Modelling of Complex Software Systems: a Reasoned Overview *

D. Krob

Laboratoire d'Informatique de l'Ecole Polytechnique (LIX)
CNRS & École Polytechnique **

Abstract. This paper is devoted to the presentation of the key concepts on which a mathematical theory of complex (industrial) systems can be based. We especially show how this formal framework can capture the realness of modern information technologies. We also present some new modelling problems that are naturally emerging in the specific context of complex software systems.

Keywords – Complex system; Information system; Integrated system; Modelling; Software system.

This paper is dedicated to the memory of M.P. Schützenberger

1 Introduction

In the modern world, complex industrial systems are just everywhere even if they are so familiar for us that we usually forgot their underlying technological complexity. Transportation systems (such as airplanes, cars or trains), industrial equipments (such as micro-electronic or telecommunication components) and information systems (such as commercial, production, financial or logistical software systems) are for instance good examples of complex industrial systems that we are using or dealing with in the everyday life.

At a superficial level, "complex" refers here to the fact that the design and the engineering of these industrial systems are incredibly complicated technical and managerial operations. Thousands of specialized engineers, dozens of different scientific domains and hundreds of millions of euros can indeed be involved in the construction of such systems. In the automobile industry, a new car project lasts for instance typically 4 years, requires a total human working effort of more than 1.500 years, involves 50 different technical fields and costs around 1 billion of euros ! In the context of software systems, important projects have also the same kind of complexity. Recently the unification of the information systems of two important French financial companies that merged, needed for example 6 months of preliminary studies followed by 2 years of work for a team of 1.000 computer specialists, in order to rebuild and to mix consistently more than 250 different business applications, leading to a total cost of around 500 millions euros.

At a deeper level, complex industrial systems are characterized by the fact that they are resulting of a complex *integration process* (cf. [38, 39] for more details). This means that such systems are obtained by integrating in a coherent way – that is to say assembling through well defined interfaces – altogether a tremendously huge number of heterogeneous sub-systems and technologies, that belong in practice to the three following main categories:

1. *Physical systems:* these types of systems are manipulating and transforming *physical quantities* (energy, momentum, etc.). The hardware components of transportation, micro-electronic or telecommunication systems are for instance typical physical systems.

* This paper was supported by the Ecole Polytechnique and Thales' chair "Engineering of complex systems".
** Address: Ecole Polytechnique – LIX – 91128 Palaiseau Cedex – France – email: dk@lix.polytechnique.fr – Web site: http://www.lix.polytechnique.fr/∼ dk

2. *Software systems:* these systems are characterized by the fact that they are managing and transforming *data*. Operating systems, compilers, databases, Web applications and Business Intelligence (BI) systems are classical examples of software systems.
3. *Human systems:* human organizations [1] can be considered as systems as soon as their internal processes have reached a certain degree of normalization. They will then be identified to the business processes that are structuring them.

Note at this point that the difficulty of integrating coherently the different parts of a complex industrial system reflects of course in the difficulty of integrating coherently the heterogeneous formal and informal models – going from partial differential equations and logical specifications to business process modelling (BPM) methods (cf. [11]) – that one must handle in order to deal globally with such systems. There is in particular still no real formal general models that can be used for dealing with complex industrial systems from a global point of view. This lack can also be seen in the fact that there are no unified tools for managing all the aspects of the realization cycle of an industrial complex system (which goes from the analysis of needs and the specification phase up to the final integration, verification, validation and qualification processes).

More generally, one must clearly face a huge lack of theoretical tools that may help to clarify the question of complexity in practice. Very few research works are for instance studying directly "heterogeneous" systems *in their whole*, though a rather important research effort has been done during the last decades to understand better several important families of homogeneous systems (such as Hamiltonian systems, dynamical systems, embedded systems, distributed systems, business organizations, etc.) which are involved within larger industrial systems. The key point is here to understand that the problematics are absolutely not the same if one studies a complex industrial system at local levels (the only ones that the classical approaches are addressing) and at a global level. We however believe that the existing formal "local" theoretical frameworks can and should be redeployed to analyze complex industrial system at a holistic level.

An interesting fact that militates in favor of the possibility of progressing in these directions is the convergence, that can be currently observed in the industry, between the approaches used for managing the engineering phases [2] of physical and of software systems. This convergence can in particular be seen at a methodological level since system engineering (see [47, 55]) and software engineering (see [48, 51]) are more or more expressing their methods in the same way, but also at the level of the architectural principles used in physical and software contexts (see [33]) and of the quasi-formal specifying and modelling tools that are now taking into account both physical and software frameworks (cf. for instance [8, 53] for the description of SysML that extends the classical Unified Modelling Language (UML) – [46] – for general systems).

The purpose of this short paper is to make a reasoned overview on what could be a general theory of systems. After some preliminaries, we therefore present in Section 3 a tentative formal framework, for approaching in a mathematical way the notion of "complex industrial system", that tries to capture the realness both of these systems and of their engineering design processes (which are very difficult to separate in practice). Section 4 is then devoted both to the analysis of the modern software industrial ecosystem using the analysis grid provided by our approach and to the illustration of new types of research problems – of practical interest – that are naturally emerging from this new point of view on complex software systems.

[1] One must obligatory take into account these non technical systems in the modelling of a global system as soon as the underlying human organizations are strongly interacting with its physical and/or software components. This situation occurs for instance naturally in the context of complex software systems (see Section 4).

[2] I.e. design, architecture, integration and qualification processes.

2 Preliminaries

As in the few previous attempts to discuss globally of systems (see for instance [14, 50, 59]), these objects will be defined here as mechanisms that are able to receive, transform and emit physical and/or informational quantities among time. This explains why we will first introduce two key definitions on which are respectively based time and quantity modelling in our approach.

2.1 Time scales

A *time scale* $\mathbb{T}$ refers to any mode of modelling all the possible moments of time starting from some initial moment $t_0 \in \mathbb{R}$. Time scales can be of two different kinds, i.e. continuous or discrete. The *continuous* time scales are of the form $\mathbb{T} = t_0 + \mathbb{R}^+$. One has more various *(regular) discrete* time scales which are of the form $\mathbb{T} = t_0 + \mathbb{N}\,\tau$ where $\tau \in \mathbb{R}_*^+$ denotes their *time step*. One can consider as well *irregular discrete* time scales that are of the form $\mathbb{T} = \{\, t_0 + \tau_1 + \cdots + \tau_n,\ n \in \mathbb{N} \,\}$ where $(\tau_i)_{i\in\mathbb{N}}$ is a given family of strictly positive real numbers. Note finally that the above time scales were always *deterministic*, but that they could also be *probabilistic* if the parameters involved in their definitions are random variables with given probabilistic laws.

2.2 Quantity spaces

A *quantity space* refers to any mode of modelling either physical quantities (like energy, temperature, speed, position, etc.) or informational quantities (that is to say data in the usual computer science meaning). There are therefore two types of quantity spaces, i.e. continuous and discrete ones. On one hand, a *continuous* quantity space can be any complete topological space such as $\mathbb{R}^n$ or $\mathbb{C}^m$. On the other hand, a *discrete* quantity space is either any finite set or a discrete infinite set such as $\mathbb{N}^n$, $\mathbb{Z}^m$ or A^* (where A stands for any finite alphabet). Note finally that a quantity space Q must also always distinguish a special element – called the *missing quantity* – that represents the absence of quantity (which is typically 0 is Q is a subset of $\mathbb{C}$).

