An Overview of The Mensurae Language: Specifying Business Processes

TSE Maibaum

Department of Computer Science
King's College London
Strand
London WC2R 2LS
UK
email: tom@maibaum.org

Abstract

We describe a language which can be used to model business processes (both technical and nontechnical). It has a formal semantics, so as to enable formal analysis and prediction on the basis of the models, and is based on principles of measurement theory, so as to enable the application of engineering and scientific principles to metricate the business processes. The language models *products* and *processes* as the constituents of business processes and uses a control construct, *gates*, to enable product and process composition. The semantics of the language is based on timed transition systems and processes can be specified using a combined temporal and modal logic with real-time operators. The language thus obtained is also relevant to modelling threads of execution in object oriented systems and may lead to appropriate constructs for defining timed, concurrent, and synchronised computations over object oriented structures.

1 Introduction

The purpose of this paper is to provide a brief overview of the Mensurae language. (See [17] for a more detailed description.) The language supports the *Mensurae method* for formal description of processes, understood here as referring to the technical and administrative activities of organisations. As such, it provides abstractions for the two main constructs of the method: *products* and *processes*.

The basics of the method are described in [15], as is the relationship of the method to Measurement Theory. (See also [21, 5] on measurement.) What we are modelling is some real process and its associated products, technical or administrative (or both). Either this process is in place and we are modelling what we observe or we are intending, via the model, to prescribe what we eventually intend to observe in the organisation. The observed or intended phenomenon is called the *empirical referent* (and is to our endeavours what specific physical phenomena are to physicists attempting to understand the world by building scientific theories/models).

The language we use for modelling processes has three basic constructs via which the empirical referent must be modelled: products, processes and, an artefact of the method, gates. The first of these, *product*, is used to model the entities manipulated by processes. A product is an 'instantaneous' entity, in that it represents the measurable attributes of the entity at a specific moment in time. On the other hand, a process is an entity that relates to behaviour and so represents a phenomenon taking place over time. Gates are, to some extent, an artefact of the method. Processes are modelled via 'single input, single output' transformers and so require products to be assembled (input from several preceding processes) and disassembled (to send parts of the output to separate subsequent processes).

Business process modelling is, of course, not a new subject! Many consultants make big money out of it! Our focus here is on the problem of modelling itself, with a particular focus on technical processes and requiring levels of detail in process definition which enable the methods and principles of measurement theory to be applied for the purposes of analysis and prediction. A standard reference to approaches to business process modelling is [12].

The idea of characterising formally the process of software development is not new. The early work of [20,

16], etc is notable in this regard and relevant recent material on process modelling in software engineering can be found in [8]. The spirit of what we are attempting is very much in the style of [20], i.e., characterising processes as programs. We see the major difference as being the larger domain of processes being modelled, the much more powerful and expressive language being used and the focus on measurement and measurability. Also, software engineering has moved on since the time of this work and concepts such as patterns and software architecture enable a more sophisticated approach. A related, but quite different formalisation appears in [10].

We also see this work as having potentially useful impact on the problem of defining threads of computation in object oriented systems. This will be elaborated further in section 3 on processes and in the conclusion.

The rest of this chapter is organised as follows. Section 2 describes products; section 3 focuses on atomic (unstructured) processes; section 4 explains the role of gates; section 5 introduces the concepts needed to use the constructs already introduced to model organisational processes and section 6 contains some concluding remarks and describes open questions.

2 Products

We wish to model product entities in terms of their measurable characteristics. So, an iron rod may be characterised in terms of its length, circumference, weight and so on. We say that it has length 2.2 m, circumference 3 cm, and weight 10.75kg. A room may be characterised in terms of its length, width, height, and floor area, as well as the numbers of windows and doors, their sizes and their position relative to some designated point in the room. The width of the room is measured directly (is directly observable) by applying a ruler in a prescribed manner to determine (with some degree of expected accuracy) the required number designating length on the appropriate scale. We may wish not to measure directly the area of the room, but to calculate it instead from the usual law relating length and width to area. Sometimes, unlike area, there is no way of directly measuring the value associated with some characteristic, as for example in the case of the utility of some decision made by management. In the theory of measurement, such utility values are the result of some well-defined function applied to some already available values for relevant characteristics (some of which may be directly measurable or calculated eventually from such directly measured values). These *derived* characteristics are *theoretical* constructs that we use in the modelling of product entities.

This distinction between what is (directly) observable and what is described via theoretical relationships is a fundamentally important one in science. See [24] for an explanation of these ideas in relation to science and [11] for an exposition of software engineering from this perspective. These chosen characteristics, the rules relating derived ones to the directly measurable ones and any other relevant relationships and laws incorporated into the model give us the vocabulary and properties characterising this (class of) product entity. If we see this model of a class of product entities as being given by a theory in some appropriate logic, we can view an individual product entity as being an interpretation (often confusingly called a model or *mathematical model*, as we will refer to it) of this theory. In particular, the characteristics associated with the product entity are given values in the appropriate scales (by means of the correspondence rules/measurement procedures) and obeying the rules and laws prescribed for the class, including the derivation of some characteristics from others.

Then, an *atomic product* P is defined as a 5-tuple $\langle code_number_p, product_name_p, time_p, attributes_p, \langle \Sigma_p, A_p \rangle \rangle$ where:

$code_number_p$ is a constant of sort *Code*, used to distinguish the product from other;

- $product_name_p$ is a constant of sort *Name* to allow reference to the product;
- $time_p$ is a constant of sort *Time*, called the *time stamp* of the product P , defining the moment at which the product's attributes have values associated with them;
- $attributes_p$ is a set of constants which forms a subset of the constants in Σ_p , designating the characteristics which we wish to use as the measures characterising our empirical referent;
- $\langle \Sigma_p, A_p \rangle$ is a theory presentation, the *proto-product* (which includes in the language all of the constants above).

The Mensurae method requires that products be marshalled to form single inputs to processes (so process initiation time is well defined) and that outputs be decoupled ('unmarshalled') so as to enable them to be directed to the appropriate subsequent processes. The use of the word *composed* will be restricted to this kind of method induced notion of structure. (We will reserve the word *structured* to refer to the normal notion of structure, as for example in OO languages in terms of aggregation, specialisation, etc. For instance, a table is an aggregation of legs, top, screws, etc.)

