Appl. Math. and Comp. Sci., 1994, vol.4, No.4, 643–660

TOWARDS AUTOMATIC CORRECTNESS VERIFICATION OF REAL-TIME PROGRAMS

TOMASZ SZMUC*, PIOTR SZWED*

An algebraic approach to correctness verification of graphical software specifications is presented. LACATRE graphical language based on multitasking and providing real-time functions is considered as a specification tool. The verification process is distributed into four stages. The two initial ones are related to a translation of LACATRE specification into lower description level (Communicating Real-Time State Machines). The third stage deals with a generation of the process in the form of a graph of states and automatic relative correctness verification. The fourth stage is related to correctness verification for user-defined criteria. The proposed verification method examines the relative correctness concept introduced in former papers.

1. Introduction

Correctness verification of specification is a very important task that should be solved in initial phases of software life cycle. This is due to the nature of specification, being a coarse grained description of software. This fact causes that errors are related to a larger domain (e.g. system structure) and their reasons (sources) may be hidden in a large context of the program code. Correctness verification of specification is then very crucial and should be completed in the specification/design phases. Otherwise, undetected errors would be passed to the next phases, increasing difficulties of their detection in the verification and testing phase.

LACATRE specification language (Schwarz *et al.*, 1991; Schwarz 1992) has been chosen for specification of real-time programs. The language is used for graphical specification of applications and provides typical real-time primitives. The specification covers the Preliminary and Detailed Design phases in the software development cycle (IEEE/ANSI Std., 1986). The language renders it possible to express the dynamical behaviour and relationships of real-time or concurrent system components providing structural approach to the designed application, however does not model a flow of data values. The LACATRE (LA4 in French abbreviation) language may be used as a higher layer over several real-time operating systems e.g. iRMX, VRTX 32, VxWorks, OS-9000 and CRAFT although it has been used at the beginning to provide the tools for design of applications running within the iRMX system. The basic concept is to let a developer to concentrate on the design of an application remaining programming details to next development phases. The current MS Windows version

* Computer Science Laboratory (CSL), Institute of Automatics, University of Mining and Metallurgy, al. Mickiewicza 30, 30–059 Kraków, Poland

of the language may be considered as a CASE tool (Schwarz *et al.*, 1993b). This version (developed in cooperation with LISPI laboratory, INSA de Lyon and CSL) provides the tools for graphical programming. Graphical specification is translated into the corresponding C code constituting a skeleton of the application program for a target operating system. The translator for iRMX system is available, those for other systems are under development at LISPI laboratory.

The paper concentrates on simulation and verification of LA4 programs. The verification is distributed into several layers, and relative correctness methods (Szmuc, 1991) are applied. Communicating Real-Time State Machines (CRSM) description (Shaw, 1992) constitutes an intermediate layer between LA4 and process specifications. The process description is used in the relative correctness verification. Roughly speaking the idea of the multilayer approach lies in modelling LA4 objects by a set of state machines and all the communications between objects as state transitions. The collection of state machines is the basis for the process generation.

2. Specification Language LA4 (an Overview)

LA4 specification may be represented in two modes: graphical (LA4_G) and textual (LA4_T) one. The LA4_G representation is syntactically equivalent to LA4_T specification. The translation between two modes is achieved on-line during programming being carried out usually in graphical mode. The LA4_T code may be translated into chosen target language and operating system.

An application specified using LA4_G language is a scheme containing LA4 *objects* i.e. the static system components and *primitives* describing communications between objects. Two types of objects: *programmable* and *configurable* are provided. A programmable object may invoke some actions in a system. This class contains objects: *task, interrupt* and *alarm*. The definitions appear in LA4_T code in the form of a sequence of instructions corresponding to performed actions (Fig. 1.). Configurable objects (*semaphore, mailbox, message, message pool, resource* and *event*) may be connected to actions performed by programmable objects. Configurable objects perform no action explicitly, therefore they are only declared but have no body in the LA4_T code.

Primitives may be divided into two groups: global (*creation, destruction* primitives) and specific for an object (for example *SEND_TO_MBX* for a mailbox, *WAIT_ON_SEM* for a semaphore).

Objects and primitives have their own graphical representations. The grammar of LA4_G distinguishes certain regions in objects called *bars*. Any object has *state* bar that may be accessed by global primitives and primitives inquiring or changing object's state (those are also denoted state calls). Programmable objects have *progress bar* where primitives start. The primitives are represented by oriented broken lines with an attached graphical symbol and some parameters. The configurable objects have *action bars*; at those regions the primitive symbols end.

The progress bar of any programmable object may be extended by *algorithmic forms*. Algorithmic forms represent program control flows and correspond to classical high-level languages structural constructs: *if-then-else, switch-case, repeat, while, forever* (infinite loop) and *procedure*.