3 Complex Systems

3.1 Abstract systems

In order to move towards the formal modelling of *complex industrial systems*, let us introduce a first notion of system, identified here to an input/output behavior.

Definition 1. *An* abstract system $\mathcal{S}$ *is defined as a* 5*-uple* $\mathcal{S} = (\mathbb{I}, \mathbb{O}, \mathbb{T}_i, \mathbb{T}_o, \mathcal{F})$ *where*

- $\mathbb{I}$ *and* $\mathbb{O}$ *are two quantity spaces respectively called the* input *and* output spaces *of* $\mathcal{S}$*,*
- $\mathbb{T}_i$ *and* $\mathbb{T}_o$ *are two time scales, respectively called the* input *and* output time scales *of* $\mathcal{S}$*,*
- $\mathcal{F}$ *is a function from* $\mathbb{I}^{\mathbb{T}_i}$ *into* $\mathbb{O}^{\mathbb{T}_o}$ *which is called the* transfer function *of* $\mathcal{S}$*.*

Observe that the discrete or continuous structure of the input and output spaces and of the input and output time scales defines naturally different kinds of abstract systems in the meaning of Definition 1. To illustrate and understand better this last definition, let us now study several examples of such systems that are distinguished according to this new criterium.

Example 1. – Discrete systems – An abstract system will said to be *discrete* when its input and output time scales are discrete. Discrete abstract systems can for instance easily be described by means of finite automaton modelling approaches which capture quite well the event reacting

dimension of a real system. These types of formalisms all basically rely on the use of rational transducers – or equivalently Mealy machines – (cf. [3, 37]) for expressing the transfer function of a discrete system. Figure 1 shows a simple example of this kind of formalism for modelling the effect of the switch button of a lamp on its lighting performance. The input space of the abstract system that models the lamp is here $\mathbb{I} = \{\rho, \pi\}$ (where ρ and π respectively model the fact that the switch button is released or pressed – note that ρ stands therefore for the empty quantity of $\mathbb{I}$) when its output space is $\mathbb{O} = \{0, e\} \subset \mathbb{R}^+$ (an output real value represents the amount of energy produced by the lamp). The corresponding input and output time scales can finally here be any discrete regular time scales that are relevant with respect to the modelling purposes (they are both re-normalized to $\mathbb{N}$ in our example for the sake of simplicity). Note in particular that the discrete structure of the output time scale is not a problem as soon as we are only interested by a computer model for simulation purposes (one just has to take a sufficiently small output time step). We will revisit this example in the sequel, especially to see how to get more realistic models with respect to the lamp real physical behavior (see Examples 2 and 3).

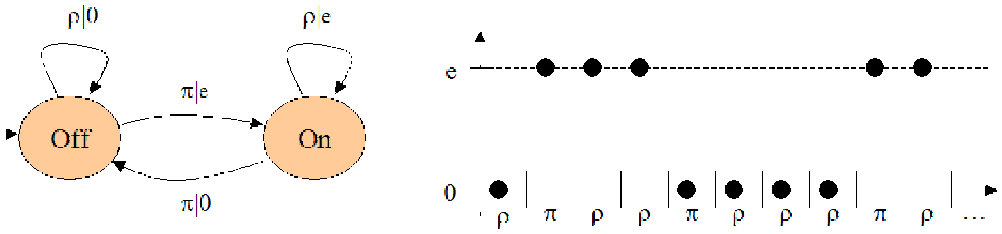

Fig. 1. *Modelling the transfer function of a lamp by a rational transducer.*

Petri nets (cf. [42, 45]), (min−max, +) systems (cf. [4]), Kahn networks (cf. [27]), etc. are other examples – among numerous others – of basic automaton-oriented formalisms that can be used (with slight modifications) for describing discrete abstract systems in our meaning.

There is also a purely logical approach for representing discrete abstract systems. The core modelling language in this direction is probably Lustre (cf. [22, 13]). This programming language is indeed structurally devoted to express transformations of typed infinite sequences. The Lustre program that models the (simple) behavior of our lamp is for instance given below.

```
node Lamp(X:bool) returns (Y:real);
var E,Z:real;
let
E = e -> pre E;
Z = 0 -> pre Z;
Y = if X then E else Z;
tel
```

Fig. 2. *Modelling the transfer function of a lamp by a Lustre program.*

In this example, X stands for the infinite sequence of boolean entries a lamp receives (`false` and `true` modelling respectively the fact that the switch button of the lamp is released or pressed) when Y represents the infinite sequence of the energy levels that can take the lamp (either 0 or e). The E and the Z variables are then just used here for defining the constant infinite real sequences $E = (e, e, e, \dots)$ and $Z = (0, 0, 0, \dots)$. The last line of the program expresses finally

that the n-th entry of Y is equal to the n-th entry of E (resp. Z), i.e. to e (resp. 0), when the n-th entry of X is true (resp. false), i.e. when the button switch is pressed (resp. released), which models correctly the expected behavior of the lamp (initially switched off) we considered.

Other reactive languages such as Signal (see [32]) or Lucid (see [12]) are using too the same global flow manipulating approach. Note that one can of course also take any usual formal specification language such as B (cf. [1, 57]), TLA+ (cf. [31]) or Z (cf. [52]), any modelling tool coming from the model checking approach (cf. [6, 49]) or even any classical programming language, to describe the step-by-step behavior of an abstract system by a "logical" formalism.

Example 2. Continuous systems – An abstract system will said to be *continuous* when its input and output time scales are continuous. Since continuous systems occur naturally in physical contexts, all the various continuous models coming from control theory, physics, signal processing, etc. can therefore be used to represent continuous systems (see [14, 50] for more details). These models rely more or less all on the use of (partial) differential equations which can be seen as the core modelling tool in such a situation. Going back again to the lamp system considered in Example 1, one can easily see that the lamp behavior can now for instance be modelled by a continuous signal $y(t)$ – giving the value of the lamp energy at each moment of time t – that respects an ordinary differential equation of the following form:

$$y'(t) = e \times x(t) - k \times y(t), \qquad y(0) = 0, \tag{1}$$

where $x(t)$ stands for a continuous $\{0, 1\}$-signal that represents the behavior of the button switch – $x(t)$ being equal to 0 (resp. 1) at time t iff the button switch is off (resp. on) at this moment of time – and where $k > 0$ is a real parameter which models the speed of reactivity of the lamp light to the opening/closing of the button switch. Figure 3 shows then the result (on the right side) of such a modelling (obtained here with $e = 2$ and $k = 1$) for a given $\{0, 1\}$-input signal (on the left side). Note that this model shows clearly the continuous initial or final evolutions of the energy of the lamp when the button is switched on or off (which are of course not immediate in reality as it was expressed in the discrete models considered in Example 1).

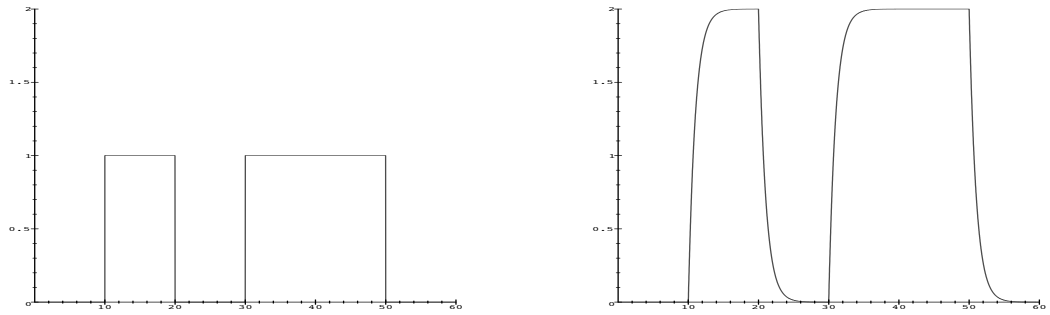

Fig. 3. *Modelling the physical behavior of a lamp by a differential equation.*

Mathlab and Simulink are the typical software tools that can be used for designing continuous systems (see [36]). Observe also that specific frameworks exist for dealing with several important families of continuous systems such as dynamical systems (cf. respectively [28] and [18] for the physical and the control theory point of views), Hamiltonian systems (cf. [40]), etc.

