THE IMPORTANCE OF FORMAL METHODS IN ADVANCED SOFTWARE DEVELOPMENT ENVIRONMENTS

ILEANA RABEGA

Software Engineering Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest,
ROMANIA

ABSTRACT

The use of formal methods in advanced software development environments is presented; generic advanced software development environments architecture is introduced and examples of such environments are given.

Keywords: advanced software development environments, formal methods/languages, operational life cycle.

INTRODUCTION

The advanced software development environments making use of formal methods may vary from experimental systems developed in research units and universities to industrial projects. Seven conclusions are drawn about formal methods' impact on users (whether these formal methods are included in software development environments or not):

1) Formal methods are very helpful in detecting errors during the early stages of development and in almost eliminating certain classes of errors.
2) The software system designer must clearly predefine his/her opinion or clarify his/her opinion about the system to be built when using formal methods.
3) Formal methods are useful for almost any kind of application; in a non–critical system, even if its further development will not be a formal one, just writing down a formal initial specification means something better than dealing with other informal/semi–formal methods; in a critical system there must be a formal specification to start with and a very rigorous step–by–step development process must be followed within which formal expression and justification hold for each step and for the whole process.
4) Formal methods are based on mathematical specifications which, being abstract, are much easier to understand than programs.
5) When using formal methods, the development costs will diminish because all the development stages, including the early ones, which would be error–prone, if informally or semi–formally approached, are formally expressed and justified.
6) Formal methods help clients understand what type of a system they are going to buy.
7) Formal methods are successfully used on practical industrial projects.

These conclusions have been reached by experimenting the CICS project at IBM, for example, or the CASE project at Praxis Systems.

1. FORMAL METHODS, LANGUAGES AND SUPPORT SOFTWARE SYSTEMS

For the last ten years, there have been carried out basic researches on formal methods, languages and support software systems, with a three–fold orientation:

1) Formal semantics building of specification/programming languages.
2) Specification of software development process semantics.

3) Integrated and advanced sequential and concurrent software development environments building which support results of 1) and 2).

One first research direction resulted on the one hand in formal methods/formal executable specification languages building and, on the other hand, in formal models for the existing programming/development languages building (e.g. Ada).

A development method is a formal one to the extent to which it makes a rigorously, mathematically-based description of the system properties. Such notions as system properties, consistency and completeness, as well as the development process itself may be precisely defined.

The description of real system properties will be possible by means of a formal specification language. A formal method may be supported or not by tools, but a formal language must be supported by tools. By means of a formal specification language the extent to which a specification is implementable can be verified by proving the system properties correctness, with no need to run the corresponding software for that.

A formal language has three characteristics:

- A formal syntax;
- A formal semantics;
- A satisfies relation.

A formal syntax is given by means of an abstract syntax, which is a function system for building and decomposing composed syntactic objects.

Functions and relations among them are defined by means of axioms.

A formal language semantics can be built in two ways:

A) **Synthetically** when specification/program meaning is defined by composing the meaning of the sub-specifications/sub-programs by appropriate operators attached to the language constructs.

Examples of synthetical semantics will be denotational semantics and algebraic semantics.

B) **Analytically**, when meaning is only given to complete systems of specifications/programs and not to fragmentary specification/program.

The explicit syntactical structure of the complete specification/program system will include a set of processes in it.

Examples of analytical techniques are the various definitions of the operational semantics of specifications/programs.

The satisfies relation of a formal language has two roles:

- To accustom with different views on system components/objects;
- To make restrictions on the system.

A second research direction resulted in semantic models of sequential software development process building (which are conceptually convergent) and in several semantic models of concurrent software development process building (which are not as conceptually convergent as the sequential models).

An example of sequential model, which is part of the PROSPECTRA advanced software development environment, refers to the development process as a formal object that does not only represent a documentation of the past but also a plan for further developments.

It can be used to abstract a class of similar developments from a particular development.

Examples of concurrent models will be labeled transition systems and synchronization trees (CCS-Milner language), Petri nets. The CSP models family is

another example, including: The Counter Model, The Trace Model (to be presented later on), The Divergence Model, The Readiness Model and The Failures Model.

, , The Counter Model is the least sophisticated one, adequately dealing with only tree-like networks of processes. The Trace Model allows the description of arbitrary networks of processes. Both the Counter Model and the Trace Model can specify safety properties (see note *) but no one can deal adequately with diverging processes (see note **). The Divergence Model is able to reason about systems that may diverge. Additionally the Readiness and Failure Models may reason about liveness (see note*) and safety.

