed by the methodology and reflect properties that remain invariant throughout the design process. (They are not unlike the consistency constraints present in a database.) Because the design rules are prescribed by the sequencing of design activities, the consistency constraints may appear to be unnecessary unless one requires a certain level of redundancy in the definition. (Redundancy is considered desirable and forms the basis for the self-consistency of the specification.) However, the presence of the consistency constraints is necessary and desirable from another point of view. Since, as shown later, many of the design activities are presented informally (natural language), the consistency constraints enable one to reduce the ambiguity level intrinsic to natural language. Furthermore, certain possible but undesirable sequences of design activities may be ruled out. (This situation occurs when using nondeterministic constructs in the activity sequencing part of the language.)

The sequencing of design activities in a methodology is not, generally speaking, different from the sequencing of instructions in programming languages. Consequently, most control abstractions (i.e., flow of control constructs) have been borrowed from structured languages, with some modifications. By necessity, they include sequential type constructs, parallel type constructs, nondeterminism and recursion. The last three require some discussion. The need for high degrees of concurrency in the methodology definition is motivated by the fact that most projects involve designers that work in parallel on different aspects of a problem. Nondeterminism is required for two key reasons. The first one is the frequent occurrence of situations where the designer has to choose among several courses of action based on personal experience and methodology supplied guidelines rather than algorithmically. The second reason is the equally common situation where the methodology is only partially defined and still under development and evaluation. Finally, regarding recursion, it is required by the use of recursive data structures in the definition of configuration items.

CONFIGURATION ITEMS DEFINITION

The simplest configuration item is one which has no recognized structure. It is called an atom. Given any number of configurations items, they may be used to create more complex configuration items: sets, which are abstractions of collections of items, and recursive structures which render the organization of documents or actual products. The BNF specification of the syntax used in specifying the configuration items looks as follows:

```
<configuration>
   ::= <item-definition> ";" |
       <item-definition> ";" <configuration>

<item-definition>
   ::= <item-name> "=" "(" <item-tuple> ")" |
       <item-name> "=" "{" <item-list> "}"

<item-tuple>
   ::= <atom> | <item-name> |
       "SEQUENCE" <item-name> |
       <item-tuple> "," <item-tuple>

<item-list>
   ::= <atom> | <atom> "," <item-list>
```

In order to better understand the approach, let us consider the relatively simple case of a top-down program design methodology. The intent is to carry out the example through completion by

starting it here with the definition of the configuration items and continuing it in the next two sections. A reasonable set of configuration items one might consider is bound to include the original program specification, the program design, and the code. The program specification may include the input and output assertions and some sample test data. The program design generally consists of the program data structures and the module definitions. The modules form a hierarchical structure which is isomorphic to that formed by the subroutines present in the program code. Keeping all these in mind, one may build the following configuration items definition:

```
METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

program-specification
 = (input-assertion, output-assertion, test-data);

program-design
 = (data-structures, module);

module
 = (module-name, module-definition,
                          SEQUENCE module);

program-code
 = (subroutine);

subroutine
 = (subroutine-name, subroutine-code,
                     SEQUENCE subroutine);

CONSISTENCY CONSTRAINTS.
...

sequencing of design activities

MEND.
```

(Note: (1) The SEQUENCE construct is used to describe lists whose length is unknown at methodology definition time, e.g., the list of modules called by a given module. (2) The language definition rules out any graphs which are not directed acyclic graphs.)

CONSISTENCY CONSTRAINTS SPECIFICATION

Most often, the consistency constraints one may want to specify involve more than mere structural properties of the configuration items. Take, for instance, the relation between modules and subroutines. They form two isomorphic structures, i.e., for each module there is a corresponding subroutine and the subroutines called by it correspond to the modules called by the said module. This relation, however, is not an invariant over the configuration items because it holds true only at the end of the design process and not throughout. One way to assist the designer in the formulation of consistency constraints such as this is through the introduction of the concept of state. States and state transitions may be included in the consistency constraints definition section and referred in the statement of other invariants. The relation above could thus be reformulated by adding the condition that both the program-design and the program-code are in the state called "frozen." The notion of state also helps capture the progress made by the designer.

