ISSN 0868-4081
0868-4154

TALLINNA TEHNIKAÜLIKOOLI

TOIMETISED

ТРУДЫ ТАЛЛИННСКОГО
ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА

TRANSACTIONS OF TALLINN
TECHNICAL UNIVERSITY

PROCESS AND CIRCUIT
MODELLING AND ANALYSIS

TALLINN 1990

722

ALUSTATUD 1937

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

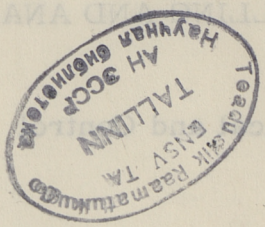
ТРУДЫ ТАЛЛИННСКОГО ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА

PROCESS AND CIRCUIT MODELLING AND ANALYSIS

Electrical and Control Engineering XL

TALLINN 1990

CONTENTS



Edited by E. K a l m

No 722

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

E. Kängsep

NON-TWO-PORT CHARACTERISTIC MODEL OF BALANCED TRANSMISSION LINE

Abstract. This paper is an attempt to find a non-two-port characteristic model of a balanced transmission line for applications in time domain analysis. The presented model is a more general case of the convolution model of transmission line presented by M. Valtonen (1978) and used by other authors.

A balanced two-wire line in Fig. 1, (a) consists of two symmetrical conductors and is described by the series distributed resistance R and series distributed proper inductance L (here values of R and L are the sum of these values of both wires). M is the distributed coupled inductance of the wires, G is the shunt distributed conductance, and C is the shunt distributed capacitance. This transmission line can be approximated by a circuit shown in Fig. 1, (b). The nodes are separated by a small distance Δz, and the node subscript t locates the node on the line according to $z=t\Delta z$. By Fig. 1,(b) the circuit equations can be written as follows:

$$\Delta u^{*}_{1,t} = i_{1,t}\,\frac{R}{2}\,\Delta z + \frac{L}{2}\,\Delta z\,\frac{di_{1,t}}{dt} + M\,\Delta z\,\frac{di_{2,t}}{dt} \tag{1a}$$

$$\Delta u^{*}_{2,t} = i_{2,t}\,\frac{R}{2}\,\Delta z + \frac{L}{2}\,\Delta z\,\frac{di_{2,t}}{dt} + M\,\Delta z\,\frac{di_{1,t}}{dt} \tag{1b}$$

$$-\Delta u_{t} = \frac{R}{2}\,\Delta z\,(i_{1,t} + i_{2,t}) + (\frac{L}{2} + M)\,\Delta z\left[\frac{di_{1,t}}{dt} + \frac{di_{2,t}}{dt}\right] \tag{1c}$$

$$-\Delta i_{1,t} = -\Delta i_{2,t} = G\,\Delta z\,u_{t+1} + C\,\Delta z\,\frac{du_{t+1}}{dt} \tag{1d}$$

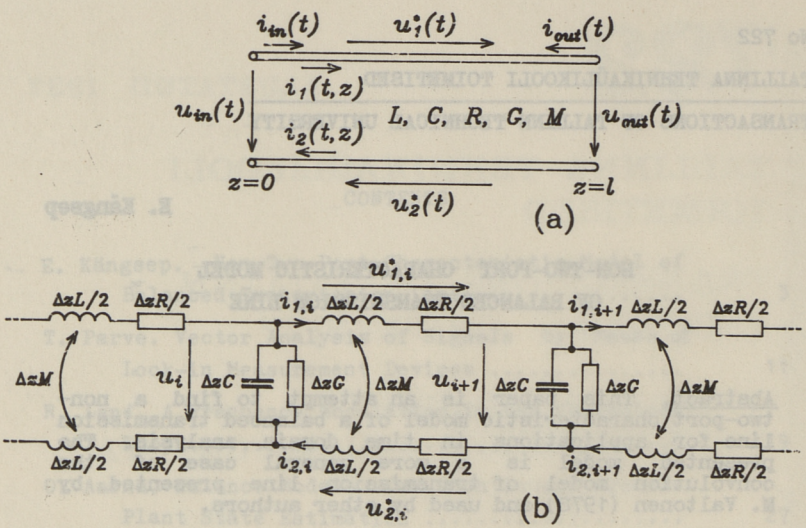


Fig. 1. Balanced two-wire line (a) and its circuit model (b).

Here
$$\Delta i_{1,i} = i_{1,i+1} - i_{1,i} \tag{2a}$$
$$\Delta i_{2,i} = i_{2,i+1} - i_{2,i} \tag{2b}$$
$$\Delta u_i = u_{i+1} - u_i \tag{2c}$$

After the division of both sides of equations (1a)...(1d) by the increment Δz we obtain:

$$\frac{\Delta u_{1,i}^*}{\Delta z} = i_{1,i}\frac{R}{2} + \frac{L}{2}\frac{di_{1,i}}{dt} + M\frac{di_{2,i}}{dt} \tag{3a}$$

$$\frac{\Delta u_{2,i}^*}{\Delta z} = i_{2,i}\frac{R}{2} + \frac{L}{2}\frac{di_{2,i}}{dt} + M\frac{di_{1,i}}{dt} \tag{3b}$$

$$-\frac{\Delta u_i}{\Delta z} = \frac{R}{2}(i_{1,i} + i_{2,i}) + (\frac{L}{2} + M)\left\{\frac{di_{1,i}}{dt} + \frac{di_{2,i}}{dt}\right\} \tag{3c}$$

$$-\frac{\Delta i_{1,i}}{\Delta z} = -\frac{\Delta i_{2,i}}{\Delta z} = G\,u_{i+1} + C\frac{du_{i+1}}{dt} \tag{3d}$$

Let us assume that the increment Δz approaches zero. The left-hand side of (3a)...(3d) would approach partial derivatives in respect to z. In the limit, (3a)...(3d) are identical to (4a)...(4d), namely,

$$\frac{\partial u_1^*}{\partial z} = i_1 \frac{R}{2} + \frac{L}{2} \frac{\partial i_1}{\partial t} + M \frac{\partial i_2}{\partial t} \tag{4a}$$

$$\frac{\partial u_2^*}{\partial z} = i_2 \frac{R}{2} + \frac{L}{2} \frac{\partial i_2}{\partial t} + M \frac{\partial i_1}{\partial t} \tag{4b}$$

$$-\frac{\partial u}{\partial z} = \frac{R}{2} (i_1 + i_2) + (\frac{L}{2} + M) \left(\frac{\partial i_1}{\partial t} + \frac{\partial i_2}{\partial t} \right) \tag{4c}$$

$$-\frac{\partial i_1}{\partial z} = -\frac{\partial i_2}{\partial z} = G u + C \frac{\partial u}{\partial t} \tag{4d}$$

Here all voltages and currents are the functions of the time t and of the distance from the beginning of the line z.

From (4d) it follows that

$$i_1 = i_2 + 2i_f(t) \tag{5}$$

for all $z = [0,l]$ (l is line length) $i_f(t)$ is a current dependent on time only and independent of z.

Then we can consider the two-port current i:

$$i = i_1 - i_f(t) = i_2 + i_f(t) \tag{6}$$

and

$$i_1 = i + i_f(t) \tag{7a}$$
$$i_2 = i - i_f(t) \tag{7b}$$

The current i_f is the same direction part of currents on wires of line (it is "a non-two-port current").

Replacing the currents in equations (4a)...(4d) by (7a) and (7b), we obtain:

$$\frac{\partial u_1^*}{\partial z} = i \frac{R}{2} + (\frac{L}{2} + M) \frac{\partial i}{\partial t} + i_f \frac{R}{2} + (\frac{L}{2} - M) \frac{di_f}{dt} \tag{8a}$$

$$\frac{\partial u_2^*}{\partial z} = i\,\frac{R}{2} + \left(\frac{L}{2} + M\right)\frac{\partial i}{\partial t} - i_f\,\frac{R}{2} - \left(\frac{L}{2} - M\right)\frac{di_f}{dt} \qquad (8b)$$

$$-\frac{\partial u}{\partial z} = R\,i + (L + 2M)\,\frac{\partial i}{\partial t} \qquad (8c)$$

$$-\frac{\partial i}{\partial z} = G\,u + C\,\frac{\partial u}{\partial t} \qquad (8d)$$

Equations (8c) and (8d) are well-known transmission line (telegraph) equations.

Equations (8a) and (8b) describe voltage below the beginning and ending nodes.

Let us take

$$u_f = \int_0^l \left(i\,\frac{R}{2} + \left(\frac{L}{2} + M\right)\frac{\partial i}{\partial t} \right) dz \qquad (9)$$

then from (8a) and (8b) we get

$$u_1^* = \int_0^l du_1^* = u_f + \frac{di_f}{dt}\int_0^l \left(\frac{L}{2} - M\right) dz + i_f \int_0^l \frac{R}{2}\,dz =$$

$$= u_f + \frac{di_f}{dt}\left(\frac{L}{2} - M\right) l + i_f\,\frac{R}{2}\,l \qquad (10a)$$

$$u_2^* = \int_0^l du_2^* = u_f - \frac{di_f}{dt}\int_0^l \left(\frac{L}{2} - M\right) dz - i_f \int_0^l \frac{R}{2}\,dz =$$

$$= u_f - \frac{di_f}{dt}\left(\frac{L}{2} - M\right) l - i_f\,\frac{R}{2}\,l \qquad (10b)$$

Comparing (10a) with (10b) we obtain

$$u_1^* + u_2^* = 2\,u_f \qquad (11)$$

By port voltages at the beginning (u_{in}) and the end (u_{out}) of the line (see Fig. 1, (a)) we can write

$$u_{in} - u_{out} = u_1^* + u_2^* \tag{12}$$

$$u_f = \frac{u_{in} - u_{out}}{2} \tag{13}$$

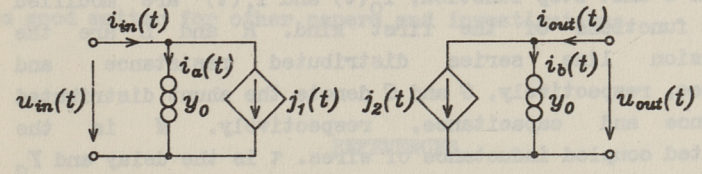


Fig.2. A model of telegraph equations solution.

Solution of telegraph equations (8c) and (8d) leads to the model in Fig. 2 that is described by equations as follows:

$$j_1(t) = i_{imp}(t) * \left[2i_b(t) + j_2(t) \right] \tag{14a}$$

$$j_2(t) = i_{imp}(t) * \left[2i_a(t) + j_1(t) \right] \tag{14b}$$

$$i_a(t) = y_0(t) * u_{in}(t) \tag{15a}$$

$$i_b(t) = y_0(t) * u_{out}(t) \tag{15a}$$

Here "*" denotes convolution, $i_{imp}(t)$ is the impulse response of matched transmission line. $y_0(t)$ is the impulse response of the characteristic admittance of line.

$$i_{imp}(t) = e^{-at}\delta(t-\tau) + 1(t-\tau)\frac{b\tau e^{-at}}{\sqrt{t^2-\tau^2}} I_1\left(b\sqrt{t^2-\tau^2}\right) \tag{16}$$

$$y_0(t) = Y_0 \left\{ \delta(t) + 1(t)\, b\, e^{-at} \left[I_1(bt) - I_0(bt) \right] \right\} \tag{17}$$

where

$$a = \frac{1}{2}\left(\frac{R}{L+2M} + \frac{G}{C} \right) \tag{18}$$

$$b = \frac{1}{2}\left(\frac{R}{L+2M} - \frac{G}{C} \right) \tag{19}$$

$$\tau = l\sqrt{(L+2M)C} \qquad (20)$$

$$Y_0 = \sqrt{\frac{C}{L+2M}} \qquad (21)$$

Here $\delta(t)$ is the Dirac's impulse function, $1(t)$ is the Heaviside's unit step function, $I_0(t)$ and $I_1(t)$ are modified Bessel's functions of the first kind. R and L are the transmission line series distributed resistance and inductance, respectively, G and C denote the shunt distributed conductance and capacitance, respectively. M is the distributed coupled inductance of wires. τ is the delay and Y_0 is the characteristic admittance on infinite frequency.

Solution of equations (8a) and (8b) leads to the equations (10)...(13). Circuit representation of these equations means additional components, shown in Fig. 3, where

$$L_X = (\frac{L}{2} - M)\, l \qquad (22)$$

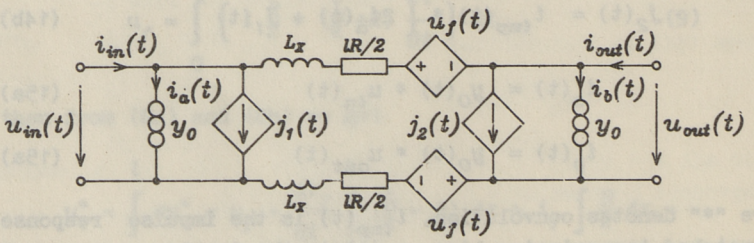


Fig.3. A non-two-port model of balanced transmission line.

CONCLUSION

To use this model in the circuit analysis program for time domain analysis of mixed lumped and distributed circuits, the line must be replaced with an equivalent circuit, shown in Fig. 3.

Currents and voltages of sources and conductances of characteristic admittances y_0 in each time step t_n can be obtained by use of numerical calculation of convolutions.

The advantage of this model is the possibility to simulate the transmission line non-two-port behavior.

Here we do not discuss numerical methods for calculation of convolution and required impulse responses. These problems are a good matter for other papers and investigations.

REFERENCES

Valtonen, M. (1978), *Computer-Aided Analysis of Mixed Lumped and Distributed Circuits in Time Domain.* Proc. of IEEE CAS Conf., New York.

Uhle, M. (1983), *Transientanalyse nichtlinear dynamischer Netzwerke mit verlustbehaften elektrischen Leitungen.* Nachrichtentechnik Elektronik, NTE 33, 12.

Mayer, D. (1981), *Úvod do teorie elektrických obvodů.* SNTL - ALFA, Praha.

Kängsep, E. (1985), *Simulace soustav se rozprostřenými parametry.* Diplomová práce, ČVUT, Praha.

Kängsep, E. (1989), *General Characteristic Model of Linear Time-Invariant Two-Ports,* TTÜ Toimetised, No.702, Tallinn.

Dworsky, L. N. (1979), *Modern Transmission Line Theory and Applications,* John Wiley & Sons.

E. Kängsep

BALANSSEERITUD PIKA LIINI MITTEKAKSPORT-LAINEMUDEL

Kokkuvõte

Käesolev artikkel on katse leida balanssseeritud pikkade liinide mittekaksportmudel kasutamiseks ajaanalüüsil. Siin esitatud liini mudel on üldisem kui M. Valtoneni esitatud ja ka teiste autorite poolt kasutatud pika liini konvolutsioonimudel.

No 722

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

T. Parve

VECTOR ANALYSIS OF SIGNALS BY MEANS OF LOCK-IN MEASUREMENT DEVICES

Abstract. The problem of determining the parameters of the harmonics of alternating current signals by measuring their module and phase or inphase and quadrature components using different measurement instruments, especially the lock-in amplifiers, is discussed in this article.

INTRODUCTION

During the recent years a number of devices designed for spectrum analysis have been put into production by different firms (Tekelec, 1979; Ono Sokki, 1983; Hewlett-Packard, 1986a; 1986b; Wandel & Goltermann, 1987). Most of them are highly modern measuring devices, where the newest technical solutions and signal conversion methods, like the FFT, are used. These devices also seem to be the best means for accomplishing vector analysis.

Nevertheless, in many cases the use of these devices is not the best solution because of some specific features they have. For example, what would you say about the possibility to use a device for measuring the module and phase of a certain harmonic, knowing that the device is able to characterize the signal through 256 spectral lines and has a synchronisation input? Can all these spectral lines be the harmonics of the signal? And what does it mean from the point of view of the signal conversion accuracy that the 13-bit A/D converter is used?

Some new devices have specially been designed for measuring the magnitude and phase of signal harmonics (Solartron, 1986; NF Electronic Instruments, 1984; Dranetz, 1986; North Atlantic, 1986). These devices, being essentially of the same kind as the ones mentioned above, have also many similar characteristics. For example, the use of a 13-bit A/D converter gives approximately 80 dB input dynamic range that in many cases is not very much, especially if compared with the 100 dB or even more of the lock-in amplifiers.

LOCK-IN AMPLIFIERS

Lock-in amplifiers are well known vector measuring instruments in the field of scientific research and technical experiments (Meade, 1983). Though they have been widely used for more than 30 years, new types of lock-in amplifiers have appeared on the market almost every year, and are, therefore, easily obtainable for every experimentator. Why then not to use the lock-in amplifiers for harmonic analysis ? The answer is that most of them are designed to measure only the first, and, possibly also the second harmonic of the signal. The missing link between the signal source and the lock-in amplifier is the frequency multiplier, as shown in Fig.1. By setting the frequency multiplication factor h of the frequency multiplier equal to the number

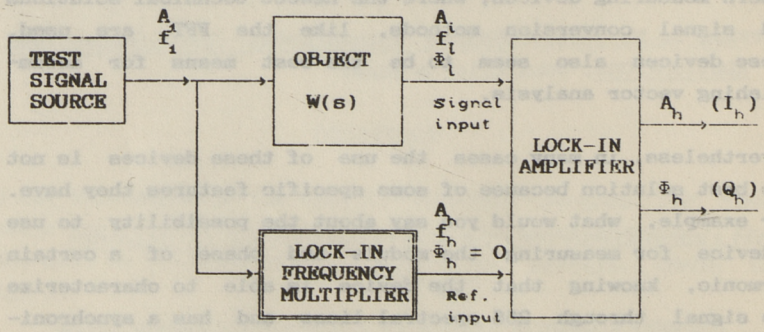


Figure 1. Block diagram of the harmonic vector analysis system on the basis of lock-in amplifier and lock-in frequency multiplier.

of the harmonic to be measured , one can accomplish the harmonic analysis of the signal that has passed through the object under the test. It is essential for vector analysis that the phase relations between the output signal of the frequency multiplier and the signal from the source be kept precise (Min and Parve, 1988).

Still, there are some other problems arising, the most significant of them is the sensitivity of the lock-in's measurement channel to higher odd harmonics of the signal to be measured (Min, 1984). This is due to synchronous detectors (demodulators, rectifiers) used to realize the correlator of the measured and the reference signals (to multiply those signals).

THE HARMONIC SENSITIVITY OF LOCK-INS

The sensitivity of lock-in amplifiers to harmonics is nearly negligible if more sophisticated means are used. For example, the heterodyning lock-ins have the harmonic attenuation of the order of 60 dB (Munroe, 1983). Another possibility is to use harmonic rejecting synchronous demodulators in lock-ins (Min and co-writers, 1981). In the case of stepwise approximation of sine typically 60 dB attenuation of the nearest higher harmonics is achieved (NF Lock-in Amplifiers, 1985), but some harmonics (in accordance with the used sine approximation) have the same level as when using detectors, i.e. the rectangular waveform (Sillamaa and Trumm, 1984).

So the user himself must take care of the methodical side of measurement when a harmonic-sensitive lock-in device is used. The problem is that there is no universal rule predicting how the harmonic effects will affect the results of the measurements. For some types of signals it is found that the errors due to the harmonic effects do not exceed certain limits (Sillamaa and Trumm, 1984). It is shown that errors due to the sensitivity of lock-ins to the harmonics do not exceed a few per cent in the worst case and are only a few tenth parts of per cent in most typical cases.

TABLE 1

Limits of the first harmonic measurement error dependent on the shape (harmonic content) of the signal and the signal conversion mode (sensitivity of the signal converter to the higher harmonics)

Signal conversion mode	Number of approx. levels	Error limit*, %	
		Signal shape (spectrum type)	
		Step wave (1/f spectrum)	Slope wave ($1/f^2$ spectrum)
Rectangular mode	1	22.5	5.2
Approximated sine mode	2	5.1 (5.3)	0.48 (0.51)
	3	2.0 (2.3)	0.14 (0.14)
	4	1.1 (1.3)	0.06 (0.06)
	5	0.6 (0.83)	0.03 (0.03)

* The sum of errors from 50 harmonics. The data given in brackets are asymptotic values by Sillamaa and Trumm, 1984.

The data given in Table 1 are valid for the first harmonic measurement. The situation appears to be quite similar in the case of higher harmonics. Let us presume at first that the relative magnitude of the n-th harmonic $A_n = A_1 / n$, as it is with the rectangular waveform. Then, if the harmonic to be measured is of order h, the relative magnitude of it $A_h = A_1 / h$.