A third research direction resulted in building integrated and advanced sequential and concurrent software development environments of which three main characteristics are:

A) The operational life cycle;

B) The existence of a formal model which the software development environment data processing is based on;

C) The existence of a knowledge base, which represents the development environment expert domain.

Besides the above mentioned research directions, formal methods/languages and advanced software development environments supporting them are used in industrial projects.

Some examples are given below:

TRANSACTION PROCESSING : IBM's CICS is a large, twenty-year old transaction-processing system. It contains more than half a million lines of code. The Z formal specification method has been used by IBM in specifying again the key CICS interfaces with a view at enhancing maintainability.

HARDWARE : Tektronix has been using Z method to specify the functionality of oscilloscope families.

COMPILERS : The Danish Datamatik Center has for many years been developing industrial compilers using formal methods.

SOFTWARE TOOLS : The CASE Project (Praxis Systems).

REACTOR CONTROL : Rolls - Royce and Associates used a combination of English and formal specifications to specify nuclear-reactor control software. They used animation to explore the specification.

2. EXAMPLES OF FORMAL METHODS/LANGUAGES TAXONOMIES

Formal methods/languages may be classified according to several criteria [7]. Formal methods/languages fall into two classes defining the system behaviour:

A) Model-oriented methods/languages, in which system behaviour is specified directly, by constructing a model of the system in terms of mathematical structures (tuples, relations, functions, sets, maps, trees, sequences, etc.).

---

* Safety for concurrent processes corresponds to partial correctness for sequential programs. Intuitively, safety properties specify that some condition does not hold whereas liveness properties specify that some condition will hold.

** A concurrent and distributed system (network of processes) is non-divergent if recurrence goes on and there is no infinite consecutive sequence of hidden events which the system may get involved in.

B) Property-oriented methods/languages, in which system behaviour is specified indirectly by stating a set of properties (usually axioms) that must be satisfied by the system; in this case a minimum number of restrictions has to be made (that is, the minimum number of properties satisfying the system requirements). The larger the number of implementations, the smaller the number of restrictions.

When referring to formal languages semantics we associate denotational semantics with model-oriented languages, and algebraic semantics with property-oriented languages. Both types of formal languages may additionally support operational semantics. In order to describe systems , either distributed or not, analytical techniques will be used, i.e. we will proceed with operational semantics; while describing the specification/programming languages and modular design methods, synthetical techniques will be used, i.e. we will proceed with denotational semantics and algebraic semantics.

For example, operational and algebraic semantics are known for OBJ3 language, operational and denotational semantics are known for CSP language, etc.

The following examples refer to model-oriented methods/languages:

- For sequential software systems: VDM (Vienna Development Method with its language, VDL), method Z.
- For concurrent and distributed systems : Petri nets, languages CCS(Milner) and CSP(Hoare).

Here are examples of property–oriented methods/languages:

- For sequential software systems: the Larch language family, languages OBJ2, OBJ3, Act One, P-AnndA-S, Pluss, method Z.
- For concurrent and distributed systems: temporal logic, language LOTOS (a combination of Act One and CCS).

Formal methods/languages can fall under two categories as to their **specifications presentation form**: textual and graphical. Generally, both forms are needed and actually they co–exist, the graphical form being elaborated after the textual one and aiming at making an as friendly as possible user interface.

Specification behaviour shall determine that formal methods/languages belong to two categories:

- executable (e.g. languages OBJ2, OBJ3);
- nonexecutable (e.g. language VDL, method Z).

Last but not least, formal methods/languages fall into two classes if their **support tools types** are considered:

- Method-oriented tools (e.g. syntax-directed editors);
- General tools [e.g. theorem-provers - Rewrite Rule Laboratory (RRL), the Boyer-Moore Theorem Prover, etc.].

3. LIFE CYCLE SUPPORTED BY AN ADVANCED SOFTWARE DEVELOPMENT ENVIRONMENT BASED ON FORMAL METHODS

An advanced software development environment based on formal methods supports a life cycle consisting of several phases [1], [2], [5]:

A) Requirements analysis;
B) Formal specification;
C) Formal specification verification;
D) Specification implementation in some executable specification language and in programming languages.

This life cycle is called operational [6] for executable specifications.

