Proc. Estonian Acad. Sci. Phys. Math., 1993, 42, 1, 63—76
https://doi.org/10.3176/phys.math.1993.1.07

UDC 681.3.06:62—52:681.324

Leo MÕTUS *

TIME CONCEPTS IN REAL-TIME SOFTWARE **

(Presented by J. Engelbrecht)

Abstract. An attempt is made to systematize the variety of time-bound terms, understandings and theories used for describing, analysing and verifying the properties of real-time (embedded) software.

Time concepts usable in real-time software are projected to the background of those used in other fields of science and technology. This topic has not been too widely discussed earlier. Recently some interest in the essence of time as used in programming has been shown by Hoogeboom and Halang [1], Mõtus [2], Mõtus and Rodd [3].

The conclusion has been reached that real-time computing in a wide sense — i. e. including specification, design, implementation and maintenance — actually makes use of at least three different philosophical time concepts within one system.

Introduction

Our everyday life becomes, with surprising speed, more and more dependent on real-time systems. This is accompanied, quite naturally, with an increasing number of specialists working in the area of developing and maintaining such systems. Many disparate methods, methodologies and theories have emerged, all claimed to be extremely useful in specifying, designing, implementing, testing and/or maintaining a real-time system. It is no wonder that even an old-timer in the domain may get confused with all the various notions, concepts and theories. At least, this happened to the author of this paper. Therefore it was decided to take a quasi-philosophical look at some of the existing results in a hope that this might somewhat clear the essence of real-time systems. Consequently, the paper contains elements of a survey intertwined with subjective speculations about the essence of time in general and time in computer systems in particular.

As stated by Hoogeboom and Halang [1], practical consensus has been reached on the definition of real-time system's functioning — DIN 44300 fixes that real-time functioning is "an operation of a computer system in which the programs for processing of incoming data are constantly operational so that the processing results are available within a given time interval; depending on the application, the data may appear at random or at predetermined times".

Arguments about the essence of "real-time" still do not indicate any tendency to converge to a generally agreed and unambiguous definition. One of the goals of this paper is to propagate our understanding that the attribute "real-time" does not denote time as such. Rather, it characterizes computer system's ability to establish correspondence between different time-measuring and/or counting systems.

This understanding is concordant with the image of a real-time system as being a computer system that functions in direct and immedi-

* Tallinna Tehnikaülikool (Tallinn Technical University), Ehitajate tee 5, EE-0026 Tallinn, Estonia.

** This paper was originally presented at the IFAC/IFIP Workshop on Real-time Programming, 23—26 June, 1992, Bruges, Belgium.

ate interaction with one or more technical and/or natural systems (the latter systems form the environment that embraces the computer system, also called the surroundings of a computer system). Usually, in each of the mentioned systems there exists its own time-measuring/counting method(s). For a successful cooperation it is essential that the computer system (which is supposed to influence the behaviour of the other systems in a clearly-defined direction) is able to understand, analyse and forecast the functioning of its partners.

Difficulties. One can distinguish two sources of difficulties hindering the understanding of and agreeing upon the basics of real-time systems. The first source follows from the fact that a real-time system must be considered as part and parcel with its environment — consequently different specialists with different understandings and priorities must be involved. Certain level of mutual understanding has been achieved, it still needs considerable improvement.

The second source of difficulties has been pointed out by Kurki-Suonio [4] and can be reduced to the fact that in the software process resulting finally in a real-time system, it is often needed to consider real, existing systems, as well as their models — and, of course, we often mix up what is what. Obviously we have to be aware of what belongs to the models of reality and what is a property of a real system.

Connections to traditions. Computing science people study a class of reactive systems (see, for example, [5]) — i.e. computer systems with an on-going interaction with its environment. The class of reactive systems is usually opposed to transformational systems (computation-oriented systems having strictly limited and well guarded interaction with humans only). From an engineer's point of view, the class of reactive systems embraces both soft real-time and hard real-time systems — a good definition of those is given by Hoogeboom and Halang [1].

Reactive and transformational systems resemble open and closed systems (as used in thermodynamics to denote energy-wise closed and open systems). This analogy was pointed out by Mõtus and Rodd [3]. Reactive systems are information-wise open, whereas transformational systems may be considered as informationally closed systems. This analogy adds to the importance of a joint study of a real-time computer system and its environment, and to the importance of a clear correspondence between time-counting methods in the interacting partners. This analogy also helps us to understand more exactly the essential differences between reactive and transformational systems, and to properly emphasise the peculiarities of hard real-time systems as a sub-class of reactive systems.