A *product* is either an atomic product or a composed product. A *composed product* P is either:

- a pair $P = \langle \text{code_number}_P, \otimes(P_{c1}, \dots, P_{cn}) \rangle$ where $\otimes(P_{c1}, \dots, P_{cn})$ is an injection of the components P_{ci} into the Cartesian product ('marshalling'); or
- a pair $\langle \text{code_number}_P, \iota(\mathcal{P}) \rangle$ where $\mathcal{P}$ is a finite non-empty set of products and $\iota(\mathcal{P}) \in \mathcal{P}$ (i.e., choice, a kind of 'unmarshalling', the other kind being projection from a tuple in $\otimes(P_{c1}, \dots, P_{cn})$).

(Note that composed products do not have names or timestamps of their own. Essentially, the timestamp of the latest atomic constituent can be said to be the timestamp of tupled product, while the timestamp of the chosen product in a choice construct is its timestamp. There is a rationale for this in the method.) When are two products equal? Clearly they have to have the same proto-product, i.e., specification. Further, we require that the codes, names, and time stamps and attributes be the same. Thus $P = P'$, where $P = \langle c, n, t, att, \langle \Sigma, A \rangle \rangle$ and $P' = \langle c', n', t', att', \langle \Sigma', A' \rangle \rangle$, iff $c = c', n = n', t = t', att = att', \Sigma = \Sigma',$ and $A = A'$.

We have been looking at what OO calls classes. That is we have been building models of classes of empirical referents. We now wish to model individual referents, with specific values (of the appropriate kinds) associated with the attributes, code, name, and timestamp. Logic supplies us with the natural definition of *what* such 'individualisation' means and measurement theory tells us *how* to use measurement procedures to determine the appropriate 'individualisation'. We use the word *instance* for the individual referents in the real world corresponding to our theoretical (class) constructs). It will be instances of products that will be involved in executions of processes.

Simply put, an instance of a product $P = \langle c, n, t, att, T \rangle$ is an interpretation/model (in the sense used in logic) of T . The reader should note that we are not in fact considering all models of the proto-product/presentation associated with a product. We are only interested in some 'standard' interpretations where: the codes are always interpreted in some organisation's standard code domain; ditto names and name domain; the time stamp is interpreted in the standard time domain (as given by correspondence rules and measurement theory); and certain sorts are interpreted as measurement scales according to the organisation's (and scientific) standards.

3 Processes

A process models an empirical referent that transforms an input product into an output product. Examples include: the process of assembling a telephone from the component products as part of an assembly line in a factory; developing, testing, and delivering a software application from a given set of requirements; ascertaining and stating the requirements for a software application from the appropriate information sources; performing an experiment in a scientific laboratory; servicing defect reports after products have been delivered to customers; (a thread of computation over a given object system – see the discussion below of the virtual machine); etc.

As for products, processes may be atomic (i.e., no internal structure in terms of 'subprocesses') or structured. We call the latter *framework* processes or process *explosions* and treat these in section 5 after introducing *gates*, which will play the role of control elements (as well as that of 'marshalling' devices) in our process language.

Because processes model input/output transformations, we are immediately put in mind of conventional programs, as conceived in the early days of programming language semantics. Should we think of processes as simply implementing such an input/output transformation? Programs are defined over some *virtual machine* representing the execution environment for the language. This execution environment provides direct interpretations for the basic commands of the language and is able to sequence instruction execution according to the control flow prescribed by the program.

Our processes differ from conventional programs in a number of ways. One such difference is the fact that there may be more than one virtual machine at work in implementing the process and these virtual machines are co-ordinated and synchronised in their behaviour in some well-specified way. This reminds us of the world of *object oriented* (OO) programming and design, where systems of objects play the role of our virtual machines. An application is realised in terms of a number of such machines co-ordinated by one such machine that controls the order in which objects' methods are invoked with appropriate data so as to achieve the application's objectives.

In order to model processes, we will pursue this analogy of the virtual machine executing our process prescriptions. Thus, our virtual machine will be an object (system) with methods, the basic actions of which our machine is capable (analogous to assignments in a conventional language and machine). Its states will be defined in terms of attributes or state variables (analogous to program variables in conventional languages and some of which will be used as attributes of products). This object may be built from a number of other objects via the appropriate structuring mechanisms (e.g., see [6,7]) and will in realistic applications be very large. For example, to model software development, we must have modelled the basic actions of which individuals or groups are capable: design a test, execute a test, build a specification of a small program, implement the specification in a given programming language, integrate a program with an application, retrieve a program from a depository, fix a bug in a program, etc. In practice, we have noted that the most difficult aspect of modelling is deciding on the granularity of the basic actions of our virtual machine.

Now, what our virtual machine models is the potential activities that the empirical referent is capable of performing. Using our analogy with the virtual machine executing a program, the former is capable of executing many programs; in fact, it can execute any syntactically correct program in the programming language. Analogously, our virtual machine is capable of exhibiting many behaviours, all consistent with the 'rules' of the machine. What we want is to choose some behaviour(s) that is (are) consistent with our intention of transforming some input product into a corresponding well defined output product. This is where processes come into the picture.

Any process will be a specification of what input/output transformation is required so as to model the empirical referent in which we are interested. Some such specifications will be like the programs in our analogy, i.e., an explicit combination of the actions of our virtual machine via some appropriate control structures. Others will be more abstract and describe implicitly the effect required via more abstract actions. Eventually, we will have to *refine* such an abstract specification into a 'program' using only basic actions so as to be able to execute them on our virtual machine.

As with conventional programs and machines, there are a number of other factors to consider. One such question is the nature of the control structures to be allowed. We will introduce standard combinators for sequencing, and parallel combination of processes and we will need some form(s) of loop structure to deal with iterative processes. See section 5 for the definition of these structures. Our first task is to formalise the concept of virtual machine. We do this by borrowing from [6,7] the concept of *object specification*.

The formal counterpart to the real-time virtual machine outlined above consists of a logical framework based on timed transition systems, called RETOOL (REal Time Object Oriented Logic). RETOOL [2] is a modal action logic based on an extension of Segerberg's δ operator [22] with time bounds. The semantics of RETOOL is based on the notion of a computation over a timed object frame, which is the formal counterpart to a potential behaviour of some object described, in our methodology, by a timed transition system.

By a specification, we mean a theory (presentation) in the proposed action logic, that is a pair consisting of a signature and a collection of sentences describing the behaviour of the system. A signature consists of a declaration of the non-logical symbols that are used to model the specific features of the object/system. In the case of RETOOL, signatures consist of collections of attributes (state variables) and actions. For simplicity, we will present in this paper a propositional version of RETOOL. In the more general setting, attributes can be typed with data sorts and actions can take arguments.