Phase order is given in Figure 1:

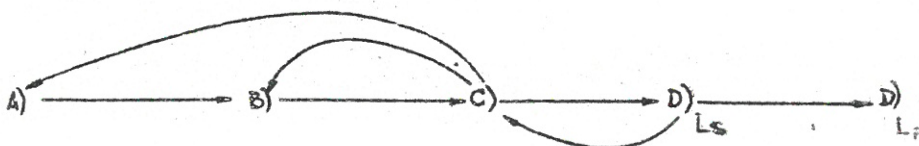

Figure 1. Operational life cycle

3.1. REQUIREMENTS ANALYSIS

Requirements analysis is a complex process of vital importance for the future system. Besides system data collecting, requirements analysis includes data organization so as to deduce the relevant requirements as well as the absent ones. Being an intuitive analysis, requirements analysis may not be easy to support by a formal method, which needs rigour, that is well–defined requirements. An advanced software development environment coherently gathers both formal methods and heuristic/informal or semi-formal methods in order to support requirements analysis.

If sequential software development is supported by the environment, static analysis techniques of structural properties can be introduced (e.g. data flow diagrams). If concurrent software development is supported by the environment, a solution for hinting at the software dynamic behaviour will be to build up requirements exploration prototypes, which use animation techniques. With a minimum of system requirements, an initial system specification is built and portions of it are identified and selected in order to be animated. Animation will then involve (dynamically) stepping through each process action and examining the output behaviour. Animation can be used to determine causal relationships embedded in the specification, or simply as a means of browsing through the specification to ensure adequacy and accuracy by reflecting the specified behaviour back to the user. In particular, the need for reflecting specified behaviour under different circumstances (i.e. animation replay with different data values) must be considered.

By animation new requirements for the future systems can be identified. This technique is generally applied to critical system components.

There is no clear–cut separation between requirements analysis and formal specification.However, in order to formally specify a software system, requirements identified at this phase must be accurate and non–ambiguous.

3.2. FORMAL SPECIFICATION

Building formal specifications takes several steps:

- State the critical requirements, which are usually a natural language-like statement of what is desired, in precise mathematical terms;
- Produce a high-level formal specification of the system using a formal specification language;
- Refine system formal specification in the same formal specification language by producing more and more detailed formal specifications;

- Code final system formal specification in a high-level programming language.

To demonstrate the consistency of the high-level programming language code with the critical requirements will obviously be a difficult task. One can manage this by verifying specification on each step (see 3.3).

The rest of the paragraph will show how simple formal specifications in well-known specification languages are built, based on the already mentioned taxonomic criterion that is the definition of system behaviour, which points to two method/language types:

- Model-oriented methods/languages;
- Property-oriented methods/languages.

Sequential case, model-oriented specifications

A symbol table specification is to be modelled in VDL (Vienna Development Language) (see Fig. 2).

VDL Language is model-oriented and supports four models: tuple (for both sequences and lists), set, map and tree.

The state of the table is modelled by a mapping from keys to values (ST = map Key to Val).

The table contains four operations:

- INIT, which initializes the symbol table to be empty;
- INSERT, which adds a new binding to the symbol table in case key k is not to be found in the domain of the table;
- LOOKUP, which returns the value to which key k is mapped, requiring that key k should be in the domain of the table;
- DELETE, which retracts the binding associated with k from the table, requiring that key k should be in the domain of the table (where $\trianglelefteq$ is the domain subtraction operator).

By convention, unprimed variables in VDL stand for the state before an operation is performed and primed variables for the state afterwards.

Supplementarily some predicates appear in VDL specification in Figure 2, named as pre- and postconditions.

A precondition of an operation is a predicate that must hold in the state on each call of the operation; if it does not hold, the operation's behaviour is unspecified.

A postcondition is a predicate that holds in the state upon return.

Additional declarations of external state variables in Figure 2 are the following:

- **rd** (for read-only-access) expresses the fact that LOOKUP does not modify the symbol table (st = st');
- **wr** (for write-and-read-access) expresses the fact that INSERT and DELETE modify the symbol table.

```
ST = map Key to Val
INIT ()
ext wr st: ST
post st' = {}
INSERT (k:Key, v:Val)
ext wr st: ST
pre k ∉ dom st
post st' = st ∪ {k → v}

LOOKUP (k:Key)v:Val
ext rd st:ST
pre k ∈dom st
post v' = st(k)
DELETE(k:Key)
ext wr st:ST
pre k ∈dom st
post st' = {k} ⩤ st
```

Figure 2. - VDL **symbol table specification [7]**

Sequential case, property-oriented specifications

The same symbol table is modelled in the property-oriented Larch language family. In Larch two levels of specifying can be addressed:

- A state-dependent level, functionally similar to a VDL specification, known as an interface specification;
- A state-independent level, expressed as an algebraic specification, known as a trait.