Basic differences between reactive and transformational systems have been thoroughly discussed in [3]. In the context of this paper (and real-time systems) two of them are of special interest:

— truly asynchronous (in addition to well-known synchronous and asynchronous modes) execution mode of interacting processes, i.e. the activation instants of interacting processes are independent of each other; this mode usually results from incomplete knowledge of causal relations and/or from their substitution by timing constraints;

— time-selective interprocess/intercycle communication. The latter is closely related to a wide-spread understanding that real-time systems consist of non-terminating programs. This interpretation has remarkably hindered the study of time-selective communication. In many cases it is sensible, and even more correct, to consider a non-terminating program as repeated activation (may be for an infinite number of times) of a terminating program — this results in a concept of cyclic execution of

terminating programs, which leads us to the problem of time-selective interprocess/intercycle communication.

Timing problems are, according to many authors, reducible to safety properties of a program. Henzinger, Manna and Pnueli [5] demonstrate that depending on a particular timing problem it is reducible either to a safety property or to a liveness property. Another area that needs additional research is that of linking fairness with time constraints.

About this paper. Hoogeboom and Halang [1] have considered time in real-time systems against the background of human civilization and general philosophy of time, providing thus an excellent basis for this paper. We have tried to take another step towards systematizing the existing (in programming) time-bound terms. From a very pragmatic point of view, we stress some philosophical connections and survey some papers in computing science domain dealing with various aspects of time and its processing.

Some pragmatic terms used in software practice

Traditionally, real-time systems (and their software) has been considered to be closely connected to time and its properties. Quite often the presence of explicit time in a computer system has been considered as the main hallmark of a real-time system. As stated by Stankovic [6], the former statement is a slight exaggeration. Mõtus and Rodd, [3], suggest that the most important problem in real-time systems is matching the behaviour of co-operating dynamic systems (one of which is the computer system). Anyway, time seems to be a suitable tool for obtaining and proving the match between the co-operating dynamic systems that form the real-time system.

Kopetz, [7], states that in real-time systems time and data form an atomic unit of information. This does not mean, however, that time can be considered as just another variable. The computing science community has learned during the past decades that time handling, its modelling and its presentation in computers is quite tricky, and it needs special care. The author of this paper has a feeling that the necessary paradigms are already present, the majority of required theories have been suggested and published. The next problem to be solved seems to be the selection of the conceptually well-founded minimal sufficient set of paradigms, theories and notions. Probably the general solution to this problem cannot be found. Still, this paper tries to give some hints that might help to find a way through the "time jungle" for a particular case of real-time systems.

The "time jungle" can be illustrated by a sample list of questions about time characteristics as stated by Shoham [8]:

— is time discrete or continuous?

— is time unbounded?

— if time is continuous, is it dense, and if so, is it complete, in other words, can continuous time be modelled by rational or real numbers?

— is time branching or linear; cyclic or acyclic?

— if time branches, should past and future be handled differently?

Note that sometimes philosophers use "real time" to denote the fact that time has been modelled by real numbers (see, for example, Benthem [9]).

This list of questions can be extended by adding problems of manipulating with different time scales (see, for example, Corsetti and colleagues [10]), and by considering the huge variety of possible ways that may define time when specifying system's requirements (see, for example, Halbwachs [11]).

In this section, some of the practically used time-bound terms are surveyed. The terms are classified according to three different standpoints of the observer — time as seen by an "implementor", time as seen by a "verifier" and time as seen by a "specifier". Terms are grouped according to their most frequent use, though they make sense, can be used and are used in the other life-cycle stages as well.

Time as seen by an "implementor"

These are terms that usually take their origin from the implementation stage of a software life-cycle. For the "implementor" the only reality is computer system, the rest of the world is a model. The model is rather detailed in the case of a real-time system (reactive system, open computer system), and may be very schematic in the case of a transformational (closed) computer system.

"Implementor" is interested in time that provides a basis for organizing the execution of programs, or may be used in the implementation-bound theories (like scheduling).

The following four definitions are taken from Kopetz, [12].

Physical time — a reference is established by counting cycles of a physical, strictly periodic process (ticks of a physical clock). Usually there is a measure to express the distance of one tick from another, i.e. physical time is usually a metric time.