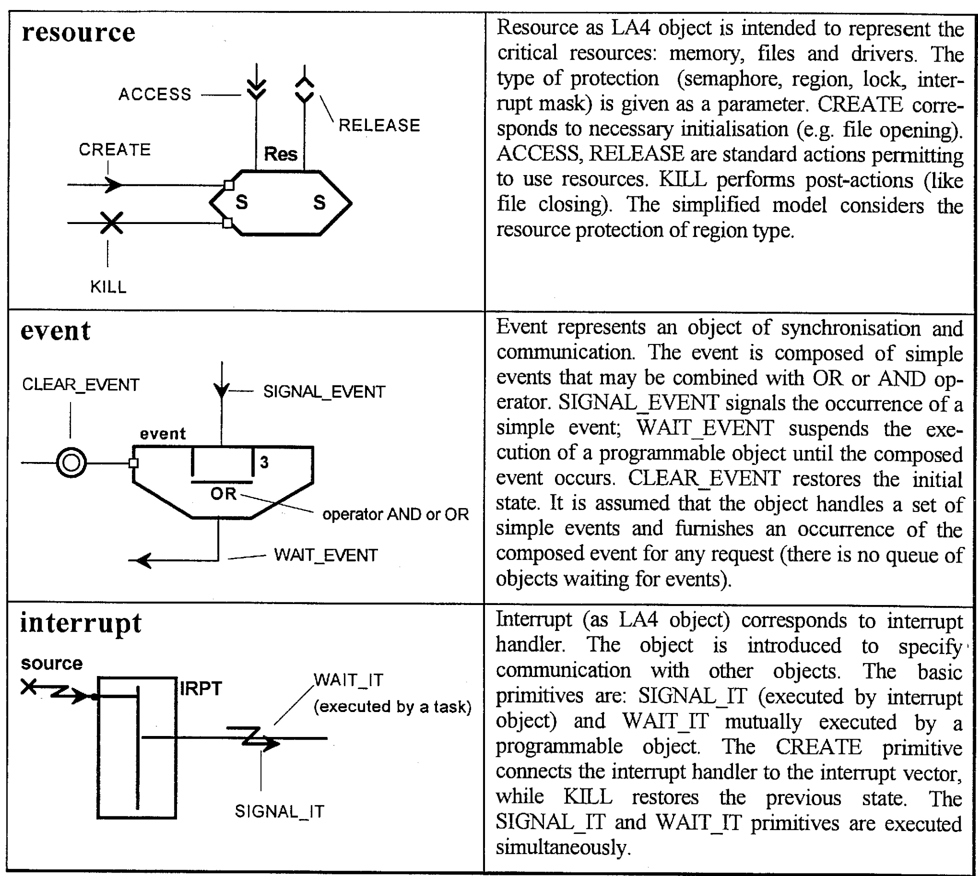
An example of LA4 application is presented in Fig. 1.

2.1. Objects and Primitives

A short description of basic LA4 components is given in the section. The detailed definition of the language may be found in (Schwarz, 1992) and some supplements in (Schwarz *et al.*, 1993a; 1993b). Selected objects and primitives are presented in Table 1, to make a brief introduction to further considerations.

Tab. 1. Selected LA4 objects and primitives.

Graphical representation	Description
task	Task is the basic LA4 object corresponding to a sequence of activities nearly always simple and repetitive. The initialisation sequence is placed before FOREVER algorithmic form. Other algorithmic forms permit to change the execution path. (In the example if cond is TRUE, then the primitives placed in the IF-THEN-ELSE brackets will be executed). The DELAY primitive suspends the task for a given period of time. SUSPEND makes the task inactive until RESUME is invoked. CREATE and KILL are global primitives for creation and destruction. Priority is the only one parameter of a task. More detailed task model will be presented afterwards.
semaphore	Semaphore is a classic object of real-time systems. It consists of a unit counter and a queue storing the requests of objects waiting for the units. SEND_TO_SEM stores numbers of units in a semaphore; WAIT_ON_SEM (called by a programmable object) places the request for units in the queue. The queues may be managed using FIFO or priority polices. Programmable object is allowed to take the units if it is the first in the queue and if the demand may be satisfied.
mailbox and message	Mailbox stores messages. Any message is a data structure or a pointer to the structure (depending on the target operating system). It is assumed that any programmable object after sending a message to a mailbox has no right to access the data stored in the message; on the contrary, the object receiving a message has unlimited access to its content. The mailbox has two queues: the first (SEND_TO_MBX) stocking messages and the second (WAIT_ON_MBX) storing requests (as for semaphores). LA4 allows us to specify also timeout and overflow control.



Object	Description
resource	Resource as LA4 object is intended to represent the critical resources: memory, files and drivers. The type of protection (semaphore, region, lock, interrupt mask) is given as a parameter. CREATE corresponds to necessary initialisation (e.g. file opening). ACCESS, RELEASE are standard actions permitting to use resources. KILL performs post-actions (like file closing). The simplified model considers the resource protection of region type.
event	Event represents an object of synchronisation and communication. The event is composed of simple events that may be combined with OR or AND operator. SIGNAL_EVENT signals the occurrence of a simple event; WAIT_EVENT suspends the execution of a programmable object until the composed event occurs. CLEAR_EVENT restores the initial state. It is assumed that the object handles a set of simple events and furnishes an occurrence of the composed event for any request (there is no queue of objects waiting for events).
interrupt	Interrupt (as LA4 object) corresponds to interrupt handler. The object is introduced to specify communication with other objects. The basic primitives are: SIGNAL_IT (executed by interrupt object) and WAIT_IT mutually executed by a programmable object. The CREATE primitive connects the interrupt handler to the interrupt vector, while KILL restores the previous state. The SIGNAL_IT and WAIT_IT primitives are executed simultaneously.

2.2. Example of LA4 Application

An example of LA4 application is given in Fig. 1. The system is composed of three tasks: Producer, Consumer and MainTask. Producer creates a message (i.e. allocates memory) and stores it (its pointer) in BufMbx mailbox. Consumer fetches a message and processes it and then destroys the message (frees memory). BufMbx may contain at most 50 messages. If the buffer is full, Producer should wait until Consumer takes at least one message. If BufMbx is empty, Consumer will wait until a message is stored. The process halts after finish condition is satisfied (it may be set in additional code in a target system). All objects apart MainTask are explicitly created and destroyed. MainTask is created and destroyed by the operating system in initial and final phases of the application execution.

