Introduction

Whether one likes it or not, a software system[1] in operation is a formal, mathematical machine, even if its function is to support people in non-mathematical, informal work. This observation has important implications. First, it must then be possible to acquire a comprehensive understanding of a software system, abstracted from implementation issues, that is sufficiently concise for intellectually managing its complexity. Second, such an understanding would be a proper basis for verifying the correctness of software systems; the role of testing can only be secondary, because it may show the presence of errors but never their absence. Third, such an understanding would also be a proper basis for studying the construction and operation of the system, through mathematical or logical analysis, as well as through simulation, including animation and gaming. Fourth, such an understanding would be a proper starting point for generating software whose correctness can be proven.

The π-theory (the Greek letter "π" is pronounced as "PI", which stands for "Performance in Interaction") aims at achieving all these objectives. It is a theory about the ontological essence of discrete event systems of which the elements are non-human. In order to avoid confusion with social systems (as covered by the ψ-theory [JDM-5]), we will call these systems "technical systems". Note that a technical system may be (originally) a social system, only technically implemented, like automated teller machines (ATM), automated check-in systems, and web shops. So, the π-theory clarifies and explains the construction and operation of technical systems. It is discussed to some extent in [Dietz, 2005]. It builds on the δ-theory [JDM-3], the τ-theory [JDM-2], and the ϕ-theory [JDM-4].

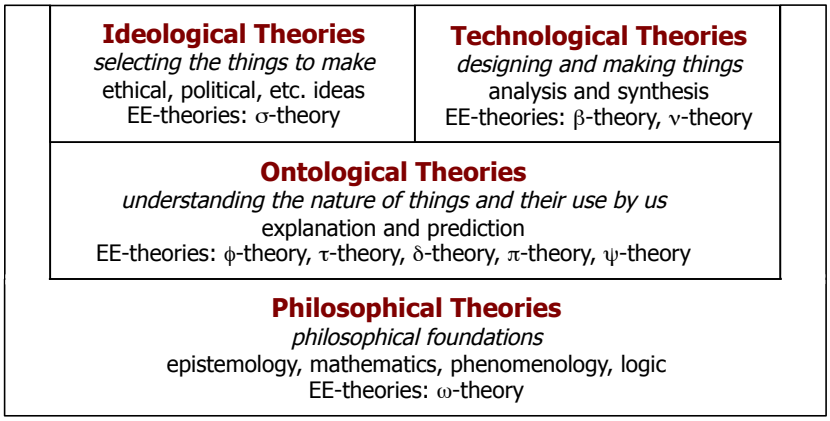

Figure 1.1 Classification scheme for enterprise engineering theories

This extended summary presents and discusses the main subjects and outcomes of the π-theory, including its relationship with other enterprise engineering theories, as exhibited in the classification scheme in figure 1, taken from [Dietz, Hoogervorst et al. 2013] (Note: in this scheme, an arrow from A to B means that A is a basis of B).

The π-theory is an *ontological* theory, which means that it concerns the nature of things, in this case of technical systems. From other EE-theories, these notions are taken: system, model (τ-theory); information (ϕ-theory); operation, state, action, event, process (δ-theory).

In the π-theory, the delta automaton from the δ-theory gets a concrete application in the domain of technical systems. Instead of actels and statels, we will speak of acts and facts respectively. These acts and facts are instances of act types and fact types respectively.

---

[1] By a software system is understood a discrete event system that is implemented by means of some information and communication technology (ICT).

Formal definition of a smartie

Smarties operate in a discrete linear time dimension, which means that we assume the existence of distinct points in time and that the difference between any two consecutive points in time is always the same.

At every point in time, the *world* (Cf. the τ-theory [JDM-2]) of a smartie is in some state. A *state* is a set of *facts*. The *state base* of a smartie is defined as the set of fact types whose instances may belong to a state.

In order to model the dynamics of a smartie, the notion of *act* is introduced. The set of act types, to the instances of which a smartie can respond, is called its *action base*.

The *operation* of a smartie can be explained as follows. At every moment a smartie disposes of a set of *agenda* (things to do). An *agendum* (singular of agenda) is a pair (*a*, *t*) in which *a* is an act, and *t* is a point in time. The act in an agendum (*a*, *t*) is said to be *current* at the point in time *t*. The becoming current of an act is identical to the occurrence of the *event* (*a*, *t*). The set of current acts at a point in time *t* is called the *action* at time *t*. A smartie responds instantly to an action by performing a *transition*, which consists of evaluating a partial function, called the *transition base* of the smartie. The result of the evaluation is a finite set of facts and a finite set of agenda. The set of facts is called the *mutation*. These facts are instances of fact types in the *mutation base* of the smartie. The set of agenda is called the *reaction*. These acts are instances of act types in the *reaction base* of the smartie.

Hereafter, a formal definition of a smartie is presented, fully based on the formal definition of a delta [JDM-3]. In this definition, the union of the extensions of a set of concept types (act types or fact types) C is denoted as $\underline{C}$, and the power set of a set X is denoted as $\wp X$. Points in time are represented by elements of the set $\mathbb{T}$; the current point in time is denoted by Now; (positive) time durations are elements of the set $\mathbb{D}$.

def

A *smartie* can formally be defined as a tuple $< \mathbf{S}, \mathbf{M}, \mathbf{A}, \mathbf{R}, \mathbf{T} >$, where:

S	:	a set of fact types, called the *state base*
M	:	a set of fact types, called the *mutation base*
A	:	a set of act types, called the *action base*
R	:	a set of act types, called the *reaction base*
T	:	a partial function, called the *transition base* :
		$\mathbf{T} \in \wp\underline{\mathbf{A}} * \wp\underline{\mathbf{S}} \rightarrow \wp(\underline{\mathbf{R}} * \mathbb{D}) * \wp\underline{\mathbf{M}}$

fed

The function **T** can conveniently be represented by its extension, i.e. the set of *transition rules* of the form <A,S,R,M> where:

A is a set of acts, called the *action*; $A \subset \underline{\mathbf{A}}$;

S is a set of facts, called the *state*; $S \subset \underline{\mathbf{S}}$;

R is a set of pairs (*r*, *d*) with $r \in \underline{\mathbf{R}}$ and $d \in \mathbb{D}$, called the *reaction*; *d* is the *delay* of *r*; it means that act *r* will become current at time Now+*d*;

M is a set of facts, called the *mutation*; $M \subset \underline{\mathbf{M}}$;

A smartie is called *elementary* if its action base consists of <u>one</u> act type. Smarties that are not elementary, are called *composite*.