Logical time — a reference is established by counting specified significant events during the execution of a program (logical ticks). The time, i.e. the distance between two adjacent logical ticks, is not measurable. Most often the "implementor" is interested only in the order of logical ticks (events).

Absolute time — a reference is established in relation to a global event for a given system (e.g. the origin of absolute time may be the instant of switching the system on).

Relative time — a reference is established in relation to a local event in the given system. Usually we may have more than one relative time simultaneously in a system.

In the case of distributed programs (i.e. those executed on multiprocessors or across computer network), one has to distinguish between **global time** and **local time,** depending on whether the reference is valid for the whole system or only for a part of it. Global and local attributes may be used for both logical and physical time. As a rule, absolute time is global, and relative time is normally considered to be local.

Only one absolute time may be defined in a system, within one and the same system several relative times are usually in use. Whenever necessary, a correspondence between physical and logical time may be established. Then, since the other types of time — absolute, relative, global, and local — are derivative terms, we will have a set of comparable times. Metric properties of the physical time are used as the basis for comparison.

A common feature of the above-defined terms is that they are intrinsic to a program (or to a system consisting of a program and a computer). The same terms may be used in a real system (in an implemented program) and in its formal models. The set of terms enables to introduce total ordering (natural for sequential programs and for interleaving models of concurrency), as well as partial ordering of events (e.g. for maximal parallelism model of concurrency).

Note that the above-presented set of terms is strictly oriented to one execution of a program, and during this execution it accepts no explicit influence from outside the computer system.

Time as seen by a "verifier"

This viewpoint is typical for the design stage of the life-cycle. The "verifier" is interested in all possible executions of a program, and usually concentrates on certain properties of a program — e.g. safety properties, liveness properties, fairness. Time is often reduced to the ordering of events (i.e. logical time), without really using metric properties of time. This is true (and works well) even for many reactive systems (see, for example, [13]).

"Verifiers" work, as a rule, with computational models of programs. Consequently, the time they use need not (but may) be related to a time used by the "implementor".

The temporal logic community, for example, is using linear time and branching time. **Linear time** allows a "verifier" to concentrate on the universal properties of a program, i.e. on the properties that hold for all execution sequencies of that program. **Branching time** considers the collection of all execution trees generated by a program, and usually helps to concentrate on its existential properties — i.e. there exists at least one execution sequence with that particular property. Emerson and Halpern, [14], compare branching versus linear-time temporal logics and point out advantages of both. They give a slight preference to the branching-time temporal logic.

Explicit metric properties have been recently introduced into temporal logic. Two basic approaches can be distinguished, [5].

1. **Bounded-operator approach** introduces for each temporal operator one or more time-bound versions (usually upper-bound and lower-bound temporal operators). This leads to two separate proof principles: upper-bound properties are close to liveness properties, whereas lower-bound properties closely resemble safety properties. Correspondingly methods usable for proof of safety and liveness properties are used.

2. **Explicit-clock approach** uses the traditional temporal logic proof system, time is introduced as a separate variable and must usually have metric properties (see, for example, [15]).

Some researchers argue that continuous time is needed (see, for example, [16]), others claim that they manage with discrete time, [5]. The arguments for and against continuous time are often linked with the computational model used ("maximal parallelism" or "interleaving" model).

Note that the "implementors" use discrete time, whereas "verifiers" use discrete as well as continuous time, "specifiers" have many specific problems of their own. Still, from pragmatic considerations, the "specifiers" tend to use discrete time.

Time as seen by a "specifier"

The specification is the most complex stage of a life-cycle having major impact on the quality of a future system. Experience shows that up to 60% of errors discovered during implementation, testing and maintenance take their origin from the specification stage. The complexity of this stage has two basic reasons:

— a "specifier" has to move from actually existing objects and phenomena (to be influenced by the computer system) to their more or less formal models since during the design, verification and implementation of the system, he/she works with these models only; transition from real objects to their models is complicated because of the famous "semantic gap" between informal and formal descriptions;

— a "specifier" has to fix the time constraints on the behaviour of the computer system and time requirements that guarantee the necessary match between behavioural characteristics of the computer system and those processes in the real world that are to be influenced by the computer; a minor inconsistency here may cause a major error in an applied system.

In the context of this paper we are interested in time presentation and in matching time counting in co-operating systems.