We assume given an ordered field $(\text{TIME}, \leq)$ with a maximum ∞ . An *object signature* is a pair $\langle A, \Gamma \rangle$ where A and Γ are sets of attribute and action symbols, respectively. A semantic interpretation structure for an object signature is given based on the notion of timed transition system by Henzinger *et al* (1992). A *timed object frame* $\mathcal{F}$

for a signature θ is a sextuple $(\mathcal{W}, \rightarrow, l, u, I, w_0)$ where: $\mathcal{W}$ is the set of states (or possible worlds); $\rightarrow: \mathcal{W} \times \Gamma \rightarrow \mathcal{W}$ is the transition relation; l maps each $g \in \Gamma$ to a delay $l(g) \in \text{TIME} \setminus \{\infty\}$; u maps each $g \in \Gamma$ to a delay $u(g) \in \text{TIME}$ such that $u(g) \geq l(g)$; $I: A \rightarrow 2^{\mathcal{W}}$ interprets the attributes; and $w_0 \in \mathcal{W}$ is the initial world/state. We say that $g \in \Gamma$ is *enabled* in state w iff $\rightarrow$ is defined on the pair $\langle w, g \rangle$, and define, for every $g \in \Gamma$, the set $\text{en}(g)$ of all states in which g is enabled.

Attribute symbols are interpreted as defining subsets of worlds: the worlds in which the attributes are true (recall that we are working in a propositional language). Action symbols are interpreted as transition relations over states. As in [14], two time bounds are defined for every action: a lower and an upper time bound. The intuitive semantics of these bounds, formalised below, is that the lower bound defines the minimum delay that must be observed for the action to take place (be committed), whereas the upper bound defines the maximum delay before the action's effects become observable.

Given a signature θ , we define the set $\text{SP}(\theta)$ of state propositions over θ as follows:

$$p ::= a \mid \neg p \mid p \rightarrow p'$$

where $a \in A$. The denotation of a state proposition in a timed object frame $\mathcal{F}$ is the set of worlds defined as follows: $\llbracket a \rrbracket^{\mathcal{F}} = I(a)$; $\llbracket \neg p \rrbracket^{\mathcal{F}} = \mathcal{W} \setminus \llbracket p \rrbracket^{\mathcal{F}}$; $\llbracket p \rightarrow p' \rrbracket^{\mathcal{F}} = \mathcal{W} \setminus \llbracket p \rrbracket^{\mathcal{F}} \cup \llbracket p' \rrbracket^{\mathcal{F}}$.

Given an object signature θ , the set of *action terms* $\text{AT}(\theta)$ is defined as follows:

$$t ::= g \mid q_l \delta^u p$$

where $g \in \Gamma$, $p, q \in \text{SP}(\theta)$, $u \in \text{TIME}$ and $l \in \text{TIME} \setminus \{\infty\}$, $l \leq u$. Action terms are interpreted over 2^{Γ} as follows:

- $\llbracket g \rrbracket^{\mathcal{F}} = \{g\}$;
- $\llbracket q_l \delta^u p \rrbracket^{\mathcal{F}} = \{g: l \leq l(g) \leq u(g) \leq u, \text{en}(g) = \llbracket \neg p \wedge q \rrbracket^{\mathcal{F}} \text{ and } w \xrightarrow{g} w' \text{ implies } w' \in \llbracket p \rrbracket^{\mathcal{F}}\}$.

By $q_l \delta^u p$ we denote the actions that establish p before u units of time provided that they were enabled for at least l units of time in states satisfying $\neg p \wedge q$. This is a generalisation of the δ operator defined by [22] in two ways: the inclusion of the precondition q and the time bounds. Finally, we have the language $\text{FOR}(\theta)$ of the sentences that can be used to express properties of objects over an object signature θ :

$$\phi ::= a \mid t_1 \supset t_2 \mid \neg \phi \mid \phi \rightarrow \phi' \mid [t]\phi \mid []\phi$$

where $a \in A$ and t, t_1 and $t_2 \in \text{AT}(\theta)$. The *satisfaction* of a formula by a timed object frame $\mathcal{F}$ at a world w is defined as follows: $\mathcal{F}, w \models a$ iff $w \in I(a)$; $\mathcal{F}, w \models (t_1 \supset t_2)$ iff $\llbracket t_1 \rrbracket^{\mathcal{F}} \subseteq \llbracket t_2 \rrbracket^{\mathcal{F}}$; $\mathcal{F}, w \models \neg \phi$ iff $\mathcal{F}, w \not\models \phi$; $\mathcal{F}, w \models \phi \rightarrow \phi'$ iff $\mathcal{F}, w \models \phi$ implies $\mathcal{F}, w \models \phi'$; $\mathcal{F}, w \models [t]\phi$ iff $\mathcal{F}, w' \models \phi$ for every $g \in \llbracket t \rrbracket^{\mathcal{F}}$ and $w \xrightarrow{g} w'$; and $\mathcal{F}, w \models []\phi$ iff $\mathcal{F}, w_0 \models \phi$ (where $[]$ is a 'beginning of time' operator).

A formula ϕ is said to be *valid* in a timed object frame $\mathcal{F}$ iff $\mathcal{F}, w \models \phi$ for every world w . Given a set Φ of formulae, and a formula ϕ , ϕ is a *consequence* of Φ ($\Phi \models \phi$) iff ϕ is valid in all the timed object frames in which all the formulae in Φ are valid. An *object specification* is a theory presentation $\langle \theta, \Phi \rangle$ where θ is an object signature and Φ is a (finite) set of θ -formulae. We say that a frame $\mathcal{F}$ is a *model* of the specification $\langle \theta, \Phi \rangle$ iff ϕ is valid in $\mathcal{F}$ for every ϕ in Φ . (There is a proof theory for the logic presented in [2,17].)

As mentioned before, the behaviour of a system is given by the integration of the behaviour of each of its component objects together with their interactions. The integration will be discussed further below.

In fact, an extension of Timed Transition Systems may be used, Timed Action Transition Diagrams (TATD), which, instead of just assignments as in [14], takes arbitrary actions in the labels of the transitions. The idea is that actions can be understood as a semantic counterpart of methods and messages in OO. For a definition of TATD, see [14,2]. A mapping from TATDs to object specifications in RETOOL is also provided.

The integration of the behaviour of the objects of the involved classes is achieved by means of colimits in the appropriate category of object specifications (CRETOOL). The categorical framework for this integration, which takes into account the possibility of various objects of the same class to be active at the same time, follows the approach

presented in [6,7].