If the demodulator used for measuring the 1st harmonic of the signal has sensitivity Sn to the higher harmonic of the n-th order $S_n = S_1 / n$, as by stepwise approximation of sine, the error of the magnitude measurement of the first harmonic from the harmonic h

$$\Delta A_{1h} = A_n \cdot S_n / S_1 = A_1 / n^2,$$

or in the relative form

$$\delta A_{1h} = A_{1h} / S_1 A_1 = 1 / n^2.$$

If the same demodulator is used to measure the h-th higher harmonic of the same signal, the error of measurement due to the sensitivity of the demodulator to the n-th harmonic can be expressed by the formula

$$\Delta A_{hn} = A_{hn} \cdot S_n = \frac{A_1}{h \cdot n} \cdot \frac{K_1}{n}$$

or in the relative mode

$$\delta A_{hn} = A_{hn} / S_h = 1 / n^2 .$$

So the errors due to the sensitivity of a stepwise approximated sine demodulator to the higher harmonics when measuring either higher harmonic or the first harmonic are equivalent if the signal spectrum is of $1 / f$ shape. The same can be shown for some other types of spectra, e.g. for the $1 / f^2$ shape.

The problem of subharmonic sensitivity of the system consisting of a lock-in amplifier and a lock-in frequency multiplier is also quite complicated. For the lock-ins having the 2f measurement regime, the subharmonic sensitivity level is typically normed at about 0.1 per cent (Land, 1988).

If 1f regime is used, low subharmonic content of the frequency multiplier output signal is required. This complicated problem needs further discussion (Min and Parve, 1988; Land, 1988).

When using a lock-in amplifier phase and magnitude errors due to phase shift and gain error of the preamplifier are also to be taken care of. They become significant at the lower and upper limits of the operating frequency range of the device, and, of course, if the prefiltering is used for some reason.

It is quite well known that a phase shift caused by the preamplifier is usually compensated by introducing a corresponding time delay circuit into the reference channel of the device. It is effective for compensating the high frequency phase lag of the preamplifier. Near the upper limit of the operating frequency range the error appears

to stay quite significant because of the nonlinearity of the amplifier phase lag frequency response.

Commonly the first order transfer function is used. So at the operating frequency of 1/2 of the upper limit the magnitude measurement error reaches the value of the order of -10 % and the phase error appears to be nearly + 2^{o}.

CONCLUSIONS

1. Contemporary lock-in amplifiers can be used for vector analysis of harmonics higher than the 2nd only if the external phase stable frequency multiplier is used;

2. Better accuracy will be achieved if the used lock-in has good attenuation of the harmonics, at least of the nearest higher harmonics;

3. The subharmonic content of the frequency multiplier output signal must be little enough to keep the measurement errors from the first and other subharmonics below the accepted level.

4. The frequency response of the preamplifier must be taken into account when operating at relatively low and relatively high frequencies.

REFERENCES

Model 7530A Spectrum Analyzer. Tekelec Airtronic Catalogue Genaral 1979. Rockland - Tekelec Airtronic, p.106.

FT 512/S Real Time Spectrum Analyzer. Tekelec Airtronic Catalogue General 1979. Rockland - Tekelec Airtronic, p.106.

CF-300 portable FFT analyzer. Ono Sokki Digital Instruments and Control Systems. Cat. No. 1138-3. Japan, 1983.

HP 3562A Spectrum Analyzer / Prospect. Hewlett-Packard Co., 1986.

HP 8590A Portable Spectrum Analyzer / Prospect. Hewlett - Packard Co., 1986.

SNA-1 Spectrum and Network Analyzer / Prospect. Wandel & Goltermann Electronic Measurement Technology, Eningen, BRD. (Data Sheet No.E.03.87/109/8).

Solartron 1250 Series Frequency Response Analyzers/ Prospect. Solartron-Schlumberger, 1986.

Model S-5720 Frequency Response Analyzer. NF Electronic Instruments Short Form Catalog. Japan, 1984. P.7.

Model 3110 Wide Band Phase Sensitive Voltmeter. Mesures, 1986, 10, 130. (Dranetz Co., U.S.A.)

Voltmetre de phase analyseur d´onde. Mesures, 1986, 8, 135. (North Atlantic Industries, U.S.A.).

LI-570 Lock-in Amplifier. NF Lock-in Amplifiers. A Guide. Japan, 1985, p.7 - 10.

Meade, M. L. (1983), Lock-in amplifiers: Principles and applications. Peregrinus, London.

Min, M., and T. Parve (1988), Phase-locked signal processing in vector analyzer. Signal Processing in Measurement: Proc. 6th TC7 MEKO Symposium. IMEKO TC Series No. 16, 97 - 101. Nova Science Publishers, Commac, New York.

Min, M. (1984), Phase-locked signal processing in measurement technique. Trans. Tallinn Techn. Univ., No. 583, 3 - 15, (in Russian, Special issue "Synchronous measuring transducers: Theory, circuits and applications").

Munroe, D. M. (1983), The heterodyning lock-in analyzer. Ithaco Corp. Bulletin.

Min, M., T. Parve, H. Harm, and T. Pungas (1981), Quadrature stepwise frequency converter. U.S. Patent No. 4,409,555.

Sillamaa, H.,and T. Trumm (1984), Methodic Error Analysis of a Synchronous Phase Detector with Multistep Reference Signal. Trans. Tallinn Techn. Univ., No. 583, 73 - 84, (in Russian, special issue "Synchronous measuring transducers: Theory, circuits and applications").

Model 186A Lock-in Amplifier. Operation manual. EG&G Princeton Applied Research Corp., 1975.

Land, R. (1988), Problems of Spectrum Analysis by Means of Synchronous Transducers of Discrete Operation. Trans. Tallinn Techn. Univ., No. 682, 57 -62, (in Russian).

T. Parve

SIGNAALI VEKTORANALÜÜS SÜNKROONMÕÕTERIISTADE ABIL

Kokkuvõte

Artiklis käsitletakse võimalust kasutada laialdaselt levinud sünkroondetektoriga mõõtevõimendeid (sünkroonvoltmeetreid) signaalide vektoranalüüsi teostamiseks. Peamine tähelepanu on pööratud harmooniku vektorparameetrite mõõtmise metoodilistele vigadele, mis sel juhul tekivad analüüsitava signaali keeruka spektraalkoostise puhul.

No 722

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

R. Land

A FREQUENCY MODULATION IN THE PHASE-LOCK LOOP

Abstract. This paper discusses operation principles of the phase-lock loop influenced by frequency modulation. The key specification points of phase-lock loop for signal recovery are reviewed and it is shown how to compute frequency modulating effects. Examples have been presented for examining a few practical problems.

INTRODUCTION

The increasing requirements to the experiment and experimental system parameters cause the trend to examine the familiar solutions better. Frequency modulation in the phase-lock loop is a phenomenon which has not received proper attention in the literature. A classic phase-lock loop contains three basic components (Fig. 1.): a phase detector (PD), a loop filter (LPF) and a voltage-controlled oscillator (VCO) (Gardner, 1979; Meade, 1983). The PD output voltage is proportional to the product of the amplitudes of two inputs and to cosine of the phase between them. The output of the PD, proportional to the phase error, is applied to the low-pass filter LPF, which smooths out the ripple component and delivers a d.c. voltage. Frequency of the VCO is determined by the control voltage v_c. When the loop is locked, the control voltage is such that the frequency of the VCO is exactly equal to the frequency of the input signal.

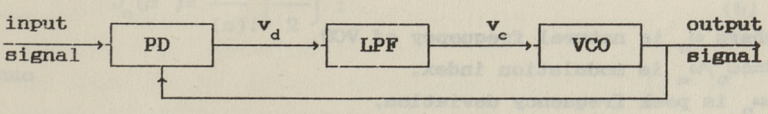


Fig. 1. Basic phase-lock loop.

There are many requirements placed on loop filter in different applications. These requirements are usually in conflict with one another, and therefore a compromise is needed. Two important characteristics of the filter contradicting each other are that the bandwidth must be very small to reject the large amount of noise and high frequency signal components, and that the filter automatically tracks the signal frequency.

ESSENTIAL PRINCIPLES OF FREQUENCY MODULATION

In practice, there is an unwanted unsuppressed ripple at double input frequency. Usually ignored, the ripple is still a serious disturbance in many applications. For example, for spectrum analysis by means of synchronous transducers of discrete operation, the double frequency ripple causes significant "subharmonic response" of the Harmonic Analyzer. Ripple must be suppressed to prevent sidebands from appearing on the VCO (Land, 1988).

Usually, when the loop is locked, some alternating drive signal v_c in the input of VCO causes some voltage v_d, which must compensate the influence of v_c. If the modulating frequency is too high, the compensating voltage v_d cannot pass the filter, and the alternating drive signal v_c causes the frequency modulating of the VCO. All other frequency components inside the filter bandwidth will be compensated by v_d and cause the phase modulation.

Next, let us investigate loop behaviour in the presence of modulating voltage $\cos(\omega_m t + \phi_m)$ in the input of VCO. For sinusoidal frequency modulation

$$u=U_o\cos[\omega_o t + \beta\sin(\omega_m t + \phi_m) + \phi_o], \qquad (1)$$

where ω_o is natural frequency of VCO,
$\beta=\Delta\omega_o/\omega_m$ is modulation index,
$\Delta\omega_o$ is peak frequency deviation,
ω_m is modulating frequency,
ϕ_m is phase angle of modulating voltage and
ϕ_o is phase angle of VCO frequency (Cartianu, 1964).

The modulated signal may be rewritten as

$$u=U_o \sum_{n=-\infty}^{+\infty} J_n(\beta)\cos[(\omega_o+n\omega_m\ t + n\phi_m + \phi_o], \qquad (2)$$

where $J_n(\beta)$ is the Bessel function of the first kind of order n (n=1,2,3,...) and it is equal to (Churchill, 1987)

$$J_n(\beta)= \sum_{k=o}^{+\infty} \frac{(-1)^k}{k!(n+k)!} \left(\frac{\beta}{2}\right)^{n+2k} \qquad (3)$$

and

$$J_{-n}(\beta)=(-1)^n J_n(\beta). \qquad (4)$$

Examining the expression (2), we note that there are all spectrum components of frequency modulated signal. Spectrum of signal (2) consists of the fundamental frequency ω_o, and the infinite number of sideband components placed symmetrically in pairs with regard to main frequency $\omega_o \pm n\omega_m$. According to (3) and (4), the lower and higher odd components are in paraphase. The peak value of the n-th component is $U_o J_n(\beta)$ and it is proportional to the value of the n-th order Bessel function $|J_n(\beta)|$ at modulation index β.

The value of Bessel functions is founded by summation (3). Increasing the modulation index β more members are taken into consideration to find the Bessel function value. In practice the modulation index is not very large, and for $\beta<0.2$ the first sum term is used only. For this situation, computing error is less than 1% for each order of Bessel function and (3) can be written

$$J_n(\beta)= \frac{1}{(n)!} \left(\frac{\beta}{2}\right)^n; \qquad (5)$$

thus

$$J_0(\beta)=1.0\ \beta^0$$
$$J_1(\beta)=0.5\ \beta^1$$
$$J_2(\beta)=0.125\ \beta^2 \qquad (6)$$
$$J_3(\beta)=0.0208\ \beta^3$$
$$J_4(\beta)=0.0026\ \beta^4$$
$$\text{----------}$$

That is, Bessel function $J_n(\beta)$ is a usual power function if $\beta<0.2$.

The output VCO signal in sophisticated nonsinusoidal modulating frequency can be written

$$u=U_0 \sum_{p,r,q,\ldots=-\infty}^{+\infty} J_p(\beta_1)\ J_r(\beta_2)\ J_q(\beta_3)\ldots \cos[(\omega_0+$$

$$+p\omega_{m1}+r\omega_{m2}+q\omega_{m3}+\ldots)t+p\phi_{m1}+r\phi_{m2}+q\phi_{m3}+$$

$$+\ldots+\phi_0], \qquad (7)$$

where ω_{m1}, ω_{m2}, ω_{m3},... are the sinusoidal signals of Fourier series terms. Examining expression (7), we note that here as in (2) in the modulated signal the components ω_0, $\omega_0\pm p\omega_{m1}$, $\omega_0\pm r\omega_{m2}$,... exist. In addition, here are the intermodulated components, which are combined frequencies between ω_0, $p\omega_{m1}$, $r\omega_{m2}$, $q\omega_{m3}$,... . The peak value of these components is proportional to the product of Bessel functions in simple signal, according to (2). At first it seems very capacious to compute all these components, but in reality the Bessel functions of higher order have neglible magnitude, and a large number of the sum terms may be ignored.

In many experiments, the output signal of phase-lock loop does not only appear at the fundamental excitation frequency, but bears a harmonic relationship to it. A classic frequency multiplying phase-lock loop is supposed to generate H equidistant output pulses during a period of the input signal (Fig.2). Applying this signal as a reference to the multiplier, we can design a Harmonic Analyzer.

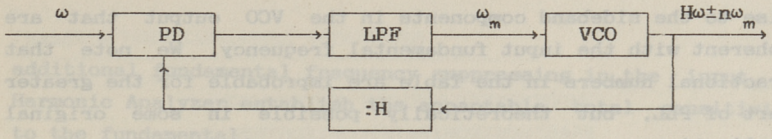


Fig. 2. Basic multiplying phase-lock loop.

Any modulating signal ω_m at the input VCO of multiplying phase-lock loop causes the frequency modulation of VCO

$$u=U_o \sum_{n=-\infty}^{+\infty} J_n(\beta)\cos[(H\omega + n\omega_m)t + n\phi_m + \phi]. \qquad (8)$$

Examination of the expression reveals that the VCO output signal consists of the fundamental term $H\omega$ and the sidebands components $H\omega \pm n\omega_m$ (n=1,2,3,...). The output of the Harmonic Analyzer is the sum of each individual term of the Fourier series multiplied by the input signal. If no higher harmonics, except fundamental, are present at the input, it is easy to show that the only multiplier product containing a d.c. component is the one associated with the sidebands term $H\omega - n\omega_m$ (n=1,2,3,...). All other products only contribute to high frequency ripple and are suppressed.

In order to produce a d.c. response, however, the signal must be coherent with one or more of the reference Fourier components:

$$\omega = H\omega - n\omega_m$$
$$-\omega = H\omega - n\omega_m$$

or

$$\frac{\omega_m}{\omega} = \frac{H-1}{n}$$
$$\frac{\omega_m}{\omega} = \frac{H+1}{n} \qquad (9)$$

We know that H and n are the integer, hence we can find the critical value of modulating signal ω_m for each H and n. The relationship between modulating frequency ω_m and fundamental ω (TABLE 1) determines the frequency ω_m giving

rise to the sideband components in the VCO output that are coherent with the input fundamental frequency. We note that fractional numbers in the Table are improbable for the greater part of PLL, but theoretically possible in some original applications.

TABLE 1 Critical Frequency Relationship ω_m/ω for the Harmonic Analyzer.

$$\frac{\omega_m}{\omega} = \frac{H+1}{n} \qquad \frac{\omega_m}{\omega} = \frac{H-1}{n}$$

n				H	n			
4	3	2	1		1	2	3	4
3/4	1	3/2	3	2	1	1/2	1/3	1/4
1	4/3	2	4	3	2	1	2/3	1/2
5/4	5/3	5/2	5	4	3	3/2	1	3/4
3/2	2	3	6	5	4	2	4/3	1
7/4	7/3	7/2	7	6	5	5/2	5/3	5/4
2	8/3	4	8	7	6	3	2	3/2
9/4	3	9/2	9	8	7	7/2	7/3	7/4

In a well-designed phase-lock loop with a true multiplier the casual signal does not occur in the input of VCO, there is only the residual PD ripple at double input frequency. For this situation $\omega_m/\omega=2$ and there are a few critical combinations in the Table. For a switching multiplier the modulating signal consists of even harmonics terms of Fourier series (numbers 2,4,6,... in Table). Examination of the Table reveals that the response to the fundamental, due to frequency modulation in the phase-lock loop, occurs only for the measurements of higher odd harmonics.

If we now consider the problem of measuring very small nonlinear distortion of sine wave, the measurement uncertainty is absolute, when the sensitivity to fundamental is comparable with the real amount of higher harmonic. To decrease the minimum detectable harmonic in practical systems, substantial effort is needed to eliminate the modulation. One way to suppress the modulating voltage is to use in the PLL a Sample and Hold (S/H) circuit, which operates at the double input frequency. The S/H circuit makes possible to reduce the unwanted ripple component in the input of VCO without superfluous decreasing the bandwidth of the loop filter. The

additional fundamental frequency suppressing in the input of Harmonic Analyzer establish the acceptable total sensitivity to the fundamental.

CONCLUSIONS

The phase-lock loop behaviour in presence of modulating voltage in the input of VCO has been described. Theoretical results have been complemented by simple examples to obtain a quantitative understanding of frequency modulating effects in phase-lock loops. The slightness of the modulating index in most applications of the PLL reduces the Bessel function to a usual power function and decreases the spectrum components computing capacity. The critical modulating signal frequencies, which determine the Harmonic Analyzer response to the fundamental have been presented. The Harmonic Analyzer with the true or switching multiplier has response to the fundamental only for the measurements of higher odd harmonics.

REFERENCES

Cartianu, Gh. (1964). Frequency Modulation. Meridiane, Bucharest, (in Russian).

Churchill, R. V. (1987). Fourier Series and Boundary Value Problems, 4th ed. McGraw Hill, New York.

Gardner, F. M. (1979). Phaselock Techniques, 2nd ed. Wiley, New York.

Land, R. (1988). Problems of Spectrum Analysis by Means of Synchronous Transducers of Discrete Operation. Trans. Tallinn Tech. Univ., 682, 57-62, (in Russian).

Meade, M. L. (1983). Lock-in Amplifiers: Principles and Applications, Peter Peregrinus Ltd, London.

R. Land

SAGEDUSMODULATSIOON AUTOMAATSES FAASISÜNKRONISATSIOONISÜSTEEMIS

Kokkuvõte

Automaatset faasisünkronisatsioonisüsteemi (AFSS) kasutatakse laialdaselt sünkroonmuundite ning nende baasil loodud harmooniliste analüsaatori juhtsignaalide formeerimiseks. Artiklis vaadeldakse perioodilisest signaalist põhjustatud parasiitset sagedusmodulatsiooni AFSS-is ja selle mõju sünkroonmuundite kvaliteedinäitajatele. Üldkujul on näidatud siinuselise ja mittesiinuselise moduleeriva signaali mõju väljundsignaali spektraalkoostisele ja spektrikomponentide arvutus Besseli funktsioonide abil. Analüüsist selgub, et reaalsetes AFSS-des väheneb arvutuste maht tunduvalt, sest väikeste modulatsiooniindeksite korral lihtsustuvad Besseli funktsioonid tavalisteks astmefunktsioonideks.

Harmooniliste analüsaatori kasutamisel kõrgemate harmooniliste mõõtmisel on oluline teada AFSS-is esineva moduleeriva sageduse väärtust. Nimelt tekib uuritava ja moduleeriva signaali sageduste teatud vahekordadel moduleeritud juhtsignaali külgribas spektraalkomponent, mille sagedus langeb kokku analüüsitava signaali põhiharmoonilise sagedusega, põhjustades sellega analüsaatori tundlikkuse põhiharmoonilisele. Artiklis toodud tabeli põhjal on võimalik vastavalt valitud sünkroonmuundi tüübile leida kriitiliste modulatsioonsageduste väärtused ning selle põhjal otsustada, millist tehnilist võtet kasutada nende modulatsioonsageduste kõrvaldamiseks AFSS-ist.

No 722
TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

O. Aarna

BALANCE MODELS FOR CONTINUOUS PROCESS PLANT STATE ESTIMATION

Abstract. This paper deals with a class of continuous process plant (CPP) models designed to meet the requirements of plant state estimation. CPP state estimation relies on the temporal and spatial redundancy of information, incorporated in the results of process measurements, in the measurement system topology, and in the mass, energy, and momentum balance equations. This redundancy is described mathematically by the plant state estimation balance models, relating the plant state to the measured variables. In this paper the genesis of two types of CPP balance models: state prediction and state estimation models is discussed. The relationships between stochastic systems with dynamic and static states, CPP state prediction models and three basic types of CPP state estimation models: static, quasi-stationary, and dynamic balance models are studied. A simple flow vessel model with one inlet and one outlet flow is used to illustrate the main concepts.

INTRODUCTION

Continuous process plants (chemical plants, power plants, etc.) comprise a variety of control problems for modern process control systems. The problem of plant state estimation is critical for advanced model-based and knowledge-based control concepts. This paper deals with a class of continuous process plant (CPP) models designed to meet the requirements of plant state estimation.

Mathematical formulation of the fundamental principles postulating the conservation of momentum, energy, and mass for any given system constitute the basis of their mathematical description (Stanislav, 1982). During the last two decades a large number of publications have appeared dealing with various aspects of using CPP models based on mass, energy, and momentum balance principles for process measurement reconciliation and rectification (Vaclavek, 1969; Mah, Stanley, and Downing, 1976; Aarna, 1978; Romagnoli and Stephanopoulos, 1980), gross error detection (Mah and Tamhane, 1982; Serth and Heenan, 1986), and plant diagnosis (Watanabe and

Himmelblau, 1982). All these techniques use the CPP state estimation as a primary tool.