Presentation of time constraints and requirements. When stating time constraints and requirements for the future system, we face two basic problems.

1. The necessity to handle different time scales — in a normal control system, in order to characterize some action, we need milliseconds, for others we need minutes, hours or even days. When moving gradually to implementation, the different time scales are to be presented in the units of system time (usually defined by a timer in the computer system). As demonstrated by Corsetti and colleagues, [10], this is not a straightforward task.

2. The necessity to handle different ways of presenting time constraints and requirements. For example, to maintain control of a car when braking on a slippery road, the rotation of wheels must be well synchronized. We can state that brakes on different wheels must start to work simultaneously and that the wheels are to be blocked simultaneously. This statement sounds theoretically nice but is of no use in practice. Simultaneity here can only mean that we fix a tolerated (admissible) difference in wheels' angular speed. For a computer system which controls brakes, this must be translated into actual time constraint (e. g. milliseconds) required to synchronize the angular speed of wheels. Another example of a similar problem is given in [11].

According to Hoogeboom and Halang, [1], the philosophers call this view on time "reductionism" — i. e. time is reducible to the history system of events, and vice versa, — as opposed to "Platonism" where time is a primary system and is non-reducible to the history of events.

Matching different time-counting systems. Time is a widely used tool for controlling the match of behaviour and dynamic properties of the co-operating systems. More properly, time provides a common measure against which we can compare the behaviour and dynamics of co-operating systems.

Speaking about a computer system, the common measure is, of course, global physical time. In practice, however, a computer system co-operates with many technical systems (that form the computer's surroundings). Some of these systems may essentially function in continuous time, some others are characterized by a sequence of events etc. The notions and terms used for comparing a variety of times with the global physical time in a computer system, or with any globally accessible metric time, are grouped into two, [3].

1. Synchronization terms

Tolerance interval (for (part of) a computer system's surroundings) — if two or more events characterizing, or influencing, the surroundings occur within this interval, they are considered to occur simultaneously from the point of view of (part of) the surroundings.

The tolerance interval will determine the necessary granularity of time required for describing (this part of) the surroundings.

Equivalence interval (for the embedded (into the surroundings) computer system) — if two or more events occur inside a computing system within this interval, then they are considered to have occurred simultaneously (from the point of view of the embedded computer system).

A trivial consequence is that the equivalence interval may not be larger than the tolerance interval for the corresponding events.

Simultaneity interval (for the embedded computer system) is the time that elapses from the occurrence of the first of a generated group of events until the occurrence of the last event of the same group. The generating event and the group of events generated by it should be in the same equivalence interval.

2. Acceptability terms

This group of time notions is concerned with timeliness of events and the resulting data.

Validity time is a time interval during which an event in the system can be considered legal or the data resulting from this event can legally be used.

Validity time notion is essentially based on the ideology of time-stamping of data (see, for example, [7]). The validity time is usually determined by the producer of the data who also time-stamps it.

Response time is a time interval during which specific input values must be accepted and corresponding output values must be generated (it could also be interpreted as a specific form of validity time).

Overstressing the importance of response time in connection with real-time systems has caused a number of misinterpretations discussed by Stankovic [6].

Timeout is used to denote a deadline for response/validity time.

What are "timing properties"?

In this subsection, we discuss some of the software properties usually termed as "timing properties", or "real-time properties" or "time correctness". We have tried to collect and systematize as many different time-bound properties as we could find in the literature. It is suggested that the timing properties be classified into three groups.

1. **Performance-bound properties** comprising integral time characteristics for the system as a whole, or for a part of it. Examples of this group are response time, timeout, execution time for a sequence or loop of programs. This is the most thoroughly studied group of timing properties.

The presence or absence of certain performance characteristics can be proved/evaluated by using:

— temporal logic, more specifically either bounded-operator or explicit-clock approach (see, for example, [5] and [15]); in both cases upper and lower bounds of the properties can be verified; while explicit-clock approach always reduces performance to safety properties, the bounded-operator approach results in liveness properties (the upper-bound case) or safety properties (the lower-bound case);

— algebraic methods [16, 17] and timed Petri nets (see, for example, [18]), provide formulae for evaluating response time for a collection of programs whose co-operation is guided by causality (or before/after) relationship;

— the *Q*-model formalism that considers a more sophisticated case of loosely co-ordinated group of cyclically executed programs where some of the unknown (or incompletely known) causality relations have been

substituted by timing constraints (see, for example, [3]); formulae are provided for evaluating upper and lower bounds for the message passing through the given collection of programs;

— simulational approach, which probably is the first, and still the most widely used method for performance studies.