Properties of behaviours of objects specified in RETOOL can be formulated and proved using a Metric Temporal Logic in the style of [3]. We start by defining the possible behaviours of an object in terms of computations over a timed transition diagram. A *timed state sequence* for a timed object frame $\mathcal{F}$ is a pair $\rho = \langle \sigma, T \rangle$ where σ is an infinite sequence of states ($\sigma_i \in \mathcal{W}_{\mathcal{F}}$) and T is an infinite sequence of corresponding times ($T_i \in \text{TIME} \setminus \{\infty\}$), satisfying:

- (monotonicity): for all $i \geq 0$, $T_{i+1} = T_i$ or ($T_{i+1} > T_i$ and $\sigma_{i+1} = \sigma_i$);
- (progress): for every $t \in \text{TIME} \setminus \{\infty\}$, there is $i \geq 0$ such that $T_i \geq t$.

A computation over a timed object frame $\mathcal{F}$ is a timed state sequence $\langle \sigma, T \rangle$ such that σ is a computation of the underlying transition system (i.e., for every $i \geq 0$, if $T_{i+1} \neq T_i$, then σ defines a transition $\sigma_i \xrightarrow{\sigma(i)} \sigma_{i+1}$ with $g = \sigma(i)$ for some $g \in \Gamma$) and

- (lower bound): for every $i \geq 0$ in the domain of σ there is a $j \leq i$ such that $T_i - T_j > \ell(\sigma(i))$ and $\sigma(i)$ is enabled in every state σ_k for $j \leq k \leq i$;
- (upper bound): for every $g \in \Gamma$ and $i \geq 0$, there is $j \geq i$ with $T_j - T_i \leq u(g)$ such that either g is not enabled at σ_j or $g = \sigma(j)$.

A computation is said to be *complete* if it is either infinite or it cannot be extended. Let $O = \langle \theta, \Phi \rangle$ be an object specification. The *set of computations over O* , denoted $\text{comp}(O)$, is the set of all computations over any timed object frame $\mathcal{F}$ which is a model of O . We also use $\text{comp}_{\mathcal{F}}(O)$ to denote computations over the fixed model $\mathcal{F}$.

The temporal language over a signature $\theta = (A, \Gamma)$ is defined as follows:

$$\tau ::= a \mid t \mid t_1 \supset t_2 \mid \neg \tau \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{X}_{\sim c} \tau \mid \mathbf{Y}_{\sim c} \tau \mid \tau_1 \mathbf{U}_{\sim c} \tau_2 \mid \tau_1 \mathbf{S}_{\sim c} \tau_2$$

where $a \in A$, $t \in \text{AT}(\theta)$, $\sim$ is a relation on TIME , $c \in \text{TIME}$. The semantics of this language is straightforward, see [3].

In summary, the virtual machine provides basic actions whose behaviour is specified in terms of state variables. There are necessary conditions for the action to be performed. There are preconditions that define the situations in which the execution of the action is to have a particular effect. There are upper and lower bounds on the duration of an action (with the latter indicating the time after which the action will be committed and the former indicating the maximum time until the effect of the action becomes observable).

We now wish to define atomic processes. We will use the diagrammatic notation for processes illustrated in Figure 1:

Figure 1: a basic process

Here p denotes a process code (as we will see, an attribute of a process) and P_i , P_o denote the codes of the input and the output products, respectively. These kinds of diagrams hide other information about the process and we will have occasion to use various devices to exhibit some of this information, as and when necessary. Our intention is to define a process as a *transaction* consisting of a segment of a computation of the underlying virtual machine. Hence, we want a process to specify such a segment of a computation by:

- defining an *initial condition* (identifying states of the virtual machine's computations which can serve as an initial state of the transaction);

- defining a *final condition* (identifying states of the virtual machine's computations which can serve as a final state of the transaction);
- defining an *invariant* which must be satisfied throughout the transaction in order for the transaction to complete successfully; and
- defining lower and upper bounds on the time to be taken by the transaction where the lower bound gives a minimum time from some point of the computation within which the transaction must start and the upper bound gives a maximum time within which it must complete.

In order to specify transactions, we need a new operator that allows us to describe segments of computations of such a virtual machine. We use Δ to denote this operator and introduce it into the class of formulae, $\text{FOR}(\theta)$, as follows:

$$\tau ::= a \mid t \mid t_1 \supset t_2 \mid \dots \mid \tau_1 \mathbf{S}_{\sim c} \tau_2 \mid (q, I)_l \Delta^u p$$

where: q is the initial condition, I is the invariant, l is the lower time bound, u is the upper time bound and p is the final condition.

The semantics of this new formula is defined as follows: $\sigma, T, i \models (q, I)_l \Delta^u p$ iff for some j, k such that $i+l \leq j < k \leq i+u$, we have for each $i \leq m \leq j$ that $\sigma, T, m \models q$, $\sigma, T, k \models p$, and for each $j \leq n \leq k$ we have $\sigma, T, n \models I$. In fact, we can reformulate this as: $\sigma, T, i \models (q, I)_l \Delta^u p$ iff for some j such that $i+l \leq j < i+u$ we have for each $i \leq m \leq j$ that $\sigma, T, m \models q$ and $\sigma, T, j \models \mathbf{I} \mathbf{U}_{\leq (i+u-j)} p$ where the latter relation holds if at state j the invariant begins to hold, continues to hold until p becomes true and p must become true before $i+u-j$ time units pass (so as to assure termination of the transaction before the maximum time u allowed for the transaction).

The reason we have included the invariant I in the specification of the transaction is because the transaction is not atomic, in the sense that the actions of the virtual machine are. Once committed, the latter actions complete without interruption or interference, 'ignoring' whatever else may be happening during the elapsed time of the execution. In contrast, transactions may be influenced by changes caused by the execution of other actions in the virtual machine. For example, if someone turns off the printer in the middle of its printing transaction, the latter cannot complete properly. We can represent the availability of the printer throughout the printing transaction via an invariant on the states of the virtual machine. In fact, we will use this feature to represent environmental influence on the characteristics of a transaction. Analogously, q may be thought of as a 'set-up invariant', i.e., a property that has to hold while the transaction is made ready. This is analogous to the enabling condition on actions having to hold for long enough for the action to commit, as defined above.

We say that $(q, I)_l \Delta^u p$ is *satisfied* by the specification $\langle \theta, \Phi \rangle$ iff there is a computation over a timed frame $\mathcal{F}$ such that $(q, I)_l \Delta^u p$ is satisfied in the computation. In other words, the specification has a non-empty set of models satisfying the specification of the transaction.