3. Relative Correctness Verification

The relative correctness concept (Szmuc, 1991; 1992; 1993) has been applied for verification of LA4 specifications. The idea of the relative correctness consists in checking if the program (system) meets requirements. It is an analogy (on the verbal

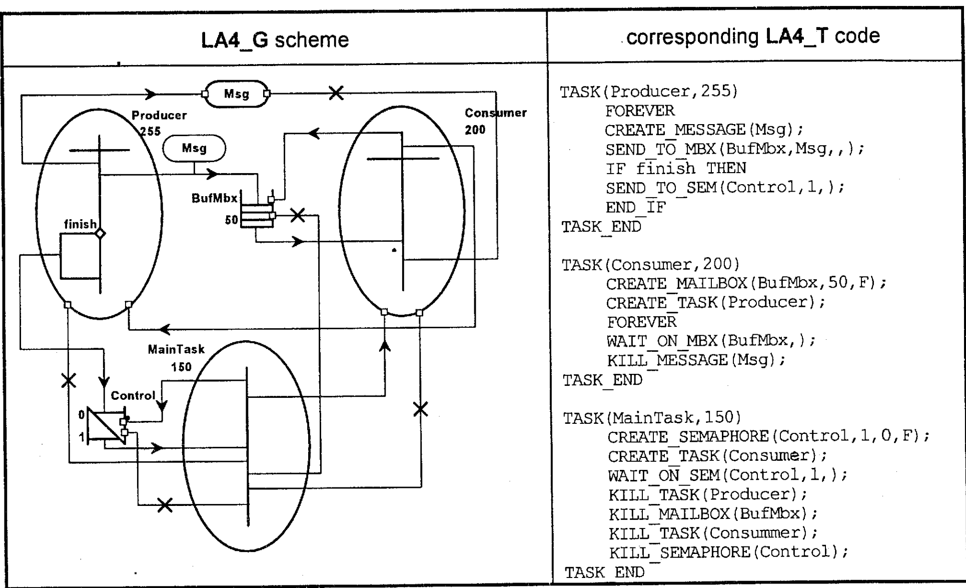


Fig. 1. The LA4 application: LA4_G scheme and corresponding LA4_T code.

description level) to the correctness verification defined in IEEE Software Standards (IEEE/ANSI Std., 1986). As regards the formal description, the following three objects are considered:

- verified process – described by a graph;
- criterion process – defined by another graph, but usually simpler than the first one;
- correctness relation, which defines correspondence between selected (characteristic) states in the verified and the criterion processes.

The relative correctness notion is described using algebraic language – algebra of processes (Szmuc, 1989). This property causes that the methodology is independent of a language specifying the verified system and may be used on any layer of system description as well as in any phase of the software life cycle (IEEE/ANSI Std., 1986). The generality of the tools provides homogenous description of a system and correctness requirements but, on the other hand, causes additional operation, i.e. a translation from applied specification (programming) language into the algebraic form (graph) has to be accomplished. The correctness verification consists in checking whether characteristic states of verified process appear (are encountered during the execution) in order specified by a criterion process. The verification is accomplished with checking a kind of homomorphism between the graphs describing verified and criterion processes correspondingly. This homomorphism is checked in an automatic way by a symbolic execution of the verified reduced process with an analysis of the corresponding changes in the criterion process. A translation of the verified system and correctness requirements into the corresponding process (graph) forms has to be completed before the relative correctness method is applied.

The following conclusion may be drawn as a result of the above considerations. The relative correctness methods require three components (objects): process to be verified P, criterion relation k and criterion process P'. The correctness criterion is the pair $(k,\ P')$. Considering application of relative correctness methods to any specification one should answer the following two questions:

- how to generate a process describing the system starting from the specification?
- how to choose the correctness criterion?

In the presented approach the verified process is generated starting from LA4 specification via an intermediate description layer called CRSM (Communicating Real-Time State Machines) Layer. Two types of criteria have been introduced: criteria generated automatically starting from specification (typical for specifying language primitives and paradigms) and user-defined criteria. The proposed distribution of criteria definition increases the level of automation of the verification process and makes the verification easier and more comfortable for the user. The idea of verification is presented in Fig. 2.

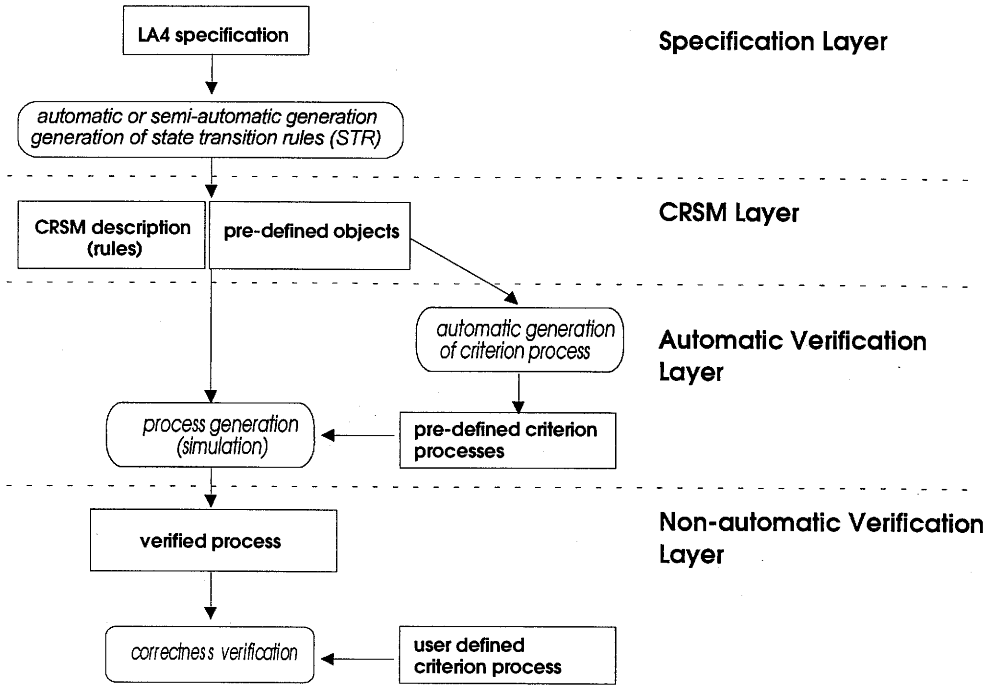


Fig. 2. The verification layers.

The verification process is divided into four stages:

- Specification Layer (LACATRE specification in the case);
- CRSM description layer;
- Automatic Verification Layer;
- Non-automatic Verification Layer.