CPP state estimation relies on the temporal and spatial redundancy of information, incorporated in the results of process measurements, in the measurement system topology, and in the mass, energy, and momentum balance equations. This redundancy is described mathematically by CPP state estimation balance models, relating the plant state to the measured variables. The CPP state estimation balance models fall into three groups: static, quasi-stationary, and dynamic, depending on the content of the state vector and the nature of state equations (Aarna, 1985). Most of the existing CPP state estimation applications are based on static balance models.

Until now little attention has been paid to the problems of CPP state estimation balance models genesis, typology, and comparative analysis (Aarna, 1985). In this paper the genesis of two types of CPP balance models: state prediction and state estimation models are discussed. The relationships between stochastic systems with dynamic and static states (Fathi, Ramirez, and Aarna, 1990), CPP state prediction models and three basic types of CPP state estimation models: static, quasi-stationary, and dynamic balance models are studied. A simple flow vessel model with one inlet and one outlet flow is used to illustrate the main concepts.

Throughout this paper bold lower-case letters denote vectors and bold upper-case letters denote matrices. The main variables are:

$\mathbf{q}$ - generalized flows,

$\mathbf{p}$ - parameters,

$\mathbf{u}$ - inputs,

$\mathbf{v}$ - measurement noise,

$\mathbf{w}$ - process noise,

$\mathbf{x}$ - system state,

$\mathbf{y}$ - outputs,

$\mathbf{z}$ - accumulations in the process elements.

Scalar entities are denoted by lower-case letters with or without superscripts.

MASS, ENERGY, AND MOMENTUM BALANCES IN CPP MODELING

A large variety of continuous processes is encountered in process industries. The common feature among these processes is that some single-phase or multiphase continuous medium is processed in a set of units (e.g., reactors, mixers, separators, distillation columns, etc.) connected together by mass and energy flows, thus constituting a CPP. From the control engineering point of view a CPP can be considered as consisting of a technological process itself and a measurement system providing temperature, flow, level, pressure, and other measurable signals.

If we divide a continuous process into a number of finite elements, each capable of accumulating mass, energy, and/or momentum, then, the entire process in terms of the inter-element flows, $q_o(t)$, can be modeled by a linear set of algebraic equations:

$$A_o q_o(t) = 0 \qquad (1)$$

where A_o is an incidence matrix of the process multigraph describing the adjacency of process elements and flows. The matrix A_o has dimensions (n_z n_{qo}), where n_z is the number of process elements, n_{qo} is the number of generalized flows (total mass, components, energy, and momentum flows), and $n_{qo} < n_z$.

Upon considering the integral relation between accumulations $z(t)$ (total amount of mass, components, energy, and momentum in the process elements) and accumulation flows $q_z(t)$ (a subvector of $q_o(t)$):

$$z(t) = z(t_o) + \int_{to}^{t} q_z(t)\, dt \qquad (2a)$$

or

$$q_z(t) = dz(t)/dt \qquad (2b)$$

and using the partitioning $A_o = (A\ I)$ and $q_o = (q^T\ q_z^T)^T$, the system (1) can be rearranged as

$$z(t) = Aq(t) \qquad (3)$$

where the flow vector $q(t)$ has dimension $n^q = n_{qo} - n_z$, and the incidence matrix A has dimensions $(n_z\ .\ n_q)$. The only difference between

balance models (1) and (3) is that the former describes the static (momentary) balance of generalized flows around the process elements while the latter expresses the dynamic balance of these flows over an infinite time interval. Both models are as exact as the process flow structure (matrix **A**). The continuous process balance models in the form (1) or (3) are quite useless and ineffective in control engineering applications. In the following, model transformations are represented using two different procedures.

The dynamic balances (3) can be transformed by substituting some or all components of the flow vector $q(t)$ by their models or constitutive equations:

$$q(t) = g[z(t),u(t),p(t)] \tag{4}$$

describing the cause - effect relations between the process flows $q(t)$, accumulations $z(t)$, inputs $u(t)$ (both controls and measurable disturbances), and parameters $p(t)$. Substituting (4) into (3) yields

$$\dot{z}(t) = f_c[z(t),u(t),p(t)] \tag{5}$$

an ordinary ***CPP state prediction model***. This name is due to the fact that the model (5) describes the causal relationship between the process states and its inputs, providing a means to predict the process state given an initial state $z(t_0)$. Optionally, the output equations

$$y(t) = h[z(t),p(t)] + v(t) \tag{6}$$

can be added to the model (5) and an additive process noise, $w(t)$, can also be associated with the state equations (5). Equations (5) and (6) comprise a standard continuous-time state-space model of a lumped-parameter control plant. Notice that the state vector of the model equations (5) and (6) consists of only accumulation-type process variables.

Another way of proceeding from the process model (3) is to ignore the cause - effect relationships among the components of vectors $z(t)$, $q(t)$, $u(t)$ and $p(t)$, and describe the dynamic behavior of the flow vector $q(t)$ by some black box model, e.g., random walk model:

$$q(t) = w_q(t) \tag{7}$$

where $w_q(t)$ is a n_q-dimensional random process with known stochastic properties. The system equations (3) and (7) plus the output equations give us the following process model:

$$z(t) = Aq(t) \tag{8a}$$

$$q(t) = w_{qc}(t) \tag{8b}$$

$$y(t) = h[z(t),q(t),p(t)] + v(t) \tag{8c}$$

which can be called a **CPP state estimation model** (Aarna, 1984). This model does not describe explicitly any dependence of the process states on its physical inputs and therefore has only a limited prediction power. However, it gives us a clear description of the relations between process states (accumulations $z(t)$ and generalized flows $q(t)$) and measured outputs (temperatures, flow rates, levels, pressures, concentrations, etc.). Notice that the dimension of the state vector in the process state estimation model (8), $(n_z + n_q)$, is higher than the number of states, n_z, in the process state prediction model (5) and (6).

In discrete-time framework, the process state prediction model has the form

$$z(t) = f[z(t-1),u(t-1),p(t-1)] + w(t) \tag{9a}$$

$$y(t) = h[z(t),p(t)] + v(t) \tag{9b}$$

and the process state estimation model is

$$z(t) = z(t-1) + s^{t-1}Aq(t-1) \tag{10a}$$

$$q(t) = q(t-1) + w_q(t-1) \tag{10b}$$

$$y(t) = h[z(t),q(t),p(t)] + v(t) \tag{10c}$$

where s^{t-1} is the sampling interval.

INTERPRETATION OF CPP BALANCE MODELS IN TERMS OF STOCHASTIC SYSTEMS WITH DYNAMIC AND STATIC STATES

As shown in (Fathi, Ramirez and Aarna, 1990) a stochastic discrete-time system with coupled dynamic and static models, and randomly varying parameters can be represented by the following model:

$$x_d(t) = f_d[t,x_d(t-1),x_s(t-1),p(t-1)] + w_{xd} \tag{11a}$$

$$0 = f_s[t,x_d(t),x_s(t),p(t)] + w_{xs} \tag{11b}$$

$$p(t) = p(t-1) + w_p \tag{11c}$$

$$y(t) = h[t,x_d(t),x_s(t),p(t)] + v \tag{11d}$$

where

x_d - system state with slow dynamics

x_s - system state with fast dynamics

In continuous process control applications, the plant model (11) is usually obtained via discretizing the corresponding continuous-time system model

$$\dot{x}_d(t) = f_{dc}[t,x_d(t),x_s(t),p(t)] + w_{xdc} \tag{12a}$$

$$0 = f_s[t,x_d(t),x_s(t),p(t)] + w_{xs} \tag{12b}$$

$$\dot{p}(t) = w_{pc} \tag{12c}$$

$$y(t) = h[t,x_d(t),x_s(t),p(t)] + v \tag{12d}$$

where

$$f_d[t,x_d(t-1),x_s(t-1),p(t-1)] = x_d(t-1) + \int_{t-1}^{t} f_{dc}[r,x_d(r),x_s(r),p(r)]dr \tag{13a}$$

$$w_{xd} = \int_{t-1}^{t} w_{sdc}(r)dr \tag{13b}$$

$$w_p = \int_{t-1}^{t} w_{pc}(r)dr \tag{13c}$$

The majority of continuous-time process models (12) are stationary, i.e., the vector-functions $f_{dc}[.]$,$f_s[.]$, and $h[.]$ do not depend explicitly on time. Even in these cases, the discrete-time model (11) can be nonstationary.

The main interpretation of the state variables, $x_d(t)$ and $x_s(t)$, in terms of the CPP models, is that $x_d(t)$ corresponds to the slowly-varying process states, and $x^s(t)$ - to the fast-responding process states.

In this case, it is quite natural and essential to regard the static state equations

$$0 = f_s[t,x_d(t),x_s(t),p(t)] + w_{xs} \tag{14}$$

as a limiting case of the corresponding continuous-time dynamic state equations

$$\dot{x}_s(t) = f_s[t,x_d(t),x_s(t),p(t)] + w_{xs} \tag{15}$$

where $\dot{x}_s(t) = 0$ or as a limiting case of the corresponding discrete-time dynamic state equations

$$x_s(t) = x_s(t-1) + f_s[t,x_d(t-1),x_s(t-1),p(t-1)] + w_x \tag{16}$$

where $x_s(t) = x_s(t-1)$. Thus, the main interpretation is that the model (11) is a stochastic system with two distinctly separated groups of modes (slow and fast) and randomly varying parameters.

Another way to interpret the set of static state equations (14) is that some additional algebraic or transcendental relations are imposed upon the dynamic state vector $x_d(t)$ and the parameters $p(t)$ such as the constitutive equations (4) (deterministic flow models). In many applications, equation (14) in its explicit form,

$$x_s(t) = f_s[t,x_d(t),x_s(t),p(t)] + w_{xs} \tag{17}$$

simply defines some new entities of interest, $x_s(t)$, as functions of $x_d(t)$, $x_s(t)$, and $p(t)$. Evidently, in this case, $x_s(t)$ can be called the static state vector only conditionally.

Now let the system model be described by the following equations

$$x_d(t) = A[x_d(t-1),x_s(t-1),p(t-1)]x_d(t-1) + B[x_d(t-1),x_s(t-1),p(t-1)]x_s(t-1) + w_{xd} \tag{18a}$$

$$0 = f_s[x_d(t),x_s(t),p(t)] + w_{xs} \tag{18b}$$

$$p(t) = p(t-1) + w_p \tag{18c}$$

$$y(t) = h[x_d(t),x_s(t),p(t)] + v \tag{18d}$$

where $A[.]$ and $B[.]$ are functional matrices of the proper dimensions. The system (18) can be readily interpreted in terms of the CPP mass, energy, and momentum balance models.

By a proper change of dynamic and static state coordinates

$$x'_d(t) = R[x_d(t), x_s(t), p(t)]x_d(t) \quad (19a)$$

$$x'_s(t) = S[x_d(t), x_s(t), p(t)]x_s(t) \quad (19b)$$

where det $R[.] = 0$ and det $S[.] = 0$, the process model (18) can be transformed into

$$x'_d(t) = x'_d(t-1) + B'(t-1)x'_s(t-1) + w'_{xd} \quad (20a)$$

$$0 = f'_s[x'_d(t), x'_s(t), p(t)] + w'_{xs} \quad (20b)$$

$$p(t) = p(t-1) + w_p \quad (20c)$$

$$y(t) = h'[x'_d(t), x'_s(t), p(t)] + v \quad (20d)$$

where

$$B'(t-1) = s^{t-1}A \quad (21)$$

The transformation (19) has the following semantics. The dynamic and static state vectors of the initial quasi-linear process model (18) contain both intensive and extensive process thermodynamic state variables. By using equation (19), we transform the mixed system of process state variables into exceptionally extensive system of state variables, i.e., $x'_d(t) = z(t)$ accumulated amounts of total mass, components, energy, and momentum in the process elements, and $x'_s(t) = q(t)$ generalized flows of total mass, components, energy, and momentum.

Now let the dynamic state equations be

$$z(t) = z(t-1) + s^{t-1}Aq(t-1) \quad (22a)$$

$$q(t) = q(t-1) + w_q \quad (22b)$$

and define

$$x_d(t) = [z^T(t) \; q^T(t)]^T, \quad w_{xd} = [0^T \; w_q^T]^T$$

Further, suppose that the static state equations (14) or (17) define:

- some dependent generalized flows $x_{sd}(t)$,
- some additional process variables of interest $x_{sv}(t)$, related to $q(t)$, $z(t)$, and $p(t)$,
- stationary mass energy, and momentum balances ($q_s(t) = 0$) in some process elements.

Then, the set of equations

$$x_d(t) = G(t-1)x_d(t-1) + w_{xd} \tag{23a}$$

$$0 = f_s[x_d(t),x_s(t),p(t)] + w_{xs} \tag{23b}$$

$$p(t) = p(t-1) + w_p \tag{23c}$$

$$y(t) = h[x_d(t),x_s(t),p(t)] + v \tag{23d}$$

where

$$G(t-1) = \begin{matrix} I_x & S^{t-1}A \\ 0 & I_q \end{matrix} \qquad x_s(t) = \begin{matrix} x_{sf}(t) \\ x_{sv}(t) \end{matrix}$$

represents a CPP ***dynamic balance model*** with partially stationary process elements and known flow models. If all process elements are stationary, then we obtain a ***quasi-stationary balance model*** with given flow models as

$$q(t) = q(t-1) + w_q \tag{24a}$$

$$0 = f_s[q(t),x_s(t),p(t)] + w_{xs} \tag{24b}$$

$$p(t) = p(t-1) + w_p \tag{24c}$$

$$y(t) = h[q(t),x_s(t),p(t)] + v \tag{24d}$$

where the vector-function $f_s[.]$ also contains the stationary balance equations

$$Aq(t) = 0 \tag{25}$$

Static balance models are a special case of CPP model (24) without time-varying generalized flows and parameters:

$$0 = f_s[q(t),x_s(t),p(t)] + w_{xs} \tag{26a}$$

$$y(t) = h[q(t),x_s(t),p(t)] + v \tag{26b}$$

Finally, the process state estimation model (10) can be interpreted as a special case of the general stochastic system (11), where $x_d(t) = z(t)$, $p(t) = q(t)$, and the static state vector is absent. The above discussion shows that the stochastic system model (11) has a wide variety of interpretations and potential applications in the field of CPP and parameter estimation.

EXAMPLE

As a simple example of different state estimation balance models, we consider a vertical cylindric vessel with one inlet liquid flow, $q^1(t)$, and one outlet flow, $q^2(t)$. The outlet flow from the vessel is governed by the liquid hydrostatic pressure. The measured variables are the liquid level in the vessel, $y^1(t)$, and the inlet flow rate, $y^2(t)$. The following gives dimensionless process state estimation models in the discrete-time form for a sampling interval of one ($s = 1$).

The total mass balance around the vessel is described by

$$q^a(t) - q^1(t) + q^2(t) = 0 \tag{27}$$

Taking into consideration that

$$q^a(t) = z(t+1) - z(t) \tag{28}$$

the momentary (static) total mass balance (27) can be expressed in a dynamic balance form

$$z(t) = z(t-1) + q^1(t-1) - q^2(t-1) \tag{29}$$

where z - total accumulated liquid mass in the vessel. The outlet mass flow rate is related to the liquid mass in the vessel by

$$q^2(t) = a\ z(t) + w^{q2} \tag{30}$$

where a - a known parameter determined by the liquid viscosity and the outlet pipe geometry.

According to the above assumptions we can derive two static balance models for this sample problem: without deterministic flow models and with known outlet flow model (30). In the first case the flow vessel static balance model is

$$q^a(t) - q^1(t) + q^2(t) = 0 \tag{31a}$$

$$y^1(t) = c^1 q^a(t) + v^1 \tag{31b}$$

$$y^2(t) = c^2 q^1(t) + v^2 \tag{31c}$$

where y^1 = measured accumulation flow rate as a difference of two consecutive level measurements ($y^{1'}(t) = y^1(t) - y^1(t-1)$), c^1 = a known

parameter converting accumulated mass to liquid level, c^2 = a known parameter converting inlet mass flow rate to measurable volumetric flow rate. The flow vessel static balance model with known outlet flow model can be expressed as

$$q^a(t) - q^1(t) + q^2(t) = 0 \qquad (32a)$$

$$q^2(t) = a\ z(t) + w^{q2} \qquad (32b)$$

$$y^1(t) = c^1 q^a(t) + v^1 \qquad (32c)$$

$$y^1(t) = c^1 z(t) + v^1 \qquad (32d)$$

$$y^2(t) = c^2 q^1(t) + v^2 \qquad (32e)$$

Notice that the model (31) can be used to estimate the flows $q^a(t)$, $q^1(t)$ and $q^2(t)$ while the model (32) contains also an accumulation variable $z(t)$. The models (31) and (32) are specific cases of a general static balance model (26).

With the given set of measurable parameters the flow vessel static balance models do not contain no spatial redundancy of information and therefore the state estimation reduces to straightforward sequential use of the model equations, e.g.

$$q^a(t) = y^1(t)/c^1 \qquad (33a)$$

$$q^1(t) = y^2(t)/c^2 \qquad (33b)$$

$$q^2(t) = q^1(t) - q^a(t) \qquad (33c)$$

where $q^a(t)$ and $q^1(t)$ are selected as the independent flows and $q^2(t)$ is the dependent flow. The same is valid for the corresponding quasi-stationary balance models, e.g.

$$q^a(t) = q^a(t-1) + w^{qa} \qquad (34a)$$

$$q^1(t) = q^1(t-1) + w^{q1} \qquad (34b)$$

$$y^1(t) = c^1 q^a(t) + v^1 \qquad (34c)$$

$$y^2(t) = c^2 q^1(t) + v^2 \qquad (34d)$$

$$q^2(t) = q^a(t) + q^1(t) \qquad (34e)$$

which is a quasi-stationary counterpart of the static balance model (31). The model (34) consists of two independent first-order models

$$q^i(t) = q^i(t-1) + w^{qi} \tag{35a}$$

$$y^i(t) = c^i q^i(t) + v^i \tag{35b}$$

for two independent flows, $q^a(t)$, and $q^1(t)$, which can be estimated separately as simple time-series using the first-order Kalman filter, and a static balance model (34c) used to estimate the dependent flow, $q^1(t)$. This means that our quasi-stationary sample model contains only temporal redundancy of information.

The flow vessel dynamic balance model can be expressed as

$$z(t) = z(t-1) + q^1(t-1) - q^2(t-1) \tag{36a}$$

$$q^1(t) = q^1(t-1) + w^{q1} \tag{36b}$$

$$q^2(t) = q^2(t-1) + w^{q2} \tag{36c}$$

$$y^1(t) = c^1 z(t) + v^1 \tag{36d}$$

$$y^2(t) = c^2 q^2(t) + v^2 \tag{36e}$$

In the dynamic balance model with known deterministic outlet flow model the equation (36c) is replaced by the equation (30). As shown in the previous section, the state estimation model (36) with known deterministic flow model can be interpreted as a special case of a stochastic system with dynamic and static states, and randomly varying parameters where $x_d = z$, $x_s = q^2$, $p = q^1$. Notice that the dynamic state equation (36a) does not contain any uncertainty due to exactness of the mass balance. Fathi, Ramirez, and Aarna (1990) show that an Extended Kalman Filter type algorithm can be effectively used to estimate the state and parameters of this kind of systems.

CONCLUSIONS

The main result of this paper is that all the known CPP state estimation balance models can be treated as special cases of a stochastic system with

dynamic and static states, and randomly varying parameters. As a generalization of existing static, quasi-stationary, and dynamic balance models the corresponding CPP state estimation models with partially given deterministic flow models appear as an intermediate class of models between the "pure" state prediction and state estimation models. A more detailed study of these models and their state estimation technique is the aim of our further research.

REFERENCES

Aarna, O. (1978). Dynamic chemical plant model and its application. *Preprints of the 7th Triennial IFAC World Congress*, Vol.2, Helsinki, Finland, 279-286.

Aarna, O.A. (1984) Balance models for evaluating the state of chemical-engineering systems. *Automation and Remote_Control* 45 (5), Part 2, 655-662.

Aarna, O. (1985). Chemical plant state estimation. *Transactions of Tallinn Technical University* 592, 17-38 (in Russian).

Fathi, Z., W.F.Ramirez, and O.Aarna (1989). Joint state and parameter estimation/identification for systems with coupled static and dynamic models. *Report V. University of Colorado, Department of Chemical Engineering*, 88pp.

Mah, R.S.H., Stanley G.M., and D.M.Downing (1976). Reconciliation and rectification of process flow and inventory data. *Ind. Eng. Chem. Process Des. Dev.* 15, 175-183.

Mah, R.S.H. and A.C.Tamhane (1982). Detection of gross errors in process data. *AIChE Journal* 28, 828-830.

Romagnoli, J.A., and G.Stephanopoulos (1980). On the rectification of measurement errors for complex chemical plants. *Chem. Eng. Sci.* 35, 1067-1081.

Serth, R.W., and W.A.Heenan (1986). Gross error detection and data reconciliation in steam-metering systems. *AIChE Journal* 32, 733-742.

Stanislav, J.F. (1982). *Mathematical Modeling of Transport Phenomena Processes.* Ann Arbor Science Publishers, Ann Arbor, Michigan.