Mutual influencing of smarties

Based on the distinction between acts and facts, we distinguish between two ways in which smarties influence each other, respectively called *activating* and *conditioning*. They are discussed hereafter.

Smartie1 is said to *activate* smartie2 if $\mathbf{R}1 \cap \mathbf{A}2 \neq \varnothing$. If this is the case, then every agendum in a reaction of smartie1, whose act belongs to the extension of this intersection, will change the agenda of smartie2 instantly.

The new agenda of smartie2 is the symmetric set difference[2] of its current agenda and the reaction. Note that in this way, items can not only be added to the agenda of smartie 2, but also removed from it. This offers the possibility to cancel planned actions.

If smartie1 and smartie2 are identical, we speak of *self-activation*: apparently, the smartie is able to cause its own future transitions. In this way periodic operations can be modelled elegantly.

Smartie1 is said to *condition* smartie2 if $\mathbf{M}1 \cap \mathbf{S}2 \neq \varnothing$. If this is the case, then every fact in a mutation of smartie1 that belongs to the extension of this intersection, will change the state of smartie2 instantly. The new state of smartie2 is the symmetric set difference of its current state and the mutation. Note that in this way, facts can not only be added to the state of smartie 2, but also removed from it. This offers the possibility to update the value of variables.

If smartie1 and smartie2 are identical, we speak of *self-conditioning*: the smartie is apparently able to change it own state. Self-conditioning is the classical concept of the (internal) state of a system.

The mutual activating of smarties is collectively called *interaction*, and the mutual conditioning of smarties is collectively called *interstriction*[3]. The distinctive difference between interaction and interstriction is that interaction causes smarties to perform transitions, whereas interstriction does not. Consequently, a smartie is not 'aware' of a state change at the time it takes place. Instead, it 'takes notice' of a state change at the next point in time at which it processes an action. So, between two adjacent points in time at which a smartie performs a transition, it 'sleeps'. In that period however a number of state changes may occur. (For a deeper discussion, see the δ-theory [JDM-3]).

The smartienet

The influencing relationships among a collection of smarties can be made more comprehensible if this collection is modelled as a smartienet. A *smartienet* is a network consisting of three kinds of components: processors, banks and channels. Figure 1 exhibits the symbolic representations of the components of a smartienet diagram, as well as its basic constructs.

Processors represent the transition mechanisms of smarties, consisting of their transition base and the operating cycle. By the *operating cycle* is understood the periodic checking of a processor (at high frequency) whether there is an action to be processed. Therefore, the processor is also called the *kernel* of the smartie. The *environment* of a smartie is defined as the set of processors with which its own processor has influencing relationships (interaction and/or interstriction). The kernel of an elementary smartie is an elementary processor; the kernel of a composite smartie is a composite processor.

Banks enable the interstriction of smarties through their capability of containing facts. A bank B_k is defined by its *contents base* $\mathcal{CB}_k$, which is the set of fact types whose instances it can contain. The set of contained facts at some moment, is called the *contents* of the bank at that moment. The content base of an *elementary bank* is precisely one fact type. The contents bases of the elementary banks in a smartienet are disjoint. Consequently, every single fact can only be contained in precisely one bank. It is not possible to have duplicates. An *aggregate bank* is a collection of elementary banks. The contents base of an aggregate bank is the union of the contents bases of the composing elementary banks. The contents of an aggregate bank at some moment is the union of the contents of the composing elementary banks at that moment.

Channels enable the interaction of smarties through their possibility of transmitting acts. A channel C_n is defined by its *transmission base* $\mathcal{TB}_n$, which is the set of act types whose instances it can transmit. The set of transmitted acts at some moment, is called the *transmission* of the channel at that moment. The transmission

---

[2] The symmetric set difference Δ is defined as follows: $A \Delta B = (A \setminus B) \cup (B \setminus A)$. Its effect is that every element in B that is not in A will be 'added', and that every element in B that is also element in A, will be 'removed'.

[3] The noun "interstriction" comes from the Latin verb "stringere", which is also the root of "restriction". The term "interstriction" expresses that smarties restrict each others freedom of action or 'play area'.

base of an *elementary channel* is precisely one act type. Consequently, every single act can only be transmitted in precisely one channel. It is not possible to have duplicates. The transmission bases of the elementary channels in a smartienet are disjoint. An *aggregate channel* is a collection of elementary channels. The transmission base of an aggregate channel is the union of the transmission bases of the composing elementary channels. The transmission of an aggregate channel at some moment is the union of the transmissions of the composing elementary channels at that moment.