Example 3. Hybrid systems – An abstract system will said to be *hybrid* when one of its input or output time scales is discrete when the other one is continuous. It is interesting to know that

two types of approaches exist for studying hybrid systems, depending respectively whether one stresses on the discrete (see for instance [2, 24]) or the continuous point of view (see for instance [58]) with respect to such systems. However hybrid systems will of course always be represented by hybrid formalisms that mix discrete and continuous frameworks. Hybrid automata (see [24]) are for instance classical models for representing abstract hybrid systems – in our meaning – with a discrete input time scale and a continuous output time scale (but not for the converse situation, which shows that our hybrid systems *must not be mixed up* with these last systems). Figure 4 shows an hybrid automaton that models the physical behavior of the lamp which was already considered in the two previous examples. Three modes of the lamp are modelled here: the "Init" mode (the switch button was never touched), the "On" mode (the lamp was switched on at least once) and the "Off" mode (the lamp was switched off at least once). The states corresponding to these three different modes contain then the three generic evolution modes – modelled by ordinary differential equations (see Figure 4) – of the continuous signal $y(t)$ that represents the output lamp energy at each moment of time t, taking here again the notations of Example 2. On the other hand, the inputs are here just sequences of π and ρ (that model respectively the fact that the switch button of the lamp is either pressed or released).

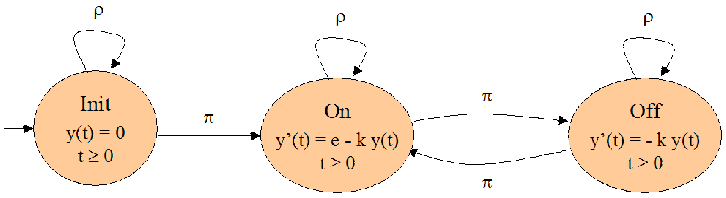

Fig. 4. *Modelling the physical behavior of a lamp by an hybrid automaton.*

Other families of hybrid formalisms – in our meaning – can be typically found in signal processing (see [43]) for modelling demodulation or sampling (transformation of a continuous signal into a discrete one) and modulation (transformation of discrete signal into a continuous one). These last formalisms are radically different from the previous one since they are all based on complex analysis (i.e. z or Laplace transforms) or on distribution theory (see again [43]).

Example 4. Non functional properties are functional ... – Let us now focus on how to express some engineering oriented system aspects. The key point we would like to stress is the fact that the classical non functional properties of a real system – that is to say response times, costs, delays of realization, availability, safety, quality of service, etc. – can easily be modelled by transfer functions in our framework. A non functional property $\mathcal{N}$ of a system can indeed typically always be measured either by some suited numerical indicator $f_N(t)$ or by an adapted boolean predicate $P_N(t)$ (see for instance [60]), depending on internal parameters of the considered system, that can be measured at each moment t of the input time. Such non functional properties can then be expressed in our framework by extending the output space of the underlying system in order to send out the corresponding indicator or predicate values.

Note finally that the "real" systems that can be found in practice form only a "small" subfamily of the abstract systems covered by Definition 1 (which was only given here in such a generality for the sake of simplicity). One may found in [29, 30] a formal definition of a "good" global more restricted family of abstract systems that tries to capture the full realness of systems, using a Turing machine type formalism mixed with non standard analysis (cf. [16]) for taking into account the continuous and discrete dimensions of systems in the same framework.

3.2 Abstract integration

Up to now, we only focused on "simple" models for dealing with systems. Quite all these models are however not really very well adapted for describing hierarchical systems, i.e. systems – in our meaning – that are recursively defined as a coherent interfacing – i.e. an integration – of a collection of sub-systems of different nature. Very surprisingly, while there is a large modelling diversity for usual systems (as we saw in the previous subsection), it indeed appears that there are only a few models that support *homogeneous hierarchical* design (the key difficulty being to be able to take into account both quantities and temporal hierarchies) when the formal models that support *heterogeneous hierarchical* design are even less (to our knowledge, the only framework which handles this last situation is SysML – see [53] – which remains a rather informal modelling approach). We will therefore devote this new subsection to introduce the key concepts on which system integration rely. To this purpose, let us first define the notion of abstract multi-system that extends slightly the notion of system introduced in the previous section.

Definition 2. *An* abstract (n,m)-system $\mathcal{S}$ *is defined as a* 5*-uple* $\mathcal{S}=(\mathcal{I},\mathcal{O},\mathcal{T}_i,\mathcal{T}_o,\mathcal{F})$ *where*

- $\mathcal{I}=(\mathbb{I}_k)_{k=1\ldots n}$ *and* $\mathcal{O}=(\mathbb{O}_l)_{l=1\ldots m}$ *are two families of quantity spaces, whose direct products are respectively called the* input *and* output spaces *of* $\mathcal{S}$*,*
- $\mathcal{T}_i=(\mathbb{T}_i^k)_{k=1\ldots n}$ *and* $\mathcal{T}_o=(\mathbb{T}_o^l)_{l=1\ldots m}$ *are two families of time scales, whose direct products are respectively called the* input *and* output time scales *of* $\mathcal{S}$*,*
- $\mathcal{F}$ *is a function from* $\prod_{k=1}^{n}\mathbb{I}_k^{\mathbb{T}_i^k}$ *into* $\prod_{l=1}^{m}\mathbb{O}_l^{\mathbb{T}_o^l}$ *which is called the* transfer function *of* $\mathcal{S}$*.*

This last definition just expresses that a (n,m)-system – or equivalently a *multi-system* – has several different typed and temporized input and output mechanisms (see Figure 5 for an example of "hybrid" multi-system with a mix of discrete and continuous input and output time scales). Note also that systems in the meaning of Definition 1 are now $(1,1)$-multi-systems.

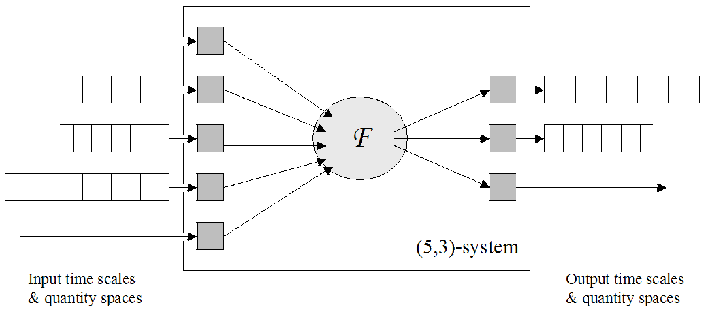

Fig. 5. *Schematic description of a* (n,m)*-system.*

Multi-systems can easily be composed, using typed and temporized *interaction channels*, in a way that reflects the realness of system integration. An interaction channel stands for a medium between an output O and an input I of two multi-systems that can only transmit quantities of a given quantity space, at a time rate associated with some fixed time scale and with a constant temporal delay (for bringing a quantity from O to I). This leads us to the following definition.

