System Ontology and its role in Software Development

Jan L.G. Dietz

Delft University of Technology
Chair of Information Systems Design
j.l.g.dietz@ewi.tudelft.nl

Abstract. The research reported upon in this paper aims at reducing errors in both the requirements engineering and the system specification phase of software development, by means of system ontology. The notion of system ontology is introduced and discussed. A particular generic ontology is presented as well as an accompanying diagramming technique. The application of system ontology in software development is demonstrated and discussed, using the design of a traffic control system as an example.

1 Introduction

Software development has been put in a bad light for decades. Huge amounts of money are wasted in software development because of cancelled projects, projects coming in over time and budget, and delivered software systems that are never or scarcely used. In [16] it is estimated that these failures amount to an annual and world-wide loss of about $120 billion. Most failures are caused in the early stages, in determining requirements and in devising specifications.

The research that is reported upon in this paper concerns the use of system ontologies in the design process of a software system. By doing this we aim at reducing two major kinds of errors. The first one comprises the failures that are the results of imperfectly performed requirements engineering. They are largely due to the fact that the common strategy to perform this crucial activity is still the 'waiter strategy'. By this is meant the traditional way of asking the user what (s)he wants. The fallacy in this approach is that users can only partly tell what they need, and also that they ask for things they don't need [6]. The second kind of failures are the results of imperfectly devising the specifications of the system to be developed. It is still hard to provide software specifications that are really and fully independent of the programming code and that thus do not restrict or influence the programming freedom in any way (cf. [14]).

The means by which a substantial reduction of these errors could be achieved is system ontology. The notion of ontology is rather new in software engineering. In its original meaning ontology is about what reality is, in fact what being is [2, 3]. It has got almost world-wide attention through the work in the Semantic Web community [1, 17]. In this community, ontology is mainly restricted to the specification of state spaces, largely comparable to conceptual schemata [10]. In our view, the ontology of a system should comprise the complete knowledge of the system, covering all relevant aspects. We define the ontology of a system as the specification of its construction and its operation, independent of its realization and independent of any technology. In this paper we mean by system a technical system or an information system or any software system. Other system categories, in particular social systems like organizations, are excluded. Regarding the ontlogy of enterprises, the reader is referred to [8].

The outline of the paper is as follows. In section 2, the SMART generic ontology is introduced, both in an informal and a formal way. It builds on earlier work in [4, 5, 12, 13]. A graphical notation of ontologies according to the SMART model, which improves the understanding of the generic model, is introduced in section 3. In section 4, a generic model of the design process is presented and the role of ontology is demonstrated in the design of a traffic control system. Discussions of the findings as well as the conclusions that can be drawn are provided in section 5.

2 System Ontology

A definition of the notion of ontology that is well received in the AI community, is that an ontology is a specification of a conceptualization [11]. We like to add that this conceptualization must be a true ontological one in Bunge's sense [2], otherwise said that it is a true white-box model of a piece of reality. Moreover, the ontology of a system should not only cover the static perspective (the state space, which is the one that has got the dominant attention in the Semantic Web community) but also the kinematic perspective (process space), the dynamic perspective (action rules), and the constructional perspective (composition, environment and structure), as well explained by Bunge in [3]. As general requirements for a system ontology we therefore propose that it be coherent, comprehensive, consistent, concise, and essential. By *coherent* we mean that the distinct aspect models constitute a logical and truly integral whole. By *comprehensive* we mean that all relevant issues are covered, that the whole is complete. By *consistent* we mean that the aspect models are free from contradictions or irregularities. By *concise* we mean that no redundant matters are contained in it, that the whole is compact and succinct. The most important property however is that an ontological model is *essential*, that it only shows the essence of the system, its deep structure, abstracted from all realization and implementation issues.

In this section we present the SMART model as an ontological generic model for information systems. We consider these systems to operate in a discrete linear time dimension, which means that there are distinct points in time and that the difference between any two consecutive points in time is always the same. This time difference is called the *time unit*. The variable *Now* represents the current time.