An atomic process is a pair $\langle \text{proc}, \text{VM} \rangle$ where $\text{VM} = \langle \theta, \Phi \rangle$ is as defined above and $\text{proc} = \langle \text{process_code}, \text{process_name}, \text{process_owner}, P_i, P_o, (q, I)_l \Delta^u p \rangle$ where:

- *process_code* is a state variable of the appropriate sort (used for process codes) and which is rigid (i.e., does not change value during computations);
- *process_name* is a state variable of the appropriate sort (used for process names) and which is rigid (i.e., does not change value during computations);
- *process_owner* is a state variable of the appropriate sort (used for process owner names) and which is rigid (i.e., does not change value during computations);
- P_i, P_o are the input and output products, respectively, for the process with the attributes of the products interpreted as state variables of the virtual machine; and
- $(q, I)_l \Delta^u p$ is the specification of the process's various properties, which we call the *transfer function* of the process.

The semantics of an atomic process $p = \langle proc, VM \rangle$, denoted by $comp(p)$, is given by the subset of $comp(VM)$ which satisfies $(q, I)_i \Delta^u p$. That is, we take the timed state sequences over frames, which are models of the virtual machine and choose those computations that satisfy the process specification.

We can use TATDs to specify an atomic process by associating the initial condition q with the initial state of a TATD, the final condition p with the final state of a TATD, and the invariant and time bounds with the transition from the former to the latter. The following diagram illustrates this:

Figure 2: a simple transaction diagram defining an atomic process

In fact we can use arbitrary such TATDs as long as a few restrictions are observed. Firstly, we require that there are unique initial and final states in the TATD. We also require that each transition be labelled by a formula of the form $(q, I)_i \Delta^u p$. We then obtain ‘complex’ invariants from the process thus defined. We will not detail here how this works.

An instance p^i of an atomic process $p = \langle proc, VM \rangle$ is a computation $\langle \sigma, T \rangle$ over a timed object frame $\mathcal{F}$ such that there is an i such that σ_i satisfies the transaction specification in $proc$. We also call such an instance an *execution trace* for p . So, an execution trace is a sequence of states and transitions in some model of the underlying virtual machine that satisfies the transaction specification.

Let us consider an example, which will be completed in section 5. The example shows how the real-time features of the specification language play a decisive role in the selection of the refinements of processes. We concentrate first on the prescription of the required process. Our products will be very simple and their role in the example completely de-emphasised.

A manufacturer of candy vending machines wants to manufacture machines with the following characteristics. If the machine has candy, then, after money has been deposited, at most a time K_1 passes before candy is delivered. If the machine is empty, then at most a time K_2 can pass before the transaction is finished. If the machine can be fully replenished in time to meet the K_2 upper bound, then it should be replenished; otherwise, the money should be given back.

Let us model the process that forms the part of the behavior of the machine after money has been introduced and until candy has been delivered or the money was given back. We do this by defining an appropriate formula $(q, I)_i \Delta^u p$; we will actually do this by giving I , u and l and characterising q and p in corresponding ‘parts’.

I : true

The invariant is the trivial condition. (But see below!)

$p1$: $\$_{in} \wedge x = \#candy > 0 \wedge u \geq K_1 \wedge l = 0$

$q1$: $\#candy = x - 1 \wedge delivered$

This covers the situation where there are candies in the machine and the upper time bound for the required process is greater than that needed to deliver the candy.

$p2$: $\$_{in} \wedge \#candy = 0 \wedge u \geq K_2 \wedge l = 0$

$q2$: $\#candy = \max_candy - 1 \wedge delivered$

This covers the situation where there are no more candies in the machine, but there is enough time to replenish the machine before the customer gets tired.

$p3$: $\$_{in} \wedge \#candy = 0 \wedge u < K_2 \wedge l = 0$

$q3$: $delivered \wedge money_back$

If the manufacturer believes that a consumer can wait 3 minutes for candy without losing his patience, then K_2 can be set to 3 minutes in the specification. Let us assume that as a constraint, this part of the machine must be built using the following (subset of) actions of the assumed general vending machine (i.e., it is the basic actions of

which a general vending machine is capable; it must be programmed to exhibit the behaviour described by the formulae above):

RETURN_PRODUCT (which returns candy provided the machine is not empty; its lower time bound is 0 and the upper time bound is 3 seconds);

REPLENISH (which fully replenishes the machine; its lower time bound is 0 and the upper time bound will be discussed later);

RETURN_MONEY (which gives the customer his/her money back; its lower time bound is 0 and the upper time bound is 3 seconds).

We will complete the example in section 5 when we have defined the concept of refinement and we have conceptual tools to check the correctness of implementations.

4 Gates

The Mensurae method requires that the processes used to model empirical referents have a single input product (so as to make the start time of the process unique in any execution trace) and a single output product (so as to make the termination time of the process unique in any execution trace). Hence, it becomes problematic to model a process that may have many input sources without finding some way to synchronise the arrival of the inputs (in time), and some way to ‘package’ the various products into the single input product.

Similarly, the output product needs to be ‘unpacked’ so as to enable the sending of various constituent products to the appropriate subsequent processes. There is also the problem of dealing with context conditions defining when a part of a process may proceed. For example, it may be that some product can move to a process only after some condition is met, as in the case of a print job being held in a queue until the printer is ready (e.g., has been turned on).

We use the concept of a *gate* to model the constructs we need in the language. Gates are simply artefacts of the method, in that they do not appear explicitly in empirical referents and are used only to achieve the ‘artificial’ conditions set by the method. There are three basic types of gates which we will discuss: the multiplexer, the demultiplexer, and the semaphore.

A *multiplexer* m_c (see Figure 3 for a graphical representation) defined over some virtual machine $VM = \langle \theta, \Phi \rangle$ is a four-tuple $\langle \text{multiplexer_code}, \mathcal{P}_c, P_c, F_c \rangle$ where:

- *multiplexer_code* is a rigid variable of the underlying virtual machine used as a unique identifier of the gate in some model;
- $\mathcal{P}_c = \{P_1, \dots, P_n\}$ is the set of input products to the multiplexer;
- $P_c = \langle P_1', \dots, P_m' \rangle$ is the output product of the multiplexer; and
- F_c is the multiplexer action function defining each P_i' explicitly in terms of the set of input products $\mathcal{P}_c$.

Figures 3-5: gates

A *demultiplexer* d_c (see Figure 4) and a *semaphore* s_c (see Figure 5) are defined similarly.

Gates are used to connect processes by means of their input/output products. We now want to define what the result is of such a connection in terms of the properties and semantics of the resulting constructs. These constructions will also be useful when defining abstractions and explosions of processes in section 5. Suppose that we have a multiplexer $m_c = \langle c, \mathcal{P}_c, P_c, F_c \rangle$ and that the process p_i outputs the corresponding product P_i to the gate m_c and that the output product P_c of the gate is the input to process p_c . Diagrammatically, we have the situation of Figure 6a.

