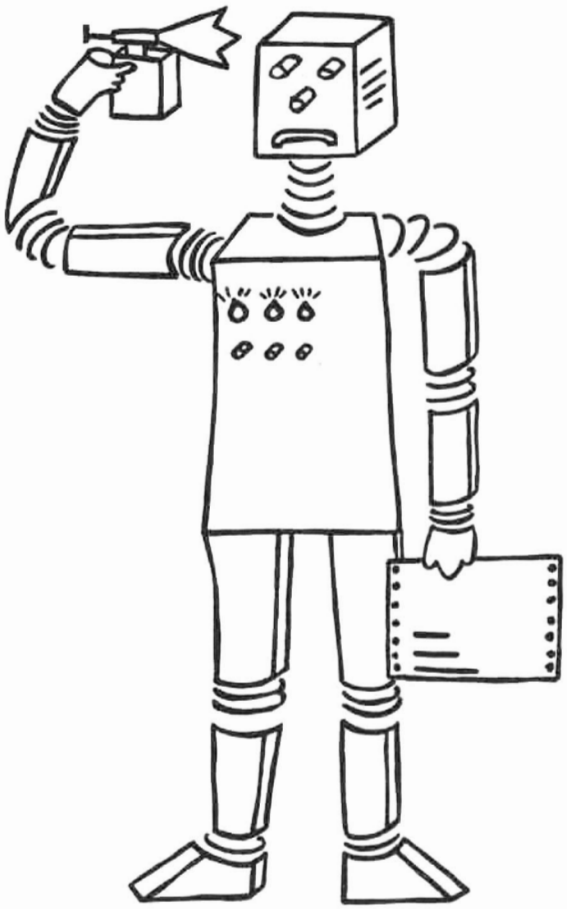
SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Integrated Software Development and Verification: A Case Study Using the SPESY System

Christoph Beierle
Horst Lichter
Walter Olthoff
Angelika Voss

Memo SEKI 85-11

Integrated Software Development and Verification: A Case Study Using the SPESY System

Christoph Beierle
Horst Lichter
Walter Olthoff
Angelika Voss

SEKI Memo SEKI-85-11

University of Kaiserslautern
FB Informatik
P.O. 3049
6750 Kaiserslautern
FRG

Contents

1. Introduction

The SPESY system is the central component of the Integrated Software Development and Verification (ISDV) system. The ISDV project was carried out at the Universities of Bonn, Kaiserslautern, and Karlsruhe and was supported by the Bundesministerium für Forschung und Technologie under contract IT.8302363.

SPESY supports the development of verified software from requirements specifications to Pascal programs according to the "stepwise refinement" and "verify while develop" paradigms. The methodology is elaborated in [BV 85] covering the range from high level axiomatic to intermediate level algorithmic specifications via refinements and implementations. All these development steps are expressed in the algebraic specification development language ASPIK. The realization of algorithmic specifications in an extension of Pascal by modules and generic features is described in [Olt 86]. Surveys of the methodology are given in [BOV 86].

While this report does not comprise any of the papers mentioned nor the complete description of SPESY in the manual [SBV 85], it is rather intended as a supplement. Its purpose is twofold: It can be used at the terminal getting acquainted with SPESY since it contains repeatable protocols of SPESY sessions. Besides it contains a complete example of a development from axiomatic specifications via algorithmic ones to modules in ModPascal, which exhibits many interesting features, as a demonstration of the feasability of our approach to software development.

2. The SPEcification development SYstem (SPESY)

2.1. System Overview

SPESY is a support environment for the development of algebraic specifications (specs) into programs and it is the central component of the Integrated Software Development and Verification System (ISDV System, [BGGORV 83], [BOV 86]). The ISDV System supports hierarchical development of programs according to the paradigms 'stepwise-refinement' and 'verify-while-develop'; it provides appropriate tools for all phases of the software development process ranging from formal specifications to imperative Pascal programs. The general proceeding when working with SPESY can be divided into a sequence of abstract and concrete steps (fig. 2.1.-1).

The abstract steps start from a formal requirements definition (axiomatically) and end with a constructively defined model (algorithmically). These steps can be distinguished into refinement or implementation steps. The former replace parts of specifications by more precise requirements and thereby reduce the set of models. For example, the replacement of axiomatically defined operations by algorithmically defined operations would constitute a refinement step. Implementation steps replace abstract specifications by more concrete ones. The set of models is changed such that all computations valid in the old set of models can be simulated in the new set of models.

The concrete steps start from constructively defined specifications and end with executable Pascal programs. One can distinguish between realization steps and (pre)compilation steps: first specifications are connected to module constructs written in ModPascal, an extension of standard Pascal by a module and parameterization concept. The specification hierarchy is remodelled as module hierarchy. Then precompilation and compilation steps generate the executable code.

All abstract steps are expressed in the algebraic specification language ASPIK ([BV 83a], [BV 85]). An ASPIK specification defines a class of algebras via its signature (sort and operation symbols), its properties (first order predicate calculus formulas) and its algorithmic definitions. The specifications are structured hierarchically; the parameterization concept allows to instantiate arbitrary used specifications ([BGV 83]). Another kind of objects are so-called map objects which are used in the definition of refinement steps and implementation steps.

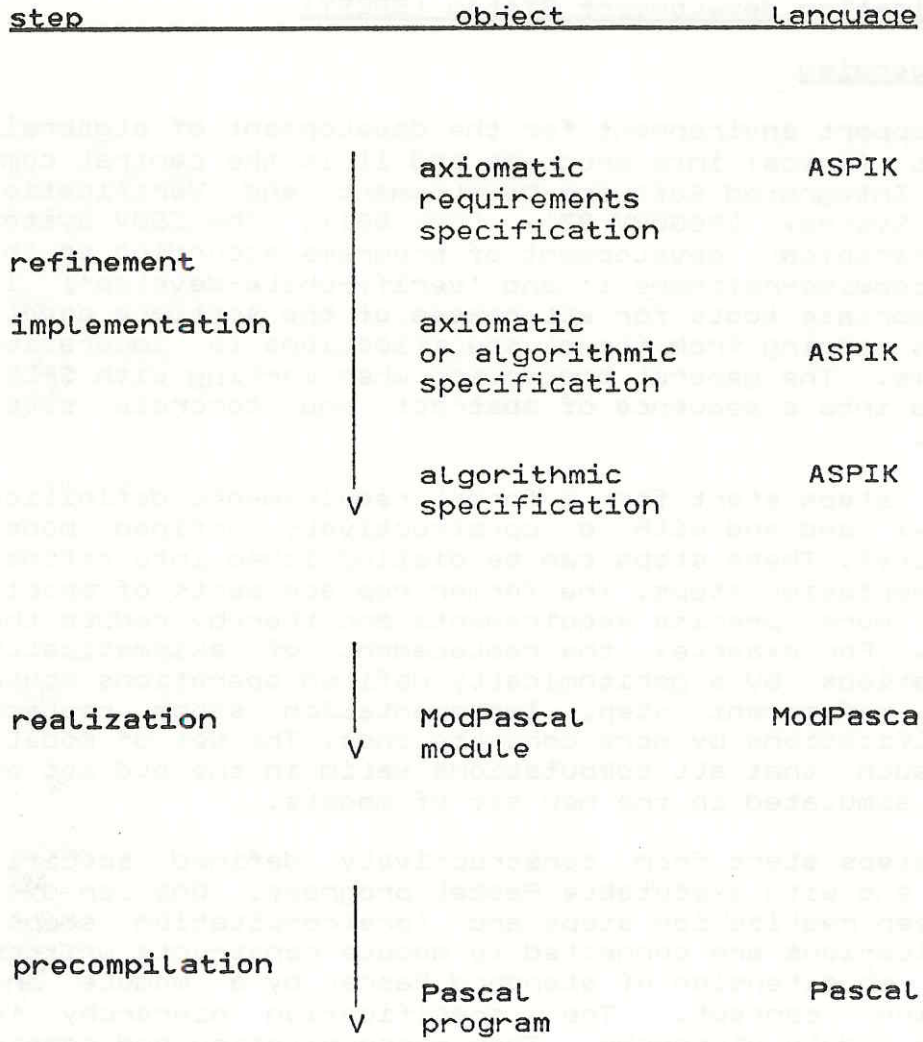


Fig. 2.1.-1: Stepwise Software Development

ASPIK supports two structuring mechanisms: The USE-relation allows the hierarchical composition of specifications and maps. Especially for specifications this means that they may contain occurrences of sorts and operations which are defined in some used specification.
The 'Parameterization-By-Use' concept ([BV 83a]) does not distinguish between formal and actual parameters at definition time of a specification. Only when an instance has to be generated the association to actual parameter specifications indicates which subspecifications are considered as formal parameters.

SPESY is a multi-user system which allows to assign to each user a collection of separate name spaces for identification of his objects. Additionally there is a set of system objects that are

accessible by all users.

To administrate all objects SPESY uses a sophisticated file system. There are private files which are accessible only by their owners. System files are public; they contain standard objects that are useful in various development scenarios (data structures like lists, stacks, queues etc.) or that contain objects that are of central importance. For example, there is a system file SYS.BOOLFILE that contains the specification BOOL. BOOL is a specification being used by every other specification treated by SPESY.
A file constitutes a name space in which names of objects have to be unique. Files may be composed hierarchically whenever it causes no name conflicts. Correctness conditions are associated to the objects in a file concerning for example the context-sensitive and syntactical correctness or involvation in proof tasks.

Beside subsystems for interactive syntax-oriented input and editing of objects the following additional tools are available:

- A symbolic interpreter for testing of constructive specifications (currently in a stand-alone version, [KRST 83]).

- A program transformation system for recursion removal (currently in a stand-alone version, [Gei 84], [Pet 83], [Ge 83]).

- A connection to proof systems which enables the user to pass proof tasks to connected provers and to integrate the results of proof efforts in SPESY. Currently there are two provers connected to SPESY:
  The resolution-based proof system "Markgraf Karl Refutation Procedure" ([BES 81]) and the rewrite rule based "Rewrite Rule Laboratory" [Tho 84] which is used in consistency proofs.

- A subsystem to realize constructive ASPIK specifications by ModPascal modules (realization level; [BEORSW 85]).

The tools are offered on various levels depending on the kind of the actual object. There is also strong support of information retrieval functions in SPESY. For example, via simple commands a user can get information about names of his files or objects, their semantical status, about access rights, which he has on objects or which he has given to other users, and many others. There is also a message system that supports the communication between users.

2.2. Working with SPESY

The functionality of SPESY is distributed over several levels which are hierarchically structured (c.f. Figure 2.2.-2). The main commands of the levels will be presented below. Levels are left by use of the END command that saves the changes, or by the ABORT command that restores the state of entering the level.

The top level is the system level. It comprises the system administration and the data management. System administration

functions and commands may only be invoked by the system manager.

The next lower level is the file level. It is entered via the OPEN command. The user can either open an already existing file or create a new one; in the latter case he has additionally to define the environment of hierarchically lower files including SYS.BOOLFILE, which defines the initial name space of the newly created file. The objects that are administrated by the file level are the objects of the opened file (specifications and maps).

Below the file level there are several levels. For specifications and maps SPESY provides an editor level (command EDIT) for syntax-oriented manipulation of objects. From the edit level the user can call the CHECK system that performs context-sensitive checks on recently manipulated objects; results of the checker are established in the semantical status of objects.

The input level is accessible either from the file or the edit level (command INPUT). INPUT may be executed in interactive as well as in autonomous mode (where an object definition is read from a text file). Only syntactically correct objects are accepted and stored in the data base. A complete description of correctness conditions may be found in [Lic 85] (for specification objects) and [Spa 85] (for map objects). To enter objects without immediate context-sensitive checking, one has to use the edit level.

Another editor and input level for implementation of specifications have not yet been implemented.

A third sublevel of the file level is the realization level (command RL). It deals with ModPascal objects or representation objects which are both used to realize (= correctly implement in ModPascal) constructive ASPIK specifications. Since the realization level is entered via an opened file the name space of this file is visible. Sublevels are the ModPascal Programming System (MPPS; dedicated to input, editing, precompiling and compiling of ModPascal objects (c.f. [Eck 84], [HR 86])), the Representation Object Programming System (ROPS; dedicated to input and editing of representation objects (c.f. [BR 85])), and the Compatibility Checker (CC; dedicated to the correctness check of a realization (c.f. [Wei 85])).

Further tools and interfaces are available at the file level. In order to carry out proof tasks the user can access connected proof systems (see above) by employing the PROOVE command. Typical proof tasks are termination proofs, consistency proofs or deduction proofs from which semantical properties of objects are derived. It is also possible to establish semantical properties directly by the user which speeds up software development in applications where automatic verification is not necessary.
A previous version of SPESY also allowed the use of an interpreter for algorithmic specifications and of a program transformation tool for recursion removal; in the actual version these accesses are still subject of ongoing implementation work.