For each operation in an interface specification, the **requires** and **ensures** clauses are used as VDL pre-and postconditions. The **modifies** clause lists those objects whose value may be altered as a result of executing the operation. Hence LOOKUP does not modify the symbol table, whereas INSERT and DELETE do modify it.

The trait is an algebraic specification consisting of two parts:

- A signature, which contains the declaration of a set of function symbols (the syntactic part);
- A set of equations that define the semantics (meaning) of the function symbols (the semantic part).

The symbol table abstract data type specified in the interface specification ranges over values denoted by terms of sort S (see Fig. 3).

```
symbol_table is data type based on S from SymTab
    init = proc () returns (s:symbol_table)
        ensures s' = emp ∧ new (s)
    insert = proc (s:symbol_table, k:key,v:val)
        requires ~ isin (s,k)
        modifies (s)
        ensures s' = add (s,k,v)
    lookup = proc (s:symbol_table,k:key) returns (v:val)
        requires isin (s,k)
        ensures v' = find (s,k)
    delete = proc (s:symbol_table,k:key)
        requires isin (s,k)
        modifies (s)
        ensures s' = rem (s,k)
    end symbol_table
SymTab: trait
    introduces
        emp :              →S
        add : S, K, V      →S
        rem : S, K         →S
        find: S, K         →V
        isin: S, K         →Bool
    asserts
        S generated by (emp, add)
        S partitioned by (find, isin)
        for all (s:S, k,k1:K v:V)
            rem (add(s, k, v), k1) = =  if k = k1 then s
                                        else add (rem(s, k1), k, v)
        find (add(s,k,v),k1) = = if k = k1 then v
                                 else find (s,k1)

        isin(emp,k) = = false
        isin(add(s,k,v),k1) = = (k = k1) V isin(s,k1)
    implies
        converts(rem,find,isin) exempting (rem(emp),find(emp))
end SymTab
```

Figure 3 . Larch symbol table specification [7]

There are several clauses (in Fig. 3):

- The **generated by** clause states that all symbol-table values can be represented by terms composed of two function symbols only, emp and add;
- The **partitioned by** clause intuitively states that two terms are equal if they cannot be distinguished by any of the functions listed in the clause;
- The **exempting** clause expresses exceptions from the equation writing rule, meaning that there are no right hand sides for rem(emp) and find(emp).

One can notice that the user-defined function symbols in a Larch trait are the same as those which appear in the pre- and postconditions of the interface specification.

It can also be noticed that Larch has no special built-in notation (as, for example, VDL) which the user can benefit (must memory nothing) or not (large sets of user-defined symbols and equations for them must be provided).

Concurrent case, model-oriented specifications

An unbounded buffer specification in CSP model-oriented specification language is given in Figure 4.

CSP language in this example is based on The Trace Model (see commentary in Chapter 1 on CSP model family) where every process is represented by a set of traces, a trace being a finite or infinite sequence of actions. CSP supports interleaving type of concurrency, which means that in case of communicating processes only one action of one process is performed at a time.

A process can communicate with other processes and with an external environment via a finite number of input-output channels by sending messages. Processes synchronize on actions so that the action of sending output message m on named channel c should be synchronized with the action of simultaneously receiving an input message on C.

The unbounded buffer in Figure 4 is defined recursively with two clauses to handle the empty and non--empty cases.

The first clause,

$P_{<>} = left?m \rightarrow P_{<m>}$

expresses the fact that if the buffer is empty, in the event that there is a message m on the left channel (left?m), it will input it. The notation $x \rightarrow P$, for x = action and P = process, denotes a process which engages in action x and then behaves like P.

The second clause,

$P_{<m>\hat{}s} = (left?n \rightarrow P_{<m>\hat{}s\hat{}<n>} \mid right!m \rightarrow P_s)$

expresses the fact that, if the buffer is not empty, two things can happen:

1) The buffer will input another message n from the left channel, appending it to the end of the buffer;

or

2) The buffer will output the first message through the right channel.

Notation $s \hat{} t$ denotes the concatenation of sequence s to sequence t. CSP uses $\mid$ to denote choice operator, that is, if x and y are distinct actions, $x \rightarrow P \mid y \rightarrow Q$ describes a process that initially engages in x and then behaves like P, or initially engages in y and then behaves like Q.