The syntax employed in the state specification is given below.

```
<state-assignment>
 ::= <item-name> ":" <initial-state> ";" |
     <item-name> ":" <initial-state> ","
                              <transitions> ";" |
     <atom> ":" <initial-state> ";" |
     <atom> ":" <initial-state> ","
                              <transitions> ";"

<transitions>
 ::= <state> "-->" <state> |
     <transitions> "," <transitions>
```

Assisted by this notation system, the following state definitions may be added to the example.

```
METHODOLOGY top-down-design.

CONFIGURATION ITEMS.
...

CONSISTENCY CONSTRAINTS.

 STATES.
 program-specification: given;
 program-design:  not-started,
                  not-started --> in-progress,
                  in-progress --> frozen;
 data-structures: null,
                  null --> designed;
 module:          null,
                  null --> designed;
 program-code:    not-started,
                  not-started --> in-progress,
                  in-progress --> frozen;
 subroutine:      null,
                  null --> stubbed
                  stubbed --> coded
                  coded --> tested;
 INVARIANTS.
  ...

sequencing of design activities

MEND.
```

The approach to invariants definition is illustrated by constructing two invariants required by the example. Some knowledge of LISP is assumed on the part of the reader. A more convenient notation is being planned but its introduction would increase the size of the paper without adding to its technical contents. A configuration item name followed by a state name in square brackets should be treated as a predicate that evaluates to true if the item is in the specified state, and to nil, otherwise.

The first invariant states that the coding may not start until the completion of the design:

```
(COND ( program-code[in-progress]
                  program-design[frozen])
      ( T T) )
```

The second invariant originates in the requirement that the design may not be frozen unless all modules have been designed:

```
(COND
  (program-design[frozen]
     (AND data-structures[designed]
        (NIL ((check LAMBDA (X)
               (COND ((NIL X) NIL)
                     (X[designed]
                       ((checkseq LAMBDA (Y)
                          (COND
                            ((check (CAR Y)) T)
                            (T (checkseq (CDR Y))))
                          ) (CDDR X)))
                     (T T)
               )
              ) (CADR program-design))
        )
     )
  )
  (T T))
```

In the above invariant the function check performs a depth-first search of the module call tree and returns T if and only if a module which has not been designed is encountered. This search is performed recursively through the checkseq function which invokes check for each of the modules called by a given module.

In a similar manner, all other invariants may be constructed. A complete specification would require at least one more invariant. It is needed to establish the isomorphism between the program structure identified in the program design and that present in the code.

SEQUENCING OF DESIGN ACTIVITIES

The main body of the methodology is described by a combination of formal and informal statements sequenced by means similar to those employed by programming languages. (See Figure 2.) Tasks, subtasks, and procedures are subject to the same scoping and invocation rules as procedures in block-structured languages. If there are small differences, they are due to the need to capture some concepts peculiar to methodologies. Both tasks and subtasks, for instance, have a section dedicated to project review activities. The section is entered just before the return from a task or subtask. Another distinction is the fact that a task may not be invoked recursively or from other tasks, subtasks, or procedures. The motivation for this limitation stems from the intent that tasks be used to describe major design baselines.

Before continuing the example, one unusual feature of the language must be pointed out in order to avoid possible confusion. It concerns the place of definition for tasks, subtasks, and procedures. A definition always appears at the place where the name of the respective task, subtask, or procedure occurs in the text for the first time. The choice has been made based on the results of early experiments in the use of the language. These experiments indicated that the placement of the definitions at any other place distracts the reader who would encounter definitions prior to understanding their role in the description or would have to search for definitions when the first mention of some task, subtask, or procedure is made in the text. It is felt, however, that in the future both in-line and separate definition capabilities may have to be provided, particularly for use with very large descriptions.