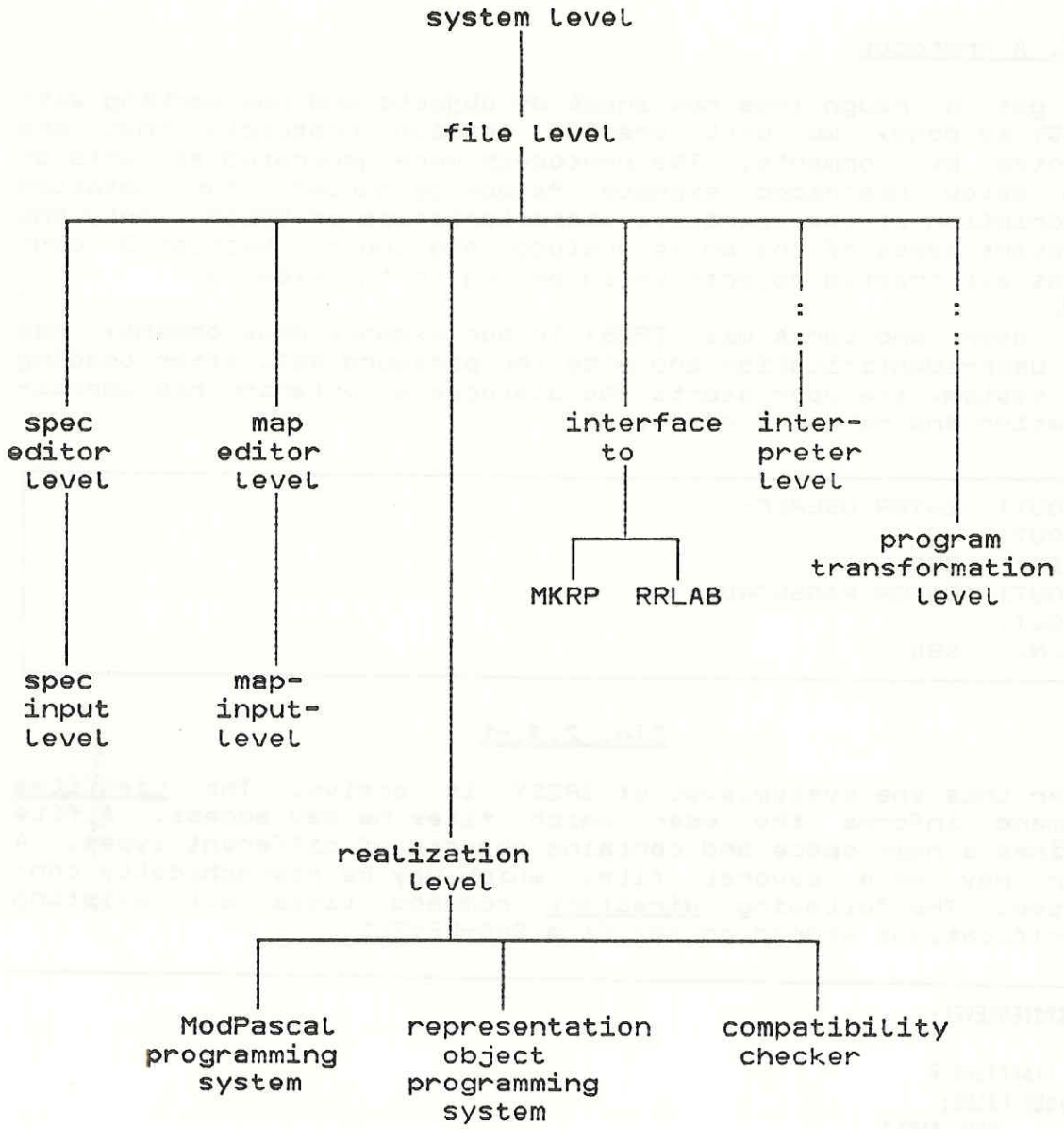


Fig. 2.1.-2: Command Levels of SPESY

2.3. A Protocol

To get a rough idea how input of objects and how working with SPESY is done, we will present session protocols that are annoted by comments. The protocols were generated as parts of the below described example "stack-by-queue". For detailed description of the operative characteristics of SPESY, only the relevant parts of the whole protocol are shown. Section 3 contains all created objects which belong to the example.

The user who works with SPESY in our example development, has the user-identification and also the password SBQ. After loading the system, the user starts the dialogue by entering his identification and personal password.

```
(OUT)  ENTER USERID:
(OUT)
(IN)   SBQ
(OUT)  ENTER PASSWORD:
(OUT)
(IN)   SBQ
```

Fig. 2.3.-1

After this the systemlevel of SPESY is active. The listfiles command informs the user which files he may access. A file defines a name space and contains objects of different types. A user may have several files which may be hierachically connected. The following directory command lists all existing specifications stored on the file SBQ-LEVEL1.

```
(OUT)  SYSTEMLEVEL:
(OUT)
(IN)    listfiles @
(OUT)  YOUR FILES:
(OUT)        SBQ-LEVEL1
(OUT)        SBQ-LEVEL2
(OUT)        SBQ-LEVEL3
(OUT)        SBQ-LEVEL4
(OUT)
(OUT)  SYSTEMLEVEL:
(OUT)
(IN)    directory sbq-level1 spec
(OUT)  SPECS ON SBQ-LEVEL1:    ELEM
```

Fig. 2.3.-2

In section 3, objects will be introduced as belonging to certain (logical) levels of our example (which should not be confused with the 'SPESY input level' or 'SPESY editor level').
To complete level 1 of our example, file SBQ-LEVEL1 will be opened for creating the specification LIMITED-LIFO using the input-system.

```
(OUT)  SYSTEMLEVEL:
(OUT)
(IN)    open sbq-level1
(OUT)  FILE OPENED !
(OUT)  MESSAGE(S) FOR SBQ-LEVEL1: TO COMPLETE THIS LEVEL YOU HAVE TO CREATE THE SPEC LIMITED-LIFO
(OUT)  FILELVEL:
(OUT)
(IN)    input spec limited-lifo
(OUT)          ************************************************************
(OUT)          ***                                                      ***
(OUT)          ***                  S P E C - E D I T O R               ***
(OUT)          ***                VERSION VOM:  10. 12. 1984            ***
(OUT)          ***                                                      ***
(OUT)          ************************************************************
(OUT)  ***************    I N P U T L E V E L     FOR      SPECS     ***************
(OUT)  ***************                VERSION :  02.01.1985           ***************
(OUT)  ENTER COMMENT OR ;
```

Fig. 2.3.-3

The input-level will be entered via the editor-level. The system expects the input of the specification LIMITED-LIFO. Syntactical and semantical errors which may occur in the user input are detected. The following protocol which describes the creation of the specification LIMITED-LIFO will show the input of a specification and the functionality of the inputlevel.

```
(OUT)  ENTER COMMENT OR ;
(OUT)
(IN)   /* loose specification of a limited container behaving lifo-like as long as it is not full */
(OUT)  use
(OUT)  ENTER SPECTERM COMMENT
(OUT)
(IN)   elem ; sorts
(OUT)  ENTER SORTID COMMENT OR ;
(OUT)
(IN)   container ; ops
(OUT)  ENTER OP-HEADER COMMENT OR ;
(OUT)
(IN)    into : container elem --> conteiner
(OUT)  *** THE FOLLOWING SORTIDS ARE NEITHER IN THE INTERFACE NOR IN SORTS-CLAUSE ***
(OUT)   CONTEINER
(OUT)  ENTER OP-HEADER COMMENT OR ;
(OUT)
(IN)    into : container elem --> container, last-in : container --> elem
(OUT)  ENTER OP-HEADER COMMENT OR ;
(OUT)
(IN)   first-out : container --> elem , filled? : container --> bool
```

```
(OUT)  ENTER OP-HEADER COMMENT OR ;
(OUT)
(IN)   ;
(OUT)  props
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)   help
```

Fig. 2.3.-4

Using the help command the user can get informations about the syntax of the actual clause at every prompt-position. Focussed on the property-clause a sequence of help or ? commands leads for example, to the following information:

```
(OUT)  PROPERTY ::= [LABEL] ATTRIBUTE | QUANTIFICATION
(OUT)
(IN)    ?
(OUT)  ATTRIBUTE ::= ATRIBUTENAME : OPID
(OUT)
(IN)    ?
(OUT)  ATTRIBUTENAME ::= COIMPLICATION | QUANTIFIER VARID[, VARID] : SORTID QUANTIFICATION
(OUT)
(IN)    ?
(OUT)  COIMPLICATION ::= IMPLICATION | IMPLICATION <==> IMPLICATION
(OUT)  IMLPICATION   ::= DISJUNCTION | DISJUNCTION ==> DISJUNCTION
(OUT)  QUANTIFICATION ::= COIMPLICATION | QUANTIFIER VARID[, VARID] : SORTID QUANTIFICATION
(OUT)
(IN)    ?
(OUT)  COIMPLICATION ::= IMPLICATION | IMPLICATION <==> IMPLICATION
(OUT)  IMLPICATION   ::= DISJUNCTION | DISJUNCTION ==> DISJUNCTION
(OUT)  DISJUNCTION   ::= CONJUNCTION | CONJUNCTION ' | ' CONJUNCTION
(OUT)  CONJUNCTION   ::= NEGATION | NEGATION & NEGATION
(OUT)  NEGATION      ::= ATOMIC-FORMULA | _ ATOMIC-FORMULA
(OUT)  ATOMIC-FORMULA ::= (QUANTIFICATION) | ATOM
(OUT)  ATOM          ::= TERM | EQUATION | INEQUATION
(OUT)  EQUATION      ::= TERM == TERM
(OUT)  INEQUATION    ::= TERM =|= TERM
```

Fig. 2.3.-5

The property-clause consists of formulas in first order predicate calculus which constrain the introduced operations by stating relations between them. The following protocol shows the input of two properties of a LIFO structure; note that the system detects several errors in the user input.

```
(OUT)  ENTER PROPERTY COMMENT OR ;
(IN)    all c: container all e elem ¬filled(c) ==> (last-in(into(c,a)) == e & first-aut( into(c,e)  == c)
(OUT)
(OUT)  *** MISSING COLON AFTER VARID-LIST ***
(OUT)   E,ELEM,_FILLED(C),==>,(LAST-IN(INTO(C,A)),==,E,&,FIRST-AUT(,INTO(C,E),==,C)
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)    all c: container all e:elem ¬filled?(c) ==> (last-in(into(c,a)) == e & first-aut (into(c,e)  == c)
```

```
(OUT)
(OUT)  *** INPUT IS IGNORED AT :     ) ) == E & FIRST-AUT ( INTO ( C,E ) . . .
(OUT)  *** THE FOLLOWING SYMBOL IS NEITHER A VALID VARIABLE OR CONSTANT , NOR OPID ***
(OUT)  A
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)    all c: container all e:elem ¬filled?(c) ==> (last-in(into(c,e)) == e & first-aut (into(c,e)  == c)
(OUT)
(OUT)  *** INPUT IS IGNORED AT :     == C )
(OUT)  *** MISSSING CLOSING-PARENTHESIS BEHIND THE FOLLOWING PREFIX-OPID ***
(OUT)  FIRST-OUT
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)    all c: container all e:elem ¬filled?(c) ==> (last-in(into(c,e)) == e & first-out (into(c,e)) == c)
(OUT)
(OUT)  *** THE FOLLOWING TERMS ARE OF DIFFERENT DOMAIN ***
(OUT)    (FIRST-OUT (INTO (C)
(OUT)                     (E)))
(OUT)    (C)
(OUT)
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)   all c:container all e:elem ¬filled?(c) ==> (last-in(into(c,e)) == e & first-out (into(c,e)) == c)
(OUT)  ENTER PROPERTY COMMENT OR ;
(OUT)
(IN)   end
(OUT)  ***************          E N D          I N P U T          ***************
(OUT)  CPU TIME USED : 40378 ms.
```

Fig. 2.3.-6

The input system is left by entering the end command. If at least one clause is not syntactically or not context-sensitively correct, the editor for specifications is activated automatically. The specification is listed whenever the dialog controlparameter LIST is set.

```
(OUT)  POSITION: TOP OF SPEC LIMITED-LIFO    LIST
(OUT)  spec LIMITED-LIFO
(OUT)      /* LOOSE SPECIFICATION OF A LIMITED CONTAINER BEHAVING LIFO-LIKE AS    */
(OUT)      /* LONG AS IT IS NOT FULL */
(OUT)     use   ELEM ;
(OUT)     sorts CONTAINER;
(OUT)     ops   LE: CONTAINER CONTAINER  --> BOOL
(OUT)           GE: CONTAINER CONTAINER  --> BOOL
(OUT)           INTO: CONTAINER ELEM  --> CONTAINER
(OUT)           LAST-IN: CONTAINER  --> ELEM
(OUT)           FIRST-OUT: CONTAINER  --> ELEM
(OUT)           FILLED?: CONTAINER  --> BOOL;
(OUT)     props [PROP1] ALL C:CONTAINER   ALL E:ELEM
(OUT)                     ¬FILLED?(C)
(OUT)                      ==> LAST-IN(INTO(C,E)) == E & FIRST-OUT(INTO(C,E)) == C;
(OUT)  endspec
```

Fig. 2.3.-7

When declaring the operation First-Out the user entered the wrong sort. He can correct this mistake using the change command of the editor.

```
(OUT)  SPEC-EDIT-LEVEL:
(OUT)
(IN)    ops ch 5 first-out : container --> container
(OUT)
(OUT)  POSITION: TOP OF OPS-CLAUSE OF SPEC LIMITED-LIFO    LIST
(OUT)     ops   LE: CONTAINER CONTAINER  --> BOOL
(OUT)           GE: CONTAINER CONTAINER  --> BOOL
(OUT)           INTO: CONTAINER ELEM  --> CONTAINER
(OUT)           LAST-IN: CONTAINER  --> ELEM
(OUT)           FIRST-OUT: CONTAINER  --> CONTAINER
(OUT)           FILLED?: CONTAINER  --> BOOL;
(OUT)
(OUT)  SPEC-EDIT-LEVEL:
(OUT)
(IN)    end
(OUT)  EDITOR-LEVEL ENDED.
(OUT)  FILELEVEL:
(OUT)
(IN)    end
```