2. **Timewise correctness of events and data** is concerned with execution time of programs and delays between events. The latter covers a wide variety of questions connected to:

— measuring/stating validity time of data;

— checking whether (requiring that) data is consumed within its validity time;

— checking (requiring that) the length of the specified simultaneity (or equivalence) interval;

— requiring (scheduling) repeated activation of programs (periodic or quasiperiodic execution);

— etc.

These questions have been somewhat less thoroughly studied than the performance-bound properties but are still well covered. The corresponding examples can be found in temporal logic (e. g. [5]), in compositional verification [19], in algebraic methods [16], in the *Q*-model [3].

3. **Time correctness of interprocess communication** is a typical problem of hard real-time system, although it can also be met in other reactive systems.

Intuitively it is clear that time correctness of interactions is a good indication of whether or not time parameters and constraints imposed upon the interacting parties (usually independently of each other) are consistent and non-contradicting. Amazingly little attention has been paid to time correctness of interprocess interaction (even for the case of a non-cyclical execution of processes). Most of the researchers concentrate on the transport delays in the communication media. This is certainly a very important feature, yet it can be useful and applicable in the implementation stage only.

A few researchers dwell on interaction timing in the pre-implementation stages. For example, Hooman, [19] whose assertion language enables to specify the time instant when the interaction should start.

The case of repeatedly executed processes, where time-selectivity of the transmitted data really becomes important, has drawn even less attention. Time-selective interactions were first introduced by Quirk and Gilbert, [20]. The idea has been further developed by Mõtus, [21]. As demonstrated in [3], the concept of time-selective interaction, if properly handled, is an excellent tool for checking the consistency and non-contradiction of time parameters and constraints imposed upon the interacting parties.

Time-selective interaction means, in a nutshell, that the data required by the consumer (addressee) must be of a certain, prefixed, age — i. e. not too old and not too new. This age requirement remains unchanged throughout all the execution cycles of the interacting parties, even if they are activated with different periods. The idea is based on the understanding that in a real-time system the most recent data are not necessarily the most suitable for consumption — for example, the requirements of the data integrity, in many cases, overrun the traditional desire for the most recent information.

From the traditional computing science viewpoint the consequences of adopting time-selective interaction are rather heretical — it becomes natural that the consumer process may never consume some of the mess-

ages sent to it, whereas some other messages may be consumed several times. As a by-product, this concept explains and enables to forecast timing errors that occur at seemingly random instants (see, [3]), caused by a sawtooth graph of a non-transport delay occurring inevitably during the interaction of truly asynchronously executed processes.

It seems highly probable that little attention to the interprocess communication and time-selective interaction in particular is mostly caused by incomplete understanding of the essence of time in real-time systems. Interaction is a complex activity where different time parameters, constraints, requirements, and possibly different philosophical concepts of time, are involved.

Concerning philosophy of time

Time-bound terms surveyed in the previous section reflect a healthy and pragmatic attitude of programmers — a notion of time was introduced only when it was inevitable and useful. Not too much attention has been paid to the resulting metasystem that could describe the essence of time in programs. This has led to some discrepancies in understanding the role of time in the software process, which has considerably hindered the development of theoretically well-founded software engineering environments for real-time systems. The rest of this paper is devoted to propagating an interpretation of time in software (as a combination of several philosophical concepts of time) that might clarify some misconceptions.

The philosophical essence of time is still an object of active research. An extensive survey of contemporary time research can be found in Benthem [9], Denbigh [22], Shoham [8]. In the context of this paper, it seems useful to emphasize that, as a rule, only one basic concept of time is being used in each major research area (see, for example, Denbigh [22]). In the following, three time concepts have been described — each of them is also being used in programming as demonstrated in the following sections of this paper.

Time in theoretical physics is just like any other space co-ordinate; it has no intrinsic direction, and is fully reversible.

Time in thermodynamics and in other evolutionary sciences (e. g. biology) has a fixed intrinsic direction and is generally not reversible. Under strict assumptions (e. g. complete control of the causality) it is sometimes possible to reverse time for a brief period.