Stanley, G.M. and R.S.H.Mah (1977). Estimation of flows and temperatures in process networks. *AIChE Journal* 23 (5), 642-650.

Vaclavek, V. (1969). Studies on systems engineering II.- On the application of the calculus of observations in calculations of chemical engineering balances. *Coll. Czechoslov. Chem. Commun.* 34, 364-372.

Watanabe, K., and D.M.Himmelblau (1982). Instrument fault detection in a system with uncertainties. *Int. J. System Sci.* 13, 137-158.

O.Aarna

Pidevate tehnoloogiliste protsesside oleku hindamise bilansimudelid

Kokkuvõte. Artiklis käsitletakse pidevate tehnoloogiliste protsesside (PTP) oleku hindamiseks mõeldud mudelite klassi. PTP oleku hindamine põhineb informatsiooni ajalisel ja ruumilisel liiasusel, mis sisaldub tehnoloogiliste mõõtmiste tulemustes, mõõtesüsteemi topoloogias ja massi, energia ning liikumishulga bilansi võrrandites. Seda liiasust kirjeldatakse matemaatiliselt protsessi oleku hindamise bilansimudelitega, mis seovad protsessi olekut ja tehnoloogiliste mõõtmiste tulemusi. Artiklis analüüsitakse PTP bilansimudelite kahe tüübi -oleku prognoosi ja oleku hindamise mudelite geneesi. On uuritud seoseid dünaamiliste ja staatiliste olekumuutujatega stohhastiliste süsteemide, PTP oleku prognoosi mudelite ja kolme oleku hindamise mudelite klassi (staatiliste, kvaasistatsionaarsete ja dünaamiliste mudelite) vahel. Vaadeldavaid põhimõisteid ja mudeleid illustreeritakse lihtsa, ühe siseneva ja ühe väljuva vooga anuma kui PTP näitel.

No 722

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

A. Kiitam

ON THE SELECTION OF THE UTILITY FUNCTION FOR PROCESS ADJUSTMENT

Abstract. Some aspects concerning the application of different utility functions for process adjustment are discussed. It is argued that in case the specification limits are vaguely defined, the conventional utility functions based on yield or mean square error criteria might not be the best choice. Some alternative utility functions are considered, including dispersed utility functions, which are based on additive stochastic dependencies between quality parameters.

INTRODUCTION

A crucial task in any quality assurance program is the formulation of an adequate utility function, which describes the requirements to the product quality parameters in a proper way. To select the utility function, one has to evaluate customer needs and expectations, anticipated revenue from selling the manufactured products, and sometimes the computational complexity of the corresponding optimal adjustment algorithm. Thus, the proper formulation of the utility function can be a complicated task, and different aspects of a specific situation must be taken into account. In this paper we will discuss some aspects of the selection of the utility function, primarily for the manufacturing process adjustment (however, some considerations are also applicable for statistical design centering). We will concentrate on the "nominal-the-best" situation for the case when the product quality is essentially determined by a one-dimensional parameter. Let this quality parameter be y and the utility

function be g(y). The use of the utility function g(y) for process adjustment leads to the optimal adjustment problem

$$\int_R g(y)f(y,X)dx \to \max_x , \qquad (1)$$

where f(y,X) is the distribution density function for a random variable y which depends on the adjustment parameter vector X. Usually in (1) it is suitable to use process centering or shift model f(y,X)=f(y-t), t=t(X), i.e. the influence of the adjustment parameters X is representable through the shift t. Then we obtain the optimum shift problem

$$\int_R g(y)f(y-t)dy \to \max_t , \qquad (2)$$

The solution of problem (2) gives the optimal shift value t^*, which implies the best process adjustment for given utility function g(y). As y-t=e can be interpreted as process noise, it is natural to assume zero mean M(e)=0 for the density function f(e).

We will first discuss the use of the well-known yield and mean square error criteria. Second, we will list some alternatives and third, consider the use of so-called dispersed utility functions.

YIELD VERSUS MEAN SQUARE ERROR

The two widely used criteria for process adjustment are: (a) yield maximization; (b) mean square error minimization (Kapur and Wang,1987).

The use of parametric field as an adjustment criterion is based on the concept of conformity to specifications. Products are classified into two categories, as shown in Fig.1,a. Items that conform to the specification limits are acceptable and items outside the specification limits are not acceptable. Thus, the quality evaluation has binary nature. All the items whose quality parameter y lies between the lower specification

limit A and the upper specification limit B are estimated equally good giving some income I. Unacceptable items are estimated all equally bad resulting in some loss L. For the solution of the optimum shift problem (2) we can fix the values I=1 and L=0. Thus, the yield criterion leads to the utility function , shown in Fig.1,b:

$$g(y)=\begin{cases}1, & A \le y \le B,\\ 0, & \text{otherwise.}\end{cases} \qquad (3)$$

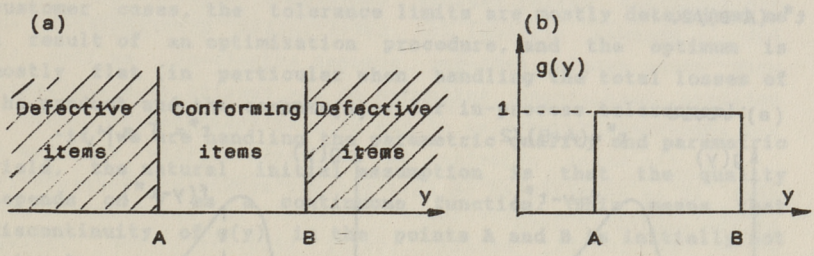


Fig.1. Yield criterion.

The use of mean square error (MSE) criterion is based on the concept of some "nominal-the -best" value, which we denote as target value T. The products with y=T have the best quality , and if y moves away from T, then the quality is not so good. The loss in quality is described by a quadratic loss function. Thus, as shown in Fig.2, the utility function is

$$g(y)= -(y-T)^2. \qquad (4)$$

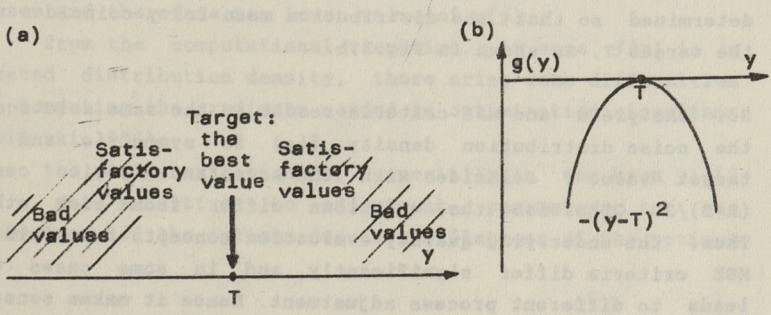


Fig.2. Mean square error criterion.

The use of different utility functions (3) and(4) in general leads to different solutions for the optimum shift problem (2). The use of yield criterion with the utility function (3) leads to the equation

$$f(A-t^*) = f(B-t^*),$$

whose solution t^* is the optimal shift. Geometrically it implies the equal density condition, as shown in Fig.3. For symmetrical distributions with $f(-e)=f(e)$ we obtain $t^*=(A+B)/2$.

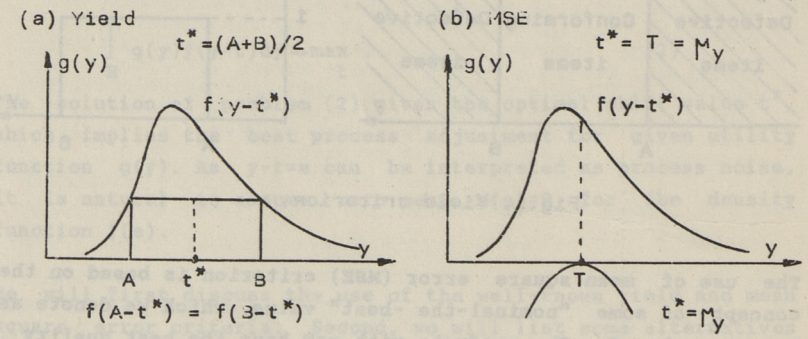


Fig.3. Optimal shift for yield and MSE criteria.

The use of the MSE criterion with the utility function (4) leads to the solution $t^*=T$. This implies that the shift is determined so that the distribution mean for y coincides with the target T, as shown in Fig.3,b.

So, the yield and MSE criteria result in the same solution if the noise distribution density $f(e)$ is symmetric and the target value T coincides with the acceptance region center $(A+B)/2$. Otherwise, the solutions differ from each other. Thus, the underlying quality evaluation concepts for yield and MSE criteria differ significantly, and in some cases this leads to different process adjustment. Hence it makes sense to discuss in some details the objections related to the application of those quality criteria.

The main objections to the application of the yield criterion may be listed as follows:

-for the yield criterion it is assumed that the acceptability region is uniquely defined and fixed. This assumption might be incorrect, as often the tolerance limits are vaguely defined or negotiable: they can be more tight or more relaxed for different customers or different price/cost situations. If we omit the overall standards and severe customer cases, the tolerance limits are mostly determined as a result of an optimization procedure, and the optimum is mostly flat (in particular when handling the total losses of the product and the consumer, or for in-process tolerances);

-if we are handling the parametric quality and parametric yield, the natural initial assumption is that the quality depends on y as a continuous function. This means that discontinuity of g(y) in the points A and B is initially not natural;

- for the "nominal-the-best" situations it might be inadequate to use the same value of g(y) within the tolerance (A,B), since sometimes the products from the central part of the tolerance can be sold with a higher price due to their superior or more guaranteed quality, and the inferior products farther off from the center can be sold with a lower price for less critical applications. For in-process adjustment problems, the items outside the tolerance limits can be sometimes made acceptable by applying some additional corrective processing or trim, which requires additional expenses and therefore has lower value of g(y);

- from the computational viewpoint, in case f(y) is a truncated distribution density, there arise some difficulties when computing the gradients needed in optimization algorithms (Styblinski,1986);

- the yield criterion is, especially in the high yield cases, not sensitive to adjustment parameters in the neighborhood of the optimum due to the flatness of the optimum curve.

The objections to the application of the MSE criterion may be listed as follows:

- the main objection is that the use of the utility function $-(y-T)^2$ implies too large losses when y differs from T significantly, as $g(y) \to -\infty$ if $y \to \pm\infty$. This is not natural from the economic point of view, since the fabrication of a defective product does not imply infinite losses: if the defective device is recognized and rejected, then the losses are determined by a finite cost of product fabrication. This aspect could be not critical when operating with theoretical distributions having rapidly converging distribution tails. But it becomes critical for heavy-tailed distributions and when using sample data for process adjustment, as then the outliers have strong influence on the results of optimization and robustness properties of the utility function are needed;

- the minor objection to quadratic utility function is that it is complicated to have a clear economic interpretation of the quadratic loss.

The above considerations permit to conclude that in some cases, primarily due to the vagueness of quality specifications or economic aspects, yield and MSE are not satisfactory criteria for process adjustment. If so, alternatives are needed.

SOME ALTERNATIVES

As related to the above discussion, we will now list some recent alternative formulations of utility functions, which take into account more economic and other details.

First, let us mention the limited square loss approach (Taguchi, 1986; Adams and Woodall, 1989). The corresponding utility function can be presented as shown in Fig.4:

$$g(y) = \begin{cases} L-(L/W^2)/(y-T)^2, & T-W \le y \le T+W \\ 0, & \text{otherwise,} \end{cases} \qquad (5)$$

where L is the loss for the manufacturer due to the production of nonconforming items and W is the tolerance width. Such an

utility function eliminates the complications related to the infinity of losses when $y \to \pm\infty$.

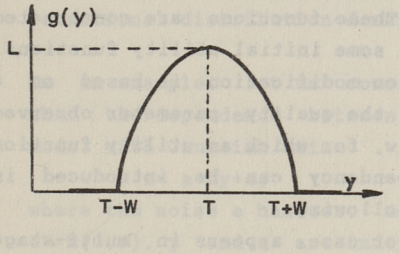


Fig.4. Limited square utility function.

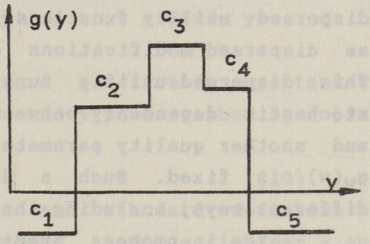


Fig.5. Piecewise constant utility function.

Second, the differentiated quality groups approach, which has been considered by Abramov, Bernatskii and Zdor (1982), Opalski and Styblinski (1986), Kiitam (1986b) and other authors. The products are divided into several quality groups characterized by different prices c_i which are usually higher in the neighborhood of the target value. This leads to piecewise constant utility function, as shown in Fig.5. The approach enables to evaluate better the economic results and is due to its higher sensitivity especially useful for high yield situations.

Third, the signal-to-noise (S/N) ratios as performance criteria are considered by Taguchi (1986) and other authors. For "nominal-the-best" situation S/N ratio SN_T is recommended:

$$SN_T = -10\log_{10}\gamma^2 = -10\log_{10}(\mu^2/\sigma^2), \qquad (6)$$

where γ is the variation coefficient as the ratio of standard deviation σ to mean μ. The motivation for the use of S/N ratio is caused by the dependency between μ and σ, which occurs in most cases. Correspondingly, the efficient use of S/N ratio is based on an adequate process model (León, Shoemaker and Kackar, 1987; Box, 1988) and thus requires modeling analysis for a specific process.

DISPERSED UTILITY FUNCTIONS

Let us now consider a class of utility functions which we call dispersed utility functions. These functions are constructed as dispersed modifications of some initial utility functions. This dispersed utility function modification is based on a stochastic dependency between the quality parameter observed and another quality parameter w, for which an utility function $g_w(w)$ is fixed. Such a dependency can be introduced in different ways, including the following:

- the in-process predictor case appears in multi-stage manufacturing processes when in-process quality testing is applied. In this case w is a quality parameter for final product and y is a quality parameter observed after a manufacturing stage; y is used as an in-process test variable to predict the final quality parameter w (Kiitam, 1985a). The dependency between y and w is described by a model

$$w=f(y)+e, \qquad (7)$$

where f(y) is a fixed function and e denotes the noise introduced in the intermediate manufacturing stages;

- the measurement error case appears when the variable w is observed under the influence of the measurement error e which has considerable dispersion (Sauer and Hoffmann, 1987). The corresponding dependency model becomes

$$y=w+e, \qquad (8)$$

where w and e are independent stochastic variables;

- the indirect measurement case leads to the above model (7), where y is the initial quality parameter which is tightly (with high correlation) related to y and which is much easier to measure than y;

-the computational perturbation case appears when one smoothes the initial nondifferentiable optimization problem by introducing an additional stochastic variable, in order to get a problem with continuous gradients (Tang and Styblinski, 1988). This technique also leads to the model (8).

In the above cases the determination of the dispersed utility function g(y) for y leads to the computation of an expectation which is expressed as a convolution (Kiitam, 1986a):

$$g_y(y) = M(g_w(w)|y) = \int_R g_w(w) f(w|y)\,dw, \qquad (9)$$

where $g_w(w)$ is the utility function for w and $f(w|y)$ is the conditional distribution density for w at a given y.

As an example, let us consider a practically important case when the dependence between w and y is described by a linear model with Gaussian noise

$$w = a+by+e, \qquad (10)$$

where the noise e has zero mean and standard deviation σ, $e \sim N(0,\sigma)$, y and e are independent random variables, a and b are model coefficients. Then the utility function $g_y(y)$ is expressed as

$$g_y(y) = (2\pi)^{-1/2}\sigma^{-1} \int_R g_w(w) \exp[-(w-a-by)^2/(2\sigma^2)]\,dw \qquad (11)$$

This formula gives $g_y(y)$ as a dispersed modification of $g_w(w)$. For instance, if $g_w(w)$ is the tolerance function corresponding to the yield criterion, i.e. $g_w(w)=1$ if $A_W \le w \le B_W$ and $g_w(w)=0$ otherwise, we obtain

$$g_y(y) = N_o((B_w-a-by)/\sigma) - N_o((A_w-a-by)/\sigma), \qquad (12)$$

where $N_o(x) = (2\pi)^{-1/2} \int_{-\infty}^{x} \exp(-t^2/2)dt$.

The hat-shaped utility function (12) is shown in Fig.6. When σ increases, the utility function $g_y(y)$ becomes more dispersed and closer to the normal distribution density curve. The normalized asymptotic shape of (11) does not depend on $g_w(y)$ and coincides with the normalized normal distribution density curve $n_o(y) = (2\pi)^{-1/2} \exp(-y^2/2)$.

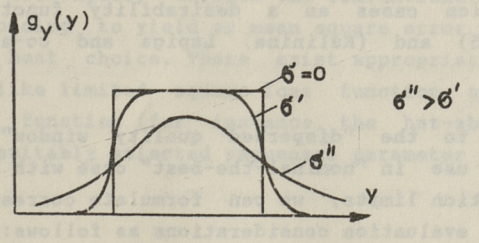


Fig.6. Dispersed utility function (12).

In Fig.7 two more examples of dispersed utility functions are shown. The first one corresponds to the tolerance type $g_w(w)$ and uniform noise distribution, the second one to the limited square type $g_w(w)$ and normal noise distribution.

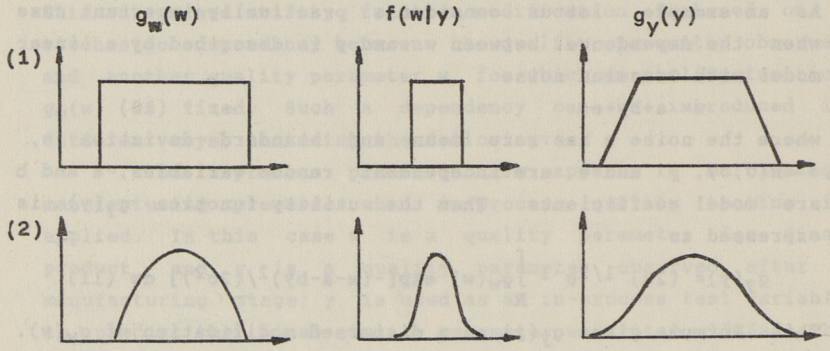


Fig.7. Two examples of dispersed utility function.

The dispersed utility functions are applicable not only in the most usual case when the utility function g(y) describes the profit or revenue effect as a function of the quality parameter y. They are also applicable in case when interpreting y as a fuzzy quality parameter (Glazunov and Lapidus, 1989). In this and other cases σ serves as a fuzziness or vagueness parameter: when σ increases, y as a quality parameter becomes more vague. The above asymptotic case $\sigma \to \infty$, which leads to the normal density curve $n_{\sigma}(y)$ as a normalized utility function, has been recommended in complex quality description cases as a desirability function in (Harrington, 1965) and (Kalinina, Lapiga and co-authors, 1989).

As a background to the "dispersed quality window" based utility functions use in "nominal-the-best" case with vaguely defined specification limits, we can formulate corresponding parametric quality evaluation considerations as follows:

1) it is natural to evaluate the quality as the highest at the target/nominal value;

2) the quality is decreasing when moving away from the target value, mostly due to some smooth continuous curve which is close to symmetrical;

3) as the fabrication expenses are finite for non-conforming product items, $g(y)$ has plateau-like asymptotes, i.e. $g(y) \to c = \text{const}$ when $y \to \pm\infty$.

When comparing the utility functions mentioned above from the computational point of view, the difference depends on specific situation. For instance, the difference between the yield and differentiated quality groups is computationally not significant (Opalski and Styblinski, 1986). The computation of dispersed quality functions is more complicated than the computation of yield, but for the computation of the corresponding gradients needed in optimal adjustment algorithms the situation can be vice versa (Tang and Styblinski, 1988). For analytical operations, the simplest cases are MSE and yield, and the greatest complications arise with dispersed quality functions, especially when their additional truncation is necessary.

CONCLUSION

In this paper we have discussed some aspects concerning the utility function selection for process adjustment. The discussion permits to conclude that in cases the specification limits are vaguely defined the conventional utility functions corresponding to yield or mean square error criteria might not be the best choice. There exist appropriate alternatives for them, like limited square loss function or some dispersed utility function (for instance, the hat-shaped function (12) with a suitably selected vagueness parameter σ).

REFERENCES

Abramov, O. V., F. I. Bernatskii, V. V. Zdor (1982). Parametric Correction of Control Systems. Energoizdat, Moscow (in Russian).

Adams, B. M., W. H. Woodall (1989). An Analysis of Taguchi's On-Line Process-Control Procedure Under a Random-Walk Model. Technometrics, v.30, no.4, pp.401-413.

Box, G. E. P. (1988). Signal-to-Noise Ratios, Performance Criteria, and Transformations (with Discussion and Reply). Technometrics, v.30, no.1, pp.1-40.

Glazunov, A. V., V. A. Lapidus (1989). Flexible Quality Control Methods for Fuzzy Quality Requirements. Zavodskaya Laboratorija, no.3, pp.85-89 (in Russian).

Harrington, E. C. (1965). The Desirable Function. Industrial Quality Control, v.21, no.10, pp.124-131.