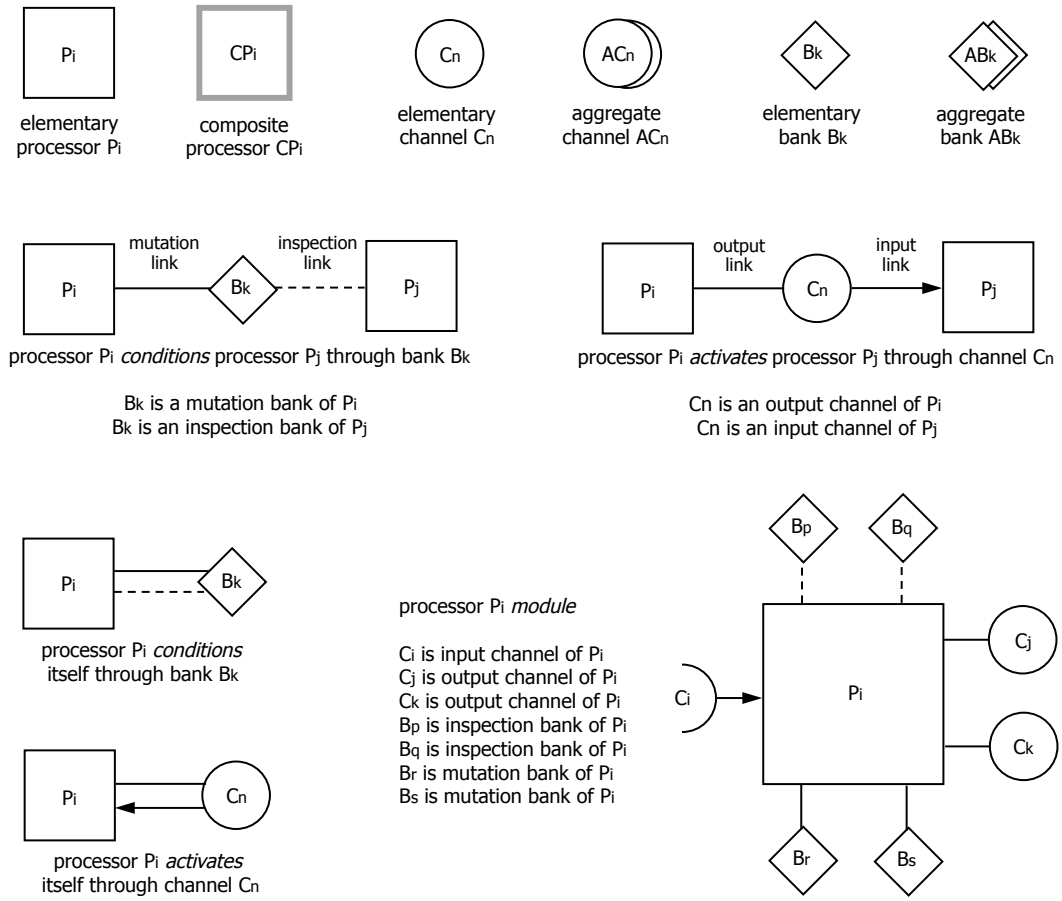

Figure 1 Legend of the smartienet diagram

Banks are connected to processors by two kinds of links: inspection links and mutation links (Cf. Figure 1). Through an *inspection* link, a processor is able to inspect the contents of a bank. Through a *mutation* link, a processor is able to mutate (change) the contents of a bank. There is an inspection link between bank B_k and processor P_j if the contents base of bank B_k is a subset of the state base of the smartie of which processor P_j is the kernel ($\mathcal{CB}_k \subset \mathbf{S}_j$). There is a mutation link between bank B_k and processor P_i if the contents base of bank B_k is a subset of the mutation base of the smartie of which processor P_i is the kernel ($\mathcal{CB}_k \subset \mathbf{M}_i$). Consequently (Cf. Figure 1), bank B_k is called a *mutation bank* of processor P_i and an *inspection bank* of processor P_j.

Channels are also connected to processors by two kinds of links: input links and output links (Cf. Figure 1). Through an *input* link, a processor is able to receive agenda. Through an *output* link, a processor is able to send agenda. There is an input link between channel C_n and processor P_j if the transmission base of channel C_n is a subset of the action base of the smartie of which processor P_j is the kernel ($\mathcal{TB}_n \subset \mathbf{A}_j$). There is an output link between channel C_n and processor P_i if the transmission base of channel C_n is a subset of the reaction base of the smartie of which processor P_i is the kernel ($\mathcal{TB}_n \subset \mathbf{R}_i$). Consequently (Cf. Figure 1), channel C_n is called an *output channel* of processor P_i and an *input channel* of processor P_j.

The notion of channel must be understood in this way (Cf. Figure 1). Suppose that processor Pi generates at time *t* an agendum (*a*,*d*), where *a* is an act and *d* is the delay (i.e. the time after which act *a* becomes current). The channel metaphor then is that act *a* is 'put' in the channel by the sender Pi at time *t* and that it 'arrives' at the receiver Pj at time *t* + *d*.

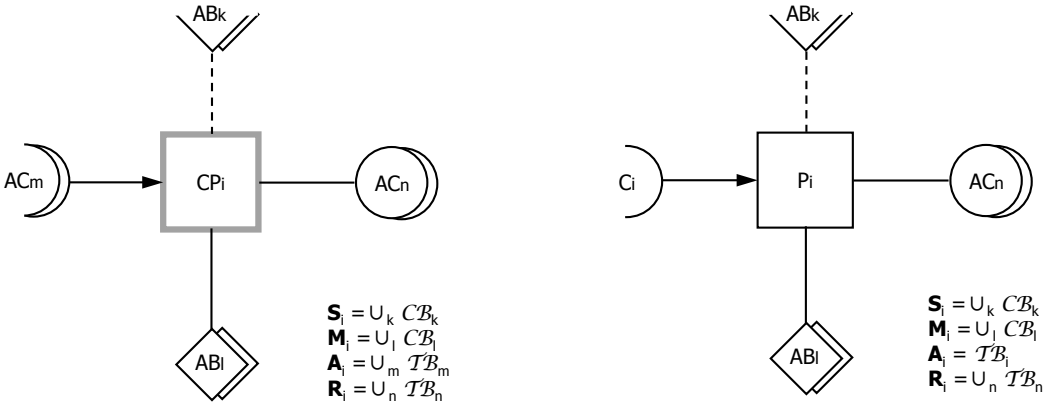

Figure 2. Module representations of smarties

Figure 2 exhibits the representation of smarties as smartienet building blocks or *modules*. On the left side is the generic module representation of a composite smartie, and on the right side is the generic module representation of an elementary smartie. Because a processor may be connected to a multitude of elementary input channels and output channels, as well as inspection banks and mutation banks, aggregate channels and banks are drawn. The only exception is the input channel of the elementary processor (right side of Figure 2), which is by definition elementary. Next, the disks of the input channels and the diamonds of the inspection banks are only drawn 'half'. It expresses that they must be 'clicked' to the corresponding disks and diamonds of other modules in order to become operating systems.

Figure 2 also shows the specification of the state base, mutation base, action base, and reaction base of the represented smarties. As an example, $\mathbf{S}_i$ is the union of the contents bases $\mathcal{CB}_k$ of all elementary banks Bk in the aggregate bank ABk. By convention, the input channel of an elementary processor gets the same number as the processor. Thus Ci is the input channel of Pi (Figure 2, right side). Consequently, $\mathbf{A}_i = \mathcal{TB}_i$.