Fig. 2.3.-8

The filelevel is left by entering the end command and now the systemlevel is re-entered. Because of using the change command on the editorlevel the specification's controlvector is set to "unknown". To test the syntactical correctness of the object the specification LIMITED-LIFO is checked by the checker for specifications. It tests the syntactical and context-sensitive constraints of a specification and sets the controlvector to "ok" if no errors are detected. If an error is found, the position in the faulty clause is printed on the screen. Now the user may correct the mistake using the editor; after this he may test the correctness of the object again. The activation of the checksystem is possible from the filelevel as well as from the editor.

```
(OUT)  FILELEVEL:
(OUT)
(IN)   check limited-lifo
(OUT)  % D800 ERASE FILE CHECKFILE-LIMITED-LIFO.00
(OUT)  ***** C H E C K S Y S T E M FOR SPECS STARTED *****
(OUT)  *****      VERSION: 25.02.1985              *****
(OUT)  *****      USE-CLAUSE CHECKED               *****
(OUT)  *****      SOPU-CLAUSE CHECKED              *****
(OUT)  *****      OPPU-CLAUSE CHECKED              *****
(OUT)  *****      PROP-CLAUSE CHECKED              *****
(OUT)  *****      E N D C H E C K S Y S T E M      *****
(OUT)  CPU TIME USED: 9988 ms.
(OUT)  % D800 ERASE FILE CHECKFILE-LIMITED-LIFO
(OUT)
(OUT)  ********************************************
(OUT)  * YOUR SPECIFICATION LIMITED-LIFO IS CORRECT *
(OUT)  ********************************************
(OUT)  FILELEVEL:
```

```
(OUT)
```

Fig. 2.3.-9

Now suppose that part of the LIMITED-STACK hierarchy described in section 3 has already been established. File SBQ-LEVEL2 already contains the necessary specifications LIMIT and NAT as well as the not yet completely created specification LIMITED-STACK. This information can be seen by performing the directory command and the following listing of the ok-status of LIMITED-STACK.

```
(OUT)  SYSTEMLEVEL:
(OUT)
(IN)    open sbq-level2
(OUT)  FILE OPENED !
(OUT)  MESSAGEFILE EMPTY
(OUT)  FILELEVEL:
(OUT)
(IN)    directory sbq-level2 spec
(OUT)  SPECS ON SBQ-LEVEL2 : LIMIT NAT LIMITED-STACK
(OUT)  FILELEVEL:
(OUT)
(IN)    list limited-stack ok
(OUT)           status of specification:   LIMITED-STACK
(OUT)
(OUT)                          syntax: -                    semantics
(OUT)                                                          |
(OUT)                                                          |— terminating: ?
(OUT)                                                          |— consistent : ?
(OUT)                                                          |— derivable  : ?
(OUT)            interface: +                rest: -
(OUT)                |                         |
(OUT)                |                         |
(OUT)                |                         |
(OUT)              use:   +                    |
(OUT)              sorts: +   props: 0+   constructors: +    def.constructors: 0-
(OUT)              ops:   +               auxiliaries : 0+   private ops     : 0+
(OUT)                                     def.-aux.   : 0+   define ops      : 0-
(OUT)                                     def.-carrier: 0-
(OUT)
(OUT)                0 = empty
(OUT)                + = ok
(OUT)                - = not ok
(OUT)                ? = unknown
(OUT)  FILELEVEL:
(OUT)
```

Fig. 2.3.-10

Using the edit command the editor for specifications is activated, and the already created first part of the specification is listed.

```
(OUT)  FILELVEL:
(OUT)
(IN)    edit spec limited-stack
(OUT)          ************************************************************
(OUT)          ***                                                      ***
(OUT)          ***                  S P E C - E D I T O R               ***
(OUT)          ***                VERSION VOM:  10. 12. 1984            ***
(OUT)          ***                                                      ***
(OUT)          ************************************************************
(OUT)  POSITION: TOP OF SPEC LIMITED-STACK    LIST
(OUT)  spec LIMITED-STACK
(OUT)      /* STANDARD ALGORITHMIC DEFINITION OF A LIMITED-STACK.PUSH ON A FULL   */
(OUT)      /* STACK, POP OR TOP OF AN EMPTY STACK RESULT IN ERRORS */
(OUT)     use   ELEM
(OUT)           LIMIT ;
(OUT)     sorts STACK;
(OUT)     ops   EMPTY:  --> STACK
(OUT)           EMPTY?,FULL: STACK  --> BOOL
(OUT)           PUSH: STACK ELEM  --> STACK
(OUT)           POP: STACK  --> STACK
(OUT)           TOP: STACK  --> ELEM
(OUT)           LT,GT: STACK STACK  --> BOOL;
(OUT)  spec-body
(OUT)     constructors EMPTY
(OUT)                  PUSH;
(OUT)
(OUT)  SPEC-EDIT-LEVEL:
(OUT)
(IN)    input
```

Fig. 2.3.-11

The entering of the command input branches to the inputlevel. It focusses on the currently last clause of the object. Now the user is able to complete the specification LIMITED-STACK.

```
(OUT)  ***************   I N P U T L E V E L     FOR      SPECS     ***************
(OUT)  ***************                VERSION :  02.01.1985          ***************
(OUT)  ***************    START   WITH   THE   FOLLOWING   CLAUSE    ***************
(OUT)  CONSTRUCTOR-CLAUSE
(OUT)  ENTER CONSTRUCTOR COMMENT OR ;
(OUT)
(IN)   ;
(OUT)  auxiliaries
(OUT)  ENTER AUXILIARY COMMENT OR ;
(OUT)
(IN)   depth: stack --> nat ;
(OUT)  !!! SPESY CAN'T DECLARE THE EQ-OPERATION FOR THE FOLLOWING SORTS !!!
(OUT)  STACK
(OUT)  define-auxiliaries
(OUT)  ENTER AUXILIARY-DEFINITION COMMENT
(OUT)
(IN)   depth(st) = case st is
(OUT)  * EMPTY :
(OUT)
(IN)   nat.zero
(OUT)  * PUSH ( ST0 , EL0 ) :
```

```
(OUT)
(IN)   suc(depth(st0))
(OUT)  ESAC
(OUT)  ENTER CARRIER-DEFINITION COMMENT OR ;
(OUT)
(IN)   is-stuck(st) =
(OUT)
(OUT)  *** INPUT IS IGNORED AT :     ( ST ) =
(OUT)  *** INVALID CHARACTERISTIC PREDICATE IN DEFINE-CARRIERS-CLAUSE ***
(OUT)   IS-STUCK
(OUT)  ENTER CARRIER-DEFINITION COMMENT OR ;
(OUT)
(IN)   is-stack(st) = if st then
(OUT)
(OUT)  *** INPUT IS IGNORED AT :     ST THEN
(OUT)  *** MISSING CASE IN DEFINITION OF CHARACTERISTIC PREDICATE ***
(OUT)
(OUT)  ENTER CARRIER-DEFINITION COMMENT OR ;
(OUT)
(IN)   is-stack(st) = case st is
(OUT)  * PUSH ( ST0 , EL0 ) :
(OUT)  IF ( NOT (IS-STACK (ST0)) )
(OUT)  THEN FALSE
(OUT)  ELSE
(OUT)
(IN)   (depht(st0) lt limit)
(OUT)
(OUT)  *** INPUT IS IGNORED AT :     ( ST0 ) LT LIMIT )
(OUT)  *** THE FOLLOWING SYMBOL IS NEITHER A VALID VARIABLE OR CONSTANT , NOR OPID ***
(OUT)  DEPHT
(OUT)  ENTER CARRIER-DEFINITION COMMENT OR ;
(OUT)
```

Fig. 2.3.-12

The last part of the protocol explicates the system's capability to generate large parts of a specification automatically. The user completes the input of the specification with the following:

```
(IN)   is-stack(st) = case st is * push(st0,el0) : if (not(is-stack(st0))) then false
(IN)    else (depth(st0) nat.lt limit)
(OUT)  ENTER THE MISSING CASE-PARTS OR 'OTHERWISE'
(OUT)
(IN)   otherwise : true
(OUT)  ESAC
(OUT)  define-constructor-ops
(OUT)  ENTER CONSTRUCTOR-DEFINITION COMMENT
(OUT)
(IN)   empty = empty
(OUT)
(OUT)  *** MISSING ' * ' IN CONSTRUCTOR DEFINITION ***
(OUT)
(OUT)  ENTER CONSTRUCTOR-DEFINITION COMMENT
(OUT)
(IN)   empty = * empty
(OUT)  ENTER CONSTRUCTOR-DEFINITION COMMENT OR ;
(OUT)
```

```
(IN)   push(st0,el0) = if  (depth(st0) nat.lt limit) then * push(st0,el0) else error-stack
(OUT)  private-ops
(OUT)  ENTER PRIVATE-OPERATION COMMENT OR ;
(OUT)
(IN)   ;
(OUT)  define-ops
(OUT)  ENTER OP-DEFINITION COMMENT
(OUT)
(IN)   ;
(OUT)
(OUT)  *** FIRST YOU MUST DEFINE THE FOLLOWING OPERATIONS ***
(OUT)   EMPTY?  FULL  POP  TOP
(OUT)  ENTER OP-DEFINITION COMMENT
(OUT)
(IN)   full(st) = bool.not((depth(st) nat.lt limit))
(OUT)  ENTER OP-DEFINITION COMMENT OR ;
(OUT)
(IN)   empty?(st) = case st is
(OUT)  * EMPTY :
(OUT)
(IN)   true otherwise : false
(OUT)  ESAC
(OUT)  ENTER OP-DEFINITION COMMENT OR ;
(OUT)
 (IN)   pop(st) = case st is * push(st1,el1) : st1 otherwise : error-stack
 (OUT)  ESAC
 (OUT)  ENTER OP-DEFINITION COMMENT OR ;
 (OUT)
 (IN)   top(st) = case st is
 (OUT)  * EMPTY :
 (OUT)
 (IN)   error-elem
 (OUT)  * PUSH ( ST0 , EL0 ) :
 (OUT)
 (IN)   el0
 (OUT)  ESAC
 (OUT)  ENTER OP-DEFINITION COMMENT OR ;
 (OUT)
 (IN)   end
 (OUT)  ***************          E N D          I N P U T          ***************
 (OUT)  CPU TIME USED : 124218 ms.
```

Fig. 2.3.-13

Now all objects of the second development level are created (c.f section 3). Next, the user requires information and the file SBQ-LEVEL2 is closed. Next, the user requires information about the use-relation of the specification LIMITED-STACK by using the List command.

```
(OUT)   FILELEVEL:
(OUT)
(IN)    list limited-stack usrel
(OUT)
(OUT)   spec LIMITED-STACK: all used specifications
(OUT)
(OUT)                        1 2 3 4 5
(OUT)     1          BOOL *
```

```
(OUT)     2            NAT * *
(OUT)     3           ELEM *   *
(OUT)     4          LIMIT * *   *
(OUT)     5 LIMITED-STACK * * * * *
(OUT)
(OUT)  FILELEVEL:
```

Fig. 2.3.-14

The map LIMIT-INCREASE→LIMIT+1 shall be generated on development level 3 of the example. To do so, the user performs the input command on the filelevel to activate the inputsystem for maps and enters the wanted map.

```
(OUT)     FILELEVEL:
(OUT)
(IN)      input map limit -increase -> limit+1
(OUT)      INPUT FOR MAPS AT CMT-CLAUSE STARTED :
(OUT)      CMT
(OUT)
(IN)      /* replace limit by limit+1 */
(OUT)      IS
(OUT)
(IN)      refinement
(OUT)      BASE
(OUT)
(IN)      help
(OUT)      SPECTERM /* COMMENT */ , ...
(OUT)      OR END BY  ;
(OUT)
(IN)      ;
(OUT)      USE
(OUT)
(IN)      ;
(OUT)     ALL USED SPECTERMS FROM SOURCE HAVE TO BE IN THE BASE-CLAUSE
(OUT)     OR EQUAL TO THE SOURCE OF AN USE-TERM
(OUT)      THE USE-CLAUSE IS NOT OK
(OUT)      SORTS
(OUT)
(IN)      elem = elem
(OUT)     ELEM = ELEM
(OUT)      ';' EXPECTED, THE MAP FOR THE SORTS IS COMPLETE :
(OUT)
(IN)      ;
(OUT)      OPS
(OUT)
(IN)     limit = suc-of-limit
(OUT)       ENDMAP ;
(OUT)      THE MAP IS NOT INTERFACE-OK
(OUT)      END OF MAP-INPUT: 1140 MSEC USED
```

Fig. 2.3.-15

The map is not interface-ok because a context sensitive condition requires that the used specification NAT has to be listed in the base-clause. This error is corrected in the editor for maps; after this the map will be tested for syntactical correctness by performing of the check command.

```
(OUT)     MAP-EDIT-LEVEL:
(OUT)
(IN)      base
(OUT)     MAP-EDIT: (LIMIT -INCREASE -> LIMIT+1) AT BASE :
(OUT)
(OUT)
(OUT)     MAP-EDIT-LEVEL:
(OUT)
(IN)      insert 1 nat
(OUT)     MAP-EDIT: (LIMIT -INCREASE -> LIMIT+1) AT BASE :
(OUT)
(OUT)        base
(OUT)            NAT ;
(OUT)     MAP-EDIT-LEVEL:
(IN)      check
(OUT)       CHECK OF THE MAPID-CLAUSE STARTED ...
(OUT)       CHECK OF THE IS-CLAUSE STARTED ...
(OUT)       CHECK OF THE BASE-CLAUSE STARTED ...
(OUT)       CHECK OF THE USE-CLAUSE STARTED ...
(OUT)       CHECK OF THE SORTS-CLAUSE STARTED ...
(OUT)       CHECK OF THE OPS-CLAUSE STARTED ...
```

Fig. 2.3.-16

To show the semantical correctness of the map, the MKRP-Prover will be activated. Thereby a proof task will be generated formulated in the input language of the prover. The prover shows that the properties of objects are derivable. To prove the consistency of the properties the parameter RRLAB is passed to the prove-command, which generates an interface to the RRLab.