It is possible that CSP states and proves properties of its traces, by using algebraic rules on them. The last line in Figure 4 states that the unbounded buffer describes a set of traces satisfying two predicates:

- The first predicate expresses that the sequence of output messages on the right channel is a prefix of the sequence of input messages on the left channel ($s \leq t$ denotes that sequence s is a prefix of sequence t);
- The second one expresses that the process never stops, not being permitted to refuse communication on either the right or the left channel.

$$\text{BUFFER} = P_{<>}$$

$$\text{where } P_{<>} = \text{left?m} \rightarrow P_{<m>}$$

$$\text{and } P_{<m>\hat{}s} = (\text{left?n} \rightarrow P_{<m>\hat{}s\hat{}<n>} \quad | \quad \text{right!m} \rightarrow P_s)$$

$$\text{BUFFER sat}(\text{right} \leq \text{left}) \wedge (\text{if right} = \text{left then left} \notin \text{ref else right} \in \text{ref})$$

Figure 4. CSP specification of an unbounded buffer and condition for proving specification correctness [7]

Concurrent case, property-oriented specifications

In Figure 5 an unbounded buffer is specified by means of temporal logic. Temporal logic is a property-oriented method for specifying properties of concurrent systems. It works with modal operators that make assertions about system behaviour; these operators refer to a temporal logic inference system, which can be analysed by means of past, present and future states.

Commonly used modal operators and their meanings are given below:

- $\Box P$ expresses that predicate P holds for all future states;
- $\Diamond$ P expresses that there is a future state for which predicate P will hold;
- $\bigcirc$ P expresses that in the next state predicate P will hold.

A temporal logic specification is represented by an unstructured set of predicates, all of which having to be satisfied by a given implementation. The formulas in Figure 5 are interpreted with respect to sequences of events. A buffer has a left input channel and a right output channel.

Expression $<c\,!m>$ denotes the action of placing message m on channel c. The first predicate $<right!m> \Diamond <left\,!m>$ states that any message transmitted to the right channel ($<right!m>$) must have been previously placed on the left channel ($\Diamond <left\,!m>$). The second predicate,

$$(<right!m> \wedge \bigcirc \Diamond <right!m'>) \Rightarrow \Diamond(<left!m> \wedge \bigcirc \Diamond <left!\,m'>)$$

expresses the fact that messages are transmitted in FIFO order; if message m placed on the right channel ($<right!m>$) is preceded by some other message m', also on the right channel ($\bigcirc \Diamond <right!m'>$), there must have been a preceding action (the second $\Diamond$) of placing m on the left channel ($<left!m>$) and an even earlier event that placed m' on the left channel ahead of m ($\bigcirc \Diamond <left\,!m'>$). The third predicate,

$$(<left!m> \wedge \bigcirc \Diamond <left!m'>) \Rightarrow (m \neq m')$$

expresses the fact that all messages are unique. For each message m on the input (left) channel and for each previously placed message m' on the left channel ($\bigcirc \Diamond <left!m'>$), m and m' are not equal. This is an assumption of the environment and it is essential for

specification validity. Without it, a buffer that transmits duplicate copies of its input to the output would be considered correct.

The first three predicates state safety properties of the system. The fourth predicate $(<left!m>) \Rightarrow \Diamond(<right!m>)$ states a liveness property, that is, each input message will eventually be transmitted.

(1) $<right!m> \Rightarrow \Diamond <left!m>$

(2) $(<right!m> \wedge O \Diamond <right!m'>) \Rightarrow \Diamond(<left!m> \wedge O \Diamond <left!m'>)$

(3) $(<left!m> \wedge O \Diamond <left!m'>) \Rightarrow (m \neq m')$

(4) $(<left!m>) \Rightarrow \Diamond(<right!m>)$

Figure 5. Temporal logic specification of an unbounded buffer [7]

3.3. FORMAL VERIFICATION OF SPECIFICATIONS

The first step in the formal verification of specifications is to informally check the way formal critical requirements reflect customers' critical requirements. This step is informal because customers' requirements are informal.

Another step is to prove that the highest-level specifications are consistent with the formal critical requirements. There are two different approaches based on whether model-oriented or property-oriented specifications are dealt with.

For model-oriented specifications, the effect of performing each operation has to be specified based on the fact that certain conditions must be satisfied when the operation is invoked; for each operation, there are entry (pre-) and exit conditions (postconditions) and if the system state, before invoking the operation, satisfies the entry conditions, the state after the operation has been executed will satisfy the exit conditions. When proving consistency of model-oriented specifications, one must verify that the initial state satisfies the formal critical requirements and that every operation preserves the critical requirements (operation invariants).