The essential model of a system

The smartie model of a system is a conceptual model, which means that it is fully independent of its, current or future, (technological) implementation. Unlike most other conceptual models, smartie models have two additional qualities. The first one is that the model is truly comprehensive, by which is meant that it includes statics, kinematics, and dynamics, and this also in an integrated way. The second quality is that smartie models abstract from informational acts, like keeping facts ('remembering') and getting facts ('recalling'), and computing derived facts from other ones. These acts are 'hidden' by the notions of conditioning and interstriction, which are represented in the smartienet by banks, mutation links and inspection links. Instead of informing other smarties about a state change, or of asking for (original or derived) facts, smarties inspect the banks to which they have access, in much the same way as higher programming languages allow the programmer to directly change and use the values of variables (or as the processor of a computer has direct access to the data in the memory of the computer). Because of this second quality, the acts in the smartie model of a system are called its essential acts, and the smartie model is called the *essential model* of the system.

Both the comprehensiveness and the conciseness of the essential model contribute substantially to the enterprise engineering goal of intellectual manageability. As will be demonstrated at two example cases hereafter, the specification of the **S**, **M**, **A**, **R**, and **T** component of an elementary smartie is rarely complicated, and the smartienet diagram offers a deep and accurate understanding of the activating and conditioning interrelationships between the constituting smarties.

Example: traffic control system

To illustrate the modelling of technical systems as smarties, we take the control of traffic at a simple crossing of two roads, numbered 1 and 2, as an example. The next requirements for the traffic control system (TCS) have been formulated (Figure 3 summarises these requirements in a graphical way).

If the traffic on road 2 is moving, the traffic on road 1 must wait. There is a minimum time that the traffic on road 2 may move on, called the (standard) move time for road 2. As long as there is no traffic waiting on road 1, the move time is prolonged. However, as soon as a car on road 1 wants to cross, and if the (standard) move time for road 2 has passed, the traffic on road 2 must be signalled to stop during an amount of time that is called the stop time for road 2. To let traffic that could not stop in time leave the crossing, there is an extra amount of time, called the clear time, during which the traffic on both roads has to wait. Next, the traffic on road 1 is signalled to move on, while the traffic on road 2 is waiting. Of course, the same requirements hold if road 1 and road 2 are exchanged. Instead of actively 'triggering' the traffic participants to move and stop, it has been decided that it is sufficient to notify the state changes (move, stop, wait) of the TCS to the traffic participants. This could be done by means of traffic lights (green, yellow, red), but that is a matter of implementation design.

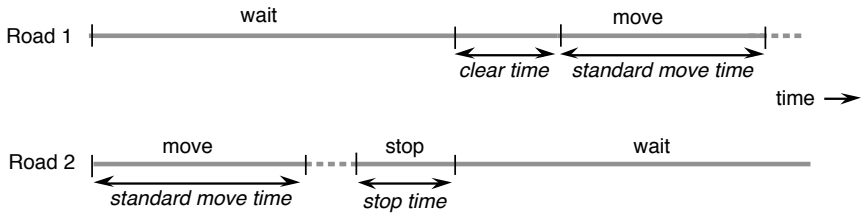

Figure 3. Functional requirements for the TCS

This description represents a *functional model* of the TCS (Cf. the τ-theory [JDM-2]) from which we will devise its *constructional model* (Cf. the τ-theory [JDM-2]), that is its essential model according to the π-theory. We first present the global construction model of the TCS, represented in a smartienet diagram, in order to clearly set the boundary of the system (Cf. Figure 4). The notion of road is replaced by the notion of cycle. At any point in time, a cycle is in one of three phases: move, stop, or wait. There are three parameters for every cycle: the (standard) move time, the stop time, and the clear time (Cf. Figure 3). Their values are set by environmental processors. The TCS is activated externally by means of 'let pass' acts.

On the basis of the previous analysis, we identify two interface banks: AB1 (containing the parameter facts) and B1 (containing 'phase' facts), and one interface channel: C1 (transmitting 'let pass' acts). Two environmental processors are identified. By convention, these processors are modelled as composite smarties. Processor CP1 (traffic) activates the traffic control system by means of 'let pass' acts. This processor also inspects bank B1. Processor CP2, called 'traffic manager', informs the traffic control system about the current value of the control parameters, through the aggregate bank AB1.

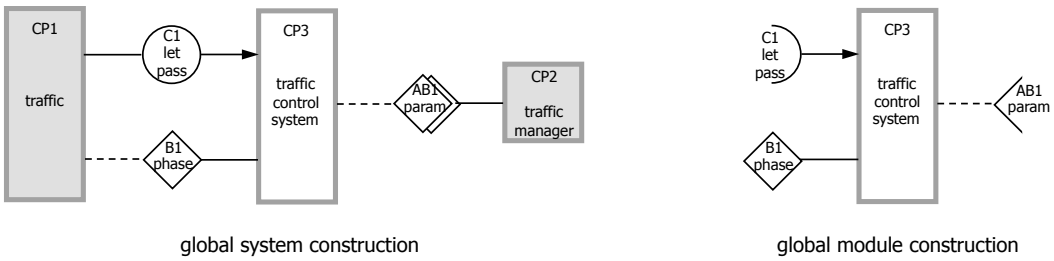

Figure 4. Global smartienet diagram of the TCS

Clearly, there must also be an internal activation in the TCS to accommodate the transition of the various phases of every cycle, next to the external activation by 'let pass' acts. Consequently, we come up with the detailed construction model as exhibited in Figure 5. The bold grey-lined rectangle represents the system boundary.