3. The Stack-by-Queue Example

This is an example for software development with the ISDV-system

- beginning with a loose axiomatic specification for a limited lifo-stack
- which is refined to a limited stack
- which is simulated by a limited queue
- which is instantiated with a concrete limit and concrete elements
- which is at last realized with ModPascal modules.

The development process is divided into five levels and four transitions (from one level to another). For each level we present the associated object hierarchy and the object definitions; for each transition we also present the verification conditions that are connected to it.

Level 1: comprises the LIMITED-LIFO hierarchy; a limited lifo-storage is loosely specified using axioms.

Level 2: presents the limited stack specification object; the LIMITED-STACK hierarchy is specified using fixed algorithmic specifications.

Transition 1-2: The refinement map (LIMITED-LIFO -FIX→ LIMITED-STACK) refines objects of level 1 to objects of level 2 by fixing LIMITED-LIFO to LIMITED-STACK.

Level 3: The LIMITED-QUEUE-CONSTRUCTION hierarchy; STACK operations are specified via an algorithmically specified limited queue object.

Transition 2-3: The implementation object =I:Stack implements objects of level 2 by 3, simulating LIMITED-STACK by LIMITED-QUEUE.

Level 4: The LIMITED-QUEUE-CONSTRUCTION {-POINTO→, -LIMIT100→} hierarchy instantiates the LIMITED-QUEUE-CONSTRUCTION hierarchy with limit 100 and nat-elements.

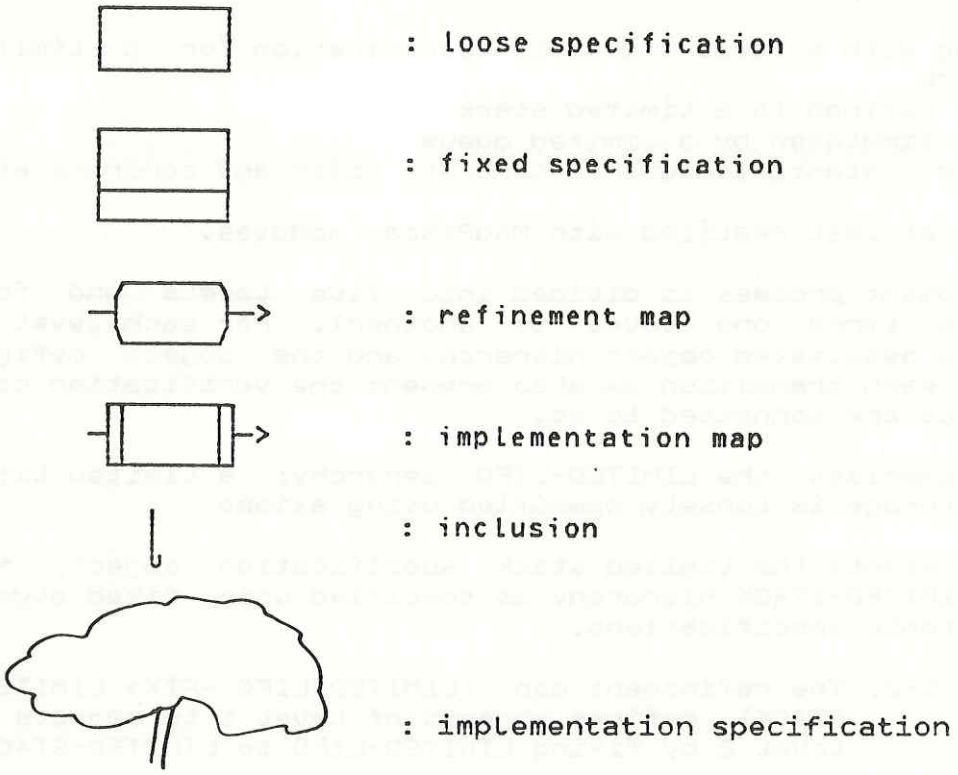
Transition 3-4: The refinement map object (LIMITED-QUEUE-CONSTRUCTION ——> LIMITED-QUEUE-CONSTRUCTION {-POINTO→, -LIMIT100→}) instantiates objects of level 3 to 4 by fixing the limit and the type of elements.

Level 5: Specification objects of level 4 are realized as ModPascal module or enrichment objects.

Transition 4-5: A set of representation objects realizes (=implements) objects of level 4 by objects of level 5.

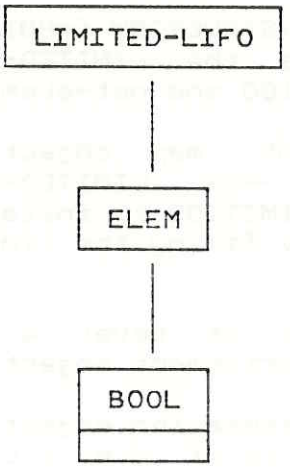
This chapter contains the objects of our example from level 1 to level 4. We will show you for each level the object-hierarchy and the transition from each level to the next one along with a

graphical representation.
To indicate the different objecttypes we will use the following symbols.



3.1. Level 1: Axiomatic LIMITED-LIFO specification

3.1.1. The Hierarchy



3.1.2. Object Definitions

```
spec BOOL
   sorts BOOL;
   ops   TRUE,FALSE:   --> BOOL
         NOT: BOOL   --> BOOL
         _AND_,_OR_,EQ-BOOL: BOOL BOOL   --> BOOL;
spec-body
   constructors TRUE
                FALSE;
   auxiliaries EQ-BOOL: BOOL BOOL   --> BOOL;
   define-auxiliaries
        EQ-BOOL(B1,B2) = case B1 is
                           * TRUE : case B2 is
                                      * TRUE : TRUE
                                     otherwise FALSE
                                    esac
                           * FALSE : case B2 is
                                       * FALSE : TRUE
                                      otherwise FALSE
                                     esac
                         esac;
   define-carriers
        IS-BOOL(B) = TRUE;
   define-constructor-ops
        TRUE = * TRUE
        FALSE = * FALSE;
   define-ops
        NOT(B) = case B is
                   * TRUE : FALSE
                   * FALSE : TRUE
                 esac
        B1 AND B2 = case B1 is
                      * TRUE : B2
                      * FALSE : B1
                    esac
        B1 OR B2 = case B1 is
                     * TRUE : B1
                     * FALSE : B2
                   esac;
endspec
```

```
spec ELEM
    /* LOOSE SPECIFICATION OF JUST A SET */
   use   BOOL ;
   sorts ELEM;
endspec
```

3.1. Level 1: Axiomatic LIMITED-LIFO specification

```
spec LIMITED-LIFO
    /* LOOSE SPECIFICATION OF A LIMITED CONTAINER BEHAVING        */
    /* LIFO-LIKE AS LONG AS IT IS NOT FULL  */
  use   ELEM ;
  sorts CONTAINER;
  ops   INTO: CONTAINER ELEM  --> CONTAINER
        LAST-IN: CONTAINER  --> ELEM
        FIRST-OUT: CONTAINER  --> CONTAINER
        FILLED?: CONTAINER  --> BOOL;
  props [PROP1] ALL C:CONTAINER   ALL E:ELEM
        ¬FILLED?(C)
        ==> LAST-IN(INTO(C,E)) == E & FIRST-OUT(INTO(C,E)) == C;
endspec
```

The symbol _ is used for logical negation.

3.1.3. Verification Conditions

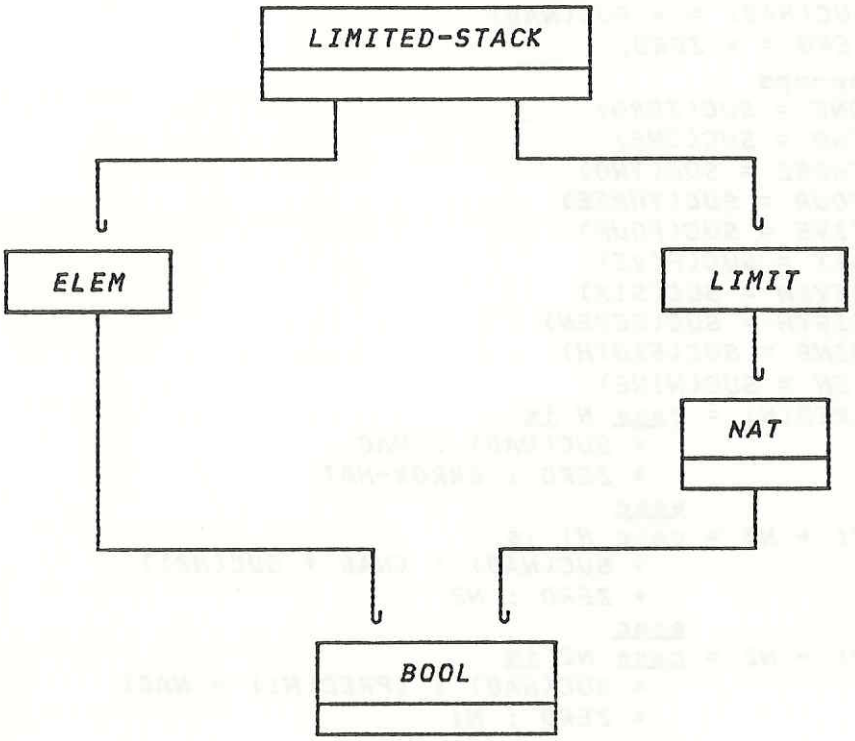
BOOL: all operations terminate

ELEM: -

LIMITED-LIFO: property PROP1 is consistent wrt. BOOL

3.2. Level 2: Algorithmic LIMITED-STACK specification

3.2.1. The Hierarchy



3.2.2. Object Definitions

```
spec NAT
    /* STANDARD ALGORITHMIC DEFINITION OF THE NATURAL NUMBERS */
   use    BOOL ;
   sorts  NAT;
   ops    ZERO:  --> NAT
          SUC,PRED: NAT  --> NAT
          MULT,EXP,_+_,_-_,_/_: NAT NAT  --> NAT
          ZERO?: NAT  --> BOOL
          _LT_,_LE_,_GT_,_GE_,EQ-NAT: NAT NAT  --> BOOL
          [__]: NAT NAT  --> NAT
          ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGTH,NINE,TEN:  --> NAT
   props [PROP1] ALL X,Y:NAT   ([ X,Y ]) == (MULT(X,TEN) + Y);
spec-body ------------------------------------------------------------
   constructors SUC, ZERO;
   auxiliaries EQ-NAT: NAT NAT  --> BOOL;
   define-auxiliaries
       EQ-NAT(N,O) = case N is
                       * SUC(NO) : case O is
                                     * SUC(OO) : EQ-NAT(NO,OO)
                                     otherwise FALSE
                                   esac
                       * ZERO : case O is
                                  * ZERO : TRUE
```

```
                                    otherwise FALSE
                                esac
                        esac;
define-carriers
    IS-NAT(N) = TRUE;
define-constructor-ops
    SUC(NA0) = * SUC(NA0)
    ZERO = * ZERO;
define-ops
    ONE = SUC(ZERO)
    TWO = SUC(ONE)
    THREE = SUC(TWO)
    FOUR = SUC(THREE)
    FIVE = SUC(FOUR)
    SIX = SUC(FIVE)
    SEVEN = SUC(SIX)
    EIGTH = SUC(SEVEN)
    NINE = SUC(EIGTH)
    TEN = SUC(NINE)
    PRED(N) = case N is
                * SUC(NA0) : NA0
                * ZERO : ERROR-NAT
              esac
    N1 + N2 = case N1 is
                * SUC(NA0) : (NA0 + SUC(N2))
                * ZERO : N2
              esac
    N1 - N2 = case N2 is
                * SUC(NA0) : (PRED(N1) - NA0)
                * ZERO : N1
              esac

    MULT(N1,N2) = case N1 is
                    * SUC(NA0) : case N2 is
                                   * SUC(NA1) :
                                       (N1 + MULT(N1,NA1))
                                   * ZERO : ZERO
                                 esac
                    * ZERO : ZERO
                  esac
    N1 / N2 = case N2 is
                * SUC(NA0) : case N1 is
                               * SUC(NA1) :
                                   if (((N1 - N2)) LT ZERO)
                                   then ZERO
                                   else (ONE + ((N1 - N2) / N2))
                               * ZERO : ZERO
                             esac
                * ZERO : ERROR-NAT
              esac
    EXP(N1,N2) = case N2 is
                   * SUC(NA0) : MULT(N1,EXP(N1,NA0))
                   * ZERO : ONE
                 esac
    ZERO?(N) = case N is
                 * SUC(NA0) : FALSE
                 * ZERO : TRUE
               esac
    N1 LT N2 = case N2 is
```

```
                    * SUC(NA0) : case N1 is
                                   * SUC(NA1) : (NA1 LT NA0)
                                   * ZERO : TRUE
                                 esac
                    * ZERO : FALSE
                  esac
      N1 LE N2 = NOT((N2 LT N1))
      N1 GT N2 = (N2 LT N1)
      N1 GE N2 = (N2 LE N1)
      [ N1,N2 ] = (MULT(N1,TEN) + N2);
endspec
```

```
spec LIMIT
    /* LOOSE SPECIFICATION OF A NON-ERROR-NATURAL NUMBER AS      */
    /* LIMIT   */
   use   NAT;
   ops   LIMIT:  --> NAT;
   props [PROP1] LIMIT =|= ERROR-NAT;
endspec
```

```
spec LIMITED-STACK
    /* STANDARD ALGORITHMIC DEFINITION OF A LIMITED-STACK. PUSH  */
    /* ON A FULL STACK, POP OR TOP OF AN EMPTY STACK RESULT IN   */
    /* ERRORS   */
   use   ELEM , LIMIT ;
   sorts STACK;
   ops   EMPTY:  --> STACK
         EMPTY?,FULL?: STACK  --> BOOL
         PUSH: STACK ELEM  --> STACK
         POP: STACK  --> STACK
         TOP: STACK  --> ELEM;
spec-body
   constructors EMPTY, PUSH;
   auxiliaries DEPTH: STACK  --> NAT;
   define-auxiliaries
       DEPTH(ST) = case ST is
                     * EMPTY : ZERO
                     * PUSH(STO,ELO) : SUC(DEPTH(STO))
                   esac;
   define-carriers
       IS-STACK(ST) = case ST is
                        * PUSH(STO,ELO) : if NOT(IS-STACK(STO))
                                          then FALSE
                                          else
                                              (DEPTH(STO) LT LIMIT)
                      otherwise TRUE
                    esac;
   define-constructor-ops
       EMPTY = * EMPTY
       PUSH(STO,ELO) = if (DEPTH(STO) LT LIMIT)
                       then * PUSH(STO,ELO)
                       else ERROR-STACK;
   define-ops
       EMPTY?(ST) = case ST is
                      * EMPTY : TRUE
                      * PUSH(STO,ELO) : FALSE
```

```
                esac
    FULL?(ST) = NOT((DEPTH(ST) LT LIMIT))
    POP(ST) = case ST is
                * EMPTY : ERROR-STACK
                * PUSH(STO,ELO) : STO
              esac
    TOP(ST) = case ST is
                * EMPTY : ERROR-ELEM
                * PUSH(STO,ELO) : ELO
              esac;
endspec
```