Definition 3. *An* interaction channel *is a triple* $\mathcal{C}=(Q,\mathbb{T},\tau)$ *where* Q *is a quantity space,* $\mathbb{T}$ *is a time scale (of initial moment* t_0*) and* $\tau\in\mathbb{T}-t_0$ *is a transmission delay.*

Multi-system composition makes only sense in the context of interacting system networks, another important notion that is defined below (see Figure 6 for a graphical vision).

Definition 4. *An* interacting system network $\mathcal{N}$ *is a triple* $\mathcal{N} = (\mathcal{S}, \chi, \mathcal{C})$ *where*

- $\mathcal{S} = (\mathcal{S}_i)_{i=1...N}$ *is a family of multi-systems,*
- $\chi : C^O \longrightarrow C^I$ *is a bijective mapping between a subset of the output indices of* $\mathcal{S}$ *into a subset of the input indices of* $\mathcal{S}$ [3]*,*
- $\mathcal{C} = \{ (\mathbb{OI}^c, \mathbb{T}^c_{io}, \tau^c),\ c \in C^O \}$ *is a family of interaction channels indexed by* C^O*,*

such that the k*-th output and* l*-th input quantity spaces and times scales of* $\mathcal{S}_i$ *and* $\mathcal{S}_j$ *are always equal – respectively to* $\mathbb{OI}^c$ *and* $\mathbb{T}^c_{io}$ *– for every* $c = (i,k) \in C^O$ *and* $(j,l) = \chi(c) \in C^I$*.*

The input and output indices of $\mathcal{S}$ that belong (resp. do not belong) to C^I or to C^O (with the above notations) are called *constrained* (resp. *free*) input or output indices within $\mathcal{S}$. Note also that an interacting system network will said to be *initialized* if it is equipped with a *initialization map* ι that associates with each constrained input index $c = (i,k) \in C^I$ of the underlying family $\mathcal{S}$ a quantity $q = \iota(c) \in \mathbb{I}^i_k$ where $\mathbb{I}^i_k$ is the k-th input quantity space of the i-th system of $\mathcal{S}$.

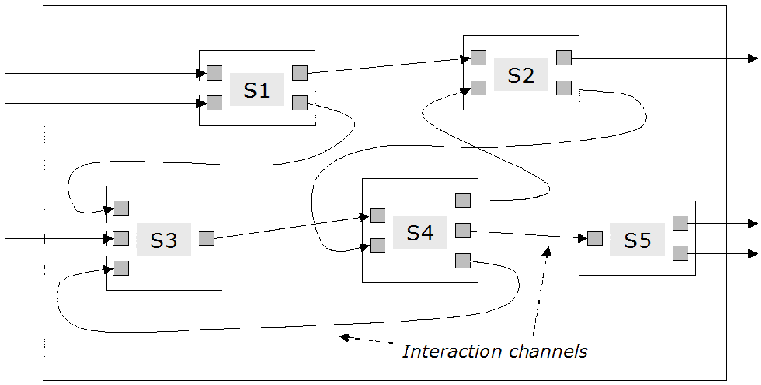

Fig. 6. *Example of an interacting system network.*

Since we will be obliged for technical reasons to restrict composition to specific types of multi-systems defined topologically, let us equip any *flow space* $\mathbb{F}$, i.e. any set of the form $\mathbb{F} = \mathbb{I}^{\mathbb{T}}$ where $\mathbb{I}$ and $\mathbb{T}$ stand respectively for a quantity space and a time scale, with a sequential topology. We will indeed say that a sequence $(x_i)_{i \geq 0}$ of *flows* of $\mathbb{F} = \mathbb{I}^{\mathbb{T}}$, i.e. of elements of the form $x_i = (x_i^t)_{t \in \mathbb{T}} \in \mathbb{F}$, has a limit $x = (x^t)_{t \in \mathbb{T}}$ in $\mathbb{F}$ iff for every $t \in \mathbb{T}$, x_i^t is always equal to x^t for i big enough. A multi-system is then said to be *continuous* if its transfer function $\mathcal{F}$ satisfies

$$\mathcal{F}(\lim_i x_i) = \lim_i \mathcal{F}(x_i)\ , \tag{2}$$

for any sequence $(x_i)_{i \geq 0}$ of input multi-flows that has a limit in the previous meaning (naturally extended to products of flows). We can now introduce the notion of system composition, which is a bit tedious to define properly, but that is easily graphically depicted (see again Figure 6).

Proposition 1. *Let* $\mathcal{N} = (\mathcal{S}, \chi, \mathcal{C})$ *be an interacting system network constructed over a family* $\mathcal{S}$ *of continuous multi-systems which is equipped with an initialization map* ι*. One defines then a new continuous multi-system* $S = (I, O, T_i, T_o, F)$ *– called the* composition *of* $\mathcal{S}$ *through the interactions* $\chi \times \mathcal{C}$ *with initialization* ι *– by setting:*

[3] (i,k) is an input (resp. output) index within $\mathcal{S}$ iff $\mathcal{S}_i$ has an k-th input (resp. output) space.

– *I and T_i are respectively the families of all input quantity spaces and time scales that are associated with free input indices within $\mathcal{S}$ (whose set will be denoted by F^I),*
– *O and T_o are respectively the families of all output quantity spaces and time scales that are associated with free output indices within $\mathcal{S}$ (whose set will be denoted by F^O),*
– *the function F associates with any possible input multi-flow $x = (x_c)_{c\in F^I}$ an output multi-flow $y = (y_c)_{c\in F^O}$ which is defined for each $c = (i,k) \in F^O$ by setting*

$$y_c = \mathcal{F}_i^k(X_{\chi^{-1}(i,1),(i,1)}, \ldots, X_{\chi^{-1}(i,N_i),(i,N_i)})$$ [4]

(N_i denoting here the number of inputs of $\mathcal{S}_i$), where $X = (X_{\chi^{-1}(c),c})_{c\in C^I\cup F^I}$ is the smallest [5] *solution of the equational system with flow variables* [6] *defined by setting*

$$\begin{cases} X_{(0,0),c} = x_c \text{ for } c \in F^I, \\ X^t_{\chi^{-1}(c),c} = \iota(c) \text{ for } c \in C^I \text{ and } t \in [t_0, t_0 + \tau^c[\cap \mathbb{T}^i_k\,, \\ X^t_{\chi-1(c),c} = \mathcal{F}_i^k(X_{\chi^{-1}(i,1),(i,1)}, \ldots, X_{\chi^{-1}(i,N_i),(i,N_i)}))^{t-\tau^c} \text{ for } c \in C^I \text{ and } t \geq \tau^c \in \mathbb{T}^i_k\,, \end{cases}$$

where we put $c = (j,l)$ and $\chi^{-1}(c) = (i,k)$ in all these last relations.

Proof. The proof follows by using a classical argument of complete partial order theory (cf. [21]). Note that our result can be also seen as an extension of a classical result of Kahn (see [27]). □

This proposition translates now immediately in the following definition which gives a formal and precise meaning to the notion of system integration.