For property-oriented specifications, formal verification is performed by means of building homomorphisms between abstract data types on the first (critical requirements) and second level (highest-level specifications) which must preserve critical requirements.

An example is given below (taken from [2]) for property-oriented specifications refinement and formal verification.

The used specification language is called ASPIK and is a part of the ISDV system developed in Germany.

Informal specification: Some sort of container is needed, that has a limited capacity. Things must be put into it and taken out, and the container should be organized so that, taking out an element, the element put before should be yielded.

The result of this informal specification could be the following specification, which assumes a sort ELEM, describing the things to be put into the container and a sort BOOL, describing the truth values; sort CONTAINER is to be built:

```
spec LIMITED-LIFO
/* 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 newly introduced sort CONTAINER has four operations, which are used for stating its essential property: as long as a container is not filled, putting an element into the container and taking an element out again yields the same element. Nothing is stated about what happens when the container is full.

The above specification is refined by adding further axioms (Figure 6). This new specification uses more keywords with the following meaning:

- Constructors denote those operation symbols (taken from the signature above) that are going to be the basis for the definition of all the others (here EMPTY and PUSH);
- Auxiliaries/define-auxiliaries denote new operation symbols based on constructors and necessary for defining the other operation symbols in the signature; in Fig. 6, DEPTH is the auxiliary, which has domain STACK and range NAT (the natural numbers); function SUC, which appears when defining DEPTH, is the successor function, SUC(N) = N + 1, transforming each natural number into its successor;
- Carrier, more precisely, a reachable carrier of sort STACK (the refinement of previous CONTAINER sort) represents the set of all reachable elements (the elements in STACK algebra which are the value of some term) in sort STACK; here the carrier definition is given by means of the constructors and auxiliaries;
- Define-constructor-ops denotes the specification section where constructor symbol operations are assigned a meaning;
- Define-ops denotes the specification section in which sort STACK remaining symbol operations are assigned a meaning.

ASPIK uses if-then-else and case control structures to express axioms for the operation symbols.

The last part of Figure 6 represents the map between the two levels, which relates the LIMITED-LIFO axiomatic specification to the LIMITED-STACK algorithmic one by associating sort CONTAINER with sort STACK and the operations, correspondingly. The ELEM specification is mapped identically to itself as indicated by the basic clause.

Each refinement or implementation step is associated with a set of correctness conditions which can be verified in a stepwise manner and immediately after the relation definition. In case of Figure 6, the correctness condition to be verified is that the algorithmic STACK version satisfies the last-in first-out property expressed by CONTAINER specification, which means that:

ALL S:STACK ALL E:ELEM ⌉ FULL?(S)
→ TOP(PUSH(S,E)) = = E & POP(PUSH(S,E)) = = S.

```
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-constructors-ops
                PUSH(STO,ELO)   =   if (DEPTH(STO) LT LIMIT)
                                    then * PUSH(STO,ELO)
                                    else ERROR-STACK
                EMPTY = * EMPTY;
   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
map(LIMITED-LIFO-FIX → LIMITED-STACK)
                is REFINEMENT;
                base