The verification starts at the Specification Layer (i.e. LA4_T specification). The next step is to derive the CRSM description of an application starting from the specification (i.e. to describe the whole system as a set of state machines and to provide state transition rules reflecting communication mechanisms). Three tasks are accomplished at the Automatic Verification Layer: automatic generation of correctness criterion, generation of the verified process and correctness verification with regard to the previously generated correctness criterion. The last layer is related to the verification of the generated process in the sense of user-defined criterion processes.

The paper focuses on CRSM Layer and Automatic Verification Layer that are the basic concepts in the proposed approach. More detailed explanations dealing with the other layers may be found in the related papers, i.e. (Schwarz, 1992; Schwarz *et al.*, 1991) for LA4 specification and (Szmuc, 1989; 1991; 1992; 1993) for relative correctness verification and user-oriented specification of correctness criteria. The basic idea of the relative correctness verification will be presented in the next subsection to give an introduction to considerations dealing with CRSM and Automatic Verification layers.

3.1. Formal Tools for Relative Correctness Verification

The main notions related to the relative correctness verification (Szmuc, 1989; 1991; 1992; 1993) are recalled to clarify considerations dealing with translation of CRSM specification and automatic correctness verification. The notion of the process is a basic tool in the verification theory. This notion may be interpreted in different ways as well as represented as a graph with selected two sets of its vertices. The definition from (Pawlak, 1968) is given below.

Definition 1. By *a process* we mean a relational structure $P = (S, B, F, T)$, where S is a countable set of states, $B \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of final states, $T \subseteq S \times S$ is a transition relation, and the following condition is satisfied:

$$B \subseteq \mathrm{Dom}\, T \wedge F \cap \mathrm{Dom}\, T \neq \emptyset$$

The notion of the state (and process) is very general and may be interpreted in different ways, e.g. a state may represent:

1. current values of data and a label of currently executed instruction – for sequential (one-process) program model;
2. a vector of current data values and a vector of instruction labels, i.e. any component of the vector may describe a state of a component process in multiprocess system;
3. a restriction of a state of process (system), e.g. values of selected data are considered only;
4. symbol of operation (instruction).

The two first interpretations may be used for description of the verified process, while the last ones may be applied to criterion process specification (selected data values, operations). The main idea of the verification is to check if a program meets its specification (assumed requirements). The verified process is a subject of verification, the criterion process describes correctness requirements, e.g. required order of

execution (appearance) of some selected states (operations). The criterion process is usually specified in a different way, hence (and by other reasons) *correctness relation* connecting characteristic states in a verified process with states in criterion process is defined.

Relative correctness is a relation between the verified process (sequential description of verified system) and the criterion process (specification of correctness requirements). The two classes: partial and total relative correctness are considered. A simplified definition of the correctness notion is given below.

Definition 2. Let $P = (S, B, F, T)$, $P' = (S', B', F', T')$ be processes and $k \subseteq S \times S'$ a relation. We say that the process P is *partially correct in the sense of a correctness criterion* (P', k) *iff* the following conditions are satisfied:

1. Any characteristic state that appears in the process P as *an initial* one is connected (by k relation) with states belonging to the set B';
2. Every characteristic state that appears in the process P as *a final* one is connected (by k relation) with states belonging to the set F';
3. For any transition $(s', s'_l) \in T'$ in the criterion process whose states belong to the counter domain of relation k there exists a semicomputation (in the process P) that:
   - begins with a state connected by k with s',
   - finishes in a state connected with s'_l,
   - any state in the semicomputation may be at most connected with other successors of s' or it should belong to B'.

Definition 3. Let $P = (S, B, F, T)$, $P' = (S', B', F', T')$ be processes and $k \subseteq S \times S'$ a relation. We say that process P is *totally correct in the sense of a correctness criterion* (P', k) *iff* P *is partially correct in the sense of* (P', k') and the following conditions are satisfied:

1. A set of characteristic states appearing as *initial* ones is equal to B'.
2. A set of characteristic states appearing as *final* ones is equal to F'.
3. $\mathrm{Ran}\, k = S'$.

Roughly speaking process P is *partially correct in the sense* of (P', k) when the behaviour of P observed via relation k is consonant with P'. The property does not mean that k-mapping of P into P' covers the whole process P'. This property is reached by processes being *totally correct in this sense.*

These notions may be treated as generalisations of partial and total correctness which are defined for sequential programs. Moreover, strong analogies between safety and liveness properties (Manna and Pnueli, 1981) may be observed.

The correctness verification consists in simulation of the verified process (usually its reduced form) and checking if the corresponding (by relation k) states transitions are in accordance with the ones specified in the criterion process. The two main constructs are used in the verification (see Szmuc, 1989; 1991; 1992; 1993 for details):

- *coupled process* describing transitions in the verified process and the corresponding transitions in the criterion process;
- *local testability theorem* forming conditions that should be tested at any state (transition) of the verified process to meet given correctness criterion.

The verification of the relative correctness may be carried out in two ways, as the so-called static verification (when the coupled process is constructed), and as dynamic correction when running the verified process and the additional module constitute the coupled process.

A process describing a real-time system and a criterion process specifying correctness requirements should be defined before the relative correctness verification is applied. The two processes are generated at the Automatic Verification Layer.

3.2. CRSM Layer of Description

The CRSM layer is composed of two basic types of elements: pre-defined classes modelling LA4 objects (tasks, mailboxes, semaphores) and transition rules (STR) for objects' states reflecting the communication mechanisms. It is assumed that any communication (thus state modification) may occur between at most two objects. Any LA4 primitive is described by one or more rules.

The goal of introducing the CRSM layer is to facilitate the simulation and the generation of the verified process in the form of traces containing subsequent system states. The set of all traces produced in the Automatic Verification Layer is then passed to the Non-Automatic Verification Layer, where relative correctness verification in the sense of user-defined correctness criterion is carried out.