Time in our consciousness (e. g. psychology) has a fixed direction, its origin is always at the present moment; therefore strict distinction is made between Past, Present and Future.

In theoretical physics and thermodynamics observers stay outside of processes under study, this is why all events happening in time are equally real to them. In our consciousness the only real events take place in the Present; events that have taken place in the Past and events that will take place in the Future differ substantially from the present ones and from each other.

The computing science world can be characterized, with a slight exaggeration, by two opposing schools of thought. One school advocates the use of logical time — if no events occur in the system, time does not proceed. The other school advocates the use of physical time — metric properties of time are essential and time forms a basis for measuring system properties. These schools are in concordance with the above-surveyed time-bound terms. However, clear connections to the above-described philosophical concepts of time have not explicitly been accepted.

Examples of using different time concepts in software

The following examples introduce another dimension of a time study in software, and demonstrate how and why time concepts traditionally used in different research areas come to be used in programming.

Time concept of theoretical physics

Programs in informationally closed computer system (i.e. transformational computer systems) operate very much like models of Nature in theoretical physics — in both cases it is assumed that all the causal relationships are known and can be controlled by the program/experimentor. This assumption is true, provided that the structure of programs and the properties of the computing system (in other words — the equivalents to the Laws of Nature for a program) do not change during the execution of the program. Since we know and control all the causal relations in transformational programs, the time can, almost at will, be reversed — exactly as we do in theoretical physics. As an example, just think of the possibility to "undo" some already completed action and "redo" it later (in some CAD system, or in a wordprocessor).

The most sophisticated and complete usage of this time concept is described by Jefferson, [23]. The "virtual time" notion, which is equivalent to the above-described global time, has been introduced. For reversing the direction of time, Jefferson uses a "time-warp" mechanism. In fact, virtual time is an additional co-ordinate of the program's state space. In this approach each process of the program executes in its own (local) time, which has a correspondence with the (global) virtual time, and sends messages to other processes whenever necessary. Each message is time-stamped with its virtual sending time (a local time mapped into the virtual time). A process accepts messages in the order of their arrival and checks that their virtual sending times are increasing. Whenever a message is received that is preceding in its sending time the already processed messages, the time-warp mechanism is applied and the process's local time is set back. All the more recent messages are annihilated by returning corresponding "antimessages" to their senders. In this way the time-warp mechanism guarantees time integrity of the concurrently executed processes.

Time concept of thermodynamics

In an open computer system (reactive system) the assumption that all the causal relationships are known is not justified. The surroundings which embed the computer system are, as a rule, too complicated to be completely formalized. Even if a complete and formal model of the surroundings exists, the computing power of the embedded system is not sufficient to consider all the required details. The other assumption — the ability to control all the causal relationships — is even less founded. Imagine, for example, a computer controlling the loading of a chemical reactor. It is practically impossible to "undo" the loading of a component after it has been loaded and the resulting reaction has started.

Consequently, time in open computer systems is normally not reversible and has a definite intrinsic direction. So, time in real-time systems is, in general, employed exactly as it is in thermodynamics or biology [3].

Another application of the thermodynamical time concept can be found in temporal logic (see, for example, [5], [14], and [15]). In temporal logic assertions are based on infinite sequences of events ordered according to increasing time. It is not possible to "redo" the history according to the verifier's will. In order to change the history, one has to go several

steps backwards and modify the transition system (i. e. a complex comprising of a computer system and its surroundings, or the corresponding computational models) that generates the history. Similar problems can be met by using algebraic methods [16]. Still another example of thermodynamical time is discussed by Ledru in [24].

Time concept used by our consciousness

Processes in real-time applications are typically executed repeatedly — periodically, quasiperiodically, or aperiodically — and are terminating. This paradigm is preferable to the competing paradigm claiming that processes in real-time systems are non-terminating. Although it saves us from the necessity to work with non-terminating processes, it also introduces the problem of reasoning about repeatedly activated, interacting processes — the number of activations may be theoretically infinite (in fact, countable).

Temporal logic, in spite of all its virtues, is rather cumbersome in handling time problems for repeatedly activated processes (see, for example, [25]). This is a direct consequence of the use of thermodynamical time which implicitly forms the time basis of temporal logic. Repeatedly activated processes are also the cause that necessitates the introduction of time-selective interprocess communication [20, 21].