                                   ELEM;
                sorts CONTAINER = STACK;
                ops                  INTO = PUSH
                                     LAST-IN = TOP
                                     FIRST-OUT = POP
                                     FILLED? = FULL?;
endmap
```

Figure 6. **Specification refinement in ASPIK**

Once proved the consistency of the highest-level specification, the consisteancy of the next lower specification is to be proved and so forth, from level to level until the lowest-level specification is proved to be consistent with the level above it.

Finally, it remains to prove that the high-level programming language implementation is consistent with the lowest-level specification.

By transitivity it can be deduced that the highest-level specification is consistent with the high-level programming language implementation and that they both are consistent with the initial formal critical requirements. Formal verification of the consistency of each specification level is a specification verification and the verification of the last level between lowest-level specification and high-level programming language implementation is a code verification (see Figure 7).

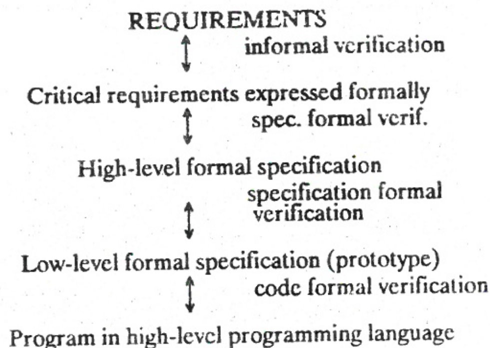

Figure 7. Formal verification hierarchy [8]

3.4. IMPLEMENTATION OF FORMAL SPECIFICATIONS INTO EXECUTABLE SPECIFICATIONS AND PROGRAMMING LANGUAGES

Implementation of formal specification into an executable specification language has been dealt with in Section 3.3. The same ISDV system is going to be mentioned concerning formal specification implementation into programming languages. Here an imperative programming language, ModPascal, is chosen as a candidate for a target language. ModPascal was designed as an extension to the widely distributed and accepted programming language, Pascal. ModPascal remodels some ASPIK concepts being object-oriented and allowing for hierarchical definitions.

Semantically, modules and enrichment constructs from ModPascal are associated with a uniquely constructable algebra; this supports in checking correctness of the transition from algorithmic specifications in ASPIK to ModPascal objects.

Correctness proof of all refinements is done in ISDV by means of two automatic theorem-proving systems:

- The MKRP system, which is a resolution–based theorem prover for first order logic augmented by an induction module;
- The Rewrite Rule Laboratory, which is based on equational logic.

4. ARCHITECTURE OF AN ADVANCED SOFTWARE DEVELOPMENT ENVIRONMENT BASED ON FORMAL METHODS

An advanced software development environment based on formal methods integrates individual tools (as presented in the next table) by means of a project database.

The database will contain representations for:

- Objects, representing data type and process specifications;
- Properties of objects or system model;
- Analysis/specification and proving steps.

phase	methods	generic tools
requirements analysis	consistency checks	- screen-oriented editors - animators
formal specification	model-or property-oriented methods	- syntactic editors - libraries of parameterized specifications
specification formal verification	- testing - consistency proofs	- symbolic interpreters - theorem–provers for consistency proofs
specification implementation into executable specification language	step-by-step refining	- libraries of parameterized specifications - theorem–provers for executable specifications
specification implementation into several programming languages	- precompiling - compiling	- editors - precompilers - compilers - module libraries - (symbolic) debuggers

Table 1. Methods and generic tools in an advanced software development environment based on formal methods

The architecture of an advanced software development environment based on formal methods is very schematically drawn in Fig. 8, from which it may be deduced that such an environment[3]:

– Provides information about its state

and

– Offers tools for

- **the constructive part** - edit, synthesis/transformation, executable specification refining;
- **the analytical part** - (semi-automated) correctness proofs of specification properties:

as well as for the combination of both parts.

If supplementarily endowed with a knowledge base (a database augmented with such capabilities as multiple inheritance, inference capability and constraint maintenance) an advanced software development environment becomes a knowledge-based system [1], which is evolving, as a result of applying its inference facilities (learning facilities) to particular software projects.

Some of the essential requirements [6] for a software engineering knowledge base are given below:

1) A representation model for the knowledge base should exist;
2) This representation model, as part of a software development environment should allow an easy access to knowledge, its modification as well as knowledge execution;
3) The knowledge base should have all the information on the initial version of the life cycle operational model (phases, activities, available techniques and tools, techniques and tools selection criteria, etc.).

The best representation model is considered to be the frame model (lots of information enabling space efficiency and rapid information access), which is similar to object-oriented modelling.