It is worthy to remark that as LA4 serves as an overlayer to many target systems and their programming languages, there is no model which is general enough to express an object that would fit directly for all the systems. The goal of the investigations is not to simulate an application in a chosen system, but to develop methodology that may be applied for a large variety of systems. This is the reason that non-deterministic application execution has been assumed whereas in iRMX system applications are usually deterministic (excluding the same priority cases). No assumptions have been made at this stage. That leads to situations when information on the verified application may be redundant for certain target systems but will be exactly satisfactory for others.

3.2.1. Predefined Classes of LA4 Objects

A class of objects describes the behaviour of LA4 objects of a given type. For example *mailbox* class is used to model messages stocking mechanism, *semaphore* class makes possible the simulation of storing and retiring units, *task* class allows a description of activities for programmable objects, etc.

For any class of predefined objects two types of activities may be distinguished:

- actions accomplishing transitions between states,
- interrogations permitting to determine an object state or a possibility to perform some selected actions.

Additionally, for any object $\boldsymbol{o}_i$ an extraction function $E_i : X_i \rightarrow \boldsymbol{I}$ is defined projecting the set of object X_i states into the set of integers. Any system state is obtained as a vector of extracted states of its component objects. In a certain sense the extraction function allows us to select observable components from the system.

Any state of object is built from elementary states expressing important (for modelling) properties. For the task in Fig. 3 we may consider the general state (INITIAL, ACTIVE, FINAL, WAITING, SUSPENDED), the label of current instruction Label, task Priority, DelayCounter supporting time flow during execution of DELAY or in waiting for communication state, MessageSet a set of messages being accessible for the task, etc.

It should be noticed that in general case states of objects cannot be enumerated *a priori*; for instance the contents of the set of messages accessible for a task MessageSet is unknown while designing the class but is to be determined during the simulation.

Any object model is supplemented by its predefined criterion process specifying correct transitions between the corresponding states. The types of transitions in a criterion process supporting the approach are discussed in (Szmuc, 1993). The criterion process is given indirectly – by enumerating all the actions that may be performed on the object at a given state without falling in error. For instance, if the task state is *INITIAL*, then *Create* action is permissible only. Invoking *Destroy* action in this state causes an error (the global Boolean variable *error* is set to TRUE) and stops the simulation.

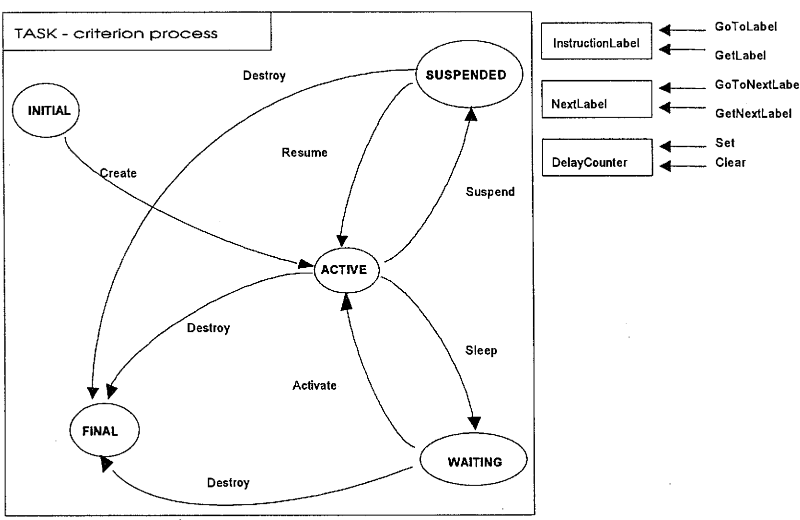


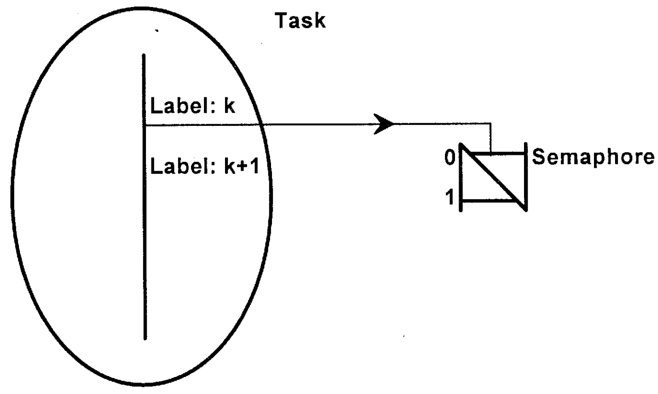
Fig. 3. Task model and its criterion process.

3.2.2. Transition Rules

The CRSM layer is completed by the data that are generated starting from the LA4 specification. The description of a RT system using CRSM language consists of two parts: a declarative part that is a declaration of objects instances (and their classes) used in the application and a descriptive part specifying dynamical behaviour by rules of transition between states.

Any rule is constructed in the same manner, i.e. it is composed of two elements – a *guard* and a *command*. A guard is a logic expression that may reach value

TRUE or FALSE. The expression contains interrogations related to objects joined by the logic operators **and**, **or**, **not**. A command is specified by the sequence of actions changing the objects state (Fig. 4.). Each rule is assigned to the corresponding LA4 primitive invoked by a programmable object. It may describe a communication performed by two objects, a primitive changing states of a programmable objects (DELAY, SUSPEND_ITSELF) or a structural command switching execution control (IF condition THEN ... ELSE).




```
GetLabel(Task)=k  and  CanDeposit(Semaphore,1)   – guard
GoToLabel(Task,k+1);   Deposit(Semaphore,1);     – command
```

Fig. 4. Example of transition rules. A task depositing one unit into the semaphore. Description using C-like syntax.

The whole system is described by a set of state transition rules (STR). It is assumed that the set of STR satisfies the following conditions:

- The language used to describe guards and commands contains a set of interrogations and actions of pre-defined classes and symbols declared in objects instances. The standard logic operators are also used.
- There are no two rules with the same guard in the set of STR.
- No guard reaches value TRUE when the global Boolean variable error is set to TRUE.