At every point in time *t*, a system is in a particular *state*, which is defined as the set of *facts* that are current at *t*. The facts must be elements of the *state base* of the system, being the set of all facts which may belong to a state of the system. A fact is said to be *current* at the point in time *t* if it has been made existent before or at *t*, and if it has not been made nonexistent since then. (Note. Making a fact existent or nonexistent is the effect of a mutation, which will be explained shortly). Systems activate each other through generating commands. A *command* is a pair <*a*,*t*> where *a* is an action and *t* is a point in time. An action is something that a system must respond to. The set of possible actions that a system can respond to is called the *action base*. At every moment a system disposes of a set of commands, which is called its *agenda*[1]. The action *a* in the agendum <*a*,*t*> is said to be *current* at the point in time *t*. Although mostly only one action will be current at a point in time *t*, there may be a number of concurrent actions. As soon as an action is current, the system will respond to it instantly by performing a *transition*, resulting in the generation of a finite set of facts, and a finite set of commands. The generated set of facts is called the *mutation* of the system. These facts are elements of the *mutation base* of the system, which is the set of all facts the system is able to generate. The facts in the mutation get a time stamp, called the creation time, of which the value is the current time. The generated set of commands is called the *reaction* of the system. The actions in these commands are elements of the *reaction base* of the system, which is the set of all actions the system is able to generate as the action part of a command. A system operates autonomously, which has to be understood as follows. At every point in time the so-called *operating cycle* is passed through. If there is a current action, it will be dealt with, i.e. the corresponding transition rule is executed. If there is no current action, nothing will happen.

The performance of a transition consists, mathematically spoken, of the evaluation of a partial function, which is called the *transition base* of the system. Below, a formal definition of a system according to the SMART model is presented. In this definition, the power set of a set X is denoted as $\wp$X. Time values are represented by natural numbers. The set of natural numbers is denoted by $\aleph$.

A system is defined by a tuple < **S**, **M**, **A**, **R**, **T** >, where:

S : a set of facts, called the *state base*
M : a set of facts, called the *mutation base*
A : a set of actions, called the *action base*
R : a set of actions, called the *reaction base*
T : a partial function, called the *transition base* :
$\mathbf{T} \in \mathbf{A} * \mathbf{S} \rightarrow \wp(\mathbf{R} * \aleph) * \wp\mathbf{M}$

The components **S**, **M**, **A**, and **R** are sufficiently explained above, the function **T** needs further explanation. It can conveniently be represented by its extension, which is a set of *transition rules* of the form <A,S,R,M> where:

A is the current *action set*; it is the set of actions $a \in \mathbf{A}$ that are current;
S is the current *state*; it is the set of facts $f \in \mathbf{S}$ that are current;
R is the current *reaction*; it is a set of pairs <*r*,*d*> with $r \in \mathbf{R}$ and $d \in \aleph$; *d* is the *delay* of the reaction; the action *r* will become current at time Now+*d*;
M is the current *mutation*; $\mathrm{M} \subset \mathbf{M}$;

[1] The word “agenda” is commonly taken as a singular noun. However, the original Latin word is a plural noun, the singular form of it being “agendum”.

If a fact is contained in both the state of a system 1 and the state of a system 2, it means that these systems 'share the knowledge' of that fact. This happens if the fact is current and if it belongs to the intersection of the two state bases, thus to $\mathbf{S}1\cap\mathbf{S}2$. The effect of a mutation by a system 1 on a system 2 is as follows. If a fact in the mutation belongs to $\mathbf{M}1\cap\mathbf{S}2$, then there is an effect, otherwise there is no effect (on the system 2 at least; it may have effect on some other system). There are two possible effects. If the fact is current (which implies that it belongs to the state of system 2), it will be made nonexistent. If the fact is not current (which implies that it does not belong to the current state of system 2), it will be made existent[2]. Likewise, the generating of a command instantly changes the agenda of every system for which the contained action belongs to its action base. More precisely, if the action in a command that is contained in the reaction of a system 1, belongs to $\mathbf{R}1\cap\mathbf{A}2$, then the command will instantly be added to the agenda of system 2.