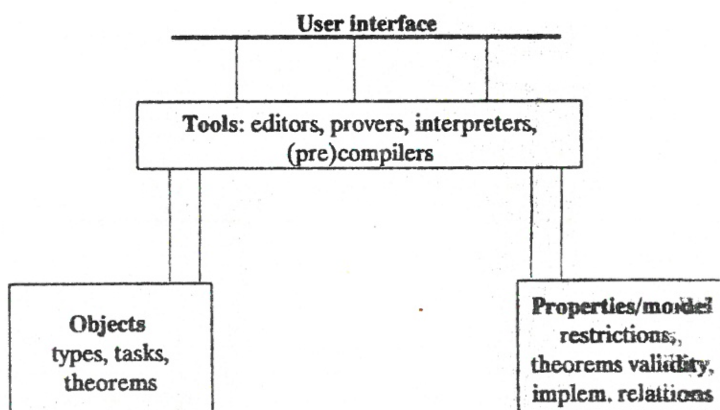

Figure 8. Architecture of an advanced software development environment based on formal methods

A knowledge base for software engineering has not been completely built yet, it takes much time and experience to perform such an action.

Small parts of a software engineering knowledge base [6] have already been built and used for non–critical components.

5. EXAMPLES OF ENVIRONMENTS FOR ADVANCED SOFTWARE DEVELOPMENT BASED ON FORMAL METHODS

Two examples of such environments (PROSPECTRA and FOR-ME-TOO) are given, both being operational for sequential and concurrent systems, as well.

PROSPECTRA (PROgram development by SPECification and TRAnsformation) has been elaborated as an ESPRIT project within a team of specialists from Germany,

Spain, France, Great Britain, Denmark. In PROSPECTRA a development is represented as a formal object having two roles:

- Documentation for already performed analysis/design actions/decisions;
- Plan for future developments.

Thus, a PROSPECTRA development can be used to abstract from a particular development to a class of similar developments, that is a development method, incorporating a certain strategy:

- An elementary development step is a program transformation, the application of a transformation rule that is generally applicable;
- A particular development then becomes a sequence of rule applications.

The PROSPECTRA support software development system guides the user by a step-by-step refinement to decide, starting from an initial set of rules, what transformations to apply, strictly having in mind the observance of specification correctness over the entire development process.

A wide-spectrum specification language starting from formal specifications to Ada programs (known as P-AnndA-S), its semantics covering concurrency aspects, is defined for PROSPECTRA. The PROSPECTRA project has made significant advances in the field of the so-called 'transformational approach' (the leader in this area being the CIP Project) and brought this approach closer to industrial use. Another ESPRIT project aiming to show the feasibility of applying PROSPECTRA methodology for developing correct software based on transformations, is PROSPECTRA-D (Demonstration of PROSPECTRA methodology). Experiments were conducted in PROSPECTRA-D for specifying some examples with the specification language P-AnndA-S, which helped to gradually introduce this new methodology to industrial projects.

FOR-ME-TOO (FORmalisms MEthods and TOOls) is another ESPRIT project elaborated by a team from Germany, France and Italy. FOR-ME-TOO objective was to define, implement and experiment both sequential and concurrent software development and systematic software verification and validation technology, based on software components reuse.

Sequential software descriptions reuse and analysis are made in terms of an algebraic specification language, LPG. Concurrent software descriptions reuse and analysis are made by means of several classes of Petri nets.

From the results of practical projects, it can be deduced that through an extensive use of case studies, FOR-ME-TOO helps user understand component reusability throughout development projects.

CONCLUSIONS

1. Formal methods/languages are helpful in detecting errors in early software development phases and can even eliminate certain classes of errors.
2. Formal methods/languages decrease development costs.
3. Formal methods/languages are successfully used on industrial-scale projects.
4. By using formal methods/languages, a software system behaviour can be specified in two ways:
   - Directly, by constructing a mathematical model of the system (model-oriented specifications);
   - Indirectly, by stating a set of properties (axioms) that must be satisfied by the system (property-oriented specifications).

5. **Advanced software development environments support operational life cycle and formal methods/executable specification languages use for both sequential and concurrent software development.**

REFERENCES

[1] BALZER, R., CHEATHAM, TH. E. Jr., GREEN, C., **Software Technology in the 1990's: Using a New Paradigm**, COMPUTER, Nov. 1983.

[2] BEIERLE, C., OLTHOFF, W., VOSS, A., **Towards a Formalization of the Software Development Process, In** D. Barnes and P. Braun (Eds.) PROC. of SOFT. ENG, UK Peter Peregrinus Ltd., London 1986.

[3] BROY,M., GESER, A., HUSSMANN, H., **Towards Advanced Programming Environments based on Algebraic Concepts, in Advanced Programming Environments.** In R. Conradi (Ed.) LNCS 244, Springer Verlag, 1986.

[4] HALL, A., **Seven Myths of Formal Methods**, IEEE SOFTWARE, Sept. 1990.

[5] JONES, C. B., **VDM Proof Obligations and their Justification**, In D. Bjorner et al (Eds.) VDM '87, LNCS 252, Springer Verlag, 1987.

[6] SYMONDS, A. J., **Creating a Software Engineering Knowledge Base, in Advanced Programming Environments**, In R. Conradi (Ed.) LNCS 244, Springer Verlag, 1986.

[7] WING, J. M., **A Specifier's Introduction to Formal Methods**, COMPUTER, Sept. 1990.

[8] KEMMERER, R. A., **Integrating Formal Methods into the Development Process**, IEEE SOFTWARE, Sept. 1990.