Definition 5. *A (continuous) multi-system will said to be an* integrated abstract multi-system *if it results of the composition of a series of other multi-systems.*

Integration leads naturally to the *fundamental design mechanism* for systems which consists in analyzing recursively any system as an *interfacing* of a series of sub-systems. This design process is quite simple to understand (in software context, it just reduces to the usual top-down design methodology), but rather difficult to realize in practice for complex heterogeneous systems (the key problem being to be sure to interface sub-systems both consistently and without any specification hole). The practical difficulty of integration reflects well in the fact that there is probably no existing formal framework for dealing with integrated systems at the generality level we tried to took here. As a consequence, one can also not really find any unique global design formal tool for real systems. To be totally complete, one should however stress that there are at least two interesting frameworks – presented in the two forthcoming examples – for helping the working engineers in his integration tasks, but which have both serious modelling limitations as soon as one arrives at a system level, and moreover quite deep semantical lacks.

Example 5. Continuous oriented formalisms – The most widely industrially used system design tool is probably the Mathlab & Simulink environment (see [36]). In this approach, systems are represented by "black boxes" whose transfer functions, inputs and outputs have to be explicitly given by the user (see Figure 7 for the graphical representation of a car window system modelled

[4] We extend here χ^{-1} to F^I by setting $\chi^{-1}(c) = (0,0)$ when c is a free input index within $\mathcal{S}$.

[5] In the meaning of the product of the (complete) partial orders that are defined on each flow space $\mathbb{F} = \mathbb{I}^{\mathbb{T}}$ by setting $f \preceq g$ for two flows f and g of $\mathbb{F}$ iff the two following conditions are fulfilled: 1. f and g coincide up to some moment $t \in \mathbb{T}$; 2. f^u is equal to the missing quantity of $\mathbb{I}$ for each moment $u > t$ in $\mathbb{T}$.

[6] Where $X_{\chi^{-1}(c),c}$ lies in the flow space $\mathbb{I}_k^{i\,\mathbb{T}^i_k}$ for every $c = (i,k) \in F^I \cup C^I$.

in this framework). The main problem of Mathlab & Simulink is however related to the fact that there is no unambiguous and/or crystal clear semantics behind the manipulated diagrams. The self loops in the graphical formalism provided by these tools does for instance not have a very well defined interpretation in this framework, which may typically create causality problems at the modelling level (i.e. lead to abstractly modelled systems whose past behavior depends on their future one ... [7]). The discrete formalism used by Mathlab & Simulink – i.e. Stateflow which is just the commercial name of the implementation of the Statecharts framework (see the next example) – is also semantically rather weak (one can find probably more than 20 different formal semantics in the literature that were proposed for Statecharts). Altogether this shows that Mathlab and Simulink, even if they are wonderful and efficient working tools for the engineer, still suffer from really fundamental flaws from a formal point of view (which limits in particular the possibility of automatically verifying and validating designs made in this formalism).

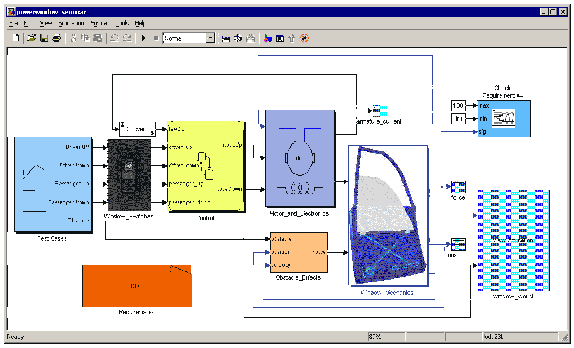

Fig. 7. *A Matlab/Simulink* ^{TM} *integrated system model* ©.

Example 6. Discrete formalisms – The last model that we would like to discuss in this section is Statecharts (see [23, 35]). It is indeed probably the very first model – introduced in 1987 – that allowed hierarchical design, one of the key idea of this formalism. In Statecharts, it is indeed possible to deal with distributed hierarchical Mealy machines (see again Example 1) which allow to model multi-flow production by *integrating* event oriented hierarchical mechanisms.

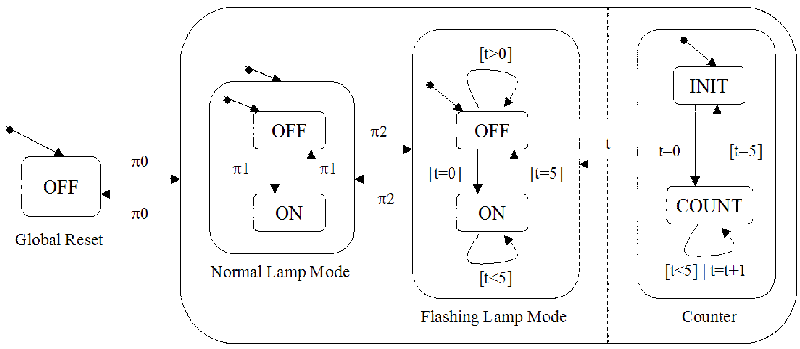

Fig. 8. *The lamp revisited with Statechart.*

The example of Figure 8 illustrates these key aspects of Statecharts. We modelled here a lamp with two working modes: a normal one, where the usual lamp button switch – associated with

[7] Note that we totally avoided this problem in the formalism we introduced above, due to the fact that our interaction channels have always a response delay !

π_1 – allows to switch on or off the lamp, and a flashing one, that one reachs or quits by pressing on a special button – represented by π_2. The lamp is also controlled by a global reset button – modelled by π_0 – which allow to stop or activate all the functions of the lamp when pressing on it. Note that from the point of view of this last button, the right state of the above figure is therefore just the "ON" state which is hierarchically decomposed into two states, corresponding to the two possible working modes for our lamp (in which one should continue to descend in order to arrive to the finer level of design in our example), plus a concurrent state representing a modulo 6 counter working totally independently from the other internal mechanisms, which gives permanently its value to the flashing mode management state [8] .

The problem of Statecharts is however its poor semantics with respect to distribution expressivity: the precise nature of the interactions between the two automata separated by the dashed line (which models concurrency) in Figure 8 is typically not totally clear. The Esterel language (see [7] or [5] where one can find a good overview of all so-called synchronous languages) was typically designed in order both to preserve the most interesting aspects of Statecharts' approach and to overcome its core flaws. For the sake of completeness, note finally that there are also other formal discrete formalisms that allow hierarchical design (see [9] and again [35]).

3.3 System abstraction and simulation

Abstraction and simulation are two classical notions that can also be re-adapted to systems (we take below all the flow notations of the previous section extended here to multi-flows).

Definition 6. *A multi-system $\mathcal{S}_1$ with input multi-flow space $\mathbb{F}_1^i$, output multi-flow space $\mathbb{F}_1^o$ and transfer function $\mathcal{F}_1$ is said to be an* abstraction *(resp. a* simulation*) of a multi-system $\mathcal{S}_2$ with input multi-flow space $\mathbb{F}_2^i$, output multi-flow space $\mathbb{F}_2^o$ and transfer function $\mathcal{F}_2$ iff there exists two injective functions σ^i and σ^o such that the following diagram is commutative:*

$$\begin{array}{ccc} \mathbb{F}_1^i & \xrightarrow{\mathcal{F}_1} & \mathbb{F}_1^o \\ {\scriptstyle\sigma^i}\downarrow & & \downarrow{\scriptstyle\sigma^o} \\ \mathbb{F}_2^i & \xrightarrow[\mathcal{F}_2]{} & \mathbb{F}_2^o \end{array} \quad \textit{(resp.} \quad \begin{array}{ccc} \mathbb{F}_2^i & \xrightarrow{\mathcal{F}_2} & \mathbb{F}_2^o \\ {\scriptstyle\sigma^i}\downarrow & & \uparrow{\scriptstyle\sigma^o} \\ \mathbb{F}_1^i & \xrightarrow[\mathcal{F}_1]{} & \mathbb{F}_1^o \end{array} \quad \textit{)}. \tag{3}$$

Hence $\mathcal{S}_1$ is an abstraction of $\mathcal{S}_2$ if these two systems have homomorphic functional behaviors, the first one being however less detailed than the second one. On the same way, $\mathcal{S}_1$ is a simulation of $\mathcal{S}_2$ if one can mimic all the functional behaviors of the second system by the first one.