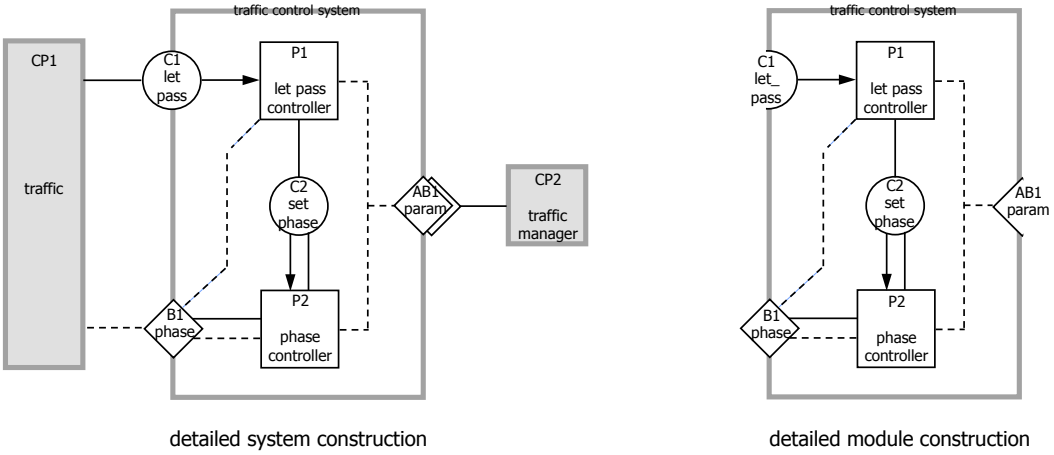

Figure 5. Detailed smartienet diagram of the TCS

There are two internal elementary processors: P1 (let pass controller) and P2 (phase controller), and one internal channel: C2 (transmitting 'set phase' acts). P2 is activated both by P1 and by itself. The contents base of the bank 'param' consists of move_time(Cycle), stop_time(Cycle) and clear_time(Cycle). The contents base of the bank 'phase' consists of phase(Cycle). The transmission base of the channel 'let_pass' consists of let_pass(Cycle), and the transmission base of the channel 'set_phase' consists of set_phase(Cycle).

We are now able to specify the TCS as the union of the specifications of the two constituting elementary smarties: one with kernel P1 and one with kernel P2. In the specifications below, Cycle is a variable that can have the value 1 or 2, corresponding with road 1 and road 2 respectively. Because of the symmetry of the cycles, we refer to these values by 'cycle' and 'other_cycle'; so, if cycle = 1, then other_cycle = 2 and vice versa. For the specification of **S**, **M**, **A**, and **R**, normal set theoretic notations are used. For the specification of **T**, we use a pseudo-algorithmic language that will be explained shortly.

Specification of smartie 1

The first smartie (with kernel P1) is specified as follows:

S1 = {phase(Cycle), move_time(Cycle)}
M1 = ∅
A1 = {let_pass(Cycle)}
R1 = {set_phase(Cycle, Phase)}

The transition base **T1** is specified as follows:

when let_pass(cycle) **occurs**
if phase(cycle) = wait **and** phase(other_cycle) = move
then set_phase(other_cycle,stop)
with delay = max(0, (move_time(other_cycle) -
(Now - creation_time(phase(other_cycle) = move)))

(Note. The *creation time* of a fact is defined as the point in time at which it becomes/became existent, i.e., comes/came into being)

Specification of smartie 2
The second smartie (with kernel P2) is specified as follows:

S2 = {phase(Cycle), stop_time(Cycle), clear_time(Cycle)}
M2 = {phase(Cycle)}
A2 = {set_phase(Cycle, Phase)}
R2 = {set_phase(Cycle, Phase)}

The transition base **T2** is specified as follows:

when set_phase(cycle,stop) **occurs**
if phase(cycle) = move
then set_phase(cycle,wait) **with** delay = stop_time(cycle);
phase(cycle) := stop

when set_phase(cycle,wait) **occurs**
if phase(cycle) = stop
then set_phase(other_cycle,move) **with** delay = clear_time(other_cycle);
phase(cycle) := wait

when set_phase(cycle,move) **occurs**
if phase(cycle) = wait **and** phase(other_cycle) = wait
then phase(cycle) := move

Discussion
The next explanation of the transition rules holds. The first line (when clause) specifies the *event* (action) that the system has to respond to. In the subsequent if clause, the current *state* is evaluated by means of a logical proposition. If the truth value of the proposition is 'true', the subsequent then clause is executed. If it is not true, nothing will happen. Executing a then clause consists of generating a *reaction* (like set_phase(cycle,wait) **with** delay = stop_time(cycle) and a *mutation* (like phase(cycle) := wait).

In the first rule, the proposition is true if cycle is in its wait phase and other_cycle is in its move phase. The reaction consists of an agendum of which the act is 'set_phase(other_cycle,stop)'. By giving the delay the specified value, the phase of other_cycle will be changed to "stop" after it has been in its "move" phase for the standard move time, provided it is in its standard "move" phase at the time of executing the rule (cf. Figure 3). If other_cycle is in its prolonged "move" phase at that time, the delay will be zero. This act will become current at time Now + the delay. The effect of the mutation 'phase(cycle) := stop in the second rule is that the phase of cycle changes from "move" to "stop". More precisely, the fact 'phase(cycle) = move' becomes nonexistent at time Now, and the fact 'phase(cycle) = stop' becomes existent at Now. A similar reasoning holds for the mutations in the other rules.

Example: elevator control system

In the period between 1980 and 2000, several 'reference' software problems have gone around in the area of software research and development, for demonstrating the qualities of a development approach, and for comparing appoaches. One of them is the Elevator Control System (ECS). Hereafter, the version of the ECS as elaborated by Edward Yourdon (in his book "Modern Structured Analysis") is copied. The assignment is to produce the essential model of this system according to the π-theory, so as a smartienet. In order to save space, each paragraph of the description will be followed by its analysis (in italics) in the π-theory.

Problem description and analysis
The general requirement is to design and implement a program to schedule and control four elevators in a building with 40 floors. The elevators will be used to carry people from one floor to another in the conventional way.

The program should schedule the elevators efficiently and reasonably. For example, if someone summons an elevator by pushing the down button on the fourth floor, the next elevator that reaches the fourth floor traveling down should stop at the fourth floor to accept the passenger(s). On the other hand, if an elevator has no passengers (no outstanding destination requests), it should park at the last floor it visited until it is needed again. An elevator should not reverse its direction of travel until its passengers who want to travel in its current direction have reached their destinations. (As we will see below, the program cannot really have information about an elevator's actual passengers; it only knows about destination button presses for a given elevator. For example, if some mischievous or sociopathic passenger boards the elevator at the first floor an then presses the destination buttons for the fourth, fifth, and twentieth floor, the program will cause the elevator to travel to and stop at the fourth, fifth, and twentieth floors. The computer and its program have no information about actual passenger boardings and exits.) An elevator that is filled to capacity should not respond to a new summons request. (There is an overweight sensor for each elevator. The computer and its program can interrogate these sensors.)