The program design methodology used for illustration purposes includes two phases: program design and coding. Because coding should not be started before the completion and the review of the design, the two phases may be separated into two distinct tasks.

```
METHODOLOGY top-down-design.

CONFIGURATION ITEMS.
...
CONSISTENCY CONSTRAINTS.
...

  TASK design.
   ...
  TREVIEW.
   ...
  TEND.

  TASK code.
   ...
  TREVIEW.
   ...
  TEND.

MEND.
```

The program design starts with the tentative selection of the program data structures. After designing the top-level module, the design proceeds with the design of the modules identified in the definition of the top-level module. A strict one level at a time strategy is followed until no more modules are identified. Appropriate checks and adjustments take place throughout the design process rendered in the following definition of the program design task. (The state changes have been omitted from the definition.)

```
TASK design.
    Design data-structures.
    Design top-level module.

    SUBTASK level-design(x='top-level module').
      Identify modules called by x.
      FOR z IN modules called by x DO
        {IF z needs to be refined
          THEN {Design z.
                Data-structures changed
                              => BACK design.
                F(Verify consistency of z with x)
                              => BACK}}
    STREVIEW.
      F(Verify refinement of x.)
               => BACK level-design(x).
      FOR z IN modules called by x
               DO { // level-design(z).}
    STEND.

TREVIEW.
    F(Verify program-design against
      program-specification.) => BACK design.
    Declare program-design frozen.
TEND.
```

In contrast with the program design, coding is assumed to follow a top-down direction but not in the strict level by level manner. The designer has the freedom to guide the coding and testing based on testing dependencies, as long as testing and coding are not done separately, but progress together. As a way to restrict the designer from coding too much before attempting testing, no more than five untested subroutines are permitted to exist at one time. The same simple style free of the formal state transitions is used to describe also the coding task.

```
TASK coding.
    Code the main program and stub all
    subroutines it calls.
    Debug main using stubs.
    LOOP
      {(All subroutine are tested.) => BREAK.
       IF fewer than 5 subroutines are untested
         THEN { T => Replace one stub by
                              actual code. |
                T => Debug available code.  }
         ELSE Debug available code.          }
TREVIEW.
    F(Check agreement between
                code and the design.) => BACK.
    F(Obtain user acceptance of the program.)
                => { T => BACK design. |
                     T => BACK coding.}
TEND.
```

The last thing to be done is to establish the entry points for program maintenance activities and the rules by which the appropriate entry point is selected.

```
METHODOLOGY top-down-design.

CONFIGURATION ITEMS.
...
CONSISTENCY CONSTRAINTS.
...

  ENTRY major-maintenance.
    Design errors and major enhancements.
  END.

  TASK design.
   ...
  TREVIEW.
   ...
  TEND.

  ENTRY minor-maintenance.
    Changes to the printout format
    and coding level errors.
  END.

  TASK code.
   ...
  TREVIEW.
   ...
  TEND.

MEND.
```

The semantics of an entry point is similar to that of backtracking to be discussed in the next section. The only difference lies in their distinct causes. One is due to checks that take place in the development of the program (backtracking), while the other has its roots in either the decision to enhance the program or the discovery of errors during production.

BACKTRACKING

The discovery of design errors, the identification of a better design path, changes in the specifications, and a newly acquired understanding of the technological impact of a proposed design are some of the most common reasons for backtracking. One can hardly conceive of a methodology that denies the possibility for backtracking to occur. Furthermore, while intentional neglect of backtracking for the sake of simplifying the presentation of some methodology is common practice, to disregard its effects is not acceptable. The cost of backtracking is an important factor in selecting one methodology over another. Methodology design is centered around cutting the cost of backtracking through automated and non-automated checks, through exercising control over the degree of backtracking being permitted, through automatic detection of the potential side effects of backtracking, etc.