3.3. Verification Layers

3.3.1. Criterion Processes

Two types of criterion processes corresponding to Automatic and Non-automatic Verification Layers have been introduced:

- the one specified as a sum of disjoint predefined criterion processes related to LA4 objects; the process is generated automatically and specifies an intuitional requirement of a program *executability*; the relative correctness regarding this process is tested dynamically during the system simulation;

- a user defined criterion process specifying correct system behaviour in the sense of the program *functionality*.

Automatic generation of criterion processes

Specification of a correctness criterion may be a difficult and time consuming task that should be completed before the relative correctness verification. The idea applied in this paper lies in determining a part of the criterion that may be generated automatically; the remaining part is application dependent and is defined by the user. The division and selection should lead to a criterion that:

- may be generated automatically,
- allows us to express basic correctness requirements of the system.

A predefined criterion is attached to an object model. The criterion for the whole application is obtained by the instantation of predefined criteria corresponding to the objects in the application. The generated disjoint criteria related to objects are then summed to obtain an automatic criterion.

The following properties may be checked using correctness criteria specified in an automatic way:

- code redundancy, reachability (if a certain rule has never been used during the generation of traces, the corresponding transition is expected to be unreachable),
- starvation (a programmable object rests at the same instruction label starting from a certain moment to the end of the current trace),
- deadlock (neither a final system state nor a branching state have been reached but there is no rule permitting a change of system state. All the guards reached FALSE value),
- typical sequence of calls to primitives affecting objects. For example, an object should be created first, then it has to perform its characteristic activities, and finally should be destroyed; a task cannot access messages that it has never received, etc.

Let us examine the application presented in Fig. 5. The correctness criterion may be generated using the three component criteria: (P'_{MT}, k_{MT}), (P'_{Sn}, k_{Sn}), (P'_{Sm}, k_{Sm}) corresponding to MainTask, Sender and Sem respectively. The resulting criterion process P' is obtained as a sum of component processes:

$$P' = P'_{MT} + P'_{Sn} + P'_{Sm}$$

and the correctness relation k is defined as

$$k = \Big\{(e, y) \mid e = (e_{MT}, e_{Sn}, e_{Sm});\ e \in \operatorname{Dom} k_{MT} \times \operatorname{Dom} k_{Sn} \times \operatorname{Dom} k_{Sm};\ y \in k_{MT}(e_{MT}) \cup k_{Sn}(e_{Sn}) \cup k_{Sm}(e_{Sm})\Big\}$$

(see Szmuc, 1991).

3.3.2. Process Describing the Real-Time System

A system state is a vector created by aggregation of extracted states of component objects. The process describing the real-time system is generated during the system simulation. The STR specifies constraints for possible transitions in the verified process (i.e. they specify indirectly the T relation from Definition 1). During a transition some actions changing object states are accomplished; any action performed on an object changes its state (when the transition is correct in the sense of the predefined criterion process) or causes that the global variable error is set to TRUE.

As at a given moment of simulation more then one guard may have the value TRUE (the maximum number is equal to the number of programmable LA4 objects existing in the application), the next transition may be chosen among several others. That leads to non-determinism in the verified process.

3.3.3. General View of the Algorithm of Simulation

The simulation algorithm is based on **do-od** construct in the Dijkstra style (Dijkstra and Scholten, 1989). All rules of STR are in the form of guarded commands; any computation begins at an initial state and is continued until at least one TRUE guard exists. The computation halts if all the guards are FALSE; the terminal state is encountered in this case. The type of the terminal state allows us to conclude whether the computation is correct in the sense of the correctness criterion or whether an error or deadlock has occurred.

A so-called MainTask appears usually in LA4 specification (see Fig. 1 and Fig. 5). This task is responsible for creation and destruction of all other elements of the system. MainTask is never explicitly created and destroyed; it is under control of the operating system. The expected correct execution of a program specified with LA4 begins when the MainTask is launched, just before an execution of its first instruction and stops after completing its last instruction. This situation corresponds to the *total correctness* property of the system.

Let us define two predicates in the domain of the extracted system states:

IsInitial(x) $\Leftrightarrow$ no object, apart MainTask, is created; MainTask rests in the state before the execution of its first instruction

IsFinal(x) $\Leftrightarrow$ all objects apart MainTask are destroyed; MainTask has reached its final state

The predicate mentioned above describes correct termination of programs. The others listed below define incorrect terminal states:

IsWarning(x) $\Leftrightarrow$ MainTask has reached final state, but there exist some objects that have not been destroyed

IsDeadlock(x) $\Leftrightarrow$ MainTask has not reached a final state

IsError(x) $\Leftrightarrow$ the global variable error has the value TRUE as a result of the command execution

It is easy to notice that the subsets of the final states corresponding to the predicates are mutually disjoint.

The correct program behaviour may be formulated using Dijksta's predicate transformers:
The system described by a process P is totally correct (the executability criteria) iff

$$IsInitial \rightarrow wp(P, IsFinal)$$

This requirement means that any computation beginning in the state that satisfies *IsInitial* finishes in the state satisfying postcondition *IsFinal.*

Loops

Infinite loops may appear in LA4 specification, e.g. application consisting of one task that executes only FOREVER and DELAY. No errors occur during execution but the process never reaches a terminal state. In such situation the *partial correctness* property is checked.

Pseudo-terminal branching states have been introduced to make simulation algorithm finite. Any system state is a branching state if its extracted state has occurred in the sequence generated earlier. Taking into account the branching states, the partial correctness conditions may be formulated by adding a supplementary predicate *IsBranching* that is satisfied by all pseudo-terminal branching states.