Because we abstract from implementation, there is no limit to the number of floors or elevators. We will use the variables Floor and Elevator to denote floors and elevators.

Three internal processors are distinguished: P1 (summons requests handler), P2 (destination requests handler), and P3 (controller). There are six environmental processors: CP1 (floor passengers), CP2 (elevator passengers), CP3 (floor sensors), CP4 (overweight sensors), CP5 (motors), and CP6 (operational manager).

The scheduling policy we adopt is that an elevator proceeds moving in a direction (up or down) as long as there are pending destination requests for the elevator in this direction, or if there are one or more summons requests from floors that the elevator will pass when proceeding in the current direction. If there is a summons request from a floor that will not be passed by one of the moving elevators, an 'idle' elevator will be selected and sent to this floor.

The overweight sensor will be modelled as an environmental bank that can be inspected by the controller.

The interior of each elevator is furnished with a panel containing an array of 40 buttons, one button for each floor, marked with the floor numbers (1 to 40). These destination buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a destination button not already lit, the circuitry behind the panel sends an interrupt to the computer (there is a separate interrupt for each elevator). When the computer receives one of these (vectored) interrupts, its program can read the appropriate memory mapped eight-bit input registers (there is one for each interrupt, hence one for each elevator) that contains the floor number corresponding to the destination button that caused the interrupt. Of course, the circuitry behind the panel writes the floor number into the appropriate memory-mapped input register when it causes the vectored interrupt. (Since there are 40 floors in this application, only the first six bits of each input register will be used by the implementation; but the hardware would support a building with up to 256 floors.)

Pressing a destination button is modelled as generating an agendum of the kind "destination_request(Elevator, Floor)" by processor CP2 (elevator passengers) for processor P2 (destination requests handler).

As mentioned earlier, the destination buttons can be illuminated (by bulbs behind the panels). When the interrupt service routine in the program receives a destination button interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by the program's loading the number of the button into the appropriate memory-mapped output register (there is one such register for each elevator). The illumination of a button notifies the passenger(s) that the system has taken note of his or her request and also prevents further interrupts caused by additional (impatient?) pressing of the button. When the controller stops an elevator at a floor, it should send a signal to its destination button panel to turn off the destination button for that floor.

After having accepted a proper destination request, P2 (destination requests handler) creates a fact of the kind "destination_requested(Elevator, Floor) = true", which is contained in bank B2 (destinations requested). This bank can be inspected by the elevator passengers. When visiting the indicated floor, the fact will be replaced by a fact of the kind "destination_requested(Floor, Direction) = false".

There is a floor sensor switch for each floor for each elevator shaft. When an elevator is within eight inches of a floor, a wheel on the elevator closes the switch for that floor and sends an interrupt to the computer (there is a separate interrupt for the set of switches in each elevator shaft). When the computer receives one of these (vec-

tored) interrupts, its program can read the appropriate memory mapped eight-bit input register (there is one for each interrupt, hence one for each elevator) that contains the floor number corresponding to the floor sensor switch that caused the interrupt.

These interrupts are modelled as facts of the kind "approaching(Elevator,Floor) = true | false", created by processor CP3 (floor sensors). They are contained in the aggregate bank AB1 (approaching), which can be inspected by the controller.

The interior of each elevator is furnished with a panel containing one illuminable indicator for each floor number. This panel is located just above the doors. The purpose of this panel is to tell the passengers in the elevator the number of the floor at which the elevator is arriving (and at which it may be stopping). The program should illuminate the indicator for a floor when it arrives at the floor and extinguish the indicator for a floor when it leaves a floor or arrives at a different floor. This signal is sent by the program's loading the number of the floor indicator into the appropriate memory-mapped output register (there is one register for each elevator).

Every time an elevator is approaching a floor, the controller creates a fact of the kind "position(Elevator)" whose value is a Floor. At every moment, there is exactly on such fact for every elevator. These facts are contained in bank B3 (positions), which can be inspected by both floor passengers and elevator passengers.

Each floor of the building is furnished with a panel containing summons button(s). Each floor except the ground floor (floor 1) and the top floor (floor 40) is furnished with a panel containing two summons buttons, one marked UP and one marked DOWN. The ground floor summons panel has only an UP button. The top floor summons panel has only a DOWN button. Thus, there are 78 summons buttons altogether, 39 UP buttons and 39 DOWN buttons. Would-be passengers press these buttons in order to summon an elevator. (Of course, the would-be passenger cannot summon a particular elevator. The scheduler decides which elevator should respond to a summons request.) These summons buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a summons button not already lit, the circuitry behind the panel sends a vectored interrupt to the computer (there is one interrupt for UP buttons and another for DOWN buttons). When the computer receives one of these two (vectored) interrupts, its program can read the appropriate memory mapped eight-bit input register that contains the floor number corresponding to the summons button that caused the interrupt. Of course, the circuitry behind the panel writes the floor number into the appropriate memory-mapped input register when it causes the vectored interrupt.

Pressing a summons button is modelled as generating an agendum of the kind "summons_request(Floor, Direction)" by processor CP1 (floor passengers) for processor P1 (summons requests handler).

The summons buttons can be illuminated (by bulbs behind the panels). When the summons button interrupt service routine in the program receives an UP or DOWN button vectored interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by the program's loading the number of the button in the appropriate memory-mapped output register, one for the UP buttons and one for the DOWN buttons. The illumination of a button notifies the passenger(s) that the system has taken note of this or her request and also prevents further interrupts caused by additional pressing of the button. When the controller stops an elevator at a floor, it should send a signal to the floor's summons button panel to turn off the appropriate (UP or DOWN) button for that floor.

After having accepted a proper summons request, processor P1 (summons requests handler) creates a fact of the kind "summons_requested(Floor, Direction) = true", which is contained in bank B1 (summons requested). This bank can be inspected by the floor passengers. When visiting the corresponding floor, the fact will be replaced by a fact of the kind "summons_requested(Floor, Direction) = false".