3.2.3. Verification Conditions

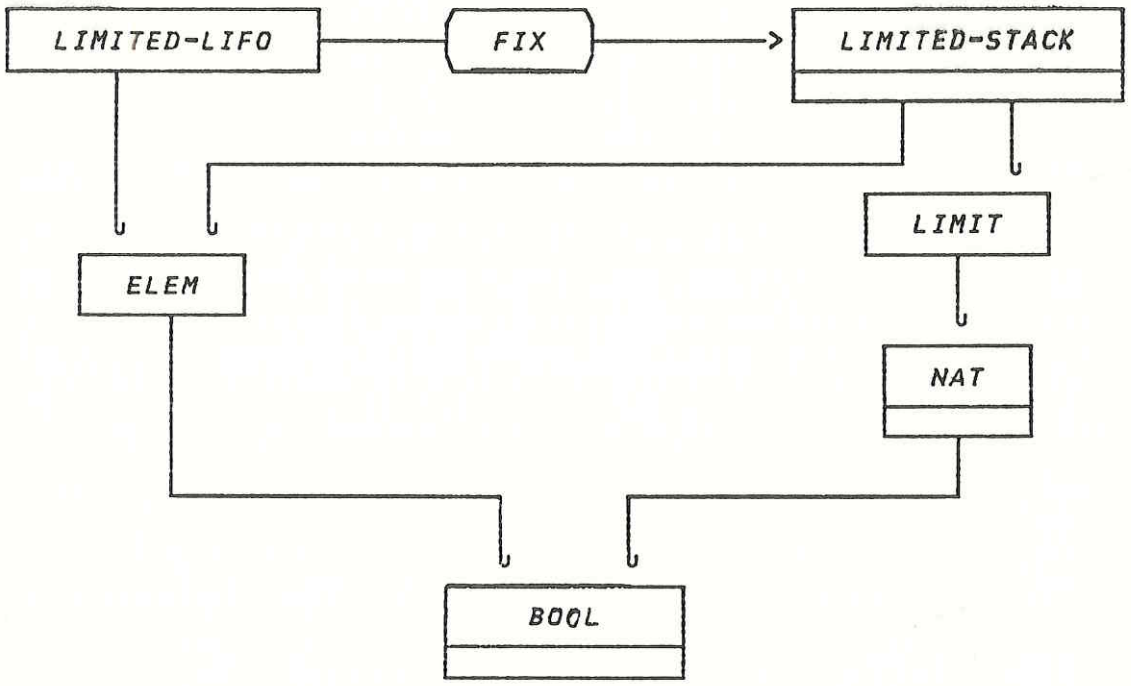
NAT: all operations terminate;
property PROP1 is (inductively) deducible by the algorithmic operation definitions of NAT and the hierarchy below

LIMIT: property PROP1 is consistent wrt. the hierarchy below

LIMITED-STACK: all operations terminate

3.3. Transition Level 1 to Level 2

3.3.1. The Hierarchy



3.3.2. Object Definitions

```
map (LIMITED-LIFO -FIX-> LIMITED-STACK)
    /* FIX LIMITED-LIFO IN THE LIMITED-STACK MODEL */
   is REFINEMENT;
   base
       ELEM ;
   sorts CONTAINER = STACK;
   ops   INTO = PUSH
         LAST-IN = TOP
         FIRST-OUT = POP
         FILLED? = FULL?;
endmap
```

3.3.3. Verification Conditions

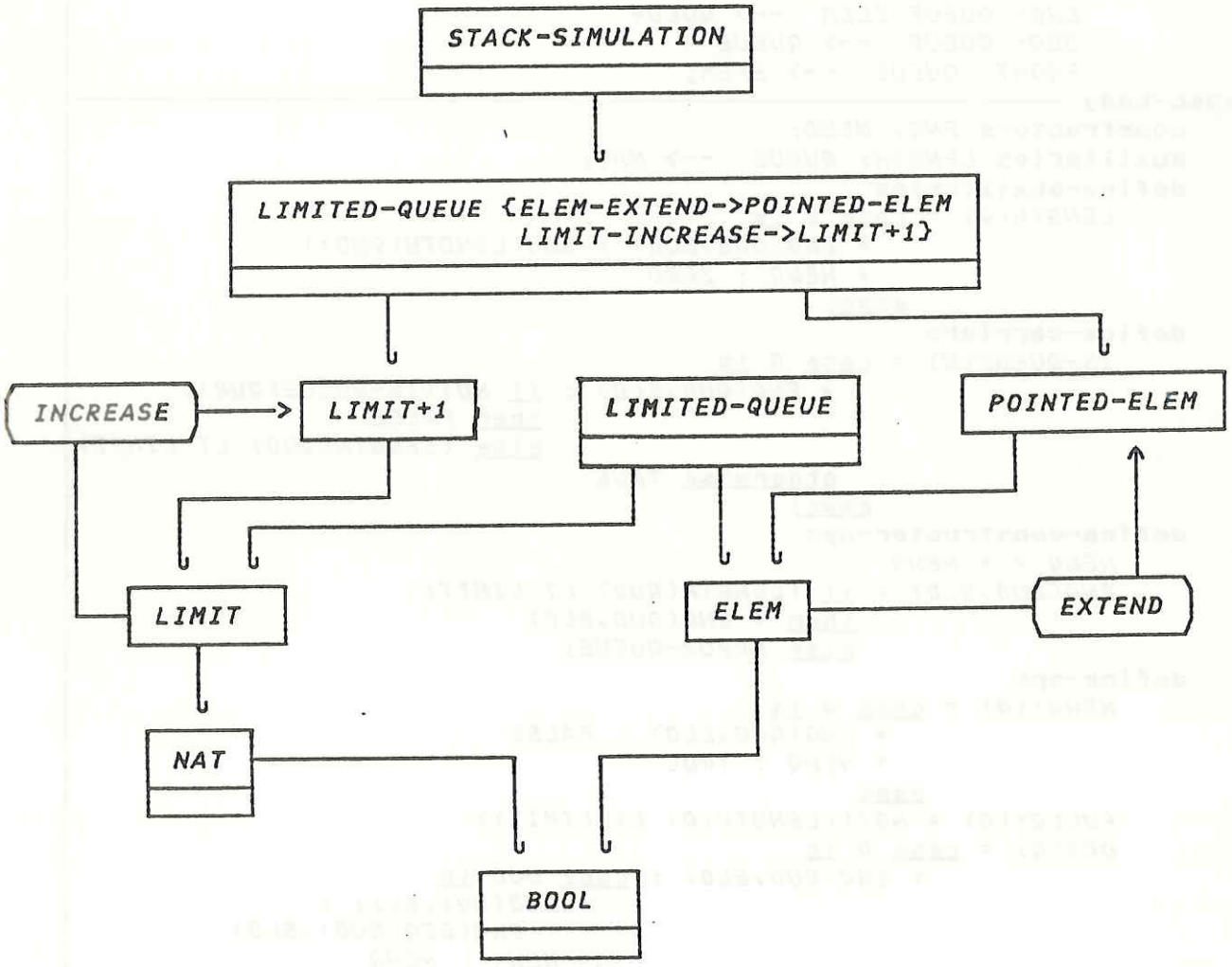
LIMITED-LIFO -FIX→ LIMITED-STACK: property PROP1 of LIMITED-LIFO, translated as defined by FIX, is (inductively) deducible in the LIMITED-STACK hierarchy.

The translated property is:

```
all c: stack all e: elem
    not(full?(st)) ==>
       (top(push(st,e)) == e &
        pop(push(st,e)) == st)
```

3.4. Level 3: Algorithmic LIMITED-QUEUE-CONSTRUCTION

3.4.1. The Hierarchy



3.4.2. Object Definitions

```
spec LIMITED-QUEUE
    /* ALGORITHMIC SPECIFICATION OF A LIMITED QUEUE. ENQUEUE IN A*/
    /* FULL QUEUE,DEQUEUE OR FRONT ON AN EMPTY QUEUE RESULT IN   */
    /* ERRORS  */
   use    ELEM , LIMIT ;
   sorts QUEUE;
   ops    NEWQ:  --> QUEUE
          NEWQ?,FULLQ?: QUEUE  --> BOOL
          ENQ: QUEUE ELEM  --> QUEUE
          DEQ: QUEUE  --> QUEUE
          FRONT: QUEUE  --> ELEM;
spec-body ------------------------------------------------------------
   constructors ENQ, NEWQ;
   auxiliaries LENGTH: QUEUE  --> NAT;
   define-auxiliaries
       LENGTH(Q) = case Q is
                     * ENQ(QUO,ELO) : SUC(LENGTH(QUO))
                     * NEWQ : ZERO
                   esac;
   define-carriers
       IS-QUEUE(Q) = case Q is
                       * ENQ(QUO,ELO) : if NOT(IS-QUEUE(QUO))
                                        then FALSE
                                        else (LENGTH(QUO) LT LIMIT)
                       otherwise TRUE
                     esac;
   define-constructor-ops
       NEWQ = * NEWQ
       ENQ(QUO,ELO) = if (LENGTH(QUO) LT LIMIT)
                      then * ENQ(QUO,ELO)
                      else ERROR-QUEUE;
   define-ops
       NEWQ?(Q) = case Q is
                    * ENQ(QUO,ELO) : FALSE
                    * NEWQ : TRUE
                  esac
       FULLQ?(Q) = NOT((LENGTH(Q) LT LIMIT))
       DEQ(Q) = case Q is
                  * ENQ(QUO,ELO) : case QUO is
                                     * ENQ(QU1,EL1) :
                                         ENQ(DEQ(QUO),ELO)
                                     * NEWQ : NEWQ
                                   esac
                  * NEWQ : ERROR-QUEUE
                esac
       FRONT(Q) = case Q is
                    * NEWQ : ERROR-ELEM
                    * ENQ(QUO,ELO) : case QUO is
                                       * NEWQ : ELO
                                       * ENQ(QU1,EL1) : FRONT(QUO)
                                     esac
                  esac;
endspec
```

```
spec LIMIT+1
    /* AXIOMATIC DEFINITION OF LIMIT + 1 */
   use   LIMIT ;
   ops   SUC-OF-LIMIT:  --> NAT;
   props [PROP1] SUC-OF-LIMIT == SUC(LIMIT);
endspec
```

```
map (LIMIT -INCREASE-> LIMIT+1)
    /* REPLACE LIMIT BY LIMIT+1 */
   is REFINEMENT;
   base
       NAT ;
   ops   LIMIT = SUC-OF-LIMIT;
endmap
```

```
spec POINTED-ELEM
    /* LOOSE SPECIFICATION OF A NON-ERROR CONSTANT OF SORT ELEM  */
    /* TOGETHER WITH AN IDENTIFICATION-TEST */
   use   ELEM ;
   ops   POINT:  --> ELEM
         POINT?: ELEM  --> BOOL;
   props [PROP1] ALL E:ELEM   POINT?(E) <==> E == POINT
         [PROP2] POINT =|= ERROR-ELEM;
endspec
```

```
map (ELEM -EXTEND-> POINTED-ELEM)
    /* THE INCLUSION OF ELEM IN POINTED-ELEM */
   is REFINEMENT;
   base
       BOOL ;
   sorts ELEM = ELEM;
endmap
```

```
spec STACK-SIMULATION
    /* SIMULATES THE LIMITED-STACK-OPERATIONS USING LIMITED-     */
    /* QUEUES WITH THE FOLLOWING SUBSTITUTIONS: -ELEM IS         */
    /* EXTENDED BY POINTED-ELEM; THE POINT WILL SEPARATE THE TOP */
    /* OF THE STACK FROM ITS BOTTOM WITHIN THE QUEUE             */
    /* REPRESENTATION - LIMIT IS INCREASED BY ONE DUE TO THE     */
    /* EXTRA POINT IN THE QUEUE REPRESENTATION. INVARIARIBLY,    */
    /* THE POINT WILL BE THE OLDEST ELEMENT OF ALL QUEUES        */
    /* GENERATED BY STACK OPERATIONS. NOTE: THE IMPLEMENTATION   */
    /* MUST GUARANTEE POINT NOT TO REPRESENT ANY ABSTRACT ELEM!  */
   use  LIMITED-QUEUE{ELEM -EXTEND-> POINTED-ELEM,
                      LIMIT -INCREASE-> LIMIT+1};
```

```
   ops  S-EMPTY:  --> QUEUE
        S-POP: QUEUE  --> QUEUE
        S-TOP: QUEUE  --> ELEM
        S-EMPTY?: QUEUE  --> BOOL;
spec-body
   define-ops
        S-EMPTY?(Q) = (POINT?(FRONT(Q)) AND NEWQ?(DEQ(Q)))
        S-EMPTY = ENQ(NEWQ,POINT)
        S-POP(Q) = let T = FRONT(Q)
                       T-1 = FRONT(DEQ(Q)) in
                   if POINT?(T-1)
                   then DEQ(Q)
                   else S-POP(ENQ(DEQ(Q),T))
        S-TOP(Q) = let T = FRONT(Q)
                       T-1 = FRONT(DEQ(Q)) in
                   if POINT?(T-1)
                   then T
                   else S-TOP(ENQ(DEQ(Q),T));
endspec
```