Kalinina, E. V., A. G. Lapiga et al. (1989). Quality Optimization. Complex Products and Processes. Himiya, Moscow (in Russian).

Kapur, K. O., C. J. Wang (1987). Economic Design of Specifications Based on Taguchi's Concept of Quality Loss Functions. Quality: Design, Planning and Control (Winter Annual Meeting of ASME, December 1987), pp.23-36.

Kiitam, A. (1986a). Synthesis of Scalar Utility Functions for Outputs of Stages of Multistage Technological Processes. Trans. Tallinn Techn. Univ. no.629, pp.45-50 (in Russian).

Kiitam, A. (1986b). Algorithmization of Some Scalar Optimal Adjustment Tasks for Discrete Technological Processes. Trans. Tallinn Techn. Univ., no.629, pp.51-68 (in Russian).

León, R. V., A. C. Shoemaker, R. N. Kackar (1987). Performance Measures Independent of Adjustment. Technometrics, v.29, no.3, pp.253-265.

Opalski, L. J., M. A. Styblinski (1986). Generalization of Yield Optimization Problem: Maximum Income Approach. IEEE Trans. CAD of Integr. Circ. and Syst., v.5, no.2, pp.346-360.

Sauer, W., W. Hoffmann (1987). Influence of Measurement Errors on Test Sensitivity. Wiss. Zeitschrift der Techn. Univ. Dresden, Jg.36, H.4, S.67-71 (in German).

Styblinski, M. A. (1986). Problems of Yield Gradient Estimation for Truncated Probability Density Functions. IEEE Trans. CAD of Integr. Circ. and Syst., v.5, no. 1, pp.30-38.

Taguchi, G. (1986). Introduction to Quality Engineering: Designing Quality Into Products and Processes. White Plains, NY, Kraus Internat. Publications.

Tang, T.-S., M. A. Styblinski (1988). Yield Optimization for Nondifferentiable Density Functions Using Convolution Techniques. IEEE Trans. CAD of Integr. Circ. and Syst., v.7, no.10, pp.1053-1067.

A. Kiitam

KASULIKKUSFUNKTSIOONI VALIKUST PROTSESSIHÄÄLESTUSEL

Kokkuvôte

On vaadeldud môningaid erinevate kasulikkusfunktsioonide rakendamise aspekte protsessihäälestusel. Väidetakse, et kui tolerantsipiirid on määratletud ebamääraselt, ei pruugi tavalised saagise vôi ruutkeskmise vea kriteeriumil pôhinevad kasulikkusfunktsioonid olla sobivaimad. On vaadeldud môningaid alternatiivseid kasulikkusfunktsioone, sealhulgas hajusaid, mis pôhinevad aditiivsetel stohhastiilistel sôltuvustel kvaliteediparameetrite vahel.

No 722

TALLINNA TEHNIKAÜLIKOOLI TOIMETISED

TRANSACTIONS OF TALLINN TECHNICAL UNIVERSITY

W. Kracht

TRUTH VALUED COMPUTING PROCESSES AND PROCESS CALCULUS: A FORMALISM FOR DESCRIBING PROGRAMMING LOGIC

Abstract. A new approach to describe programming logic and the development of high level programming languages is proposed. The approach is based on the concept of truth valued computing process and process calculus. A truth valued process is conceived as a logical function with a side effect. It expresses the applicability of data transformation as well as the transformation itself. Process calculus is presented as a generalization of predicate calculus. Logical operators (not, and, or) are defined for processes. Serial and parallel conjunction and disjunction of processes are distinguished. All traditional program control statements as well as several new control statements are interpreted as formulas of process calculus. Interpretation and derivation rules are discussed. The calculus is intended to serve as a basis for the development of real time languages.

INTRODUCTION

There exist numerous general purpose and real time languages, but only a small amount has found a wide use. Nevertheless, elaboration of new languages has continued.

The main feature of all early algorithmic languages is a purely pragmatic development in view of possibilities of computers and the existing experience rather than explicit requirements, theoretical conceptions, and mathematical formalism. The evolution cannot yield the best results, and, therefore, new approaches laid on more solid foundation are being searched long since. Such approaches are, for instance, functional programming (Bacus, 1978; Hendersen, 1980) and logic programming (Kowalski, 1979; Lloyd, 1985). Each of them yields a new language (Lisp, Prolog) based on explicit theoretical conceptions and mathematical formalisms. The languages belong to the class of declarative languages.

Nevertheless, it is hardly possible or reasonable to solve all programming tasks only on the ground of declarative languages. It seems, for instance, that the development of real time and computer control systems calls for new solutions also in the development of high level algorithmic (imperative) languages, in particular, real time languages (Young, 1982). So far they all, including Ada, have been developed on the ground of general purpose languages, first of all, on the basis of Pascal.

Each programming language is, to a certain extent, a formal system (Kleene, 1952), although usually without deductive possibilities. But a formal system has a real value only if it is a formalization for an informal theory. The latter forms the theoretical foundation and an interpretation for the formal system. The foundation is never defined explicitly for conventional programming languages, but it is widely known, for instance, for Lisp and Prolog (lambda calculus and Horn clauses logic). This is a feature and a disadvantage of all general purpose and real time languages. However, theoretical foundations of real time systems and languages are in the stage of establishing (Hoare, 1986; Fileman and Friedman, 1984).

This paper presents a new approach to the development of such languages. An attempt is made to formalize programming on the example of classical principles proposed by Hilbert for construction of formal (axiomatic) theories (Hilbert and Bernays, 1968; Kleene, 1952). It involves development of a suitable theory of computing processes, formalization of the theory, and extension of the formalism to a practical programming language.

The basic idea in our approach to evolve programming theory and languages taken over from the Hilbert's programme is associated with the constructive (inductive) way to the development of formal systems. That is why we call the approach constructive programming (CP), the theory CP-theory, the language CP-language or COPAL (COnstructive Programming and data Abstracting Language), and the whole system CP-system.

In this paper we discuss basic principles for the system construction and the language design. These principles mainly touch upon programming logic (control flow in programs), which forms the logical framework of the whole system. The rest of the theory is connected with data abstractions. Unfortunately, these problems cannot be discussed in this paper, except some general notes.

A new instrument for describing and analyzing programming logic is proposed. It is founded on the truth valued computing process concept introduced as an extension of the predicate concept. The instrument is evolved as a generalization of predicate calculus and called process calculus.

The presentation is purely theoretical. However, the theory is regarded as a foundation for developing high level real time and computer control system programming languages. The main requirements to real time languages, such as reliability, generality, simplicity, easy readability, modifiability, maintainability, efficiency, formal definability, etc. (Fisher, 1978; Young, 1982; Smith and Wood, 1987) are taken into account at language design based on the theory. Process calculus forming the logical framework of the language (Copal) introduces intrinsical correctness and reliability into it. The language itself cannot be described fully in this paper.

DATA OBJECTS AND TYPES. FUNCTIONS

The basic notions of CP-theory are the notions of data object, data type, function and (truth valued computing) process. Three of them are discussed in this section, the last and the most important one, in the next section.

Data object and type. The term 'data object', or, more precisely, 'finite data object', is used to denote processing objects - data elements, structures and sets (sequences, lists, files). They are said to be finite because only data objects with finite length, finite value range and a finite number of elements may be represented in computer memory.

Strong data typing and data abstracting principles are fully accepted in the theory, although not in quite the usual manner. Thereby, it is assumed that each data object belongs to a class of objects called data type.

Data types are divided into two main categories - individual and collective types. We call them also data entity and data set types (the term 'set' is used in the abstract sense). The two categories of data types in the CP-language correspond roughly to static and dynamic data types in the well-known programming languages. Further, individual types are divided into composed (structured) and atomary (primitive) types.

From the point of view of this investigation it is not essential which particular data types we are dealing with. It is sufficient to suppose there are certain different data types. In order to simplify presentation we suppose all types below are data entity types, i.e. individual data types. Moreover, it will do to take into account atomary types only, for instance, natural, integer, fractional and mixed number types (do not wonder three of them are not customary). Main features of our theory touching data typing and abstracting are related to data semantics.

Semantical triangle. Data entities are represented in programs by variables introduced by explicit definitions. The expression for defining a variable v of a type T has a conventional form (the sign ':' would be conceived in the sense of 'belongs to'):

$$v : T. \qquad (1)$$

The expression is interpreted in CP-theory on the example of Frege-Church's semantical triangle (Church, 1956). So, we say the variable v denotes an entity and expresses its meaning (the concept) related to the type T, and the meaning describes the entity. At that, we assume the meaning is represented in the computer by an explicit model called type model (see below).

As a matter of fact, the expression (1) is considered as a composed form (name) consisting of the common name (variable) v and the proper name (constant) T. The denotation of v is the data entity (field), the concept, the imagination of entities of the given type. For the computer the imagination is represented by the type model. The denotation of T is just the model, the concept, the general imagination of such models.

To map the interpretation (the semantical triangle) in the computer, it is presumed that the variable v does not denote the data field itself, but a pair of pointers (field'(v),type'(v)). The first of them refers to the actual data field while the second one indicates the type model. Such reference to data is an important element and feature in the theory.

Type model. The idea of an abstract data type in one's mind includes the imagination of the structure (domain) and semantics of data entities belonging to the type. In the computer, a data abstraction is usually mapped as a multiprocedural program module in which the data structure and operations on it are fixed.

Unlike to this, we assume that a data abstraction is represented by a model consisting of two parts - structural and semantical parts. Schematically: Data abstraction = Structure + Semantics.

The structural part comprises only one element - structure model for the type. It defines the data structure, (formal) parameters of the structure (see below), and, indirectly, the domain of data entities belonging to the type. The semantical part contains a number of elements called semantic modules. Each of them defines a function, a predicate or a process on the data entities of the type. Thus, in other words: Type model = Structure model + Semantic modules.

As far as there exists only one structure model for each type model the latter is identified by the former. In this relation the structure model is called also simply the type model (in the strict sense). This was kept in view above in the mapping of the semantical triangle.

An essential feature of the model is that all its elements are separately defined and compiled, and that they are associated with each other logically but not physically. The model is assumed to be defined for all data types including atomary ones. It may be constructed and extended step by step starting with the structure model and continuing with the semantic modules.

A structure model is represented as a tree (list). It consists of only one node in the case of atomary types. Semantic modules are all one-procedural program modules. Such modules for a data type all together form a partially ordered set called semantic cluster. It is ordered by the relation 'module *x* is used in the definition (construction) of module *y* '. The structure model may be regarded as the root (the least) element of the set. The cluster as a whole is stored in the semantical memory (knowledge base) of the system.

Formation of data abstractions is an essential part of the theory. Unfortunately, it cannot be discussed in more detail in this paper.

Parametrization of types. It is assumed that data types may have different parameters. For instance, the type of vectors may be expressed in the most general case in the form 'vector(*n*)[*e* : *t*]', where *n*, *e*, *t* are formal parameters denoting the number of elements, element name and type, resp.

It yields that the vector v of 10 real elements as a derived data type may be defined so: v = vector(10)[elem : real]. The value of all parameters for a data type is represented in the structure model for the type.

However, in this paper, it is only essential that all numeric atomary types have two boundary parameters and are expressed in the form of T<l,u>, where T is a type name, l and u are formal parameters denoting lower and upper bound of the value range for the type. Hence, the next definitions, for example, are legitimate: natural<0,9>, i : integer<-100,100>, m : mixed<-2.5,2.5>.

Function. A function is defined in the CP-language by the next scheme:

```
function f(x1, ..., xn)
          x1 : T1, ..., xn : Tn
   statement  F
eodef                                        (2)
```

where f is the name of the function to be defined, x_1, ..., x_n are its formal arguments, T_1, ..., T_n are types of the arguments, and F is a formula of process calculus (see the next but one section) (eodef = end of definition). Two first lines in the scheme form the head of the definition, the third line - the body. The head (without the keyword 'function') is conceived as the specification of the function, the body, or, more precisely, the formula F, the definition itself.

The function $f(x_1, ..., x_n)$ is regarded as a generator of a dependent data entity, say y, such that $y = f(x_1, ..., x_n)$ provided that $y : T_1$. In other words, it is assumed that the data value y generated by a function always belongs to the same type as the first argument of the function (there is an exception, see below). Thus, a function is conceived as a mapping:

$$f : T_1 \times ... \times T_n \rightarrow T_1 \qquad (3)$$

(the sign 'x' stands for Cartesian product). At that, T_1, ..., T_n should be interpreted as domains of variables x_1, ..., x_n. The part '$T_1 \times ... \times T_n \rightarrow T_1$' in (3) is considered as the denotation of the type of the function f. Hence, the types of the function f and of the value y of the function are distinguished. The expression (3) in whole is called the type expression for the function F.

The type of function serves for identifying and distinguishing functions. Two functions, say f_1 and f_2, are said to be identical if and only if they have the same name ($f_1 = f_2$) and belong to the same type.

Conversion function. A conversion function (for data conversion) is a special kind of one-argument functions. In case of atomary types, it is a primitive function specified by the scheme:

$$\text{conversion } T_2'(x : T_1) \text{ eospec} , \qquad (4)$$

where T_1 and T_2 are the types of the argument x and the value $y = T_2'(x)$ of the function (eospec = end of specification). The function is named by the name of the type T_2 to which an apostrophe (') is added. It is conceived as the mapping $T_2' : T_1 \rightarrow T_2$ (this is the exception: the type of the value differs from the type of the argument).

TRUTH VALUED COMPUTING PROCESSES

Computing process. The term 'computing process', or, simply, 'process', is used in our theory to denote any data transformation procedure applied to one or more data entities. In general, the notion corresponds to the notion of program unit or module (procedure, subroutine, and, also, concurrent process), except function, in the conventional programming languages. However, the analogy reflects only the functional aspect of the notion. The logical aspect is related to a property of algorithms.

As it is known, an algorithm or a program (there is no need to differentiate these notions in given context) is an instruction that defines a computing process for transformation of some argument data x into result data y. Such process may be successful or unsuccessful depending on whether the desired results may be derived from given argument data, or not (certainly if the process is terminated at all). The algorithm is said to be applicable in the former case and unapplicable in the latter one.

Validity of processes. The applicability of an algorithm is determined by the successfulness of the computing process defined by the algorithm. However, there is no need to differentiate these properties because an algorithm may be conceived as the definition of a process while a computing process may be regarded as an application of an algorithm (invocation of a program). We leave the notion of algorithm aside and discuss the problem in terms of abstract processes only. Thereby, we prefer to call the successfulness of a process (applicability of an algorithm) simply validity and say that a process is valid or invalid instead of speaking of it as successful or unsuccessful.

(The term 'validity' has in our theory the meaning which is usually, in mathematical logic, expressed by the term 'satisfiability'. Validity and satisfiability is called total and partial validity in this paper.)

The main idea of our approach is that the validity of processes is treated as a logical value, and, thereby, a process P to transform argument data (entities) x into result data (entities) y is conceived as a truth valued function with a side effect. Such process is denoted by P(x->y) or P(y<-x). The main effect of the function involves the determination of the logical value representing the validity of the process. The side effect is connected with the transformation of x into y if the process is valid (it is not essential which of the two effects is called main or side effect). In other words, the process is conceived as a mapping (of a mapping):

$$P : (X \rightarrow Y) \rightarrow L \qquad (5)$$

where X, Y are the types (domains) of argument and result data, and L = {T,F}, i.e. logical values T (true) and F (false) are used to denote validity and invalidity of the process, resp. However, such representation of processes is somewhat simplified.

Definition of processes. In general, a process is defined by the scheme having two forms - main and inverse forms:

```
process P(x->y'->z)                process P(z<-y'<-x)
   x : X, y : Y, z : Z                z : Z, y : Y, x : X
   statement  F                       statement  F
eodef                              eodef                    (6)
```

where P is the name of the process to be defined, x, y, z are interface variables of the types X, Y, Z, resp., and F is a formula of process calculus (see next section). As in case of scheme (2) two first lines in (6) represent the head (specification) of the process, the third line - the body.

Interface variables (formal parameters) of a process are divided into three classes - argument or input variables (x), result or output variables (z) and state or input/output variables (y). Only one parameter of each class is shown in the scheme (6). Actually, some or all of the expressions x : X, y : Y, z : Z may be replaced by a list of such expressions. The two schemes differ from each other only by the order in which the variables of different classes are represented. The pairs of symbols '->' and '<-' are used as separators between different classes of variables. Note that the state variable (y) in the head of the definition (in both forms) is distinguished by an apostrophe (').

Process type. A process defined by (6) (both main and inverse forms) is conceived as a mapping:

$$P : (X \times Y \rightarrow Y \times Z) \rightarrow L . \qquad (7)$$

Again, the part '(X x Y -> Y x Z) -> L' in the expression is considered as the denotation of the type of the process P. The expression (7) in the whole is called the type expression for the process P. Two processes, say P_1 and P_2, are said to be identical if and only if they have the same name ($P_1 = P_2$) and belong to the same type.

The scheme (6) is not complete, some details, such as environment interface (I/O) and import parameters, are omitted and not discussed in this paper.

Specific cases. Three interface variables of different classes were shown in the scheme (6). Actually, one or two of them may be omitted. Thus, there are six particular cases of the scheme (6). They are the following (specifications of main and inverse forms and type expressions are shown):

1. P(x->z) x:X, z:Z	P(z<-x) z:Z, x:X	P : (X -> Z) -> L
2. P(x->y') x:X, y:Y	P(y'<-x) y:Y, x:X	P : (XxY -> Y) -> L
3. P(y'->z) y:Y, z:Z	P(z<-y') z:Z, y:Y	P : (Y -> YxZ) -> L
4. P(y') y:Y	P(y') y:Y	P : (Y -> Y) -> L
5. P(->z) z:Z	P(z<-) z:Z	P : (0 -> Z) -> L
6. P(x) x:X	P(x) x:X	P : (X -> 0) -> L

Note. 0 denotes empty data type (domain).

Examples:

1. Asn(x->z)		- assign x to z,
	Eqt(z<-x)	- equate z with x,
2. Put(x->y')	-	- put x into (set) y,
3. -	Get(z<-y')	- get z from (set) y,
4. Sort(y')	Sort(y')	- sort (set) y,
5. Setnull(->z)	Setnull(z<-)	- set z to be null,
6. Null(x)	Null(x)	- x is null.

Note. The variable y in the examples 2 - 4 is assumed to be of a data set type, for instance, of type Stack(n){e : t}, Queue(n){e : t} or File(n){e : t}.

Evaluation process. In a specific case, a process P may not be connected with any argument or state data entity, but after all generates some result data. Such process corresponds to the specific case 5 described above. It is called a process without arguments or evaluation process.

Predicate. The other, a more specific case is that a process P may not be connected with any state or result data entity. It represents a logical condition or predicate P(x) held on argument data x. We call such a process a predicate. It corresponds to the specific case 6. However, in the definition

of a predicate, the keyword **process** is replaced by the keyword **predicate**. So a predicate is defined by the scheme:

```
predicate P(x)
    x : X
  statement C
eodef                                    (7)
```

where *C* is a clause of process calculus (see next section). Hence, predicates form a subclass of the class of processes.

Iteration procedure. Finally, there exists a very specific subset of processes called iteration procedures to be used simultaneously as a process and as a predicate in certain constructions. In general, it corresponds to the specific case 4 mentioned above. However, it is defined by a specific scheme described in a subsequent section.

Process interface. Presume a process (module) *P* is defined by the scheme (6) (and stored in the semantical memory of the system). Suppose the process is used (as an external procedure or subroutine) in another process (module) *R*. The situation (invoking *P* in *R*) is described, in general outline, in the conventional manner as follows (in two forms):

```
process R(...)                    process R(...)
  ...                               ...
statement                         statement
  ...                               ...
  define ...,                       define ...,
    a : X, s : Y, r : Z,              a : X, s : Y, r : Z,
    ...                               ...
  begin                             begin
    ...                               ...
    P(a -> s' -> r)                   P(r <- s' <- a)
    ...                               ...
  end                               end
  ...                               ...
eodef                             eodef                  (8)
```

where a, s, r are local variables of types X, Y, Z defined in *R* and used as actual (argument, state and result) parameters of the process *P* (a, s, r must not be obligatory local variables; some or all of them may be formal parameters of the process *R*).

The scheme shows that the type of the called process *P* may be determined in the calling process *R* (at the compile time) by the type of actual parameters and by the role in which they are used at the invocation (as an argument, state variable or result). The name *P* and the process type together determine the process module to be searched in the semantical memory. Thus, module interface consistency is supported in the CP-system.

Notation. The functional notation is used in the system for presentation of processes, predicates and functions. The notation is illustrated by the following list of examples:

(1) processes	Asn(x->y)	assign x to y	y := x
	Eqt(y<-x)	equate y to x	y := x
	Setnull(->x)	set x to be null	x := 0
(2) predicates	Eq(x,y)	x is equal to y	x = y
	Lt(x,y)	x is less than y	x < y
	Null(x)	x is equal to 0	x = 0
(3) functions	sc(x)	successor of x	x'
	sum(x,y)	sum of x and y	x + y

Process calculus. The calculus is proposed as a generalization of predicate calculus in which the role of predicates is played by processes. It is discussed in detail in the following sections of the paper. In general outline, it is characterized as follows.