There is a memory-mapped control word for each elevator motor. Bit 0 of this word commands the elevator to go up, bit 1 commands the elevator to do down, and bit 2 commands the elevator to stop at the floor whose sensor switch is closed. The elevator mechanism will not obey any inappropriate or unsafe command. If no floor sensor switch is closed when the computer issues a stop signal, the elevator mechanism ignores the stop signal until a floor sensor switch is closed. The computer program does not have to worry about controlling an elevator's doors or stopping an elevator exactly at a level (home) position at a floor. The elevator manufacturer uses conventional switches, relays, circuits, and safety interlocks for these purposes so that the manufacturer can cer-

tify the safety of the elevators without regard for the computer controller. For example, if the computer issues a stop command for an elevator when it is within eight inches of a floor (so that its floor sensor switch is closed), the conventional, approved mechanism stops and levels the elevator at that floor, opens and holds its door open appropriately, and then closes its door. If the computer issues an up or down command during this period (while the door is open, for example), the manufacturer's mechanism ignores the command until its conditions for movement are met. (Therefore, it is safe for the computer to issue and up or down command while an elevator's door is still open.) One condition for an elevator's movement is that its stop button not be depressed. Each elevator's destination button panel contains a stop button. This button does not go to the computer. Its sole purpose is to hold an elevator at a floor with its door open when the elevator is currently stopped at a floor. A red emergency stop switch stops and holds the elevator at the very next floor it reaches irrespective of computer scheduling. The red switch may also turn on an audible alarm. The red switch is not connected to the computer.

The controller can generate three kinds of move commands for the environmental processor CP5 (motors): move(Elevator,up), move (Elevator,down), and move(Elevator,stop). Processor CP5 (motors) will properly deal with these agenda, and respectively create facts of these kinds: "status(Elevator) = moving_up", "status(Elevator) = moving_down", and "status(Elevator) = still". These facts are contained in the aggregate bank AB3 (status), which can be inspected by the controller.

The controller is self-activating at a frequency that is high enough to be able to respond to all agenda in time.

In order to start the operation of each of the elevators, the environmental processor CP6 (operational manager) generates the first control agendum for the controller.

Solution

Based on the provided analysis, we produce right away the detailed smartienet diagram of the elevator control system, in Figure 6. Note: every mutation link in this diagram is considered to 'cover' an inspection link.

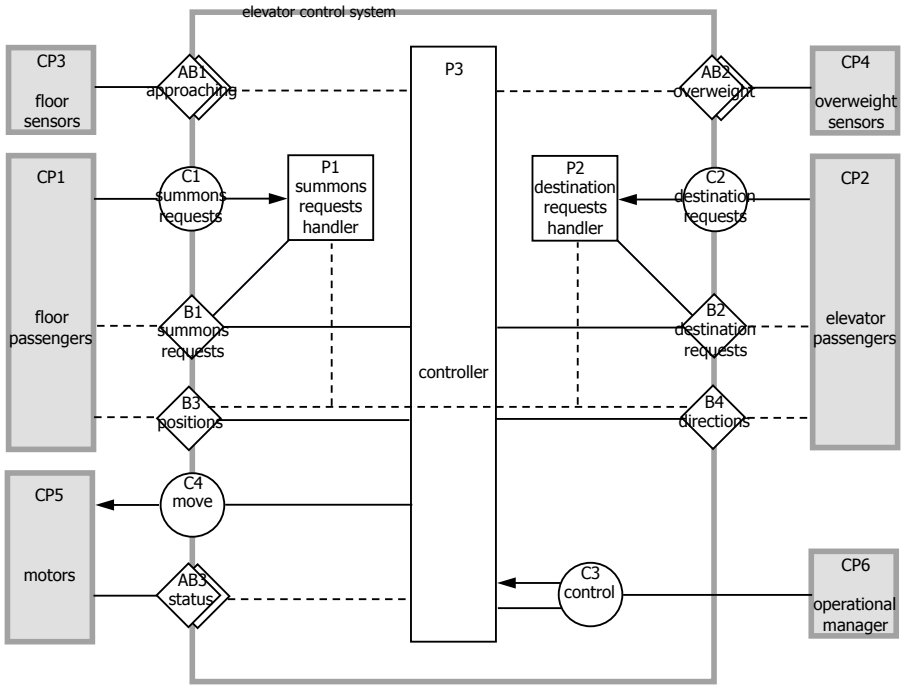

Figure 6. Detailed smartienet diagram of the ECS

The three internal smarties of the smartienet are specified thereafter. In these specifications, a derived fact kind direction(Elevator) whose instances are contained in bank B4 (directions). It is defined as follows:

direction(Elevator) = up:
∃Floor: (destination_requested(Elevator,Floor) = true **or** summons_requested(Floor,up) = true)
and Floor > position(Elevator)
direction(Elevator) = down:
∃Floor: (destination_requested(Elevator,Floor) = true **or** summons_requested(Floor,down) = true)
and Floor < position(Elevator)
direction(Elevator) = still:
direction(Elevator) ≠ up **and** direction(Elevator) ≠ down

Specification of smartie 1
The internal smartie with kernel P1 (summons requests handler) is specified as follows:

S1 = {summons_requested(Floor,Direction), position(Elevator), direction(Elevator)}
M1 = {summons_requested(Floor,Direction)}
A1 = {summons_request(Floor,Direction)}
R1 = ∅
The transition base **T1** is specified as follows:

when summons_request(Floor, Direction) **occurs**
if summons_requested(Floor, Direction) = false **and**
(**there is no** Elevator **for which** position(Elevator) = Floor **and**
(direction(Elevator) = Direction **or** direction(Elevator) = still))
then summons_requested(Floor, Direction) := true

Specification of smartie 2
The internal smartie with kernel P2 (destination requests handler) is specified as follows:

S2 = {destination_requested(Elevator,Floor), position(Elevator), direction(Elevator)}
M2 = {destination_requested(Elevator,Floor)}
A2 = {destination_request(Elevator,Floor)}
R2 = ∅

The transition base **T2** is specified as follows:

when destination_request(Elevator, Floor) **occurs**
if destination_requested(Elevator, Floor) = false **and**
((position(Elevator) < Floor **and** (direction(Elevator) = up **or** direction(Elevator) = still)) **or**
(position(Elevator) > Floor **and** (direction(Elevator) = down **or** direction(Elevator) = still))
then destination_requested(Elevator, Floor) := true