3.4.3. Verification Conditions

LIMITED-QUEUE: all operations terminate

LIMIT+1: property PROP1 of LIMIT+1 is consistent wrt. the hierachy below

LIMIT -INCREASE→ LIMIT+1:
property PROP1 of LIMIT, translated as defined by INCREASE, is (inductively) deducible in the hierarchy of LIMIT+1

The translated property is:
limit+1 =|= error-nat

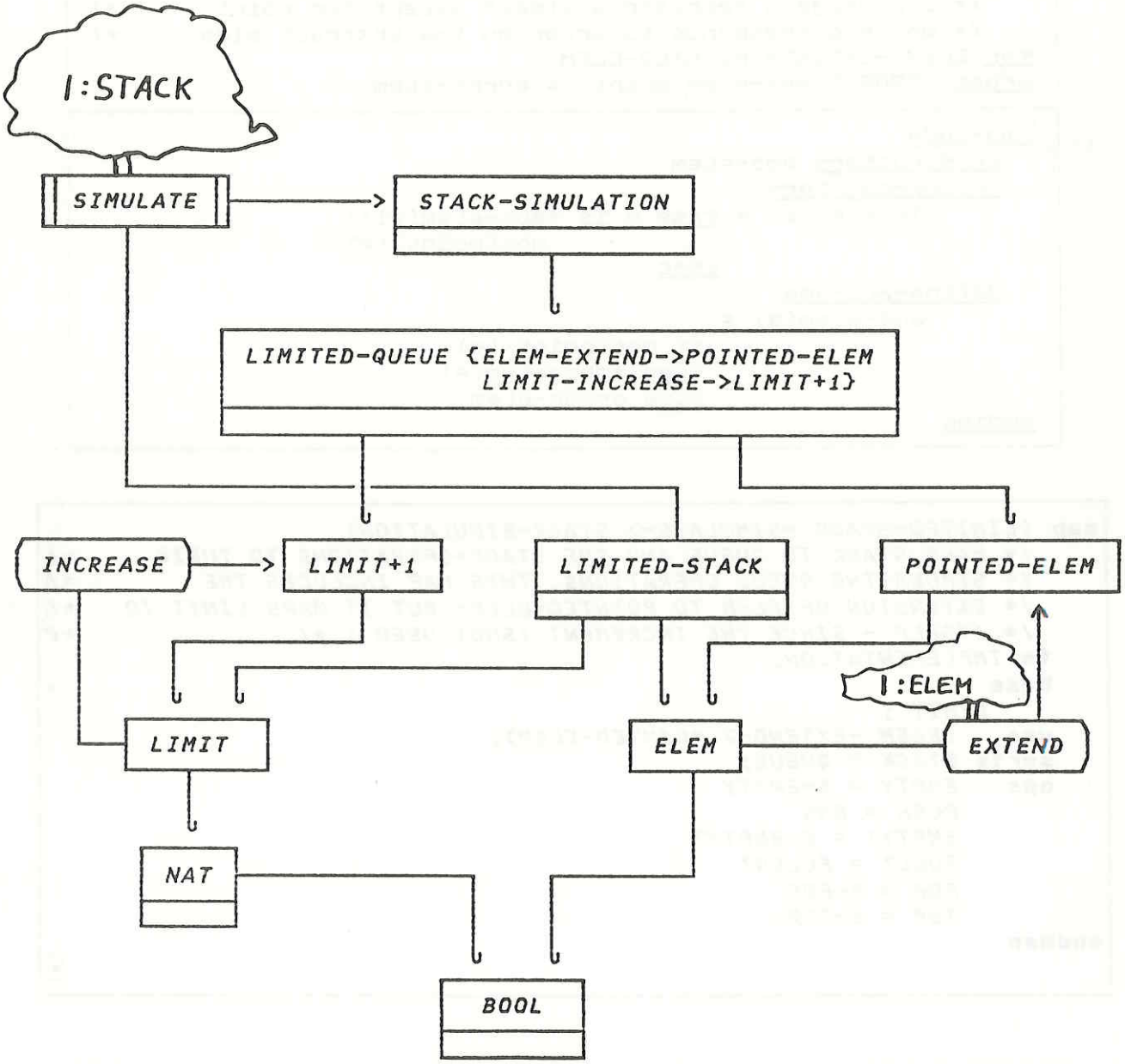
POINTED-ELEM: properties PROP1 and PROP2 of POINTED-ELEM are consistent wrt. the hierarchy below it

ELEM -EXTEND→ POINTED-ELEM: ——

LIMITED-QUEUE-CONSTRUCTION: all operations terminate

3.5. Transition Level 2 to Level 3

3.5.1. The Hierarchy



Due to technical reasons in the following paragraphs sometimes underlined keywords are used instead of bold-face keywords.

3.5.2. Object Definitions

```
imp I:ELEM
    /* algorithmic implementation definition; every          */
    /* ELEM-object represents itself except for point        */
    /* which corresponds to error on the abstract side       */
for ELEM -EXTEND-> POINTED-ELEM
props [PROP1] abs-elem(point) = error-elem
```

```
imp-body
  constructors abs-elem
  define-carriers
       is-elem(e) = case e is *abs-elem(e1):
                                  not(point?(e1))
                    esac
  define-abs-ops
       abs-elem(e) =
                    if not(point?(e))
                    then *abs-elem(e)
                    else error-elem
endimp
```

```
map (LIMITED-STACK -SIMULATE-> STACK-SIMULATION)
    /* MAPS STACK TO QUEUE AND THE STACK-OPERATIONS TO THEIR      */
    /* SIMULATING QUEUE OPERATIONS. THIS MAP INCLUDES THE        */
    /* EXTENSION OF ELEM TO POINTED-ELEM; BUT IT MAPS LIMIT TO   */
    /* ITSELF - SINCE THE INCREMENT ISNOT USED ! */               */
   is IMPLEMENTATION;
   base
       LIMIT ;
   use    (ELEM -EXTEND-> POINTED-ELEM);
   sorts STACK = QUEUE;
   ops    EMPTY = S-EMPTY
          PUSH = ENQ
          EMPTY? = S-EMPTY?
          FULL? = FULLQ?
          POP = S-POP
          TOP = S-TOP;
endmap
```

```
imp I:STACK
    /* algorithmic implementation definition for the         */
    /* simulation of LIMITED-STACK by the                    */
    /* STACK-SIMULATION.                                     */

for LIMITED-STACK -SIMULATE-> STACK-SIMULATION
use I:ELEM
     /* since abs-elem(point) must represent error-elem */
```

```
imp-body
  define-abs-ops
    abs-stack(q) = if s-empty?(q)
```

```
                    then empty
                    else push(abs-stack(s-pop(q)),
                              abs-elem(s-top(q)))
endimp
```

3.5.3. Verification Conditions

I:ELEM: all operations terminate;
property PROP1 is (inductively) deducible from the operations in I:ELEM and the hierachy below

LIMITED-STACK -SIMULATE→ STACK-SIMULATION: ——

I:STACK: its operation terminates

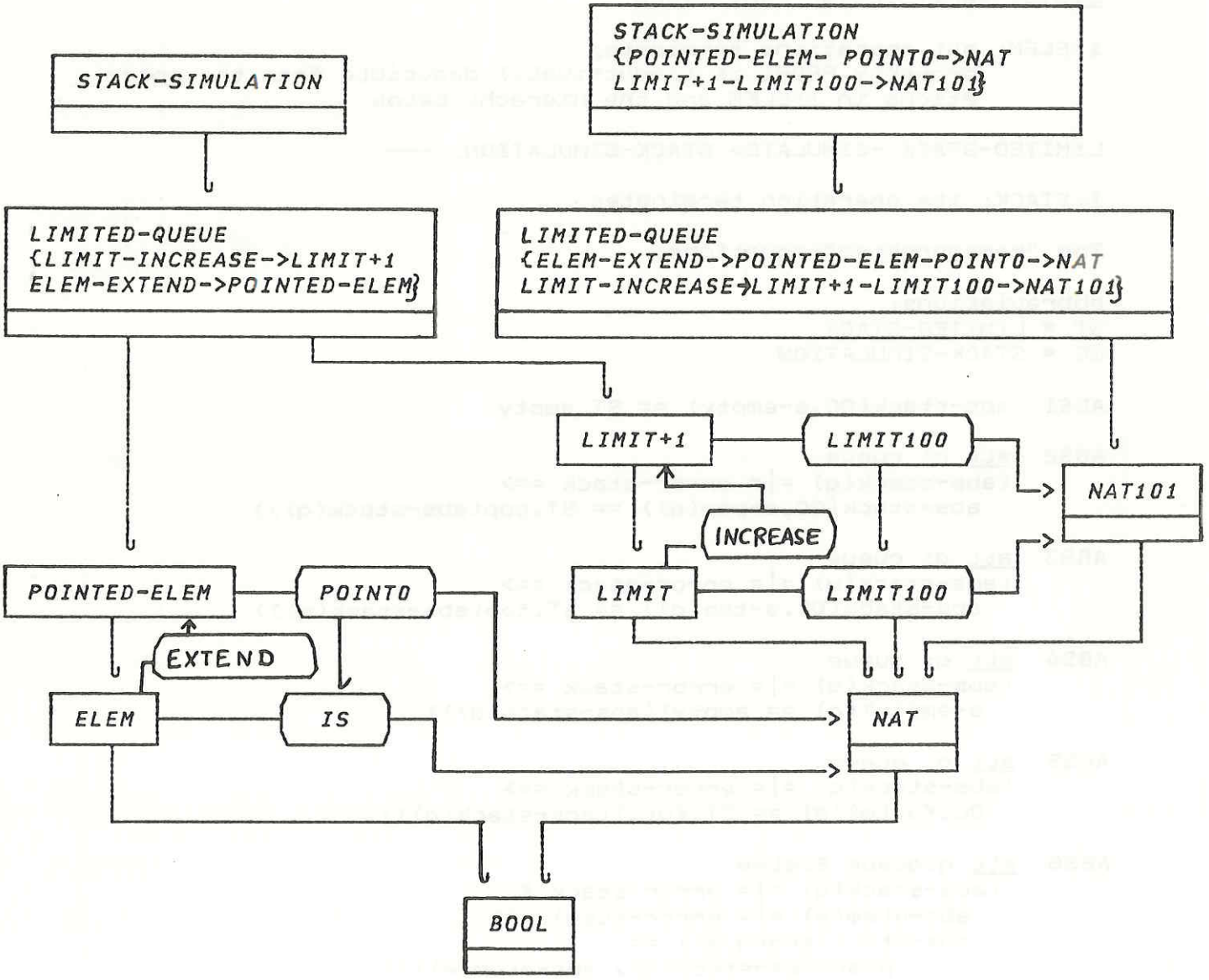
The "homomorphism"-equations:

abbreviations:
ST ≡ LIMITED-STACK
QC ≡ STACK-SIMULATION

```
ABS1  abs-stack(QC.s-empty) == ST.empty

ABS2  all q: queue
      (abs-stack(q) =|= error-stack ==>
        abs-stack(QC.s-pop(q)) == ST.pop(abs-stack(q)))

ABS3  all q: queue
      (abs-stack(q) =|= error-stack ==>
        abs-stack(QC.s-top(q)) == ST.top(abs-stack(q)))

ABS4  all q: queue
      (abs-stack(q) =|= error-stack ==>
        s-empty?(q) == empty?(abs-stack(q)))

ABS5  all q: queue
      (abs-stack(q) =|= error-stack ==>
        QC.fullq?(q) == ST.full?(abs-stack(q)))

ABS6  all q:queue E:elem
      ((abs-stack(q) =|= error-stack &
         abs-elem(e) =|= error-elem) ==>
         abs-stack(enq(q,e)) ==
              push(abs-stack(q), abs-elem(e))))
```

3.6. Level 4: Parameterization and Instantiation

3.6.1. The Hierarchy



3.6.2. Object Definitions

```
spec NAT101
    /* ENRICHES NAT BY DERIVED CONSTANTS */
   use    NAT ;
   ops    HUNDRED,HUNDRED-AND-ONE:   --> NAT;
spec-body
   define-ops
       HUNDRED = ([ TEN,ZERO ])
       HUNDRED-AND-ONE = ([ TEN,ONE ]);
endspec
```

```
map (ELEM -IS-> NAT)
    /* CHOOSES NAT AS ELEM */
   is REFINEMENT;
   base
       BOOL ;
   sorts ELEM = NAT;
endmap
```

```
map (POINTED-ELEM -POINTO-> NAT)
    /* CHOOSES ZERO AS POINT */
   is REFINEMENT;
   base
       BOOL ;
   use    (ELEM -IS-> NAT);
   ops    POINT = ZERO
          POINT? = ZERO?;
endmap
```

```
map (LIMIT -LIMIT100-> NAT101)
    /* CHOOSE 100 AS LIMIT */
   is REFINEMENT;
   base
       NAT ;
   ops    LIMIT = HUNDRED;
endmap
```

```
map (LIMIT+1 -LIMIT100-> NAT101)
    /* CONSEQUENTLY CHOOSES 101 FOR SUC-OF-LIMIT */
   is REFINEMENT;
   base
       NAT ;
   use   (LIMIT -LIMIT100-> NAT101);
   ops   SUC-OF-LIMIT = HUNDRED-AND-ONE;
endmap
```