(1) All logical operations, such as negation, conjunction and disjunction, are defined for the processes. However, generalization is needed: conjunction and disjunction of processes cannot be commutative operations. Further, these operations have two different operational interpretations: serial and parallel. Quantifiers (all, some) and descriptors (any, sole) are also used in process calculus.

(2) Processes and predicates (actions and logical conditions) may be associated by means of logical operations to describe control flow in program modules. So programming logic is represented as a whole.

(3) The validity of a composed process is determined by the validity of component processes as a result of the process performance.

(4) The total and partial validity (validity and satisfiability, in usual terms) of processes may be defined similar to those of predicates in set-theoretical interpretation of predicate calculus (Hilbert, 1968). Further, general rules can be shown for the determination of the properties for the composed processes by those for component processes. It means that certain rules of inference may be introduced into process calculus.

THE FORMALISM OF PROCESS CALCULUS

Now we shall consider process calculus as a formal system and define the main elements of the system.

Alphabet. The alphabet of the language (formal system) consists of symbols for denoting the following objects: (1) data values, (2) data entities, (3) data entity types, (4) functions, (5) processes (including predicates, evaluation processes and iteration procedures). They are represented below by small and capital letters (used as metasymbols). Actually (in the language), they are represented by identifiers (names), data values - by numerals and literals.

In addition, subsidiary symbols are needed for forming several syntactic constructions, denoting different operators and connectors, etc. They are expressed by fixed sequences of symbols in bold - keywords of the language, or by punctuation symbols. Part of them (keywords) are divided into two, in a way, dual subclasses. Main subsidiary symbols are the next (mutually dual symbols are separated by the sign '/'):

(1) formation conjunctors: **if/as, then/thus, else/othw, while/when, do/try, for, such, switch, case, define, denote;**

(2) junction brackets: **(,), begin, end;**

(3) logical operators: **not, deny, and/or, all/some, any/sole;**

(4) junction qualifiers: **ser, par.**

(Some keywords are abbreviations: **othw** = otherwise, **ser** = serially, **par** = parallel.)

Free and bounded variables. Variables, as we know, denote processing objects, i.e. data entities. In our formalism, as in any other formal theory, variables are divided into free and bounded variables (Kleene, 1952).

Module interface variables are all treated as free variables of the formula F in the module defined by (6). Their scope of influence comprises all the body of the module. All other variables can appear in the body only as bounded variables of different kind with restricted scope. They will be discussed below. The binding means that the variable is bounded by an operator, and that it is not defined (known) outside the scope but regarded as a free variable within the scope.

Terms. Terms are language elements referring to the (independent and dependent) variables to be used as actual parameters of modules, i.e. as actual arguments of functions and predicates or as actual arguments, states and results of processes. They are divided into argument, state and result terms.

Definition 1 (argument term).

1. A free argument and state variable are both argument terms.
2. If $f(x)$ is an n-argument function and a is a list of n argument terms ($n > 0$), then $f(a)$ is an argument term.

Definition 2 (state and result terms).

1. A free state variable is a state term.
2. A free result variable is a result term.

Clauses. Clauses are formal counterparts of elementary or composed predicates for expressing logical conditions in program modules. They are constructed using previously defined predicates as constituents.

Definition 3 (elementary clause).

1. Constant predicates *True* and *False* are both elementary clauses.
2. If $P(x)$ is an n-placed predicate and a is a list of n argument terms ($n > 0$), then
   $P(a)$ (C1)
   is an elementary clause.

Definition 4 (clause).

1. An elementary clause is a clause.
2. If A is a clause, then so is
   not A . (C2)
3. If $A_1, \ldots, A_n$ are clauses ($n > 1$), then
   (ser A_1 and ... and A_n) (C3.C)
   (ser A_1 or ... or A_n) (C3.D)
   are both clauses ('ser' may be omitted).
4. If $A_1, \ldots, A_n$ are clauses ($n > 1$), then
   (par A_1 and ... and A_n) (C4.C)
   (par A_1 or ... or A_n) (C4.D)
   are both clauses.

Formulas. Formulas are formal counterparts of elementary or composed processes for expressing data transformations and actions in program modules. They are constructed using previously defined processes predicates, evaluation processes and iteration procedures as constituents.

Definition 5 (elementary formula).

1. An elementary clause is an elementary formula.
2. If $P(x->y'->z)$ ($P(z<-y'<-x)$) is a (m,k,n)-placed process (i.e. a process with m, k and n formal argument, state and result variables, resp.), a, s, r are lists of m argument terms, k state terms and n result terms ($m > 0$ or $m = 0$, $k > 0$ or $k = 0$, $n > 0$), then
   $P(a->s'->r)$ (resp. $P(r<-s'<-a)$) (F1)
   is an elementary formula.

Definition 6 (formula).

1. An elementary formula is a formula.

2. If *P* is a formula, then so is
   deny *P* . (F2)

3. If *v* is a data entity symbol, *t* is a data type, P_1, ..., P_n are formulas ($n > 1$), then
   define *v* : *t* **begin ser** P_1 **and** ... **and** P_n **end** , (F3.C)
   define *v* : *t* **begin ser** P_1 **or** ... **or** P_n **end** (F3.D)
   are both formulas (the phrase '**define** *v* : *t*' is not obligatory, the expression '*v* : *t*' in the phrase may be replaced by a list of such expressions, the word '**ser**' may be omitted).

4. If *v*, *t*, P_1, ..., P_n are the same as above, then
   define *v* : *t* **begin par** P_1 **and** ... **and** P_n **end** , (F4.C)
   define *v* : *t* **begin par** P_1 **or** ... **or** P_n **end** (F4.D)
   are both formulas (the phrase '**define** *v* : *t*' is not obligatory, the expression '*v* : *t*' in the phrase may be replaced by a list of such expressions).

5. If *A* is a clause and *P*, *Q* are formulas, then
   if *A* **then** *P* **else** *Q* , (F5.C)
   as *A* **thus** *P* **othw** *Q* (F5.D)
   are both formulas (the phrases '**else** *Q*' and '**othw** *Q*' are not obligatory).

6. If *s* is a data entity symbol, *t* is a term of the natural type, *a*, ..., *b* are natural values and P_a, ..., P_b, *Q* are formulas, then
   if switch *s* = *t* : [*a*, ..., *b*]
   then case (*a*) P_a, ..., **case** (*b*) P_b **else** *Q* , (F6.C)
   as switch *s* = *t* : [*a*, ..., *b*]
   thus case (*a*) P_a, ..., **case** (*b*) P_b **othw** *Q* (F6.D)
   are both formulas (the phrases '**else** *Q*' and '**othw** *Q*' are not obligatory).

7. If *A* is a clause and *P* is a formula, then
   while *A* **do** *P* , (F7.C)
   when *A* **try** *P* (F7.D)
   are both formulas.

8. If *x* is a data entity symbol, *t* is a countable number type, *a*, *b* are values, constants or variables of the type, *N* is an iteration procedure defined for the type, *A*, *B* are clauses, and *P* is a formula, then
   for all *N*(*x* : t<a,b>) **such A while** *B* **do** *P* , (F8.C)
   for some *N*(*x* : t<a,b>) **such A when** *B* **try** *P* (F8.D)
   are both formulas (the phrases '**such A**', '**while** *B*' and '**when** *B*' are not obligatory).

9. If *x*, *t*, *a*, *b*, *N*, *A*, *P* are the same as above, then
   for any *N*(*x* : t<a,b>) **such A do** *P* , (F9.C)
   for sole *N*(*x* : t<a,b>) **such A try** *P* (F9.D)
   are both formulas (the phrase '**such A**' is not obligatory).

The phrase '**define** *v* = *t*' in the formulas (F3) and (F4) is called the (local) variable definition prefix. The phrases '**else** *Q*' and '**othw** *Q*' in (F5) and (F9) are called the alternative suffixes, the phrase '**such A**' in (F7) - (F8) - the selection infix, and the phrases '**while** *D*' and '**when** *D*' in(F7) - the continuation infixes.

Dual and quasidual forms. Clauses marked by labels with suffixes C and D are called conjunctive and disjunctive forms (C- and D-forms), resp. They are said to be mutually dual. The clause (C1) may be regarded as C-form or as D-form. It is said to be dual to itself. The same is valid for the clause (C2).

Similarly, formulas marked by labels with suffixes C and D are called conjunctive and disjunctive forms (C- and D-forms), resp. The formulas (F3.C) - (F4.C) are called dual with respect to (F3.D) - (F4.D) while the formulas (F5.C) - (F9.C) are called quasidual with respect to (F5.D) - (F9.D). The formula (F1) may be regarded as C-form or D-form, and so is conceived to be (quasi)dual to itself. The same is valid for the formula (F2).

Bounded variables. The variables introduced by the formulas (F6) and (F8) - (F9) as well as the variables defined in the declaration prefix of the formulas (F3) - (F4) are all bounded variables. We call them (proper) bounded variables in the former case and local (bounded) variables in the latter one. The variable *s* in the formulas (F6) is bounded by the selection operator **switch** and called a selection variable. Its scope contains the component formulas P_a, ..., P_b and *Q* of the formula. In the scope, the variable may be used only as an argument term.

The variable *x* in the formulas (F8) - (F9) is bounded by the quantifiers **all**, **some** (at least one), **any** (at most one) or **sole** (exactly one) and called an iteration variable. Its scope contains the component clauses *A*, *B* and the component formula *P* in the case of the formulas (F8), and the component clause *A* and the component formula *P* in the case of formulas (F9). Again, in its scope, the variable may be used only as an argument term.

Local variables and constants. The variable *v* in the formulas (F3) - (F4) is said to be bounded by the definition operator **define** and called a local (bounded) variable. Its scope includes the whole block **begin** ... **end**. But unlike selection and iteration variables a local variable may be used in its scope as an argument, state or result term.

Constants (more precisely, local constants) may be introduced in the same manner as local variables, i.e. by means of an appropriate declaration prefix to the formulas (F3) - (F4). Such prefix was not mentioned above, but actually it is foreseen: "**denote** *c* = *t'*(*d*)", where *c* is a data entity symbol, *t'* is a converion function for the data type *t*, and *d* is a data value (for

example, denote pi = real'(3.14) ...). The scope of the constant c also includes the whole block begin ... end. In the scope, a constant may be used only as an argument term. As in the case of local variable the expression' c = t'(d)' may be replaced by the list of such expressions.

CONTROL LOGIC ELEMENTS AND PROCESS NETS

Now let us go to the interpretation of the formulas as statements of the programming language represented by the formalism. In this relation we speak of the clauses and formulas also as of predicates and processes that they actually denote. In current section we introduce tools for the interpretation. Particular formulas are interpreted in subsequent sections.

Dynamic performance of processes and predicates may be simulated by means of Petri nets. However, we use for this purpose special kind of Petri nets that we call process nets. Before we can proceed to explain their features some general questions must be discussed.

Simple and composed processes. A process (predicate) is called simple if it is defined by the elementary formula (F1) (resp. clause (C1)), inverse if it is defined by the formula (F2) (clause (C2)), and composed otherwise. In turn, a composed process (predicate) is called compound if it is defined by (F3) or (F4) (resp. (C3) or (C4)), and complex otherwise (there are no complex predicates). A compound process (predicate) consists of two or more component processes (predicates). It may be defined as a serial or parallel junctions of component processes - (F3), (F4) (resp. component predicates - (C3), (C4)). A junction may be a conjunction or a disjunction.

Unlike compound processes, a complex process consists of one or more component processes as well as one or more component predicates. Processes defined by (F5) and (F6) are called selection processes with two or multiple choices, resp., by (F7) and (F8) - indefinite and definite iteration processes, and by (F9) - an approbation process.

Concurrency. The formalism is related to the representation of not only sequential but also concurrent processes and predicates. Concurrent processes (predicates) are represented as components in parallel junctions of processes (predicates) defined by formulas (F4) (resp. clauses (C4)).

The concurrency is regarded to be relative rather than absolute, and treated as a virtual (conceptual) concurrency. The relativity means that we cannot say is a particular process (predicate) a concurrent or sequential one, or not, but we can say are two (or more) component processes (predicates) of a composed process mutually concurrent or sequential, or not. The virtuality means that each of concurrent components (of a composed process) is regarded to be performed on its own virtual processor.

Performance states and levels. In order to describe the dynamic performance of processes and predicates more precisely different performance states and levels of processes are distinguished. We say that each process (predicate) is executed on a performance level L (L = 0,1,2,...), and that a (composed) process itself and its component processes are all executed in the process performance state while its component predicates are executed in the predicate performance state (substate) of one and the same or of different levels. Agree to denote the process and predicate states for the level L by S(L) and s(L), resp.

The performance state and level for a process (predicate) are determined by the following rules:

1. The process defined by a main module (program) is performed in the state S(0), i.e. in the state S on the level L = 0.
2. If a complex process is performed in the state S(L), then its component processes are performed in the same state S(L) while component predicates are performed in the state s(L).
3. If an inverse process (predicate) is performed in the state S(L) (resp. in the state s(L)), then the process (resp. predicate) to be inverted is performed in the same state S(L) (resp. in the state s(L)).
4. If a serial junction of processes (predicates) is executed in the state S(L) (s(L)), then the component processes (predicates) of the junction are performed in the same state.
5. If a parallel junction of processes (predicates) is executed in the state S(L) (s(L)), then the component processes (predicates) of the junction are performed in the state S(L') (s(L')), where L' = L + 1.

Thus, the transfer to the next performance level is caused only by the performance of parallel junctions of processes and predicates. The levels may be treated as concurrency levels. If the junction performed on the level L consists of n components, then n concurrent processes (predicates) are performed on the level L' = L + 1.

Process net. A process net is a particular Petri net introduced for the interpretation of formulas and clauses of process calculus. It is intended to

be a strict and complete model for describing control flow at the performance of the formulas. It has a number of features:

(1) Places in the process net are divided into action (main) and control (subsidiary) places.

(2) Action places have one input and two output transitions, in the particular case - one output transition. They are foreseen for mapping predicates and processes, and are denoted by symbols for the predicates (A) and processes (P) they are representing (Fig. 1, panels 1.1, 1.2). Action places have one output transition in the case of totally valid or invalid predicates and processes (panels 1.3, 1.4).

(3) Control places have one input and one output transition. They are foreseen for representing specific control functions, and are denoted by special symbols including the empty symbol (Fig. 1, 2.1 - 2.3). They will be discussed below.

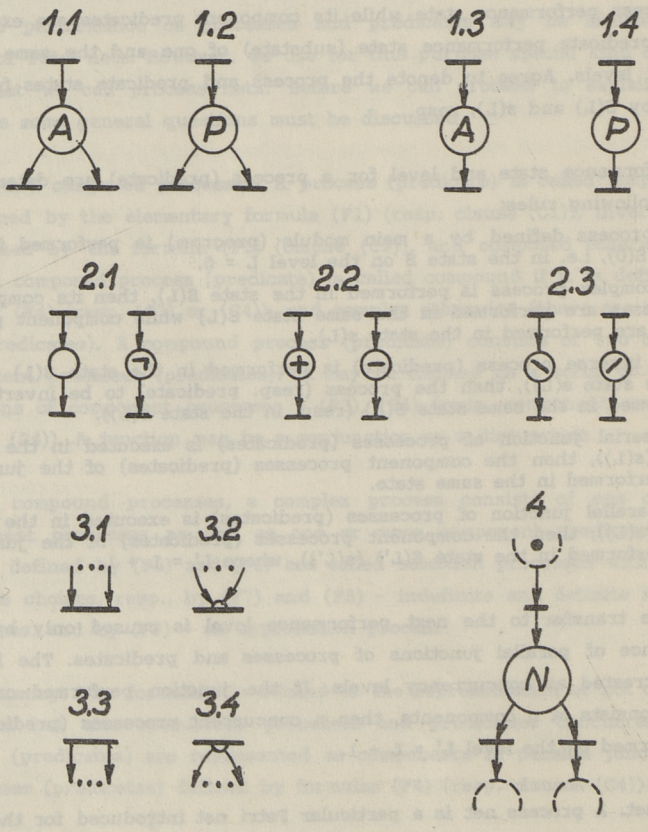


Fig. 1. Elements of process nets.

(4) A transition in a process net may be one with conjunctive or disjunctive input logic (Fig.1, panels 3.1, 3.2), or with conjunctive or disjunctive output logic (panels 3.3, 3.4).

(5) Each place may hold at most one token.

Process nets for all clauses and formulas (they are discussed in the next section) have the same general property as their main elements (action places) - they have one input transition and two output transitions (Fig. 1, panel 4). It is assumed that for each net there exists one place outside the net from which an arc is incident on the input transition of the net, and two places (outside the net) on which arcs from the output transitions terminate (dotted circles in Fig. 1). The initial arrangement of tokens is always assumed to be defined so as it is shown in Fig. 1, i.e. represented by a token in the outside input place (the outside places are not shown further).

Simulation rules for process nets are as follows:

1. A transition with conjunctive (disjunctive) input logic which has a token in each (resp. at least in one) of its input places is enabled.
2. Any enabled transition may be chosen to fire.
3. Firing a transition with conjunctive (disjunctive) output logic consists of removing the only token from each of its input places and putting one token to each (resp. to one) of its output places.

Validity test variable. In our formalism, the performance of a composed process depends on the validity of both component predicates and processes of the composed process. The validity of a composed process can be determined dynamically during the performance of the process. This is one of the most important features of the language.

In order to describe control flow during the performance of a (composed) process in detail a logical variable V for testing validity of processes and predicates is assigned to tokens in the process net for the process. Agree to denote the value of the variable in the states S and s by V(S) (V(S) = T,F) and V(s) (V(s) = t,f), resp.

Mapping and testing simple predicates and processes. A simple predicate A and process P is mapped in process nets actually so as shown in Fig. 2 (panel 1). The mappings are trivial process nets. Note that all transitions in the nets are labelled in a certain way by a value of the test variable V, and that there are two possibilities to label transitions in the nets, one for

C- and the other for D-form of the clause and formula (Fig. 2, left and right shapes).

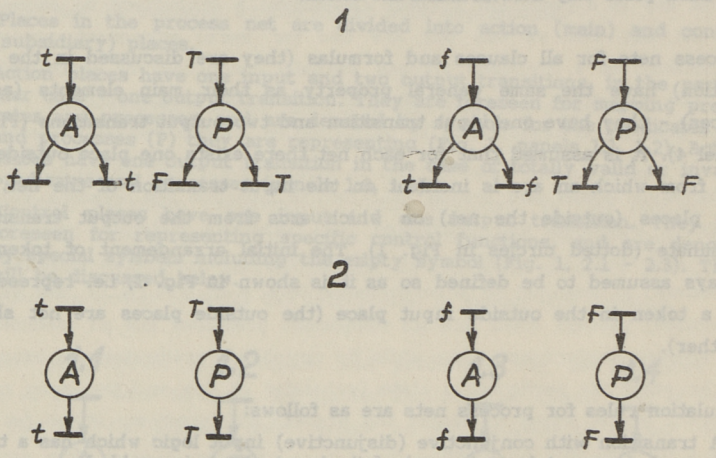


Fig. 2. Elementary predicates and processes.

Speaking informally, the labels t and f in the case of predicate A (T and F in the case of process P) at the output transitions of the nets mark which of them fires if the predicate (process) is proved to be true (valid) or false (invalid). The label t or f (T or F) at the input transition shows to which output one of another predicate (process) place the input transition may be connected. Formally, the labels are actually represented as the values of the test variable V for testing validity of predicates and processes. The value of the variable at the output transition V" is connected with its value at the input transitions V' in C- and D-forms of the nets as follows:

$V'' = V' \,\&\, A$ $\qquad$ $V'' = V' \vee A$

$V'' = V' \,\&\, P$ $\qquad$ $V'' = V' \vee P$

i.e. by the use of conjunction in the case of C-form and disjunction in the case of D-form of the mappings. It yields how a totally valid (A = t, P = T) or invalid (A = f, P = F) predicate and process must be mapped in the process nets (Fig. 2, panel 2).

Simulation. Thus, the test variable V is introduced to simulate passing tokens through the action places depending on the validity of predicates and processes. To simulate control flow fully, it was needed to introduce

control places into process nets. The value of the test variable V may change not only in the action places but also in some control places.

Control places. There are three groups of control places (Fig. 1, panels 2.1 - 2.3). The functions of the groups are:

(1) passing and inversion of the test variable V without any changes in the performance level and state (2.1);

(2) transfer to the next performance level L'= L + 1, and return to the previous level L with no changes in the performance state and in the value of the test variable (2.2);

(3) transfer to the predicate performance state (may be with changing the truth value of the test variable), and return to the process performance state with the previous value of the test variable with no changes in the level (2.3).

Iteration procedures. An iteration procedure is a special one parameter process to be used in formulas (F8) - (F9) in order to generate values of the iteration variable x, and to terminate the iteration. Iteration procedures must not be confused with iteration processes. An iteration procedure is a component of all iteration processes defined by the formula (F8) as well as of approbation processes defined by the formula (F9).