Figure 6a

Figure 6b

Figure 6c

We have presumably associated the TATDs T_j , $i = 1, \dots, n$, and T_c with the processes p_j and p_c , respectively. An obvious question to ask is: What is the TATD associated with the combination? Firstly, we have to define how a transition is made from the final state L_{jf} of T_j to the initial state L_{ci} of T_c . Clearly, not only must the initial condition p_{ci} of p_c be true, but so must the condition q_{jf} associated with the final state of T_j . Hence, we can define a transition from the final state L_{jf} to the initial state L_{ci} as in Figure 6b. Details may be found in [17].

5 Processes

When defining processes, we will be modelling very complex empirical referents, even after deciding to abstract away unnecessary details of the latter. We can proceed in two ways to overcome this complexity: work bottom up and model small subprocesses and compose them successively until we have modelled the overall process in which we are interested, or work top down and specify the overall effect we want from the process and then refine this abstract specification in terms of process combinations, eventually using the actions of the underlying abstract machine. In either case, we need to relate an ‘abstract’ atomic process to a refinement of it in terms of more concrete processes, products and gates. We start with the following definition: a *P/p framework* is either a product or a process. Whereas a product is potentially a stand-alone object, a process is not. The latter requires the existence of two other referents and their models, i.e., the two products constituting the input and the output of the process. A *process* is either an atomic process or a framework process.

Our empirical referents will, in the end, have to be modelled in terms of the atomic processes supplied as the actions/methods of our underlying abstract machine. Any process we define will eventually have to be realised by a bottom up or top down construction from these atomic constituents. We must now define what we mean by the term framework process.

We need some way of combining constituent processes and products, via the use of gates, to define process-like objects with single input and single output products, a well defined transformation from the former to the latter and well defined minimum and maximum execution times. These combinations will be defined in terms of several basic combinators: sequential combination of two processes, semaphore combination of two processes, and parallel combination of a set of processes via a multiplexer and a demultiplexer. Let $p_i = \langle code_i, name_i, owner_i, P_{ib}, P_{io}, (q_i, I_i)_1 \Delta^u p_i \rangle$, $i = 1, 2$, be two processes defined over the same virtual machine VM . Let $\tau_i = (q_i, I_i)_1 \Delta^u p_i$. Then, the *sequential composition* of p_1 and p_2 , denoted by $p_1 ; p_2$, is the process $p = \langle code, name, owner, P_{ib}, P_{io}, \tau \rangle$

where: *code* is a new (unique) code for the newly defined process; *name* is a new (unique) name for the newly defined process; *owner* is a new (not necessarily unique) entity responsible for the newly defined process;

- P_{1i} , the input product of p_1 , is the input product of the newly defined process;
- P_{2o} , the output product of p_2 , is the output product of the newly defined process;
- τ is the transfer function $(q, I)_1 \Delta^u p$ where:
 - q is q_1 ;
 - p is p_2 ;
 - I is $q_1 \ U_{>l_1} (I_1 \ U_{\leq u_1-l_1} (p_1 \rightarrow \text{true} \ U \ (q_2 \ U_{>l_2} (I_2 \ U_{\leq u_2-l_2} p_2))))$ which requires that:
 - the initial condition q_1 of p_1 holds for at least time l_1 ;
 - at which point the invariant I_1 of p_1 holds for time not exceeding u_1-l_1 ;
 - at which point the final condition p_1 of p_1 becomes true;
 - and this means that eventually the initial condition q_2 of p_2 holds for at least time l_2 ;
 - at which point the invariant I_2 of p_2 holds for time not exceeding u_2-l_2 ;
 - at which point the final condition p_2 of p_2 becomes true.
 - $l = l_1$;
 - $u = u_1 + u_2$;
 - τ is $(q_1, I)_1 \Delta^u p_2$.

Clearly, we require the resulting process to have the same initial condition as p_1 and the same final condition as p_2 . The combination will take at least as long as p_1 to commit and will take at most the sum of time that each of p_1 and p_2 would individually to produce a result. Also, we want the invariant I_1 to hold while p_1 executes and ditto I_2 and p_2 . Further, when we reach the final state of p_1 satisfying p_1 , we want that eventually the initial state of p_2 satisfying q_2 , is reached, committing p_2 to start execution. The transfer function for p is then defined from these ingredients.

The *semaphore composition* of p_1 and p_2 , denoted by $p_1 ;_s p_2$, (enabling p_2 to proceed only when the condition in semaphore s becomes true), and the *parallel composition* of p_1 and p_2 via demultiplexer d_1 and multiplexer m_o , denoted by $[p_1 ; p_2](d_1, m_o)$, are defined analogously.

Finally, we are ready to define framework processes in terms of these combinators: a *framework process* p is a process defined from some set of *constituent processes* $\{p_1, \dots, p_n\}$ in terms of the combinators defined above. We denote such a framework process by the pair $\langle p, f\text{-}exp \rangle$ where p is an atomic process obtained from the combination of processes $\{p_1, \dots, p_n\}$ by the combinators in $f\text{-}exp$. For the sake of brevity, we often write $p = \langle \text{code}, \text{name}, \text{owner}, P_{1i}, P_{2o}, f\text{-}exp \rangle$ instead of $p = \langle \text{code}, \text{name}, \text{owner}, P_{1i}, P_{2o}, \tau, f\text{-}exp \rangle$.

Processes are ‘programs’. Like programs, we require some way of describing iterative computations, not simply parallel and sequential combinations of processes. (We may indeed require recursive processes, but we leave this for further work.) The idea behind the construct that we want is the same as that for a while loop in conventional languages: If the semaphore condition S is false, then the loop body, the process p , is not executed and the loop terminates. If the semaphore condition is true, then the loop body, process p , is executed and the execution returns to the semaphore. Note that the input product to the loop, the output product of the loop and the product that is the output of the semaphore and the input of the body process are all the same. This is because a loop would not ‘type check’ if the products were different. The output product of p has to go either to the process after the loop or back to the semaphore to be (possibly) executed again. Hence it must be the same as the input product to the loop. Note, however, that the instances of the product P that would be input to the semaphore at each iteration would not be the same! (Otherwise, the loop would not be doing anything useful!)

What we are attempting to represent is the family of framework processes corresponding to the unfolding of the loop a ‘sufficient’ number of times, as in the semantics of such constructs in programming languages.