3.6.3. Verification Conditions

NAT101: all operations terminate

ELEM -IS→ NAT: ——

POINTED-ELEM -POINTO → NAT:
Properties PROP1 and PROP2 of POINTED-ELEM, translated as defined by POINTO, are (inductively) derivable in the NAT hierarchy.
The translated properties are:

all e:nat (zero?(e) <=> e == zero)
zero =|= error-nat

LIMIT -LIMIT100 → NAT:
Property PROP1 of LIMIT, translated as defined by LIMIT100, is (inductively) derivable from the NAT hierarchy.
The translated property is:

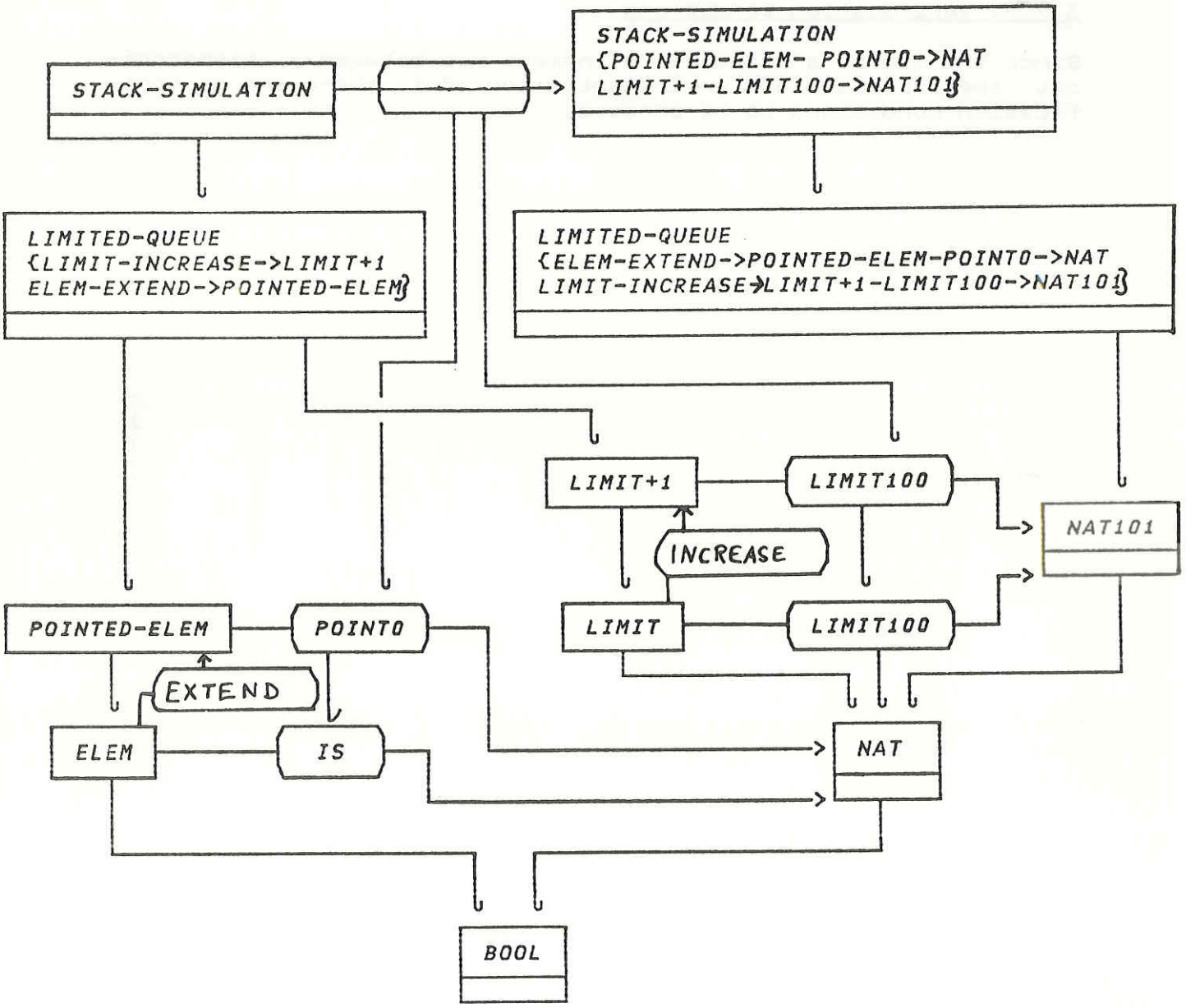
hundred =|= error-nat

LIMIT+1 -LIMIT100→ NAT101:
Property PROP1 of LIMIT+1, translated as defined by LIMIT100, is (inductively) derivable from the NAT101 hierarchy.
The translated property is:

hundred-and-one == suc(hundred)

3.7. Transition Level 3 to Level 4

3.7.1. The Hierarchy



3.7.2. Object Definitions

```
STACK-SIMULATION- → STACK-SIMULATION
          {POINTED-ELEM -POINTO→ NAT,
           LIMIT+1 -LIMIT100→ NAT101}
```

is the canonical map between a specification and an instance thereof.

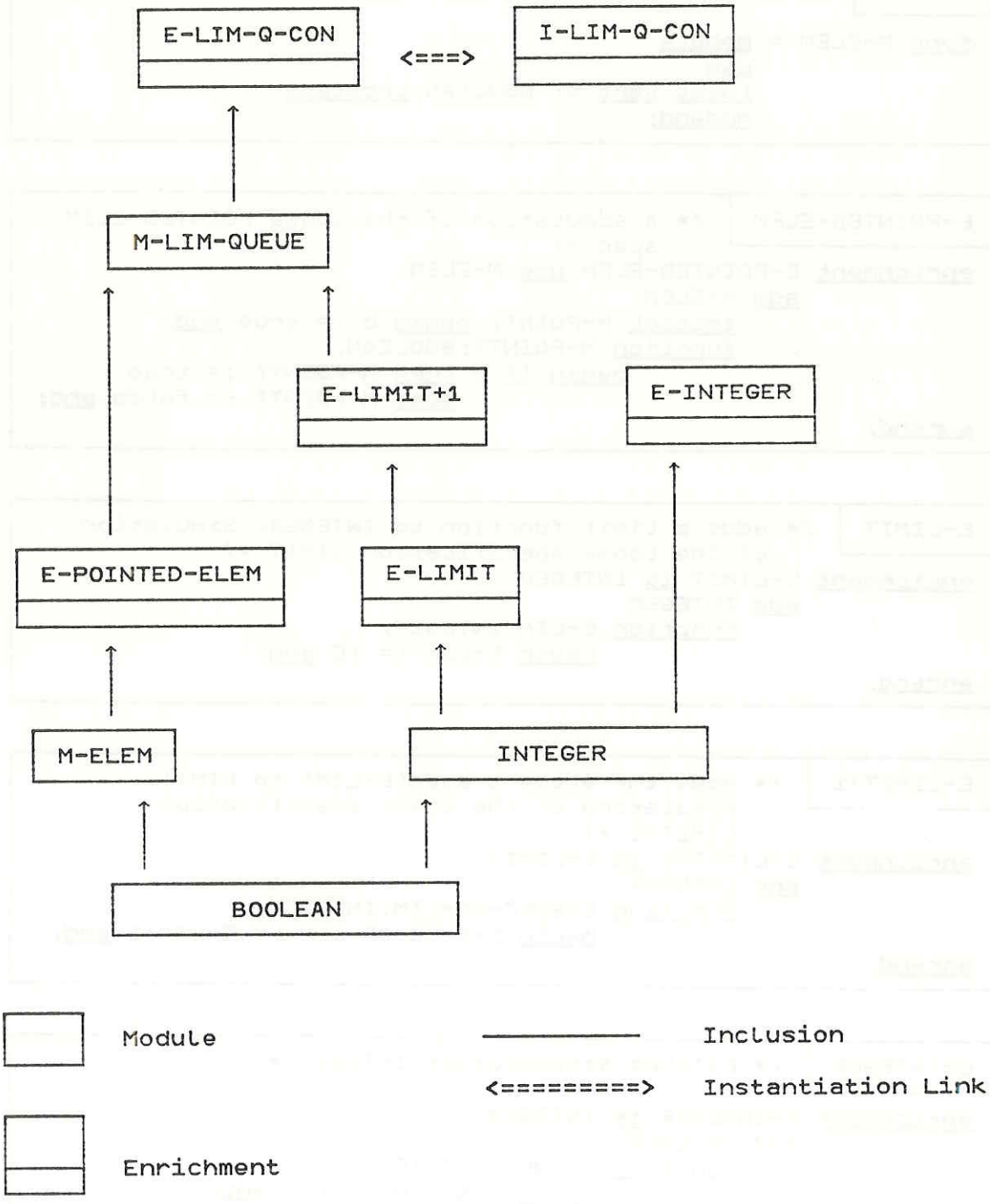
3.7.3. Verification Conditions

Since the map is a standard refinement map between a hierarchical specification and an instantiation of it there are no verification conditions to be prooved.

3.8. Level 5: Realization in ModPascal

3.8.1. The Hierarchy

The ModPascal level firstly remodels the central objects of level 4 but using the advantages of an imperative object oriented language. Then an instantiation is performed resulting in an object hierarchy that serves as counterpart to the ASPIK objects to be realized.



3.8.2. Object Definitions

BOOLEAN	/* ModPascal Standardtype */
INTEGER	/* ModPascal Standardtype */

M-ELEM /* Standard-Module with dummy representation */

```
type M-ELEM = module
              use
              local vars b: BOOLEAN localend
              modend;
```

E-POINTED-ELEM /* A simulation of the loose POINTED-ELEM spec */

```
enrichment E-POINTED-ELEM use M-ELEM
           add M-ELEM
               initial M-POINT; begin b := true end;
               function M-POINT?:BOOLEAN;
                        begin if b then M-POINT? := true
                                   else M-POINT? := false end;
enrend;
```

E-LIMIT /* adds a limit function to INTEGER: Simulation of the loose specification LIMIT */

```
enrichment E-LIMIT is INTEGER
           add INTEGER
               function E-LIM:INTEGER;
                        begin E-LIM := 10 end;
enrend;
```

E-LIMIT+1 /* adds the element succ(E-LIM) to LIMIT. Simulation of the loose specification LIMIT+1 */

```
enrichment E-LIMIT+1 is E-LIMIT
           add INTEGER
               function E-SUCC-OF-LIM:INTEGER;
                        begin E-SUCC-OF-LIM := E-LIM+1 end;
enrend;
```

E-INTEGER /* EXTENDs standardtype INTEGER */

```
enrichment E-INTEGER is INTEGER
           add INTEGER
               function HUNDRED: INTEGER;
                        begin HUNDRED := 100 end;
               function POINT: INTEGER;
```

```
                    begin POINT := 0     end;
            function Null? (I:INTEGER): BOOLEAN;
                    begin if I=0 then true
enrend;
```

M-LIM-Q

```
/* The LIMITED-QUEUE-module on E-POINTED-ELEM and
   E-LIMIT+1; local variable b1 models the
   point */
type M-LIM-Q = module
          use E-LIMIT+1, M-ELEM
          public procedure M-ENQ (E: M-ELEM);
                 procedure M-DEQ;
                 function M-EMPTY?: BOOLEAN;
                 function M-FULL?: BOOLEAN;
                 function M-OLDEST: BOOLEAN;
                 initial M-EMPTYQUEUE;
          local type A = array [1..E-LIM] of M-ELEM;
                vars a1:A; b1:BOOLEAN; j1:INTEGER
                localend;
     procedure M-ENQ
               begin if j1<E-LIM then begin j1 := j1+1;
                                            a1[j1] := E;
                                      end
                                 else ERROR.M-LIM-Q end;
     procedure M-DEQ;
               vars K1: INTEGER;
               begin if j1>0 then begin
                                  while K1 ≤ j1 do
                                  begin a1[K1] := a1[K1+1]
                                        K1 := K1+1 end;
                                        j1 := j1-1 end;
                                  else ERROR.M-LIM-Q end;
     function M-EMPTY?;
               begin if j1 = 0 then true else false
                     end;
     function M-FULL?;
               begin if j1 = E-LIM then true
                                   else false end;
     function M-OLDEST;
               begin if 1 ≤ j1 ≤ E-LIM
                     then M-OLDEST := a1[1]
                     else M-OLDEST := ERROR.M-LIM-Q
                     end;
     initial M-EMPTYQUEUE;
               begin b1 := false; j1 := 0 end;
modend;
```

E-LIM-Q-CON

```
/* adds STACK operations to M-LIM-Q.
   Stacks are queues with b1 := true. */
enrichment E-LIM-Q-CON is M-LIM-Q
          add M-LIM-Q
              initial E-EMPTYSTACK;
                      begin b1 := true; j1 := 0 end;
              procedure E-POP;
                      begin if b1 and j1>0
                                then j1 := j1-1
```

```
                                else ERROR.M-LIM-Q end;
            function E-TOP: M-ELEM;
                    begin if b1 and 1≤j1≤E-LIM
                                then E-TOP := a1[1]
                                else E-TOP := ERROR.M-ELEM
                    end;
            function E-SIZE1?: BOOLEAN
                    begin if b1 ; j1=1
                                then E-SIZE1? := true
                                else E-SIZE1? := false
                    end;
            procedure E-PUSH(E:M-ELEM);
                    begin if b1
                                then M-ENQ(E)
                                else ERROR.M-LIM-Q end;
enrend;
```