For describing timing properties of interprocess communication, especially in the case of repeatedly activated processes, it seems natural to use time in the same way as it is used in our consciousness. This means that time is relative, the instant of requesting data is taken to be the origin of the relative time. When the process requesting data is activated another time, the origin of the relative time is defined anew. The required age of the data is described with respect to the origin of the relative time used in connection with this particular interaction.

Such interpretation of relative time is close to the time concept as used in our consciousness. By applying this concept of time for each pair of interacting processes, a set of internal relative times is formed. This time concept has been used by Quirk and Gilbert [20], Mõtus [21], and Caspi and Halbwachs [16].

Another example where such relative time is effectively applicable is given by Corsetti and colleagues in [10]. They correctly observe that the models associated with temporal statements are likely to change interpretation when changing from one time unit to another. This problem could be simplified by handling the time unit change similarly to the interaction of two processes — i. e. by defining a relative time with the origin at the instant when the time-unit change is to take place.

Mixed use of multiple time concepts

In the previous section some examples were given emphasizing the usage of one or another, but always a single philosophical concept of time in programming. We claim that in order to enable full-scale analysis of timing properties — i. e. performance-bound properties, timewise correctness of events and data, and time correctness of interprocess communication — it is necessary to use several time concepts simultaneously.

The analysis of timing properties is possible only on a suitable mathematical model of computations, simulation cannot give the required level of confidence. It is essential that the model captures the necessary multitude of time notions, otherwise the analysis can cover only part of timing properties.

For example, a timed Petri net is based on causality relationships and locally defined delays on forwarding the causing factors. Since causality is usually non-reversible, one can deduce that only thermodynamical time concept is used in timed Petri nets. As a consequence, performance analysis is perfect, timewise correctness of events and data is analysable to a certain extent, and time-selective interprocess communication has not been handled in timed Petri nets. Since temporal logic is implicitly also based on thermodynamical concept of time, the same arguments can be applied for characterizing its ability of analysing time properties.

Let us consider the *Q*-model as another example [20, 21]. It is based on thermodynamical time concept, but it also relies substantially on the use of several relative times (similar to time as used in our consciousness) — one for each process and one for each pair of interacting processes. For each process the thermodynamical time advances in grains defined by the execution time interval and/or by repeated activation interval. Thermodynamical time is reversible inside one grain, if necessary. This multitude of times forms a solid basis for analysing all the above-listed timing properties of a system that are described in the *Q*-model. A software engineering environment CONRAD has been developed in order to automate the system description and analysis (see, for example, [3]).

The *Q*-model is excellent for analysing timing properties. As a price for this, it is rather awkward for describing algorithms — this is just another illustration that no panacea exists.

Conclusions

Pragmatic attitude of computer scientists and control engineers towards developing real-time systems and their software is, in principle, good and healthy. However, from time to time it is advisable to take a more abstract look at one's area of activities. This might slightly clarify some fuzzy spots in the overall picture and change one's approach to problems, hopefully fostering thus the quality and efficiency of the practical outcome.

Time-bound terms and related theories have been widely discussed recently, a huge amount of different opinions is circling around. As a result, the author of this paper got confused and attempted to take a more abstract look. The above presented speculations about time resulted together with a conclusion that computing science is somewhat exceptional — other sciences, as a rule, make use of a single philosophical time concept. Computing science needs at least three different time concepts simultaneously. It is interesting to note that simultaneous use of multiple time concepts seems to be necessary only to guarantee and demonstrate the wanted co-operation of a group of dynamic systems that form a real-time system. When solving other problems, one can manage with a simplified time system.