Example 7. Assembling and high level programs – Let us fix a finite alphabet A and a discrete time scale $\mathbb{T}$. One can then identify any halting Turing machine M – i.e. any Turing machine that eventually stop on all its entries – with entries in A with a discrete system $\mathcal{S}_M$ with A^* as input and output quantity space, $\mathbb{T}$ as input and output time scale and a transfer function $\mathcal{F}_M$ defined as follows: 1. $\mathcal{F}_M$ transforms any flow of the form $F_x = (x, 1, 1, \dots)$, into the flow $F_{M,x} = (1, 1, \dots, Mx, 1, 1, \dots)$, where Mx stands for the value computed by M on x, produced at the moment given by the number of elementary steps of M required to obtain it; 2. $\mathcal{F}_M$ transforms any input flow different from a flow of the form F_x into the empty output flow. Looking on programs in this way, one can then easily check that each high level program P is an abstraction of some assembling program A (the one produced by the corresponding compiler) and that such an assembling program A is then a simulation of the program P.

[8] Which is not a very safe approach, as one may imagine, for obvious synchronization reasons ...

Example 8. Interfaces – The interface theory which was recently developed by de Alfaro and Henziger (see for instance [17]) can easily be transferred into the system framework as presented here (with of course again a number of slight reinterpretations). System interfaces provide then new generic interesting examples of system abstractions in our meaning . In this context, note that systems appear then as simulations of their interfaces.

Note finally that there are of course other less constrained abstraction notions, typically the ones coming from static analysis (see [15]), which are also of interest in the system context.

3.4 Concrete systems

We are now in position to model formally the usual way a concrete system is designed.

Definition 7. *A* concrete system $\mathcal{CS}$ *is a pair* $(\mathcal{FS}, \mathcal{OS})$ *of abstract integrated systems, the first one (resp. the second one) being an abstraction (resp. a simulation) of the other one, which are respectively called the functional behavior* [9] *and the organic structure* [10] *of* $\mathcal{CS}$.

This definition reflects the fact that the design of a real system S follows usually two main steps. The first one is a modelling step which defines the so-called *functional architecture* of S, i.e. in other words the recursive integration structure of a high level modelling of S constructed by following the point of view of the external systems (hardware, software, users, etc.) that are interacting with S. When the functional architecture of S is fixed, one can then define its *organic architecture*, i.e. the real internal structure of S, by respecting the requirements provided by the functional architecture (which appears as an abstraction of the organic architecture).

In classical software engineering, the two architectural notions involved in Definition 7 can be seen at different places. The pairs formed by a usual program and its machine or assembling code or, at a higher level, by a software specification and its programmed implementation are typical examples of concrete systems in our meaning. However the underlying conceptual separation does only take really all its importance when one is dealing with systems whose both functional and organic decompositions are complicated, which occurs typically when a system results from an highly heterogeneous integration process. Note that this last property can in fact be seen as an informal characterization of complex systems. Observe also finally that two totally different kinds of complex systems in this meaning, that is to say embedded systems and information systems, naturally arise in the software sphere (see below and Section 4.1).

Example 9. Embedded system design – When one deals with embedded system design, one must have an holistic approach that integrates in the same common framework the software, the hardware, the physical and the control theory points of views and constraints involved within such systems (see [25]). One therefore naturally divides the design in two separated, but completely interconnected, main parts: the functional design that corresponds here to the global environment and solution modelling where one will concentrate on the high level algorithmic and mathematical problems, the organic design related then with the low level system implementation where one must be sure to respect the physical and electronic constraints, the key difficulty being of course to have a good correspondence between these two levels of representation.

Example 10. Information system design – An information system can be seen as a global (enterprise) environment where software systems, hardware devices and human organizations interact

[9] The functional behavior models the input/output behavior of S as it can observed by an external observer.
[10] The organic structure models the intrinsic structure of the considered system.

in a coherent way (usually to fulfill a number of business missions). The complexity of these systems lead therefore classically to separate the corresponding design into two architectural levels: on one hand, the functional architecture which is devoted to the description of the user services, the business processes, the user and business data, the system environment, etc. that have to be managed by the information system; on the other hand, the associated organic architecture which is the concrete organization of the software applications, servers, databases, middleware, communication networks, etc. whose interactions will result in a working information system.

4 Complex Software Systems

4.1 Hierarchies of complex software systems

Integration and abstraction mechanisms allow us to construct naturally a hierarchy of complexity – taken here in an informal way – on software systems which is organized around two axes, i.e. integration and abstraction complexity (see Figure 9). The idea consists in classifying families of software systems according both to their degree of integration, i.e. to their degree of internal systemic largeness and heterogeneity, or more formally to the size of the tree associated with their organic architecture, and to the degree of abstraction which is required to deal with them, i.e. equivalently to the size of the tree associated with their functional architecture.

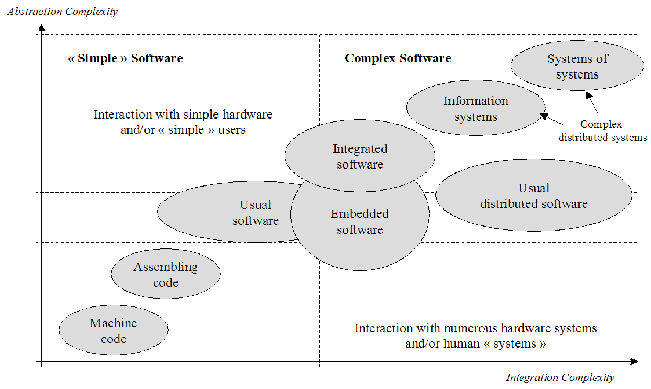

Fig. 9. *The complex software hierarchy.*

Such a classification lead us to identify two main classes of complex software systems (the term complex referring here only at first analysis to the organic integration complexity):

1. the software systems where the integration and abstraction complexity comes from the mix of computer science frameworks with *physics, signal processing* and *control theory* environments and models, that is typically to say the so-called embedded systems,
2. the software systems where the integration and abstraction complexity comes from the mix of the computer science world with mainly *"human" systems and organizations* (plus possibly hardware components), which can be themselves separated into three main subclasses that are presented hereafter by increasing degree of integration and of abstraction (i.e. from the less to the most complex underlying organic and functional architectures):
   – *integrated softwares*: this corresponds to enterprise softwares that are specifically devoted either to some category of business activities – such as BI (global information consolidation inside a large company), CRM (customer relationship management), ERP (financial

and production service management), SCM (supply chain management) or KM (documentation management) softwares – or to some type of technical integration purposes – such as B2Bi (external partner software integration), EAI (internal software integration), ETL (batch data consolidation) or EII (on the request data consolidation) softwares. We refer to [26] for an overview of these software technology (see also [54, 34]).