Specification of smartie 3
The internal smartie with kernel P3 (controller) is specified as follows:

S3 = {summons_requested(Floor,Direction), destination_requested(Elevator,Floor), position(Elevator),
direction(Elevator), approaching(Elevator,Floor), overweight(Elevator), status(Elevator)}
M3 = {summons_requested(Floor,Direction), destination_requested(Elevator,Floor), position(Elevator)}
A3 = {control(Elevator)}
R3 = {move(Elevator), control(Elevator)}

The transition base **T3** is specified as follows:

```
when control(Elevator) occurs
        generate control(Elevator) with delay 1
        if      overweight(Elevator) = false
        then    if      status(Elevator) = moving_up and
                        there is a Floor for which approaching(Elevator, Floor) = true
                then    position(Elevator) := Floor
                        if      destination_requested(Elevator, Floor) = true or
                                summons_requested(Floor, up) = true
                        then    move(Elevator, stop)
                else if status(Elevator) = moving_down  and
                        there is a Floor for which approaching(Elevator, Floor) = true
                then    position(Elevator) := Floor
                        if      destination_requested(Elevator, Floor) = true or
                                summons_requested(Floor, down) = true
                        then    move(Elevator, stop)
                else if status(Elevator) = visiting
                then    if      destination_requested(Elevator, position(Elevator)) = true
                        then    destination_requested(Elevator, position(Elevator)) := false
                        if      summons_requested(position(Elevator), direction(Elevator)) = true
                        then    summons_requested(position(Elevator), direction(Elevator)) := false
                        if      direction(Elevator) = up
                        then    move(Elevator, up)
                        else if direction(Elevator) = down
                        then    move(Elevator, down)
```

Discussion

Because of the special qualities of the smartie model (comprehensiveness and essence), the model of the ECS above is quite comprehensible and concise. Just for comparison, in his book Modern Structured Analysis, Yourdon needs 24 pages to model the ECS, and apologises in the end that his model is not complete.

Despite its conciseness, it contains everything one needs to know to understand the statics, kinematics, and dynamics of the ECS. The only things that need to be done in order to make the smartie model operational is to realise it and to implement it.

Realising smartienets

As has been discussed above, smartienets are *essential models* of concrete systems, that is, they show the (ontological) essence of a system, fully abstracted from realisation and implementation. In order to arrive at a a concrete system, using its smartienet model as the starting point, the system has to be realised and implemented.

By the *realisation* of a system is understood the design of an implementable model, starting from the essential model. The process of realisation will be explained using the TCS as the illustrating example.

The first thing is to replace interstriction by interaction. Interstriction is a nice notion that helps intellectually managing the complexity of a system, but practically it is a fiction. Concrete systems can only consist of components with internal memory (remembering the state of the component). So, we need to add informational processors and channels that completely realise the interstriction between smarties at the essential level. For the transition bases of the processors in the TCS model, it would mean that they need to be extended by the commands that are written in green. Of course, the smartienet diagram should be extended accordingly with 'green' processors and channels.

Actually, adding the informational level to the essential level of a smartie model is not sufficient for implementing the system. Also the documental level should be modelled, like it is done for crispie models in the ψ-theory [JDM-5]. We skip this level, however, knowing that almost all smartie models will be implemented in software, and that we thus can do with proper compilers that can compile models that only contain the essential and the informational level of a system.

Here is the extension of the essential level of the TCS model with the informational level:

```
when let_pass(cycle) occurs
        get phase(cycle); get phase(other_cycle);
        if      phase(cycle) = wait and phase(other_cycle) = move
        then    get move_time(other_cycle); get creation_time(phase(other_cycle));
                set_phase(other_cycle,stop)
                with delay = max(0, (move_time(other_cycle) -
                (Now - creation_time(phase(other_cycle) = move)))

when set_phase(cycle,stop) occurs
        get phase(cycle);
        if      phase(cycle) = move
        then    get stop_time(cycle);
                set_phase(cycle,wait) with delay = stop_time(cycle);
                phase(cycle) := stop;
                keep phase(cycle);

when set_phase(cycle,wait) occurs
        get phase(cycle);
        if      phase(cycle) = stop
        then    get clear_time(other_cycle);
                set_phase(other_cycle,move) with delay = clear_time(other_cycle);
                phase(cycle) := wait;
                keep phase(cycle);

when set_phase(cycle,move) occurs
        get phase(cycle); get phase(other_cycle);
        if      phase(cycle) = wait and phase(other_cycle) = wait
        then    phase(cycle) := move
                keep phase(cycle);
```

Implementing smartienets

By *implementation* is understood the allocation of suitable technological means to (the elements of) a realised implementable model, so that the smartienet can become an operating concrete system. In current practice, it means finding an appropriate IT platform on which the compiled smartie model can 'run'. In addition, proper sensors and actuators are needed for implementing the environmental processors.

References

Dietz, J.L.G.: System Ontology and its role in System Development, in: J. Castro, E. Teniente (Eds), Proc. of the CAiSE'05 workshops, University of Porto, pp 273-284, 2005

Dietz, J.L.G., Hoogervorst, J.A.P. et. al: The Discipline of Enterprise Engineering. In: Int. J. Organisational Design and Engineering, Vol. 3, No. 1, 2013, pp 86-114

List of TEEMs (Theories in Enterprise Engineering Memorandum)

TEEM-1: the ω-theory (OMEGA: Organisation's Management, Engineering & Governance Appreciation)

TEEM-2: the τ-theory (TAO: Teleology Across Ontology)

TEEM-3: the δ-theory (DELTA: Discrete Event in Linear Time Automaton)

TEEM-4: the ϕ-theory (FI: Fact and Information)

TEEM-5: the ψ-theory (PSI: Performance in Social Interaction)

TEEM-6: the σ-theory (SIGMA: Socially Inspired Governance and Management Advancement)

TEEM-7: the π-theory (PI: Performance in Interaction)

TEEM-8: the β-theory (BETA: Binding Essence to Technology under Architecture)

TEEM-9: the ν-theory (NU: Normalised Unification)