The system described by a process P is *partially correct* (the executability criterion) iff

$$IsInitial \rightarrow wp(P, IsFinal \vee IsBranching)$$

Coming back to the extraction function, the importance of the selection of this function should be emphasized. In a general case the extraction function is not a bijection and it allows us to choose a depth of the simulation by specifying how to distinguish two given states (i.e. when to conclude that the simulation looped into the previously reached state).

Algorithm for generation of state traces

The algorithm for traces generation is presented below. Variable R is a set of rules whose guards are TRUE at the given moment of simulation; rs is used to store the rule currently selected for execution; variable *stack* stocks the pairs (rs, R) to make possible return to the point where bifurcation of traces occurs. Variable x denotes the extracted system state; H stores all states that have appeared in the trace.

```
program generation;
begin extract system state x
if ¬IsInitial(x) then STOP {error in application model}
determine the set R of rules whose guards have the value TRUE
repeat
    {generate a trace tail}
    repeat
        if R = ∅ then break {the predicate IsDeadlock(x) is TRUE}
        select a rule rs ∈ R to execute;
        R := R \ {rs};
        Push( (rs, R) ); {push pair on a stack}
```

execute the command of rule rs;
extract the system state x;
if *error*, **then** *break* {*the predicate IsError(x) is TRUE*}
if $x \in H$, **then** *break* {*the predicate IsBranching(x) is TRUE*}
determine the set R *of rules whose guards have the value TRUE;*
until $\neg (IsFinal(x) \vee IsWarning(x))$;

{*find rule* rs *responsible for bifurcation*}
repeat $(rs, R) := Pop()$;
until $\neg$ empty(*stack*) $\wedge$ $R = \emptyset$;
{*restore the state before the bifurcation*}
if $R \neq \emptyset$ **then for** $i := 1$ **to** *count(stack)* **do** *reexecute rule* r_i *from pair* (r_i, R_i);
execute rule rs;
until *empty(stack)*
{*all traces were generated*}
end.

Time

Time flow has been introduced to the model to ensure the correct simulation of time dependencies of systems specified with LA4. The time is simulated to permit modelling of delay command and all the primitives where timeout control is performed. It is assumed that for any command the time of its execution may be considered. Additionally, the time of switching between the tasks may be taken into account. After a command is executed, the simulated time is increased by the time of the execution. Algorithms based on the time interval specifying the time of execution are also considered.

3.4. Example of Simulation

Two similar examples (Fig. 5) of a system with two tasks and a semaphore are tested:

- **case A** (correct) primitive `KILL_SEMAPHORE` (marked in grey) has been executed by Main Task;
- **case B** (incorrect) — the primitive has been executed by Sender.

This simple example has been chosen to provide an illustration of the presented methods and tools. The example shows the main stages of the verification considered in the paper.

The correspondence between LA4 primitives and transition rules are listed in Tab. 2.

Results of the simulation are shown in Fig. 6 in the form of traces of rules selected during execution. At the end of a trace the predicate satisfied by terminal state is indicated (OK corresponds to *IsFinal*). In the presented example all the traces for system **A** are correct. For system **B** correct traces as well as traces terminating with warnings or deadlock have appeared; there was no occurrence of error, (i.e. all executed commands succeeded, all transitions of states were correct in the sense of the automatically generated criterion process). In spite of this, the generated process is incorrect because the terminal states that do not correspond to final states of criterion process (see Definition 2 and 3) have been encountered.

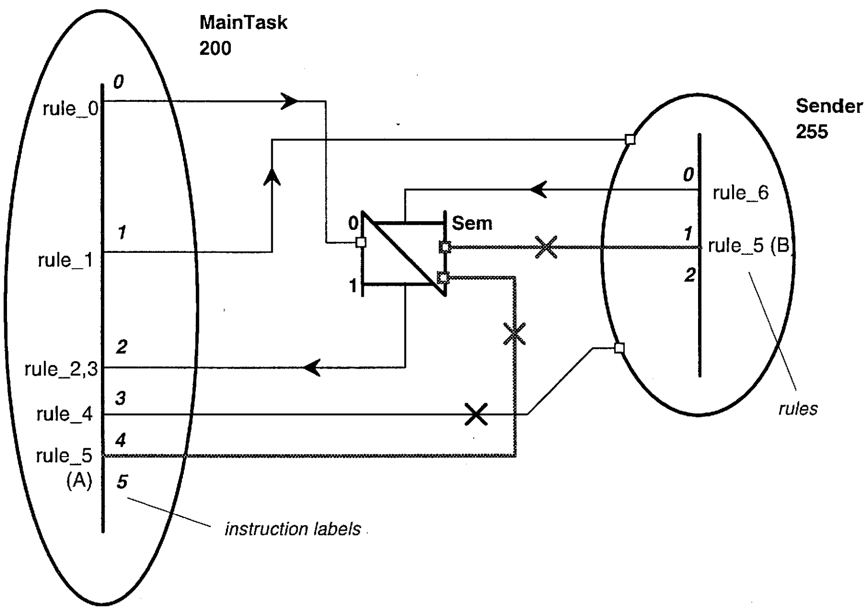


Fig. 5. Two simulated systems differing by a task that executes *KILL_SEMAPHORE.*

Tab. 2. Rules and primitives from Fig. 5.

Rule number	Corresponding LA4 primitive	Executed by
0	CREATE_SEMAPHORE(Sem,1,0,F);	MainTask
1	CREATE_TASK(Sender);	MainTask
2	WAIT_ON_SEM(Sem,1); *(registration of the request)*	MainTask
3	WAIT_ON_SEM(Sem,1); *(action of retiring units)*	MainTask
4	KILL_TASK(Sender);	MainTask
5	KILL_SEMAPHORE(Sem);	MainTask (case **A**) Sender (case **B**)
6	SEND_TO_SEM(Sem,1);	Sender

During the simulation all the traces of system states (or dual traces of selected rules) are generated. It is possible to impose their order by determining a way the rules with TRUE guards are selected for execution. In the examples given below rules have been scheduled by task priority, thus the traces generated as first (placed on the left border) correspond to iRMX operating system.