- *information systems*: an information system can be defined as a coherent integration of several integrated softwares – in the above meaning – that supports all the possible business missions of an organization, taken at a global level. An information system can therefore be seen as the integrated system that consists both of a human organization and of all the computer systems that are supporting the specific business processes of the considered organization (see [11, 41] or Example 11 for more details).
- *systems of systems*: this refers to an even higher level of integration, i.e. to the situation where a number of independently designed information systems have to cooperate in order to fulfill a common mission. Systems of systems are characterized by the loose couplings existing structurally between their organic components (that we will not discuss here since this would lead us too far with respect to the integration model we took within this survey paper). Network Centric Warfare (NCW) systems, airport platforms management systems, etc. can be typically seen as systems of systems in this meaning.

Example 11. Information system – As already pointed out, an information system can be seen as the integration of a human organization and a software system. The left side of Figure 10 shows for instance a very high level architecture of an information system focusing on this point of view: the sub-systems of the enterprise organization (i.e. the main business departments) are here at the border of this map when the technical sub-systems (i.e. the main integrated involved softwares) are in the center. One can also see on this map a number of arrows that are refering to the main business processes, that is to say to the main normalized interactions (or equivalently interfaces) existing between the corresponding human and software systems.

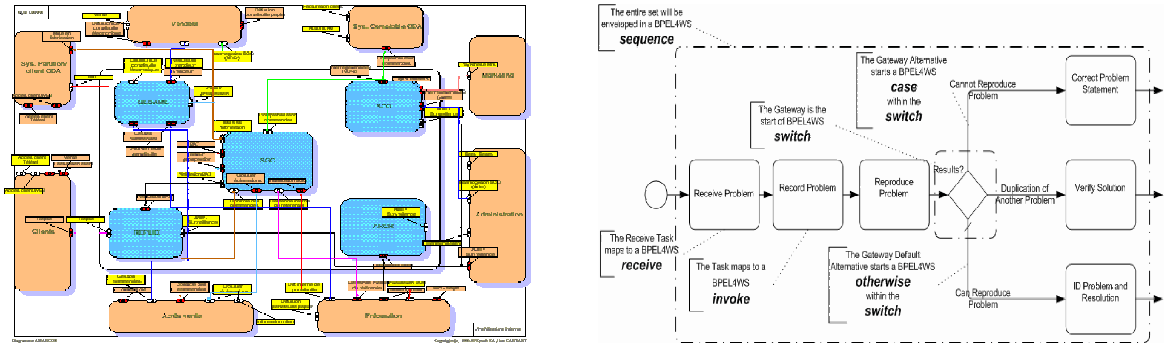

Fig. 10. *An information system architecture (Sysoft ©) and a business process model (BPMN © – [56]).*

A business profess refers typically to an enterprise process such as billing, maintening, sourcing, producing, etc. Business process modelling (BPM) is therefore naturally one of the core methodology which is presently developed to represent better the functional behavior of an information system (see [10, 41] or Figure 10 which gives an example of a software testing procedure modeling). Note however that BPM is not a formal approach in the line of the numerous models we presented in Section 3. It indeed rather belongs to the family of informal UML-like models, which limits seriously its theoretical potential (but leaves the door open for new research).

4.2 What are the new problems emerging from this framework ?

New types of problems are naturally arising with the most complex software systems. A rather important research effort is for instance presently done for understanding and designing better embedded systems, which are however probably the "most simple" complex systems due to the "nice" underlying mathematical environment in which they are living, even if they are already quite complex to handle (see [25]). We will therefore not focus here on these systems on which a lot was and continue to be made by numerous theoretical computer scientists, but rather on the "human"-oriented complex systems which were, quite surprisingly, not widely studied from a formal point of view, although they are at the center of important economic challenges and of a large technological and business literature (see for instance [11, 41, 48], etc.).

One of the key problems of these kind of software systems is clearly to be able to take more formally into account the "man in the loop", which appears to be their common denominator. There are therefore naturally several important research streams that could emerge in trying to develop operational modelling formalisms for business processes and more generally for organizational paradigms. We mean of course here formalisms with well defined semantics that would allow to express precisely and unambiguously characteristic properties of a business process (such as cost, speed of execution, probability of success, etc.) in order to be able to formally verify these properties. Such a modelling research effort could probably also help practically organizations to master better their structures and processes.

At a more global level, there is still in the same way an important high level formal modelling effort that must be done in order to give solid bases to a theory of complex software systems. If there is a real business consensus on the nature of an information system, no scientific consensus exists presently – at our knowledge – with respect to a *formal* – that is to say a mathematical – definition of an information system. For systems of systems, the situation is even worse since at this moment of time, there is even no clearly shared business definition of these more complex systems. The key point in this direction would therefore probably to be able to give sound formal definitions of information systems and systems of systems, taking of course integrated systems as primitive objects in such a modelling approach. Such a framework would probably result in the development of new methods for complex software quantitative analysis, an important subject which is still under-developed in the classical context of information systems (see [19, 44]) and basically non existing for systems of systems.

One should finally not forget all the specific problems that are of course continuously emerging in the jungle of complex software systems. As a matter of conclusion to this paper, one can find such two problems – among many others – roughly and quickly presented below.

Example 12. Information system degeneracy – A classical operational problem that arises in a real information system corresponds typically to the situation where the system begins to emit too many information flows and crashes when it is not able anymore to support the resulting treatment charge. Usually such a crash is not immediate and appears as the consequence of a long intermediate period where nothing is made to prevent it. It is therefore of main interest to be able to predict it and to analyze its origins in order to react properly when it is still time. If one models at high level an information system as a network of multi buffered applications, one sees that the problem can be rephrased as a problem of queuing networks that can probably be attacked both from a static analysis and a distributed algorithmic point of view.

Example 13. Interoperability of systems of systems – When interoperability is a well known problem which is quite well mastered for usual information systems (see [20, 34]), it is probably still an open subject at the level of systems of systems. The key difficulty at this level comes

from the fact that one must interface in a coherent way a number of information systems that were not initially intended to work together. For technical reasons, the usual interoperability approaches can therefore not totally be applied in these contexts since it is typically not easy or even possible to interface these systems through an EAI layer. New methods – mixing semantical and syntactical approaches – are therefore required to solve in a generic way this key problem.

Acknowledgements

The author would sincerely like to thank Herman Kuilder, Matthieu Martel, Marc Pouzet and Jacques Printz for the numerous discussions we had together during the maturation period that preceded the writing of this paper, leading to several key evolutions of his point of view.