The cost of backtracking, however, needs to be weighed against that of the checks employed for the sake of reducing it. In general, increased degrees of automation enable one to enjoy both frequent design checks which reduce backtracking

due to errors and greater opportunities for the exploration of the design space. The cost of backtracking is also affected by the project management procedures. For instance, when several designers work in parallel any backtracking that impacts more than one of the designers may prove very costly compared with backtracking concerning one of them alone.

Such considerations strongly suggest that a methodology definition approach ought to provide a mechanism for explicit structuring of the backtracking process and should lay the foundation for employing (methodology-based) quantitative project planning methods. These methods would require knowledge of the backtracking patterns, backtracking probabilities due to various causes, and the cost associated with various backtracking procedures. By placing explicit backtracking statements, the approach put forth in this paper enables the definition of backtracking patterns, but their use in project planning will have to be explored later on. The adoption of explicit backtracking statements (e.g., BACK name.), however, raises two issues concerning their impact on the flow of control and on the configuration items that have been generated prior to executing the backtracking statement.

The first issue is rather easy to resolve. The name that follows the backtracking command defines the backtracking point and may be the name of some group of statements or the name of some task, subtask, or procedure. In the first case, BACK could be treated as a "goto" to a label which is defined in the current referencing environment and which is always encountered prior to the execution of the BACK command. (This condition may be checked staticly by means of data flow analysis.) For instance,

```
...
label: {...}
...
BACK label.
...
IF ... THEN {...} ELSE {... BACK label. ...}
...
```

represents a correct use of the BACK command because the statement named "label" is always executed prior to the "BACK label" statement. The following sequence of statements, however, is improper.

```
...
IF ... THEN label: {...}
...
BACK label.
...
IF ... THEN {...} ELSE {... BACK label. ...}
...
```

The rule may be easily extended to procedures, tasks and subtasks by considering their names in place of the statement label. However, only tasks, subtasks, and procedures which have not been exited yet may be backtracked. Furthermore, sequences of recursive invocations are backtracked up to the first invocation of the named subtask or procedure.

The backing up of the flow of control must be accompanied by a corresponding backtracking of the configuration items state. It is easy to conceive that the state of the design is saved at every backtracking point and reestablished any time backtracking brings the flow of control back to the respective point. This way of looking at backtracking has one drawback. It seems to suggest that all the design generated prior to backtracking is lost together with the reason for the backtracking itself. Therefore, the interpretation adopted in this paper requires all results generated after encountering the backtracking point to be tagged as needing the designer's revalidation. The designer may chose to accept some results the way they are, to discard others, and to alter still other configuration items. All decisions are based on knowledge of the backtracking cause. Thus, no design decisions potentially affected by backtracking are left unchecked.

CONCLUSIONS

The methodology definition approach proposed in this paper has been used in the specification of several methodologies. Their descriptions, which vary in size from one to five single-spaced pages, have confirmed some of the advantages that were expected. Its use on a methodology development project has yielded significant quality improvements in the communication between the members of the research team. Many problems that were overlooked in the informal presentations of new proposals for a distributed systems design strategy have been rapidly uncovered during the effort of formally describing the methodology.

Despite the early successes in using the methodology definition approach, much work still lies ahead. There are four areas that seem to require immediate attention. First is the issue of incorporating this approach in some computer-aided design system for purposes of methodology enforcement and project planning. Second, project planning techniques based on formal methodology definitions need to be developed and parameterized so as to be usable over a large class of problems and by a variety of organizations. Third, more practical experience is required in using the approach. Of particular interest are methodologies characterized by high degree of backtracking and concurrency. Finally, the availability of a formal definition offers the possibility of carrying out quantitative evaluations regarding optimal placement and frequency of various design checks within methodologies. Preliminary work strongly indicates that these future research directions hold great promise for significant system design productivity payoffs.

REFERENCES

[1] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1'st Int. Conf. on Distributed Computing Systems, pp. 1-14, October 1979.