As a special kind of process, a while loop should have a specification like any other process, i.e., there should

be an input condition, an output condition, an invariant and lower and upper bounds on the execution time of the loop (as opposed to that of the constituent process). So, if we denote such a loop by $while(S, p)$, with:

- $s_w = \langle code_{s_w}, P, S_w \rangle$ being the semaphore to which the condition S corresponds,
- $p = \langle code, name, owner, P, P, (q, I)_1 \Delta^u p \rangle$ being the process to be used as the body of the while loop, and with
- $p_i = \langle code_p, name_p, owner_p, P_i, P_i, (q_i, I_i)_1 \Delta^{u_i} p_i \rangle$, $i = 1, 2$, the processes to be connected by the loop,

then $while(S_w, p) = \langle code_w, name_w, owner_w, P, P, (q_w, I_w)_1 \Delta^u p_w \rangle$ where:

- $code_w, name_w, owner_w$ are new code, name and owner for the family of processes like entities which the loop represents;
- P is the input and the output product of the loop ‘family’;
- q_w is the triggering condition for the loop. It must be the case that $q_w U(S_w \vee \neg S_w)$ since we want the triggering condition to cause the loop condition S to be evaluated (and it could have the initial value true or false);
- p_w is the output condition of the loop and we must have that:
 - $p_w \rightarrow \neg S_w$, because having exited must mean that the loop condition is false, and
 - $p_w \rightarrow I_w$, because the loop invariant should be true when the loop body terminates;
- I_w is the loop invariant and it must be true before the loop body executes and after the loop body terminates, but not necessarily during the loop body’s execution (as is the case for loops in conventional languages); and
- l_w and u_w are the lower and upper bounds on the time taken to execute the loop.

If the loop condition is false the first time it is met, then the loop connects the processes p_1 and p_2 as if it were the semaphore with condition $q_w \wedge \neg S_w$. Hence, we need the triggering condition of the loop to hold along with the semaphore condition $\neg S_w$ (when the process p_2 is available for execution). This is not quite the semaphore we started with, because there is the extra requirement that the triggering condition for the loop is to hold while the semaphore passes on the condition $\neg S_w$.

If the loop body is executed once, then the initial instance of the semaphore becomes $q_w \wedge S_w$ to indicate that the triggering condition for the loop holds when the positive result S_w holds, simultaneously with the initial condition q of the loop body. And then, after one execution of the loop body, the semaphore test fails and the loop is exited, implying that we have a semaphore condition $p \wedge \neg S_w$ connecting the loop body p with the succeeding process p_2 .

The only case left is the one when there is more than one execution of the loop body. In this case, the initial and final instances of the semaphore are connected to the first instance of the loop body p and the succeeding process p_2 , respectively, as in the paragraph above. Then there is the general case to be considered of one instance of the loop body connected by a positive result of the semaphore to the succeeding instance of the loop body. The semaphore condition in this case is just S .

Note that what we have defined is a family of framework processes consisting of: the initial process p_1 ; a number (0 or more) of positive occurrences of the semaphore, each followed by the loop body p ; and, finally, a negative occurrence of the semaphore connected to the succeeding process p_2 . The semantics of the loop construct is given in terms of this family of framework processes. For each member of the family, the framework defines a set of computations as explained above for frameworks generally. We further require that the end-to-end time from entering the first instance of the loop body to entering the final state of the final instance conforms to the loop specification.

Having mentioned the time limits set in the loop specification, as a default, the former may be 0 and the latter ∞ to reflect the fact that a semaphore takes no time to ‘execute’ and a loop may be unbounded. However, different limits may be specified, resulting in proof obligations that must be verified so as to demonstrate that the lower and

upper bounds are met by any execution of the loop.

Given a process specification $p_1 = \langle \text{code}_1, \text{name}_1, \text{owner}_1, P_{1p}, P_{1o}, \tau_1 \rangle$, we say that process $p_2 = \langle \text{code}_2, \text{name}_2, \text{owner}_2, P_{2p}, P_{2o}, \tau_2 \rangle$ is a *refinement* or *explosion* of p_1 , denoted $p_2 \sqsubseteq p_1$, iff $\text{comp}(p_2) \subseteq \text{comp}(p_1)$.

So, a refinement is exactly what one would expect – a process with a set of associated computations, which is a subset of the abstract process’s computations. This concept of refinement reminds one of the definitions used in algebraic specification: a refinement is correct iff the models of the refined specification are a subset of the models of the abstract specification. However, to obtain such a simple definition in the algebraic specification theory requires a lot of ‘contortions’: take quotients, forget junk, take a reduct along a signature morphism, etc. It is interesting that here there is a direct version of this idea which may be used and it appears that the target language (that of the virtual machine) being fixed underpins this simplicity. We might call this targeted notion of refinement *language constrained refinement* and will focus on it in a future paper. Note that this does not actually give us direct methods for proving that one process is a refinement of another. We have at the moment only the indirect method given above – compare the relevant computations. We leave as further work the development of appropriate direct methods.

Of course, if $\text{comp}(p_2) = \emptyset$, the empty set of computations, then p_2 is a refinement of p_1 ! But, in general, this would be unsatisfactory and should only occur if either we have built the refinement badly or there is a genuine problem and the original process specification already has an empty set of computations. The latter case may be an impetus to retool our factory, so that we do have in place the right constituents to realise our process specification.

Let us return to the example at the end of section 3. We want to refine our requirement for a process to control the vending machine so that the specification is implemented in terms of the three given atomic actions RETURN_PRODUCT, REPLENISH and RETURN_MONEY. If the machine is to be placed inside a convenience store, then as soon as the machine is emptied a clerk will replenish it, and therefore, a reasonable upper time bound for the replenishing action might be 2 minutes. Then, the following (not completely described) framework process shows a feasible implementation:

(Note that we have not labelled arrows with products, as we are not focusing on these in this example. The labels we have included are to help interpret the demultiplexers as implementing a conditional.)

If the machine is to be placed in an underground station, then it may be expected that it will not be replenished more than twice a day. Then, the upper time bound for the replenishing action might be, for instance, 12 hours. In this case, the previously described process does not satisfy the specification, but the following one does:

Of course, we have not given enough details to verify that the framework processes have the right functionality. (We would have to specify the underlying general vending machine.) One further variant is worth discussing. If the vending machine has a button the customer can use to retrieve the coin that he/she has inserted, then we might wish

to condition the process we have specified by requiring that the customer has not pressed this button. We can do this by changing the invariant I to $\neg \text{return_coin_pressed}$. This would have the effect of predicating the correct functioning of the framework processes defined above on the non-detection of a button press by the customer during the time that the vending machine is executing the framework process.