REFERENCES

1. *Hoogeboom, B., and Halang, W. A.* The concept of time in software engineering for real time systems. — 3rd International Conference on Software Engineering for Real Time Systems. IEE Conference Publication, 1991, 344, 156—163.
2. *Mõtus, L.* Time concepts in software. — Concise Encyclopedia of Software Engineering, Pergamon Press, Oxford et al., 1992.
3. *Mõtus, L., and Rodd, M. G.* Specification of Embedded Real-time Software, Prentice-Hall, London et al., 1992.
4. *Kurki-Suonio, R.* Some thoughts for the Real-time Session in the Como Workshop (6th Int. Workshop on software specification and design). Private communication, 1991.
5. *Henzinger, T. A., Manna, Z., Pnueli, A.* Temporal proof methodologies for real-time systems. School on Formal techniques in real-time and fault-tolerant systems. Univ. of Nijmegen, The Netherlands, 1992.
6. *Stankovic, J. A.* IEEE Computer, 1988, **21**, 10, 10—19.
7. *Kopetz, H., Kim, K. H.* Temporal uncertainties in interactions among real-time objects. Institut für Technische Informatik, Technische Universität Wien, Austria, Res. rep. no. 10/90, 1990.
8. *Shoham, Y.* Reasoning about Change. Time and Causation from the standpoint of Artificial Intelligence. MIT Press, Cambridge MA, London, 1988.
9. *van Benthem, J.* The logic of time. A model-theoretic investigation into the varieties of temporal ontology and temporal discourse. Kluwer Academic Publishers, 1991.
10. *Corsetti, E., Crivelli, E., Mandrioli, D., Montanari, A., Morzenti, A. C., San Pietro, P., Ratto, E.* Proc. 6th Int. Workshop on software specification and design, Como, 1991, 92—101.
11. *Halbwachs, N.* Synchronous programming of real-time systems. The language LUSTRE. School on Formal techniques in real-time and fault-tolerant systems, Univ. of Nijmegen, The Netherlands, 1992.
12. *Kopetz, H.* Proc. 5th IFAC Workshop on Distributed Computer Control Systems, Pergamon Press, New York, 1984, 11—15.
13. *Kurki-Suonio, R., Systä, R., Vain, J.* Proc. 6th Int. Workshop on software specification and design, Como, 1991, 84—91.
14. *Emerson, E. A. and Halpern, J. Y.* J. of the ACM, 1986, **33**, 1, 151—178.
15. *Ostroff, J.* Temporal Logic for Real-time Systems. Research Studies Press/ John Wiley and Sons Inc., 1989.
16. *Caspi, P. and Halbwachs, N.* Acta Informatica, 1986, **22**, 595—627.
17. *Caspi, P. and Halbwachs, N.* Proc. Int. Conference on Parallel Processing, 1982, 150—159.
18. *Sifakis, J.* Acta Cybernetica, 1979, **4**, 2, 185—202.
19. *Hooman, J.* Compositional Verification of Distributed Real-time Systems. School on Formal techniques in real-time and fault-tolerant systems, Univ. of Nijmegen, The Netherlands, 1992.
20. *Quirk, W. J. and Gilbert, R.* The Formal Specification of the Requirements of Complex Real-time Systems. AERE, Harwell, UK, rep. no 8602, 1977.
21. *Mõtus, L. and Kaaramees, K.* Proc. 4th IFAC Workshop on distributed computer control systems. Pergamon Press, Oxford et al., 1983, 93—101.
22. *Denbigh, K. G.* Three concepts of time. Springer, Berlin et al., 1981.
23. *Jefferson, D.* Proc. Int. Conf. on Parallel Processing, Silver Spring, Md., 1983, 384—394.
24. *Ledru, Y.* Proc. 6th Int. Workshop on software specification and design, Como, 1991, 130—139.
25. *Allen, J. F.* Artificial Intelligence, 1984, **23**, 2, 123—154.

Received
August 3, 1992

Leo MÕTUS

AJA OLEMUSEST REAALAJA TARKVARAS

On püütud süstematiseerida aja mõistega seotud termineid ja teooriaid, mis on kasutusel reaalaja süsteemide ja nende tarkvara kirjeldamisel, analüüsil ja verifitseerimisel.

Programmeerimises kasutatava aja olemuse avamiseks on võrreldud seal käibivaid ajakontseptsioone traditsioonilistes teadustes (näiteks füüsikas, termodünaamikas, psühholoogias) kasutatavate ajakontseptsioonidega.

On jõutud järeldusele, et arvutiteadus vajab kolme erineva filosoofilise ajakontseptsiooni samaaegset kasutamist reaalaja süsteemide tarkvara ajaliste omaduste täielikuks kirjeldamiseks ja analüüsiks. Traditsioonilised teadused on aja mõttes paremini fokusseeritud ja vajavad igaüks vaid ühte ajakontseptsiooni.

Лео МЫТУС

О СУЩНОСТИ ВРЕМЕНИ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ РЕАЛЬНОГО ВРЕМЕНИ

Сделана попытка систематизировать связанные с понятием времени термины и теории, которые используются при описании и анализе систем реального времени и их программного обеспечения.