[2] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer 14, No. 10, pp. 13-37, October 1981.

[3] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, 1978.

[4] Weissman, C., LISP 1.5 Primer, Dickenson Pub. Co., 1967.

FIGURE 1: METHODOLOGY SPECIFICATION STRUCTURE.

```
METHODOLOGY DEFINITION
|
|  CONFIGURATION ITEMS
|
|  CONSISTENCY CONSTRAINTS
|
|  SEQUENCE OF TASKS AND MAINTENANCE ENTRY POINTS
|
|     ENTRY
|     |  ENTRY POINT CONDITIONS
|
|     TASK
|     |
|     |  DESIGN/ANALYTIC ACTIVITIES, ENTRY POINTS
|     |  AND INVOCATIONS
|     |  OF SUBTASKS AND PROCEDURES
|     |
|     |     SUBTASK
|     |     |  DESIGN/ANALYTIC ACTIVITIES,
|     |     |  ENTRY POINTS AND INVOCATIONS
|     |     |  OF SUBTASKS AND PROCEDURES
|     |     |
|     |     |     PROCEDURE
|     |     |     |  DESIGN/ANALYTIC ACTIVITIES,
|     |     |     |  ENTRY POINTS AND INVOCATIONS
|     |     |     |  OF SUBTASKS AND PROCEDURES
|     |     |
|     |     |  SUBTASK COMPLETION REVIEW
|     |     |  |  ANALYTIC ACTIVITIES
|     |     |  |  AND INVOCATIONS
|     |     |  |  OF SUBTASKS AND PROCEDURES
|     |
|     |  TASK COMPLETION REVIEW
|     |  |  ANALYTIC ACTIVITIES
|     |  |  AND INVOCATIONS
|     |  |  OF SUBTASKS AND PROCEDURES
```

FIGURE 2: SEQUENCING OF DESIGN ACTIVITIES.

```
TASK name(parameters).          SUBTASK name(parameters).
...                             ...
TREVIEW                         STREVIEW.
...                             ...
TEND.                           STEND.

PROC name(parameters).          ENTRY name.
...                             ...
PEND.                           END.
```

activity
  informal description followed by a period
  list of state transition rules
      (e.g., item[s1-->s2, s3-->s4].)
  new state assignment (e.g., item[s1].)
  invocation of task, subtask, or procedure
      (e.g., INVOKE p.)

condition
  activity failure (e.g., F(activity));
  activity success (e.g., S(activity));
  formal and informal predicates;
  state test (e.g., item[s1]).

sequence
  a b ... c

if-then-else
  IF condition THEN a ELSE b

if-then
  condition => a
  IF condition THEN a

group
  name: { a b ... c }
  (Note: it acts as a DO group in PL/1.)

parallel group
  name: { a // b // ... // c }
  (Note: it acts as a COBEGIN-COEND block.)

nondeterminism
  name: { condition => a | ... }
  (Note: one activity preceded by a true condition
         is selected as desired by the designer.)

iteration
  name: LOOP a
  (Note: 'BREAK loop-name.' and 'NEXT loop-name.'
         are used to exit the loop and to go back
         to its beginning, respectively.)

sequential for
  name: FOR item IN item-list DO a
  (Note: activity a is repeted for each item
         in order.)

parallel for
  name: FOR item IN item-list DO { // a}
  (Note: activity a is carried out for each item
         in parallel.)

backtracking
  BACK name.
  (Note: the name may be a construct label or the
         name of a procedure, task, or subtask
         from the calling sequence; when no label
         is provided, the most recent invocation
         of a procedure, task, or subtask is
         restarted.)

others
  RETURN.
  (Note: normal return from procedures.)

  DONE.
  (Note: normal return from tasks, and subtasks;
         the task/subtask reviews are not
omitted.)

  ABORT name.
  (Note: failure return from procedures, tasks,
         and subtasks; any procedure, task, and
         subtask in the calling sequence may be
         aborted.)