An iteration procedure is involved to be defined by means of the language (formalism). But it is reasonable to allow an iteration procedure to be defined only for countable data types. Actually, it is necessary to define two iteration procedures for such types, one for the ascending and the other for the descending of the iteration variable in certain limits. We call them Next and Prev (previous), resp.

The iteration procedures Next and Prev for the natural type (nat) are defined as follows:

```
iteration Next(x : nat<a,b>)
      init  Eqt(x<-a)                    -/ x := a
      step  as    Lt(x,b)                -/ x < b
            thus  Eqt(x<-sc(x))          -/ x := x + 1
eodef ,                                                    (9)

iteration Prev(x : nat<a,b>)
      init  Eqt(x<-b)                    -/ x := b
      step  as    Gt(x,a)                -/ x > a
            thus  Eqt(x<-pd(x))          -/ x := x - 1
eodef ,                                                    (10)
```

where a, b are the lower and upper bounds of the value range, Eqt(x<-y), Lt(x,y), Gt(x,y), sc(x), pd(x) are modules (a process, two predicates and two functions) defined for the type.

The expression 'x : nat<a,b>' in the head of the procedures has the meaning 'x belongs to the interval [a,b]'. However, the procedure Next(x : nat<a,b>) is intended to support the ascending of the iteration variable x in the interval [a+1,b] while the procedure Prev(x : nat<a,b>) is foreseen to support the descending of the variable in the interval [a,b-1].

The body of the definitions consists of two parts - the iteration initiation and step parts. It is assumed that the process defined in the former part is performed once before the execution of the iteration (or approbation) process in which the procedure is contained (initiation step) while the process defined in the latter is performed before each execution of the process contained in the body of the iteration (approbation) process (iteration step). The termination of iteration in these processes is determined by the process defined in the latter part: the iteration is terminated if the process is proved to be invalid (see the interpretation of the formula (F5.D) in the next section).

INTERPRETATION OF FORMULAS

In this section, the performance of all formulas of process calculus as programming statements is interpreted by means of process nets. Interpretation of clauses is not discussed because it is very close to the interpretation of similar formulas.

Inverse process. The performance of an inverse process by (F2)

R = deny P

does not differ from the performance of the process P itself, except when

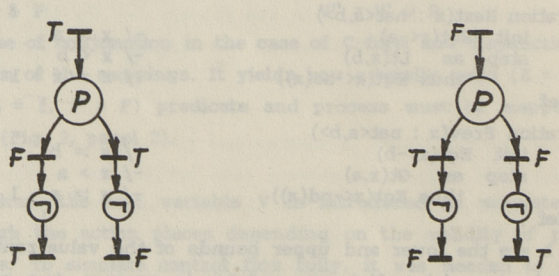


Fig. 3. Inverse processes.

the validity of R is defined inverse (denied) to the validity of P. For this appropriate passing places (Fig. 1, panel 2.1) are added to the output transitions of the place for P (Fig. 3).

Serial junctions of processes. The performance of a serial conjunction (F3.C) (in case of n = 2, P_1 = P, P_2 = Q)

R = **begin ser** P **and** Q **end**

begins with the execution of P and continues with the execution of Q, if P is valid (T) (Fig. 4, panel 1). If Q is also valid (T), then the performance of R ends and it is proved to be valid (T). Otherwise, i.e. if P or Q are invalid (F), R is proved to be invalid (F).

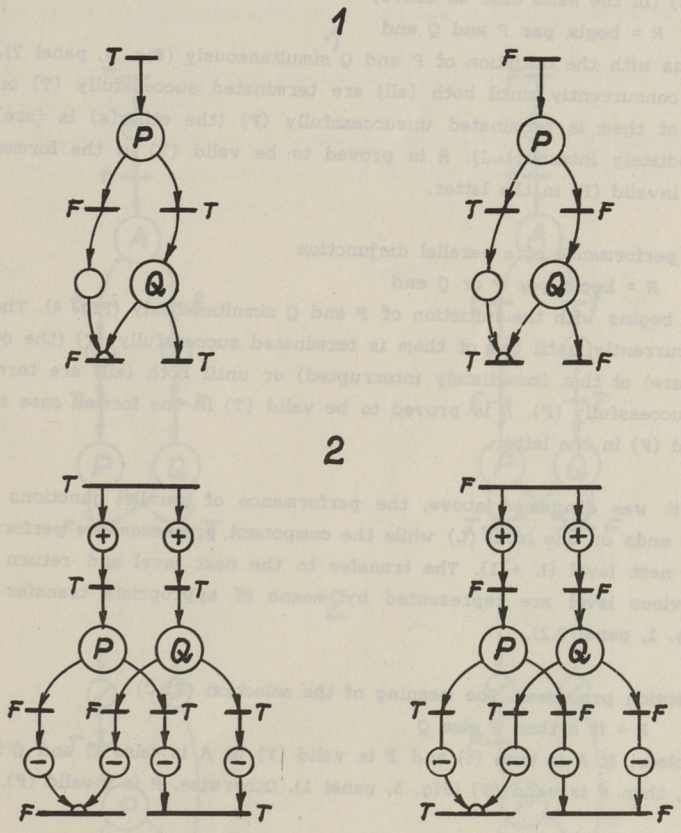


Fig. 4. Serial and parallel junctions of processes.

Unlike this, the performance of a serial disjunction (F3.D)

R = **begin ser** P **or** Q **end**

begins with the execution of P and continues with the execution of Q, if P is invalid (F) (Fig. 4, panel 1). If Q is also invalid (F), then the performance of R ends and it is proved to be invalid (F). Otherwise, i.e. if P or Q are valid (T), R is proved to be valid (T).

The presence or absence of declaration prefixes in (F3) (and (F4)) has no influence on the nets for the formulas. Note the coincidence of net shapes for C- and D-forms of the formulas discussed here and below.

Parallel junctions of processes. The performance of a parallel conjunction (F4.C) (in the same case as above)

R = **begin par** P **and** Q **end**

begins with the initiation of P and Q simultaneously (Fig. 4, panel 2). They run concurrently until both (all) are terminated successfully (T) or until one of them is terminated unsuccessfully (F) (the other(s) is (are) then immediately interrupted). R is proved to be valid (T) in the former case and invalid (F) in the latter.

The performance of a parallel disjunction

R = **begin par** P **or** Q **end**

also begins with the initiation of P and Q simultaneously (Fig. 4). They run concurrently until one of them is terminated successfully (T) (the other(s) is (are) at that immediately interrupted) or until both (all) are terminated unsuccessfully (F). R is proved to be valid (T) in the former case and invalid (F) in the latter.

As it was discussed above, the performance of parallel junctions begins and ends on one level (L) while the component processes are performed on the next level (L + 1). The transfer to the next level and return to the previous level are represented by means of appropriate transfer places (Fig. 1, panel 2.2).

Selection processes. The meaning of the selection (F5.C)

R = **if** A **then** P **else** Q

is clear. If A is true (t) and P is valid (T) or A is false f) and Q is valid (T), then R is valid (T) (Fig. 5, panel 1). Otherwise, R is invalid (F).

The meaning of the selection (F5.D)

R = as A thus P othw Q

is similar and the validity of R is the same as for the previous process (Fig. 5). A difference between the last two formulas appears if the phrases 'else Q' and 'othw Q' are omitted. In this case, if A is false (f), then the former is valid (T) while the latter is invalid (F) (Fig. 5, panel 2).

Note that these (Fig. 5) and all subsequent nets contain places for transfer to predicate performance state s and return to the process performance state S (Fig. 1, panel 2.3).

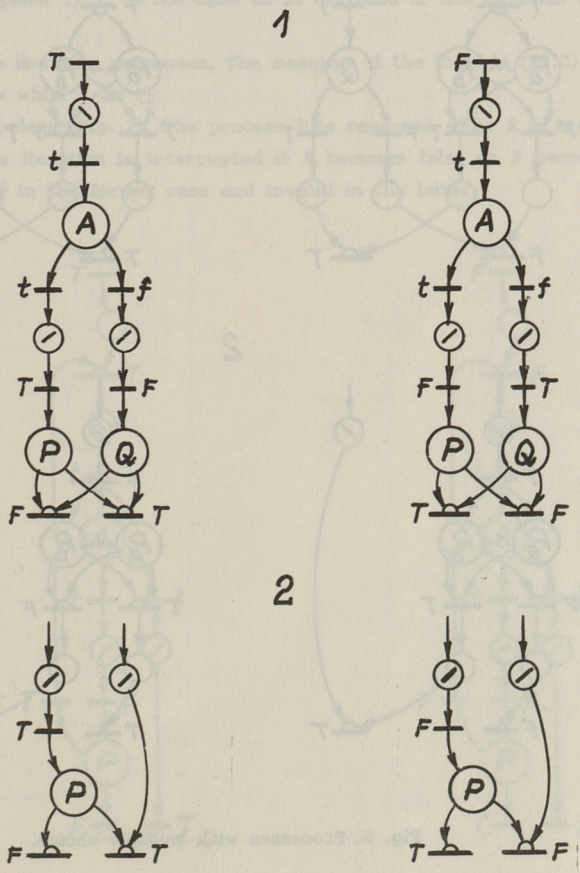


Fig. 5. Selection processes.

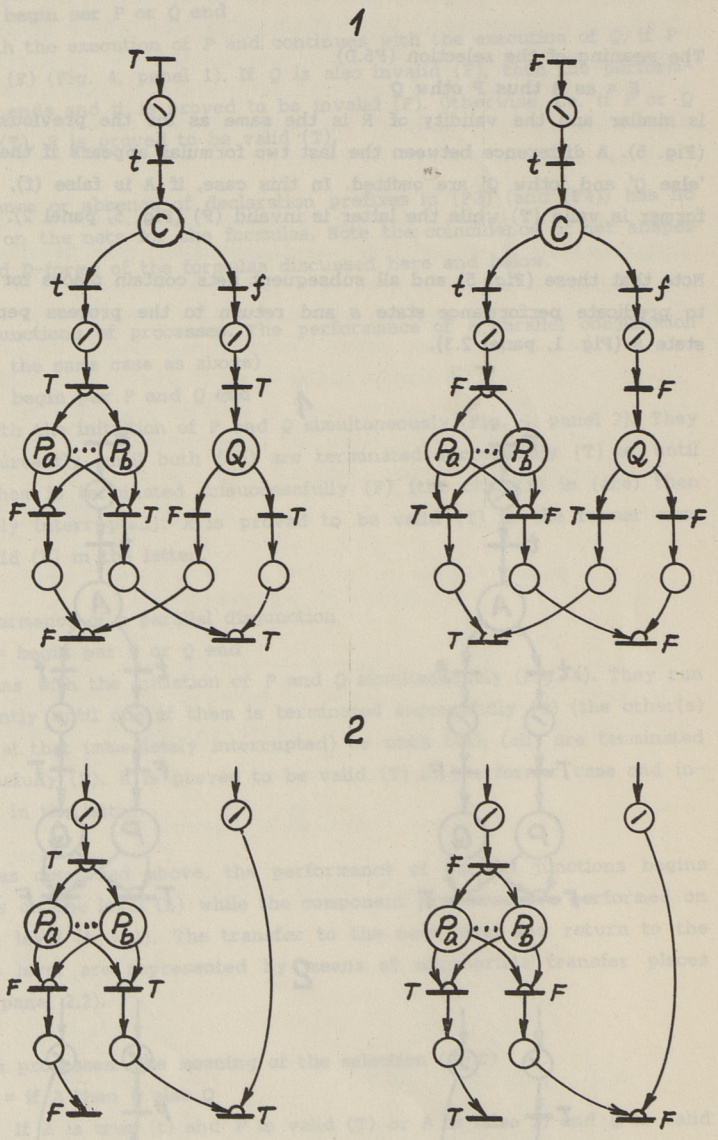


Fig. 6. Processes with multiple choice.

Processes with multiple choice. The performance of the selections with multiple choice (F6)

R = **if switch** *s* = t : [a, ..., *b*]
then case (a) P_a, ..., **case** (*b*) P_b **else** *Q*

R = **as switch** *s* = t : [a, ..., *b*]
thus case (a) P_a, ..., **case** (*b*) P_b **othw** *Q*

is described by nets in Fig. 6 (panel 1 and 2 as above) where the condition 's = t : [a, ..., *b*]' (standing for 's (= t) belongs to the set {a, ..., *b*}') is denoted by *C*. Both formulas mean that depending on the value of *s* one of the processes mentioned must be performed. The whole process *R* is valid if the selected process is found valid (panel 1). The difference between the two formulas appears if the phrases '**else** *Q*' and '**othw** *Q*' are omitted (panel 2). It is the same as in the case of the formulas (F5).

Indefinite iteration processes. The meaning of the formula (F7.C)

R = **while** *A* **do** *P*

would be clear (Fig. 7). The process *P* is repeated while *A* is true and *P* is valid. The iteration is interrupted if *A* becomes false or *P* becomes invalid. *R* is valid in the former case and invalid in the latter.

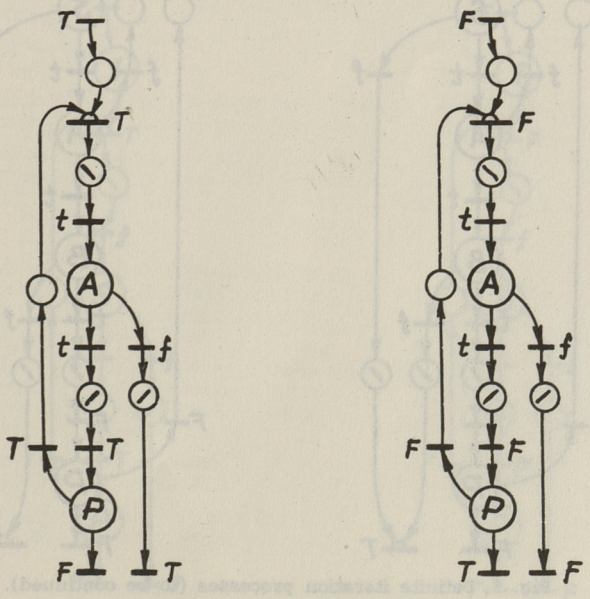


Fig. 7. Indefinite iteration processes.

The formula (F7.D)

R = when A try P

is also related to the iteration of P (Fig. 7). However, now P is repeated while A is true but P itself is invalid. Such iteration is interrupted if A becomes false or P becomes valid. R is invalid in the former case and valid in the latter.

Definite iteration processes. The performance of iterations by (F8) is explained in Fig. 8 (panels 1 and 2). One can presume the nets are represented for the formulas:

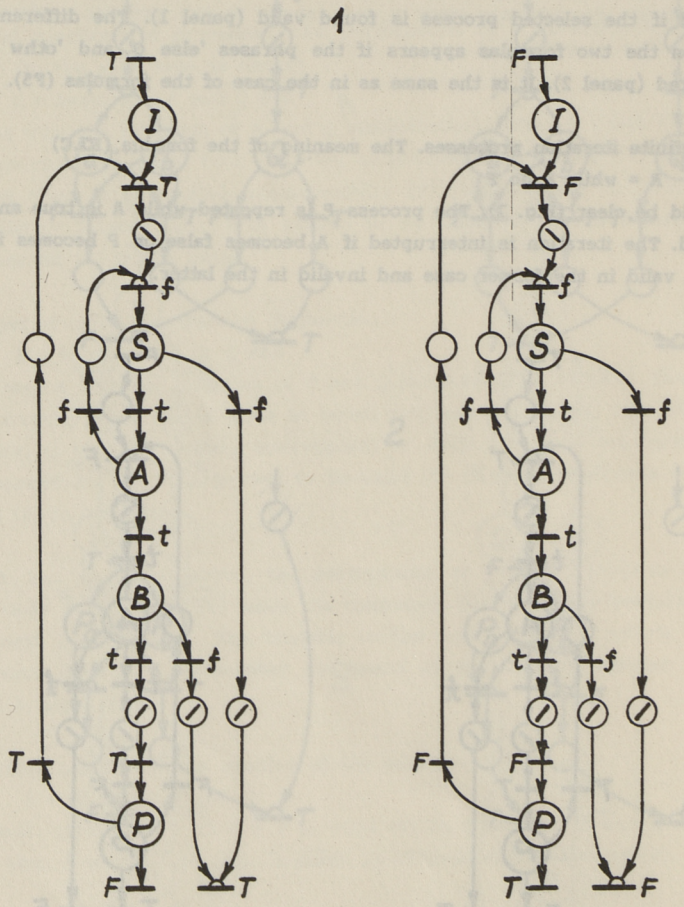


Fig. 8. Definite iteration processes (to be continued).

R = **for all** Next(x : nat<a,b>) **such A while B do P,**

R = **for some** Next(x : nat<a,b>) **such A when B try P,**

i.e. *N* and t<a,b> are replaced by Next (or Prev) and nat<a,b>, resp. Remember how the iteration procedure Next (Prev) was defined above. The places *I* and *S* in the nets (Fig. 8) are mappings for the processes defined in the initiation and step phrases in the definition of the procedure.

The first formula calls for the execution of *P* for all *x* from the interval [a+1,b] such that *A* holds while *B* is true. *R* is proved to be invalid only if *P* is found invalid for some *x* from the interval (provided *A* and *B* are both true). In all other cases *R* is proved to be valid. It is valid also if the interval is empty (a = b) (panel 1).

The second formula calls for the execution of *P* for some *x* from [a+1,b] such that *A* holds while *B* is true. *R* is proved to be valid only if *P* is found valid for some *x*. In all other cases *R* is proved to be invalid including if the interval is empty.

2

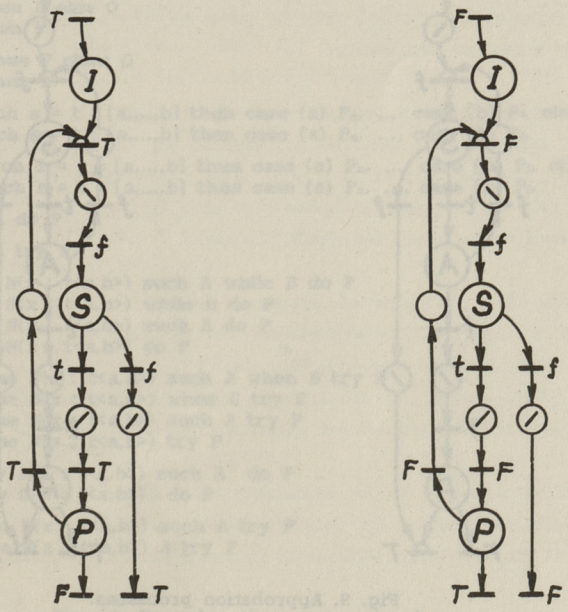


Fig. 8. Definite iteration processes (continued).

Note that the procedure Next causes the ascending of the variable x in the interval [a+1,b] while the procedure Prev causes the descending of the variable in the interval [a,b-1]. Note also that the procedures Next and Prev are defined so (see (9) and (10)) that they become invalid (false) at the end of the iteration process.

The interpretation of the formula (F8) in the case when the phrases with conditions A and B are omitted need no explanation (Fig. 8, panel 2).

Approbation processes. The interpretation of approbation processes by (F9) is explained in Fig. 9. Again, the interpretation becomes easier to understand after replacements mentioned above:

R = for any Next(x : nat<a,b> such A do P,

R = for sole Next(x : nat<a,b> such A try P.

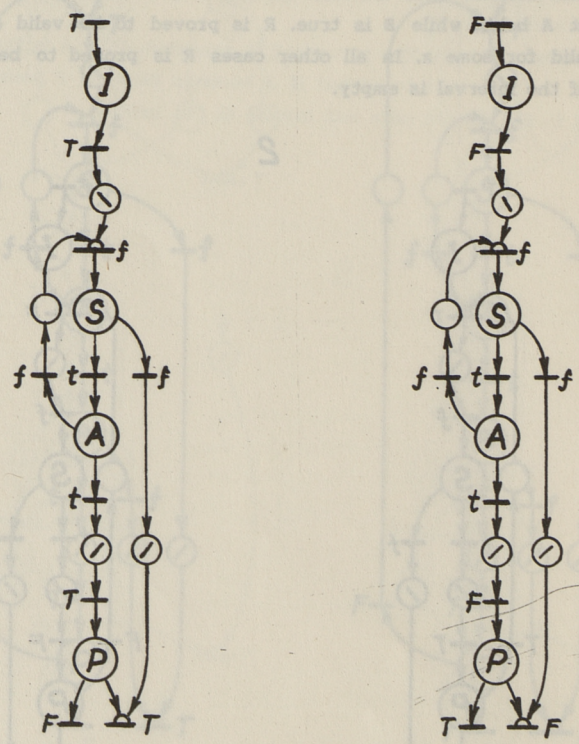


Fig. 9. Approbation processes.