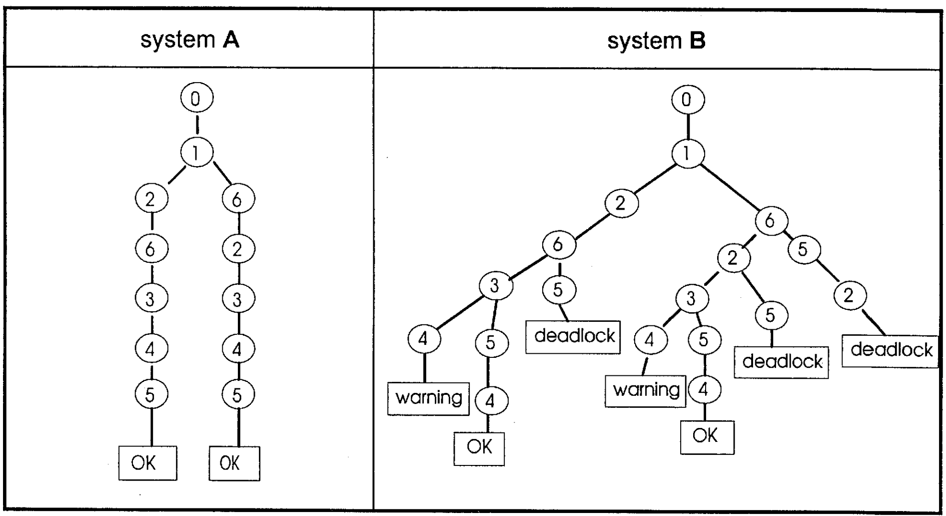


Fig. 6. Results of simulation for systems **A** and **B**.

4. Conclusions and Future Research

Correctness verification of specifications has been proposed in the paper. Graphical specification language LA4 has been chosen as a tool for description of real-time applications. The relative correctness algebraical tools have been applied for the verification. The general model provides homogenous description of the correctness in any layer and phase of the verified software. On the other hand, translation from the application description into an algebraical form has to be accomplished additionally. An intermediate layer (CRSM) has been introduced making the translation and verification more clear and supporting automatic generation of verified and criterion processes. The last property gives an opportunity for distribution of the verification into two layers:

- Automatic Verification Layer related to typical primitives and paradigms of LA4 language,
- user (application) oriented verification layer where correctness criteria should be defined by the user.

Correctness criteria for the second verification layer are determined by applications, hence generation of the correctness criteria has to be accomplished by the user and cannot be done in a fully automatic way. It seems to be purposeful, however, to create tools for fully automatic preparation of data for simulation, a user-friendly interface permitting easy specification of user defined criterion process and the module for visualisation of the execution traces.

The other directions, convergent with LA4 development tendencies (Schwarz *et al.*, 1993b) may include automatic generation of criterion processes for subsystems (**process** and **agency** in LA4) and further hierarchization of correctness verification layers.

LA4 language has been chosen as an example of a specification tool. The language is simple, clear and significantly expressive so it may be chosen as a representative for the class of specification languages. It must be mentioned, however, that the verification concept is general (see Fig. 2) and may be applied for other languages and systems.

Acknowledgements

This paper was supported by the Polish State Committee for Scientific Research under grant No.8 8528 9102 "Computer Systems for Control and Decision Making. Theoretical Foundations and Computer Aided Design".

References

Dijkstra E.W. and Scholten C.S. (1989): *Predicate Calculus and Program Semantics.* — Berlin: Springer Verlag.

IEEE/ANSI Std. (1986): *Guide for Software Quality Assurance Planning.* — IEEE Computer Society, Washington, Brussels, Tokyo.

Manna Z. and Pnueli A. (1981): *Verification of concurrent programs: The temporal framework*, In The Correctness Problem in Computer Science (Bayer R.S. and Moore J.S., Eds.). — Int. Lecture Series in Computer Science, Academic Press, pp.215–272.

Pawlak Z. (1969): *Programmed machines.* — Algorytmy, v.10, No.1, pp.5–19, (in Polish).

Schwarz J.-J., Skubich J.-J. and Miquel M. (1991): *A graphical language for multitasking real time application design. An application to iRMX programming.* — Proc. 7th Int. Conf. IRUG, Baltimore, USA, pp.1–7.

Schwarz J.-J. (1992): *Language d'aide a la conception d'applications multitaches temps reel.* — APII-AFCET., v.26, No.5, Dunod, pp.355–385.

Schwarz J.-J., Skubich J.-J., Szwed P. and Maranzana M. (1993a): *Real time multitasking design with a graphical tool.* — 1st IEEE Workshop on Real Time Applications, New York, pp.33–36.

Schwarz J.-J., Skubich J.-J., Szwed P. and Maranzana M. (1993b): *CASE tools for iRMX Applications.* — Proc. 8th Int. Conf. IRUG, Portland, Oregon, USA, pp.43–49.

Shaw A.C.(1992): *Communicating real-time state machines.* — IEEE Trans. Software Engineering, v.18, No.9, pp.805–816.

Szmuc T. (1989): *Correctness of Concurrent Software Systems.* — Scientific Bulletins of the University of Mining and Metallurgy, Automatics, Kraków, Poland, v.46, (in Polish).

Szmuc T. (1991): *Partial and total relative correctness for analysis of concurrent systems.* — Appl. Math. and Comp. Sci., v.1., No.1, pp.27–34

Szmuc T. (1992): *Relative correctness of real-time programs*, In Proc. IFAC/IFIP Int. Workshop on Real-time Programming, Bruges (Boullart L. and de la Puente J.A., Eds.). — Pergamon Press, pp.173–177.

Szmuc T. (1993): *Relative correctness verification of concurrent systems.* — Scientific Bulletins of the University of Mining and Metallurgy, Automatics, Kraków, Poland, v.64, No.1546, pp.91–106

Received: December 3, 1993
Revised: April 5, 1994