References

1. Abrial J.R., *The B-book – Assigning programs to meanings*, Cambridge University Press, 1996.
2. Alur R., Courcoubetis C., Halbwachs N., Henzinger T.A., Ho P.H., Nicollin X., Olivero A., Sifakis J., Yovine S., *The algorithmic analysis of hybrid systems*, Theor. Comp. Sci., 1995; **138** (1): 3–34.
3. Autebert J.M., Boasson L., *Transductions rationnelles – Applications aux langages algébriques*, Collection ERI, Masson, 1988.
4. Baccelli F., Cohen G., Olsder G.J., Quadrat J.P., *Synchronization and linearity – An algebra for discrete event systems*, Wiley, 1992.
5. Benveniste A., Caspi P., Edwards S.A., Halbwachs N., Le Guernic P., de Simone R., *The Synchronous Languages Twelve years later*, Proc. of the IEEE, Special issue on Embedded Systems, 2003; **91** (1): 64–83.
6. Bérard B., Bidoit M., Laroussinie F., Petit A., Schnoebelen P., *Vérification de logiciels – Techniques et outils du model-checking*, Vuibert Informatique, 1999.
7. Berry G., Gonthier G., *The Synchronous Programming Language ESTEREL: Design, Semantics*, Implementation Science of Computer Programming, **19**, 83–152, 1992.
8. Bock C., *SysML and UML2 Support for Activity Modeling*, Systems Engineering, **9**, (2), 160–186, 2006.
9. Börger E., Stärk R., *Abstract state machines – A method for high-level system design and analysis*, Springer, 2003.
10. Business Process Management Initiative – Object Management Group, *Business Process Modeling Notation*, OMG, `http://www.bpmn.org`, 2006.
11. Caseau Y., *Urbanisation et BPM : le point de vue d'un DSI*, Dunod, 2006.
12. Caspi P., Hamon G., Pouzet M., *Lucid Synchrone, un langage de programmation des systèmes réactifs*, in "Systèmes Temps-réel : Techniques de Description et de Vérification - Théorie et Outils", 217–260, Hermes International Publishing, 2006.
13. Caspi P., Pouzet M., *Synchronous Kahn networks*, Proc. of the first ACM SIGPLAN Int. Conf. on Functional Programming, 226–238, 1996.
14. Cha D.P., Rosenberg J.J., Dym C.L., *Fundamentals of Modeling and Analyzing Engineering Systems*, Cambridge University Press, 2000.
15. Cousot P., Cousot R., *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in "Conf. Record of the Sixth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages", Los Angeles, ACM Press, 238–252, 1977.
16. Cutland N., *Nonstandard analysis and its applications*, London Mathematical Society Student Texts, **10**, Cambridge University Press, 1988.
17. De Alfaro L., Henziger T.A., *Interface-based design*, in "Engineering Theories of Software-intensive Systems", M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, Eds., NATO Science Series: Mathematics, Physics, and Chemistry, Vol. 195, 83–104, Springer, 2005.
18. Fliess M., *Fonctionnelles causales non linéaires et indéterminées non commutatives*, Bull. Soc. Math. France, 1981; **109**: 3–40.
19. Garmus D., Herron D., *Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley Information Technology Series, Addison-Wesley, 2000.
20. Gold-Bernstein B., Ruh W., *Enterprise Integration: The Essential Guide to Integration Solutions*, Addison-Wesley Information Technology Series, Addsison-Wesley, 2004.

21. Gunter C.A., Scott D., *Semantic domains*, in "Handbook of Theoretical Computer Science", Vol. B, 633–674, Elsevier, 1990.
22. Halbwachs N., Caspi P., Raymond P., Pilaud D., *The synchronous data-flow programming language LUSTRE*, Proceedings of the IEEE, **79**, (9), 1305–1320, 1991.
23. Harel D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, **8**, (3), 231–274, 1987.
24. Henzinger T.A., *The theory of hybrid automata*, in Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS'96, IEEE Society Press, 1996, pp. 278–292.
25. Henzinger T.A., Sifakis J., *The embedded systems design challenge*, Proc. of the 14th Int. Symp. on Formal Methods (FM), LNCS, Springer, 2006 (to appear).
26. IT Tool box, `http://www.ittoolbox.com`.
27. Kahn G., *The semantics of a simple language for paralell programming*, Proc. of the IFIP Congress 74, 471–475, 1974.
28. Katok A., Hasselblatt B., *Introduction to the modern theory of dynamical systems*, Cambridge, 1996.
29. Krob D., Bliudze S., *Towards a Functional Formalism for Modelling Complex Industrial Systems*, in "European Conference on Complex Systems (ECCS05), P. Bourgine, F. Kps, M. Schoenauer, Eds., (article 193), 20 pages, 2005.
30. Krob D., Bliudze S., *Towards a Functional Formalism for Modelling Complex Industrial Systems*, in "Complex Systems: Selected Papers", ComPlexUs (to appear).
31. Lamport L., *Specifying systems – The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2003.
32. Le Guernic P., Gautier T., *Data-Flow to von Neumann: the Signal approach*, in "Advanced Topics in Data-Flow Computing", Gaudiot J.-L. and Bic L., Eds., Prentice-Hall, 413–438, 1991.
33. Maier M.W., *System and Software Architecture Reconciliation*, Systems Engineering, **9**, (2), 146–59, 2006.
34. Manouvrier B., *EAI – Intégration des applications d'entreprise*, Hermès, 2001.
35. Marwedel P., *Embedded systems design*, Kluwer, 2003.
36. Mathworks, *Mathlab and Simulink*; `http://www.mathworks.com`.
37. Mealy G.H., *A Method for Synthesizing Sequential Circuits*, Bell System Tech. J., **34**, 1045-1079, 1955.
38. Meinadier J.P., *Ingénierie et intégration de systèmes*, Hermès, 1998.
39. Meinadier J.P., *Le métier d'intégration de systèmes*, Hermès-Lavoisier, 2002.
40. Meyer K.R., Hall G.R., *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem*, Applied Mathematical Sciences, **90**, Springer Verlag, 1992.
41. Morley C., Hugues J., Leblanc B., Hugues O., *Processus métiers et systèmes d'information*, Dunod, 2005.
42. Petri C.A., *Fundamentals of a Theory of Asynchronous Information Flow*, Proc. of IFIP Congress 1962, 386–390, North Holland, 1963.
43. Proakis J., *Digital Communications*, 3rd Edition, McGraw Hill, 1995.
44. Printz J., Deh C., Mesdon B., Trèves B., *Coûts et durée des projets informatiques – Pratique des modèles d'estimation*, Hermès Lavoisier, 2001.
45. Reisig W., *Petri nets*, Springer Verlag, 1985.
46. Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.
47. Sage A.P., Armstrong J.E. Jr., *Introduction to Systems Engineering*, John Wiley, 2000.
48. Satzinger J.W., Jackson R.B., Burd S., Simond M., Villeneuve M., *Analyse et conception de systèmes d'information*, Les éditions Reynald Goulet, 2003.
49. Schneider K., *Verification of reactive systems – Formal methods and algorithms*, Springer, 2004.
50. Severance F.L., *System modeling and simulation – An introduction*, John Wiley, 2001.
51. Sommerville I., *Software Engineering*, Addison Wesley, 6th Edition, 2001.
52. Spivey J.M., *The Z notation – A reference manual*, Prentice Hall, 1992.
53. SysML, *Systems Modeling Language – Open Source Specification Project* – `http://www.sysml.org`.
54. Tomas J.L., *ERP et progiciels de gestion intégrés – Sélection, déploiement et utilisation opérationnelle – Les bases du SCM et du CRM*, Dunod, 2002.
55. Turner W.C., Mize J.H., Case K.E., Nazemetz J.W., *Introduction to industrial and systems engineering*, Prentice Hall, 1993.
56. White S.A., *Introduction to BPMN*, IBM, `http://www.bpmn.org`, 2006.
57. Wordsworth J.B., *Software engineering with B*, Addison-Wesley, 1996.
58. Zaytoon J., Ed., *Systèmes dynamiques hybrides*, Hermès, 2001.
59. Zeigler B.P., Praehofer H., Gon Kim T., *Theory of modeling and simulation – Integrating discrete event and continuous complex dynamic systems*, Academic Press, 2000.
60. Zschaler S., *Formal Specification of Non-functional Properties of Component-Based Software*, in "Workshop on Models for Non-functional Aspects of Component-Based Software" (NfC'04), Bruel J.M., Georg G., Hussmann H., Ober I., Pohl C. Whittle J. and Zschaler S., Eds., Technische Universität Dresden, 2004.