6 Conclusions

We have described briefly a language for modelling industrial processes, developed as part of the Mensurae project being conducted at LMF-DI, PUC-Rio de Janeiro. The language supports a method based on ideas first put forward by Agnes Kaposi (in, e.g., [15]). We have said very little about the method, which has some claim to universality in process modelling. No language, on its own, is really an effective tool. It requires systematic method in the hands of skilful engineers to turn it into that effective tool.

The language we have described is at one level a very simple one. It has only two kinds of constructs, namely products and processes. Products are modelled by first order theories, which describe the (class of) products in terms of their basic measurable characteristics. Because we are modelling industrial processes, some of these characteristics are related to the role that such products play in these processes. So, we have product codes and names, together with time stamps, which establish when the characteristics were determined by means of appropriately defined measurement procedures. Products may be composed (by means of tupling and choice operators) from other products to enable us to marshal inputs to our single input, single output processes. Products may also be complex (i.e., assembled or manufactured), in the sense that they are modelled, via a set of object oriented operations, from constituent products.

Processes are single input, single output transformations which take some non zero amount of time and that model the changes wrought by business processes on products. Like products, they are modelled in terms of appropriate characteristics, which again include codes and names. They also include further characteristics, like responsible entities (owners), timing constraints and input/output transformation. Processes may be composed from (or refined to) framework processes built from constituent processes via sequential and parallel combinators. A gate is an artefact of the method used to enforce the single input, single output regime of processes. It can also be used to define control mechanisms, including loops.

The computational model used is analogous to that for programs being executed on an abstract machine. The role of the latter is played by a system of objects modelling the basic capabilities of the organisation. A process is then like a program executing over this abstract machine, ‘causing’ it to execute the process. As such, a process may be affected by external influences, via conditions dependent on attributes not fully under the control of the executing process. Hence, other processes executing concurrently with the process in question may influence its behaviour. Examples of such external influences may be management procedures, quality procedures, as well as other technical/organisational processes.

There are many avenues of further research to be explored. We need to understand how the interaction between concurrent processes running in the same organisation may be expressed effectively. Clearly, synchronisation between processes and the notion of one process being ‘meta’ with respect to another are relevant points at issue. There is an obvious connection with the concept of threads of execution over object oriented systems. The process language presented may well be a good starting point for exploring this thorny problem in object oriented programming.

Acknowledgements The Mensurae method is the result of several years of work by many people. Agnes Kaposi was the inventor of the method and we are all grateful for her input (and that of John Kaposi). Big steps were then taken with the help of Armando Haeberer, Paulo Stark, Oscar Porto, Laura Bahiense and Luiz Tucherman, amongst others. I thank all of them for their hard work and tolerance.

7 References

1. Bicarregui J, Lano K, Maibaum T. Towards a Compositional Interpretation of Object Diagrams. In: Proceedings of IFIP Working Conference on Algorithmic Languages and Calculi, Chapman and Hall, 1997
2. Carvalho S, Fiadeiro J, Haeusler E. A Formal Approach to Real-Time Object Oriented Software. In: Proceedings of the Workshop on Real-Time Programming. IFAP/IFIP, 1997, pp91-96, Lyon, France
3. Chang E, Manna Z, Pnueli A. Compositional Verification of Real-time Systems. In: Proceedings of IEEE Symposium on Logic in Computer Science. IEEE Computer Science Press, 1994, 458-465
4. Cook S, Daniels J. Designing Object Systems. Prentice-Hall
5. Fenton N, Pfleeger S. Software Metrics: A Rigorous and Practical Approach (2nd edition). International Thomson Computer Press, 1996
6. Fiadeiro J, Maibaum T. Temporal Theories as Modularisation Units for Concurrent System Specification. Formal Aspects of Computing. 1992, 4(3):239-272
7. Fiadeiro J, Maibaum T. A Mathematical Toolbox for the Software Architect. Proceedings of 8th International Workshop on Software Specification and Design. Kramer J, Wolf A, eds. IEEE Computer Society Press, 1995, pp46-55
8. Finkelstein A, Kramer J, Nuseibeh B. Software Process Modelling and Technology. John Wiley, 1994.
9. Frias M, Baum G, Haeberer A. Fork Algebras in Algebra, Logic and Computer Science. Fundamenta Informaticae, 1997, 32:1-25
10. Frias M, Baum G, Maibaum T. A logic for Real-time Systems Specification, Its Algebraic Semantics and Equational Calculus. In: Proceedings of AMAST'98. Springer-Verlag, Heidelberg, 1999, pp91-105 (Lecture Notes in Computer Science No. 1548)
11. Haeberer A, Maibaum T. The very Idea of Software Development Environments: A Conceptual Architecture for the ARTS Environment Paradigm. In: Proceedings of ASE'98. Redmiles D, Nuseibeh B, eds. IEEE Computer Science Press, 1998
12. Hammer M, Champy J. Re-engineering the Corporation: A Manifesto for Business Revolution. Nicolas Brealey Publishing, 1993
13. Harel D. On Visual Formalisms. Communications of the ACM 1998, 31(5):514-530.
14. Henzinger T, Manna Z, Pnueli A. Timed Transition Systems. In: Real-Time: Theory in Practice. de Bakker JW, Huizing C, de Roever WP, Rozenberg G, eds. Springer-Verlag, Heidelberg, 1992, pp226-251 (Lecture Notes in Computer Science No. 600)
15. Kaposi A, Myers M. Systems, Models and Measures. Springer-Verlag, London, Formal Approaches to Computing and Information Technology, 1994.
16. Lehman M. Process Models, Process Programs, Programming Support. Proceedings of 9th International Conference on Software Engineering. IEEE Computer Society Press, 1987, pp14-16
17. Maibaum T. The Mensurae Language: Specifying Business Processes. Technical Report, King's College London, Department of Computer Science, 1999
18. Meyer B. Systematic Concurrent Object-Oriented Programming. Communications of the ACM, 1993, 36(9)

19. Mitchell S, Wellings A. Synchronization, Concurrent Object-Oriented Programming and the Inheritance Anomaly. Computer Languages, 1996, 22(1)
20. Osterweil L. Software Processes are Software Too. In: Proceedings of 9th International Conference on Software Engineering . IEEE Computer Society Press, 1987, pp2-13
21. Roberts F. Measurement Theory, with Applications to Decision-Making, Utility and the Social Sciences. Addison-Wesley
22. SegerbergK. (1989). Bringing it About. Journal of Philosophical Logic, 1989, 18:327-347
23. Selic B, Gullekson G, Ward P. Real-Time Object-Oriented Modeling. John Wiley
24. Suppe F. The Structure of Scientific Theories. University of Illinois Press, 1979