Both formulas call for the execution of *P* for the least *x* (for the greatest *x* in the case of procedure Prev instead of Next) from the interval such that *A* holds. In both cases, *R* is proved to be valid if *P* is found valid for such *x*. A difference appears if there exists no *x* such that *A* holds or the interval is empty. Then the first process is valid while the second is invalid. If the phrase with condition *A* is omitted, then *P* is performed for the first (last) *x* from the corresponding interval.

F1 *P*

F2 **deny** *P*

F3.C **define** *v* : *t* **begin ser** P_1 **and ... and** P_n **end**
begin ser P_1 **and ... and** P_n **end**

F3.D **define** *v* : *t* **begin ser** P_1 **or ... or** P_n **end**
begin ser P_1 **or ... or** P_n **end**

F4.C **define** *v* : *t* **begin par** P_1 **and ... and** P_n **end**
begin par P_1 **and ... and** P_n **end**

F4.D **define** *v* : *t* **begin par** P_1 **or ... or** P_n **end**
begin par P_1 **or ... or** P_n **end**

F5.C **if** *A* **then** *P* **else** *Q*
if *A* **then** *P*

F5.D **as** *A* **thus** *P* **othw** *Q*
as *A* **thus** *P*

F6.C **if switch** *s* = *t* : ⌈*a*,...,*b*⌉ **then case** (*a*) P_a, ... **case** (*b*) P_b **else** *Q*
if switch *s* = *t* : ⌈*a*,...,*b*⌉ **then case** (*a*) P_a, ..., **case** (*b*) P_b

F6.D **as switch** *s* = *t* : ⌈*a*,...,*b*⌉ **thus case** (*a*) P_a, ... **case** (*b*) P_b **othw** *Q*
as switch *s* = *t* : ⌈*a*,...,*b*⌉ **thus case** (*a*) P_a, ..., **case** (*b*) P_b

F7.C **while** *A* **do** *P*

F7.D **when** *A* **try** *P*

F8.C **for all** *N*(*x* : *t*<*a*,*b*>) **such** *A* **while** *B* **do** *P*
for all *N*(*x* : *t*<*a*,*b*>) **while** *B* **do** *P*
for all *N*(*x* : *t*<*a*,*b*>) **such** *A* **do** *P*
for all *N*(*x* : *t*<*a*,*b*>) **do** *P*

F8.D **for some** *N*(*x* : *t*<*a*,*b*>) **such** *A* **when** *B* **try** *P*
for some *N*(*x* : *t*<*a*,*b*>) **when** *B* **try** *P*
for some *N*(*x* : *t*<*a*,*b*>) **such** *A* **try** *P*
for some *N*(*x* : *t*<*a*,*b*>) **try** *P*

F9.C **for any** *N*(*x* : *t*<*a*,*b*>) **such** *A* **do** *P*
for any *N*(*x* : *t*<*a*,*b*>) **do** *P*

F9.D **for sole** *N*(*x* : *t*<*a*,*b*>) **such** *A* **try** *P*
for sole *N*(*x* : *t*<*a*,*b*>) *A* **try** *P*

Fig. 10. Formulas of process calcculus.

VALIDITY, REGULARITY AND EQUIVALENCE

Validity. Validity of all composed processes and predicates defined in the formalism may be expressed by means of formulas of predicate calculus (in the conventional sense and notation). In fact, validity of predicates defined in our formalism may be expressed by means of formulas of proposition calculus using only negation, conjunction and disjunction operations. But to express validity of processes all means of expression of predicate calculus including quantifiers are necessary. At that a generalization is needed - conjunction and disjunction of processes are not commutative operations except those in parallel junctions.

Validity expressions for formulas of process calculus (Fig. 10) are shown in Fig. 11. Note that the expressions for junctions (F.3) - (F.4) do not depend on whether they are serial or parallel. Validity expressions for the clauses are like the expressions for the similar formulas.

Regularity. Let us consider a process net for a formula (Fig. 3 - Fig. 9). Observe the passage of tokens through the main places of the net during the simulation of the formula provided the simulation is started from the initial arrangement of tokens (Fig. 1, panel 4). The passage may be represented by a regular expression.

Regular expressions for the formulas of process calculus (Fig. 10) are shown in Fig. 12. Note that the expressions for the (quasi)dual forms are coincident. The meaning of *C*, *I* and *S* in expressions for formulas (F6), (F8) and (F9) is the same as above (see also Fig. 6, 8 and 9). Expressions for clauses are like these for similar formulas.

Equivalence. Two formulas of process calculus are said to be logically equivalent if their validity expressions (as formulas of predicate calculus) are equivalent. For instance, the formulas (F3.C) and (F4.C) as well as the formulas (F3.D) and (F4.D) are logically equivalent (provided the commutativity of parallel junctions is not taken account of).

Two formulas are said to be eventually equivalent if regular expressions for them (as expressions for regular events) are equivalent. For instance, all C-forms are eventually equivalent to corresponding D-forms.

Two formulas are called equivalent if they are logically and eventually equivalent.

F1. P

F2. $\neg P$

F3.C $\forall(v \in t)(P_1 \& \ldots \& P_n)$
$(P_1 \& \ldots \& P_n)$

F3.D $\exists(v \in t)(P_1 \vee \ldots \vee P_n)$
$(P_1 \vee \ldots \vee P_n)$

F4.C $\forall(v \in t)(P_1 \& .:. \& P_n)$
$(P_1 \& \ldots \& P_n)$

F4.D $\exists(v \in t)(P_1 \vee \ldots \vee P_n)$
$(P_1 \vee \ldots \vee P_n)$

F5.C $(A \rightarrow P) \& (\neg A \rightarrow Q)$
$(A \rightarrow P)$

F5.C $(A \& P) \vee (\neg A \& Q)$
$(A \& P)$

F6.C $\forall(s = t)((s \in \{a,\ldots,b\} \rightarrow P_s) \& (s \notin \{a,\ldots,b\} \rightarrow Q))$
$\forall(s = t)(s \in \{a,\ldots,b\} \rightarrow P_s)$

F6.C $\exists(s = t)((s \in \{a,\ldots,b\} \& P_s) \vee (s \notin \{a,\ldots,b\} \& Q))$
$\exists(s = t)(s \in \{a,\ldots,b\} \& P_s)$

F7.C $\forall n(n < \mu(\neg A) \rightarrow P)$

F7.D $\exists n(n < \mu(\neg A) \& P)$

F8.C $\forall(x \in [a',b])(A \& x < \mu(\neg B) \rightarrow P)$
$\forall(x \in [a',b])(x < \mu(\neg B) \rightarrow P)$
$\forall(x \in [a',b])(A \rightarrow P)$
$\forall(x \in [a',b])(P)$

F8.D $\exists(x \in [a',b])(A \& x < \mu(\neg B) \& P)$
$\exists(x \in [a',b])(x < \mu(\neg B) \& P)$
$\exists(x \in [a',b])(A \& P)$
$\exists(x \in [a',b])(P)$

F9.C $\forall(x \in [a',b])(x = \mu(A) \rightarrow P)$
$\forall(x \in [a',b])(P)$

F9.D $\exists(x \in [a',b])(x = \mu(A) \& P)$
$\exists(x \in [a',b])(P)$

Fig. 11. Validity expressions for formulas.

F1.	P	
F2.	P	
F3.	$P \cup PO$	(Fig. 4)
F4.	$P \cup Q \cup PO$	(Fig. 4)
F5.	$AP \cup AQ$ $A \cup AP$	(Fig. 5)
F6.	$CP_a \cup \ldots \cup CP_b \cup CQ$ $CP_a \cup \ldots \cup CP_b$	(Fig. 6)
F7.	$(AP)^*(A \cup AP)$	(Fig. 7)
F8.	$I((SA)^*(S \cup SAB) \cup ((SA)^*BP)^*(S \cup SAB \cup SABP))$ $I(SBP)^*(S \cup SB \cup SBP)$ $I((SA)^*S \cup ((SA)^*P)^*(S \cup SAP))$ $I(SP)^*(S \cup SP)$	(Fig. 8)
F9.	$I(SA)^*(S \cup SAP)$ $I(S)^*(S \cup SP)$	(Fig. 9)

Fig. 12. Regular expressions for formulas.

Examples.

1. The formulas (F5.C) and (F5.D) as well as the formulas (F6.C) and (F6.D) are equivalent provided the phrases 'else Q' and 'othw Q' are not omitted.

2. The formulas

```
for  all N(x : t<a,b>)                for  some N(x : t<a,b>)
     such A do  P ,                        such A try P
```

are equivalent respectively to the formulas

```
for  all N(x : t<a,b>)                for  some N(x : t<a,b>)
     do if A then P ,                      try as A thus P .
```

3. The formulas

```
for  all Next(x : nat<a,b>)           for  some Next(x : nat<a,b>)
     do P ,                                try P
```

are equivalent respectively to the next formulas

```
define x : nat<a,b>                   define x : nat<a,b>
begin Eqt(x<-a)                       begin deny Eqt(x<-a)
and  while Lt(x,b)                    or   when Lt(x,b)
     do   begin Eqt(x<-sc(x))              try  begin deny Eqt(x<-sc(x))
          and P                                 or P
          end                                   end
end ,                                 end
```

provided the iteration procedure Next is defined by (9).

DERIVATION SCHEMES

Lastly, let us consider briefly how derivation schemes (deduction rules) are introduced into the formalism.

General notes. As it is known, a formal theory, for instance, the theory of (natural) numbers, may be represented as a set of axioms and theorems ordered by the relation 'a theorem *X* is deduced from an axiom or theorem *Y* by a rule of inference'. Axioms represent basic relations in the set of abstract objects to which the theory is applied (numbers in case of number theory) and so formalize their semantics. Theorems express derived relations in the set. The rules of inference (for example, modus ponens) serve to define a subset of formulas interpreted as totally valid ones. They are defined in the metatheory of the theory (Kleene, 1952), describe logical tools used to construct the theory and so formalize conclusion process in it.

Approximately the same principles are used in our approach. We consider a set of processes defined for a data entity type as an analogue to a homogeneous theory (such as number theory). At that, the role of axioms is played by the primitive processes predefined for the type, the role of theorems - by derived processes, and that of deduction rules - by the derivation schemes discussed below.

However, there are some features. Firstly, unlike formalization of theories, in programming, we are interested in derivation of formulas interpreted not only as totally valid but also as partially valid ones. Secondly, all data types are finite classes of objects, i.e. all data entities take a value from a finite domain of values. Thus we can speak immediately of total and partial validity of formulas instead of speaking of their deducibility and irrefutability. Further, semantics of data entities belonging to a data type is represented by the functions defined for the type rather than by primitive processes. Lastly, the logical tools are based on process calculus but not on predicate calculus as they are in all formal theories.

Validity and invalidity. In previous sections, it was shown how the successfulness (validity) of the process defined by a formula may be determined as a result of the performance of the process. Now the problem is how the total or partial validity of a composed formula (process) may be derived from the properties of the component ones, or, in terms of pro-

gramming practice, how the total or partial validity of processes (program modules) may be determined at the compilation time.

A formula of process calculus is called partially valid (invalid) if it is valid (resp. invalid) for some value of its argument and state variables, and totally valid (invalid) if it is valid (invalid) for all values of its argument and state variables. A formula is called neutral if it is partially valid and partially invalid simultaneously, or, what is the same, if it is not totally valid and totally invalid (not totally valid = partially invalid, not totally invalid = partially valid).

Surely, it is natural to speak of the validity or invalidity of composed formulas depending on the validity or invalidity of component formulas (processes) only, i.e. discarding component clauses (predicates).

Derivation schemes. Proceeding from the validity expressions of composed processes discussed in the previous section it is easy to understand, for instance, that all composed (not inverse) C-forms are totally valid formulas and all composed D-forms are totally invalid ones provided the component formulas are of that kind. Further, it is clear that the formulas (F7.C) - (F9.C) are partially valid and the formulas (F7.D) - (F9.D) are partially invalid independently of the validity of the only component formula (P). These simple conclusions are not exceptional. The derivation schemes presented below are all based on such elementary facts.

To determine the class a particular formula belongs to (totally valid, totally invalid or neutral) two different schemes are assigned to it. To express them we introduce four metapredicates (F denotes a formula):

PVAL{F} = 'F is partially valid', PINV{F} = 'F is partially invalid',
TVAL{F} = 'F is totally valid', TINV{F} = 'F is totally invalid'.

Now derivation schemes may be represented as follows.

F1. TVAL{P} / TVAL{P} TINV{P} / TINV{P}

F2. TVAL{P} / TINV{deny P} TINV{P} / TVAL{deny P}

F3-F4.

```
TVAL{P1(y)}  and
     ...     and
TVAL{Pn(y)}
-----------------
TVAL{define v : t
      begin  P1(v)
      and    ...
      and    Pn(v)
      end}

TINV{P1(y)}  or
     ...     or
TINV{Pn(y)}
-----------------
TINV{define v : t
      begin  P1(v)
      and    ...
      and    Pn(v)
      end}
```

```
TINV{P1(y)}  and
     ...     and
TINV{Pn(y)}
-----------------
TINV{define v : t
      begin  P1(v)
      or     ...
      or     Pn(v)
      end}

TVAL{P1(y)}  or
     ...     or
TVAL{Pn(y)}
-----------------
TVAL{define v : t
      begin  P1(v)
      or     ...
      or     Pn(v)
      end}
```

F5.1

```
TVAL{P} and TVAL{Q}
-----------------------
TVAL{if A then P else Q}

TINV{P} and TINV{Q}
-----------------------
TINV{if A then P else Q}
```

```
TINV{P} and TINV{Q}
-----------------------
TINV{as A thus P othw Q}

TVAL{P} and TVAL{Q}
-----------------------
TVAL{as A thus P othw Q}
```

F5.2

```
PVAL{if A then P}

TVAL{P}
-----------------
TVAL{if A then P}
```

```
PINV{as A thus P}

TINV{P}
-----------------
TINV{as A thus P}
```

F6.1. ...

F6.2. ...

F7.

```
PVAL{while A do P}

TVAL{P}
-----------------
TVAL{while A do P}
```

```
PINV{when A try P}

TINV{P}
-----------------
TINV{when A try P}
```

F8.

```
PVAL{for all N(t : ...)
      such A(t) while B(t)
      do P(t)}

TVAL{P(x)}
-----------------------
TVAL{for all N(t : ...)
      such A(t) while B(t)
      do P(t)}
```

```
PINV{for some N(t : ...)
      such A(t) when B(t)
      try P(t)}

TINV{P(x)}
-----------------------
TINV{for some N(t : ...)
      such A(t) when B(t)
      try P(t)}
```

F9. PVAL{for any N(t : ...)
such A(t)
do P(t)}

PINV{for sole N(t : ...)
such A(t)
try P(t)}

TVAL{P(x)}

TVAL{for any N(t : ...)
such A(t)
do P(t)}

TINV{P(x)}

TINV{for sole N(t : ...)
such A(t)
try P(t)}

Notes. P(x), P(y) denote a formula which may (but does not have to obligatorily) contain a free argument variable x, or a free argument, state or result variable y. Schemes for conjunctions and disjunctions of processes do not depend on whether the junctions are serial or parallel ones. Schemes for the formulas (F6.C) and (F6.D) are not shown. They are generalizations of those for the formulas (F5.C) and (F5.D), resp.

Simple processes. We have shown how the total validity and invalidity (and also neutrality) of composed formulas are determined in virtue of those for component formulas. If the component formulas are themselves composed the properties for them may be determined, in turn, in the same manner, and so on up to elementary formulas denoting simple processes.

A simple process represents a process module previously defined in the language and stored in the semantical memory of the system for further use. As a matter of fact, such process is simple (elementary) only from the point of view of the process in definition of which it is denoted by an elementary formula. Actually, it may be a composed process. We call such processes nonprimitive. They play the same role in CP-system as already proved theorems in the argument of new theorems in construction of formal theories.

Primitive processes. A simple process may also be purely elementary, i.e. an elementary process that cannot be defined in the language. It is introduced as a predefined process. Such processes are called primitive. As already mentioned, they are analogues to axioms in formal theories.

Actually, only one primitive process is necessary for each data type: the assignment process. It may be introduced as Asn(x->y) (assign x to y) or Eqt(y<-x) (equate y with x). The process is evident to be totally valid (provided x and y belong to one and the same data type).

The assignment (or equation) process plays the same role in process calculus as the equality relation Eq(x,y) in predicate calculus. In particular, we can speak of reflexivity, symmetry and transitivity of the process: Asn(x->x), Asn(x->y) => Asn(y->x), Asn(x->y) & Asn(y->z) => Asn(x->z).

CONCLUSION

We have introduced the notion of abstract process and presented it through a logical function with a side effect. The notion means simultaneously an algorithm (program) and the computing process defined by the algorithm. The main effect of the logical function represents the applicability of the algorithm (the validity of the process) while the side effect is related to the data processing defined by the algorithm.

We have also discussed the main elements of process calculus: logical operations on processes, application of quantifiers and descriptors to the arguments of processes, association of predicates and processes, etc. We have shown that all traditional program control statements as well as several new statements may be interpreted as formulas of process calculus.

The semantics of the formulas were interpreted by means of a special kind of Petri nets called process nets as well as of validity and regularity expressions. In addition, deduction schemes for the derivation of totally and partially valid (invalid) formulas were discussed. So the calculus was presented as a general and universal instrument for description and analysis of logic in programming and also as a high level programming language

In this paper we have been able to discuss only some basic ideas of our approach to the formalization of programming - ideas of constructive programming. They have chiefly been connected with programming logic. Many other fundamental problems have been neglected, first of all, formation of abstract data types (concepts). All special items, for instance, such as synchronization of parallel processes and data communication between them related to the real time applications, have also been omitted.

Process calculus forms a new, strict theoretical basis for the treatment of logical correctness and reliability in programming. This is essential from the point of view of programming real time, distributed computer control and embedded systems.

REFERENCES

Bacus, J. (1978). Can Programming be Liberated from the von Neumann Style. A Functional Style and Its Algebra of Programs. *Comm. ACM, 21, 8, 613-641.*

Church, A. (1956). *Introduction to Mathematical Logic, I.* Princeton, New Jersey.

Fisher, D. A. (1978). DoD's Common Programming Language Efforts. *Computer, March, 1978, 25-30.*

Fileman, R. E. and D. P. Friedman (1984). *Coordinated Computing, Tools and Techniques for Distributed Software.* McGrew-Hill.

Henderson, P. (1980). *Functional Programming. Application and Implementation.* Prentice-Hall, Englewood Cliffs, N.J.

Hilbert, D. and P. Bernays (1968). *Grundlagen der Mathematik, I.* 2nd ed. Springer, Berlin.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall International, London.

Kleene, S. C. (1952). *Introduction to Metamathematics.* Van Nostrand, New York.

Kowalski, R. (1979). Algorithm = Logic + Control. *Comm. ACM, 22, 7, 424-436.*

Lloyd, J. W. (1985). *Foundations of Logic Programming.* Springer, Berlin.

Smith, D. J. and K. B. Wood (1987). *Engineering Quality Software.* Elsevier Applied Science, London.

Young, S. J. (1982). *Real Time Languages: Design and Development.* Horwood, New York.

W. Kracht

Tõevääärtusega arvutusprotsess ja protsessiarvutus: formalism programmeerimisloogika kirjeldamiseks

Kokkuvõte

Töös tutvustatakse uut lähenemist programmeerimisloogika kirjeldamiseks ja kõrgkeelte arendamiseks. Lähenemine põhineb tõevääärtusega arvutusprotsessi mõistel ja sellele mõistele rajatud protsessiarvutusel. Sellist protsessi mõistetakse kui kõrvalefektiga loogilist funktsiooni. Temaga väljendatakse mingi andmeteisenduse algoritmi rakendatavust ja seda teisendust ennast. Protsessidele rakendatakse loogilisi tehteid ja kvantoreid. Eristatakse protsesside jada- ja rööpkonjunktsioone ja -disjunktsioone. Protsessiarvutuse valemitena interpreteeritakse kõiki programmeerimises tuntud juhtimislauseid, samuti ka uusi, töös defineeritud juhtimislauseid. Protsessiarvutuse valemite semantika esitatakse eri liiki Petri võrkude abil, samuti avaldistega, mis esitavad nende kehtivust (tõevääärtust) ja regulaarsust. Tuuakse deduktsiooniskeemid, mis määravad täielikult ja osaliselt kehtivate ja ka mittekehtivate valemite klassid. Protsessiarvutus pakutakse välja kui teoreetiline baas reaalaja programmeerimiskeelte edasiarendamiseks.

Таллиннский технический университет
Труды ТТУ № 722
МОДЕЛИРОВАНИЕ И АНАЛИЗ ПРОЦЕССОВ И ЦЕПЕЙ
Электротехника и автоматика XL
Отв. редактор Э. Калм
На английском языке

Trükkida antud 15.12.90. Formaat 60x90/16. Trükipg. 6,0+0,25 (lisa)
Arvestuspg. 4,73. Trükiarv 300. Tellimuse nr. 176/91
Hind 2 rbl. 90 kop.
Tallinna Tehnikaülikool, 200108 Tallinn, Ehitajate tee 5
TTÜ rotaprint, 200006 Tallinn, Koskla 2/9

Hind rbl. 2.90