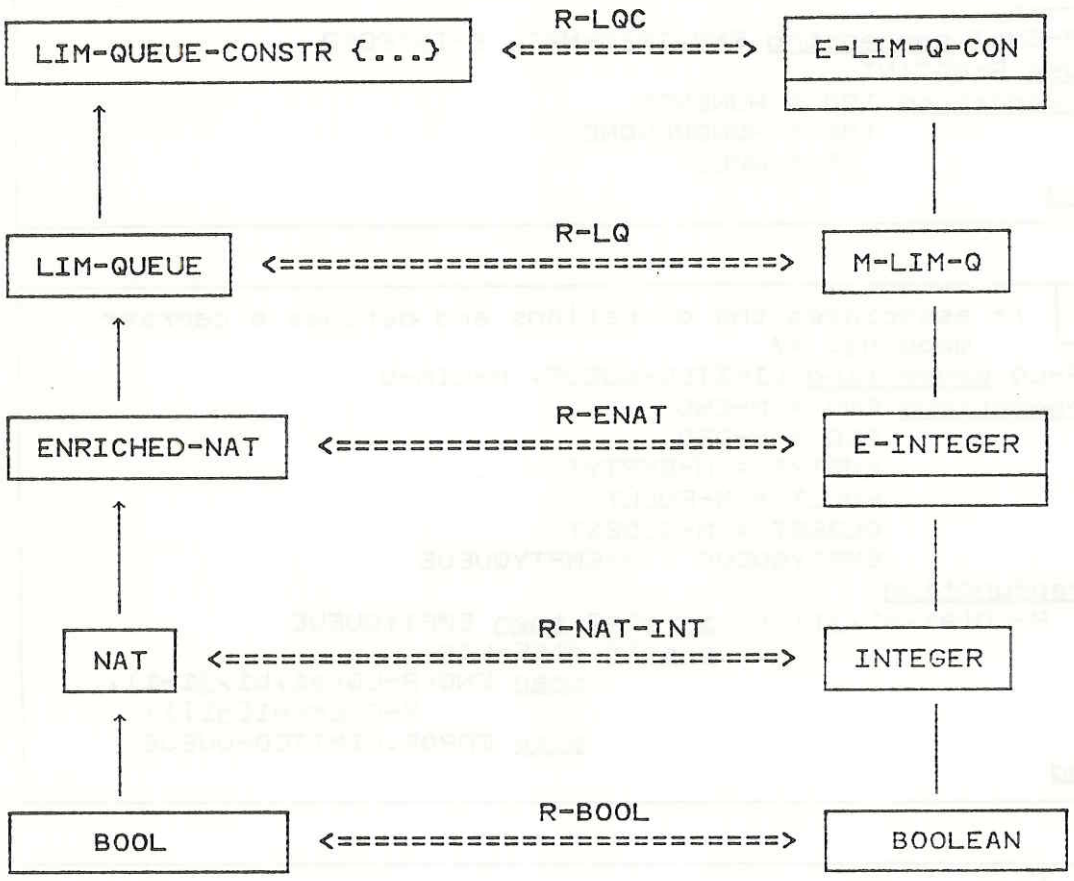
```
I-LIM-Q-CON     /* E-LIMIT and E-POINTED-ELEM are
                   substituted using E-INTEGER */
instantiation I-LIM-Q-CON of E-LIM-Q-CON is
              E-LIMIT to E-INTEGER,
              E-POINTED-ELEM to E-INTEGER;
              operations
                 E-LIM = HUNDRED,
                 M-POINT = POINT,
                 M-POINT? = NULL?;
instend;
```

3.8.3. Verification-Conditions

Checks made during the input of module /enrichment /instantiation-objects guarantee the context sensitive correctness (interface, signature morphisms, property, etc.)(c.f. [HR 86], [Olt 86]).
Further semantical checks are not necessary.

3.9. Transition Level 4 to Level 5

3.9.1. The Hierarchy



3.9.2. Object Definitions

```
R-BOOL   /* Standard object; semantics of ModPascal-BOOLEAN
            is identical to ASPIK-BOOL */
```

```
R-NATINT   /* Standard object connecting ModPascal-INTEGER
              and ASPIK NAT. Negative Integer values are
              mapped to error.nat. */
```

```
R-LIM   /* maps ASPIK-LIMIT operation to ModPascal
           operation; values are mapped by repfunction
           of R-NATINT. */
rep R-LIM connecting LIMIT, E-LIMIT
    use R-NATINT
```

```
    operations LIMIT = E-LIMIT
repend
```

```
R-ENAT | /* associates boundary operations */

rep R-ENAT connecting ENRICHED-NAT, E-INTEGER
    use R-NATINT
    operations 100 = HUNDRED
               101 = HUNDREDONE
                0? = NULL?
repend
```

```
R-LQ | /* associates the operations and defines a carrier
        mapping. */
rep R-LQ connecting LIMITED-QUEUE, M-LIM-Q
    operations ENQ = M-ENQ
               DEQ = M-DEQ
               EMPTY? = M-EMPTY?
               FULL? = M-FULL?
               OLDEST = M-OLDEST
               EMPTYQUEUE = M-EMPTYQUEUE
    repfunction
      R-LQ(a1,b1,j1) := if j1=0 then EMPTYQUEUE
                        elseif j1≤E-LIM
                               then ENQ(R-LQ(a1,b1,j1-1),
                                        R-ELEM(a1[j1]))
                               else ERROR.LIMITED-QUEUE
repend
```

```
R-LQC | /* ... */

rep R-LQC connecting LIMITED-QUEUE-CONSTRUCTION,
                     E-LIM-Q-CON
    use R-LQ
    operations EMPTY = E-EMPTYSTACK
               PUSH = E-PUSH
               POP = E-POP
               TOP = E-TOP
               SIZE1? = E-SIZE1?
repend
```

3.9.3. Verification Conditions

R-BOOL, R-ELEM, R-NATINT: Assumed to be already correct repobjects(=realizations)

R-ENAT
Assumptions: All used repobjects are realizations.
Then the following realization conditions have to be shown:
(1) R-NATINT'(HUNDRED) = 100
(2) R-NATINT'(HUNDREDONE) = 101
(3) all I ∈ INTEGER . (R-NATINT'(I) ≠ error.nat →
R-BOOL(NULL?(I)) = 0?(R-NATINT(I)))

R-LQ
Assumptions:
- All used objects are realizations.
- The types of the local variables of M-LIM-Q are realizations (for standardtypes: ARRAY, BOOLEAN, INTEGER).

Then the following realization conditions have to be shown:
(1) all Q=<a1',b1',j1'> ∈ (array x bool x integer). all I ∈ integer .
(R-LQ'(Q) ≠ error.limited-queue & R-NATINT'(I) ≠ error.integer →
(j1'<100) → R-LQ'(<assign(a1',plus(j1',1),I),b1',
plus(j1',1)>) = ENQ(R-LQ'(Q), R-NATINT'(I))
[In case of (j1'≥100) R-LQ evaluates to error.limited-queue.]

(2) all Q=<a1',b1',j1'> ∈ (array x bool x integer) .
(R-LQ'(Q) ≠ error.limited-queue) →
(a) (j1'=0) → R-BOOL'(true) = EMPTY?(R-LQ'(Q))
(b) ¬(j1'=0) → R-BOOL'(false) = EMPTY?(R-LQ'(Q))

(3) all Q=<a1',b1',j1'> ∈ (array x bool x integer) .
(R-LQ'(Q) ≠ error.limited-queue) →
(a) (j1'= 100) → R-BOOL'(true) = FULL?(R-LQ'(Q))
(b) ¬(j1'= 100) → R-BOOL'(false) = FULL?(R-LQ'(Q))

(4) all Q=<a1',b1',j1'> ∈ (array x bool x integer) .
(R-LQ'(Q) ≠ error.limited-queue) →
(1≤j1'≤100) → R-NATINT'(read(a1',1)) = OLDEST(R-LQ'(Q))
[In case of ¬(1≤j1'≤100) R-NATINT' evaluates to error.nat.]

(5) all Q=<a1',b1',j1'> ∈ (array x bool x integer) .
R-LQ'(<a1',false,0>) = EMPTYQUEUE

For the operation DEQ a homomorphism equation of the following form is generated:
(6) all Q ∈ Carrier (M-LIM-Q) . (R-LQ(Q) ≠ error.limited.queue)
→ DEQ(R-LQ(Q)) = R-LQ(M-DEQ(Q))

R-LQC
Assumptions:
- R-LQ is realization.
- All objects in E-LIM-Q-CONSTR are realized objects. Then the following realization conditions have to be shown:

(1) R-LQ'(a1',true,0>) = EMPTYSTACK.
(2) (b1' and (j1'>0)) → R-LQ'(<a1',b1',minus(j1',1)>)
= pop(RL-Q'(Q))
[In case of ¬(b1' and (j1'>0)) R-LQ' evaluates to error.limited-queue.]

(3) (b1' and (1≤j1'≤100)) → R-NATINT'(read(a1',1))
= TOP(RL-Q'(Q))
[In case of ¬(b1' and (1≤j1'≤100)) R-NATINT' evaluates to error.nat.]

(4) (b1' and (j1'=1)) → R-BOOL'(true) = SIZE1?(R-LQ'(Q))
¬(b1' and (j1'=1)) → R-BOOL'(false) = SIZE1?(R-LQ'(Q))
(5) (b1' → ((j1'<100) → R-LQ'(<assign(a1', plus(j1'1),I),b1',
plus(j1',1(>) = PUSH(R-LQ'(Q),
R-NATINT'(I))
[In case of ¬b1' R-LQ' evaluates to error.limited.queue.]

4. References

[BEORSW] Breiling, M., Eckl, G., Olthoff , W., Rainau, U., Schmitt, M., Weiss, P.: The RL-Handbook. University of Kaiserslautern, 1985.

[BES 81] Bläsius, K., Eisinger, N., Siekmann, J., Smolka, G., Herold, A., Walther, C.: The Markgraf Carl Refutation Procedure, Proc., 7th IJCAI, 1981.

[BGGORV 83] Beierle, C., Gerlach, M., Göbel, R., Olthoff, W., Raulefs, P., Voß, A.: Integrated Program Development and Verification: In: H.L. Hausen (ed.): Symposium on Software Validation, North-Holland Publ. Co., Amsterdam 1983.

[BGV 83] Beierle, C., Gerlach, M., Voß, A.: Parameterization without Parameters. In: The History of a Hierarchy of Specifications. SEKI-Project, Memo SEKI-83-09, Universität Kaiserslautern, Fachbereich Informatik, September 1983.

[BOV 86] Beierle, C., Olthoff, W., Voß, A.: Software Development Environments Integrating Specification and Programming Languages. In: H.-W. Wippermann (ed): Proc. GC-ACM Workshop on Software Architecture and Modular Programming, Teubner, Stuttgart, 1986 (to appear).

[BR 85] Breiling, M., Rainau, U.: An Object Administration System and a Representation Object Programming System. Master thesis (in German). University of Kaiserslautern, 1985.

[BV 83a] Beierle, C., Voß, A.: Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types. University of Kaiserslautern, Memo SEKI-83-07, 1983.

[BV 85] Beierle, C., Voß, A.: Algebraic Specifications and Implementations in an Integrated Software Development and Verification System. Memo SEKI-85-12, University of Kaiserslautern, 1985. (Joint SEKI-Memo containing the Ph.D. thesis by C. Beierle and the Ph.D. thesis by A. Voß).

[Eck 84] Eckl, G.: A Precompiler for ModPascal (in German). University of Kaiserslautern, Interner Bericht 121/84, 1984.

[Ge 83] Gerlach, M.: A Second-Order Matching Procedure for the Practical Use in a Program Transformation System. SEKI-Projekt, Memo SEKI-83-13, Universität Kaiserslautern, Fachbereich Informatik, 1983.

[Gei 84] Geisler, C.: MADRE - Ein Programmtransformationssystem für ASPIK-Spezifikationen, SEKI-Projekt,

Universität Kaiserslautern, Fachbereich Informatik, 1984.

[HR 86] Hammel, S., Rudolph, G.: A Re-implementation of the ModPascal Programming System. University of Kaiserslautern, Fachbereich Informatik, 1986 (in preparation).

[KRST 83] Kücke, R., Rome, E., Sommer, W., Thomas, Ch.: Das SPEC-System (SPESY): Benutzerhandbuch, SEKI-Projekt, Universität Kaiserslautern, Fachbereich Informatik, 1983.

[Lic 85] Lichter, H.: Ein interaktives und syntaxorientiertes Eingabesystem für algebraische und algorithmische Spezifikationen, SEKI-Projekt, Universität Kaiserslautern, Fachbereich Informatik, 1985.

[Olt 86] Olthoff, W.: The Connection Between Applicative and Procedural Languages in an Integrated Software Development and Verification System. Dissertation (forthcoming), Kaiserslautern, 1986.

[Pet 83] Petersen, U.: Elimination von Rekursionen, SEKI-Projekt, Memo SEKI-83-10, Universität Kaiserslautern, Fachbereich Informatik, 1983.

[SBV 85] Schoelles, V., Beierle, C., Voß, A.: SPESY - Benutzerhandbuch. SEKI-Projekt, FB Informatik, Universität Kaiserslautern, 1985.

[Sch 85] Schoelles, V.: Beschreibung des Spezifikationsentwicklungssystems SPESY und seiner Implementierung. Diplomarbeit, FB. Informatik, Univ. Kaiserslautern, 1985. (in German)

[Spa 85] Spang, H.: Implementation of a Component of SPESY. Working paper (in German), University of Kaiserslautern, 1985.

[Tho 84] Thomas, Ch.: RRLab - Rewrite Rule Labor. Entwurf, Spezifikation und Implementierung eines Softwarewerkzeuges zur Erzeugung und Vervollständigung von Rewrite-Rule Systemen. SEKI-Projekt, Memo SEKI-84-01, Universität Kaiserslautern, Fachbereich Informatik, 1984.

[Wei 85] Weis, P.: The Compatibility Checker (in German). University of Kaiserslautern, 1985.