Because of the distinction between facts and commands, we distinguish between two corresponding kinds of influencing among systems, called information and activation. A system 1 is said to *inform* a system 2 if $\mathbf{M}1\cap\mathbf{S}2 \neq \emptyset$. If this is the case, then every fact belonging to this intersection, produced by system 1, will change the state of system 2 instantly through the addition or the deletion of the fact. A system 1 is said to *activate* a system 2 if $\mathbf{R}1\cap\mathbf{A}2 \neq \emptyset$. If this is the case, then every command that is generated by system 1, and that regards an action that belongs to this intersection, will instantly be added to the agenda of system 2. It is possible for a system that $\mathbf{R}\cap\mathbf{A} \neq \emptyset$. This case is called *self-activation*: apparently, the system is able to cause its own future transitions. In this way periodic activities can be modeled elegantly. Also, it is possible that $\mathbf{M}\cap\mathbf{S} \neq \emptyset$. This case is called *self-information*: the system is able to inspect facts that are generated by itself.

The distinctive difference between information and activation is that activation implies the triggering of a system to perform a transition, whereas information does not. Consequently, a system is not 'aware' of a state change at the time it takes place. Instead, a system takes notice of a state change at some later point in time, namely the next point in time that it is activated and needs to inspect the existence or non-existence of the corresponding fact(s). So, between two adjacent points in time at which a system performs a transition, it 'sleeps'. In that period however a number of state changes may occur. Although we recognize that in the implementation of a software system there is no difference between activation and information, this distinction is crucial at the ontological level of understanding the system. Consequently, we reject the idea that events, i.e. instances of state changes, can cause something; only actions can. The cause of much confusion about this, ontologically clear, distinction, is in the way things are implemented. E.g. the floor sensors in an elevator system send interrupts to the controlling system. They must however be interpreted as informing the controlling system, not activating it. Although several other approaches, like OMG's MDA (Model Driven Architecture) [15], advocate the use of implementation independent models, they are clearly more close to implementation than the SMART model. Moreover, they lack a theoretically sound basis from which the independence becomes evident.

[2] The set-theoretic operator that produces the desired effect is the symmetric set difference, denoted by Δ. For sets A and B it is defined as $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

3 An ontology diagramming technique

The influencing relationships between systems can be made more comprehensible if a collection of co-operating systems is modeled as a smartienet. A *smartienet* is a network consisting of three kinds of components: processors, banks and channels. Four kinds of links are distinguished. The smartienet representation of a system consists of a processor (the kernel of the system) and a number of connected banks and channels. If this processor is composite, the system can be decomposed into a collection of co-operating elementary systems. A system of which the kernel is an elementary processor is called an *elementary system* or a *smartie*. (Note. The notion of elementary processor will be defined shortly). The *environment* of a system is defined as the set of processors with which the kernel of the system has influence relationships, be it activation or information.

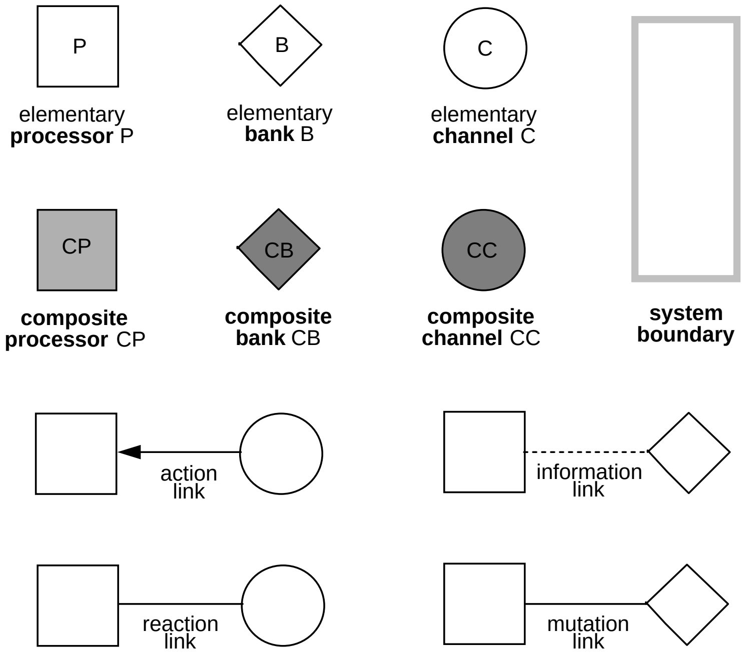
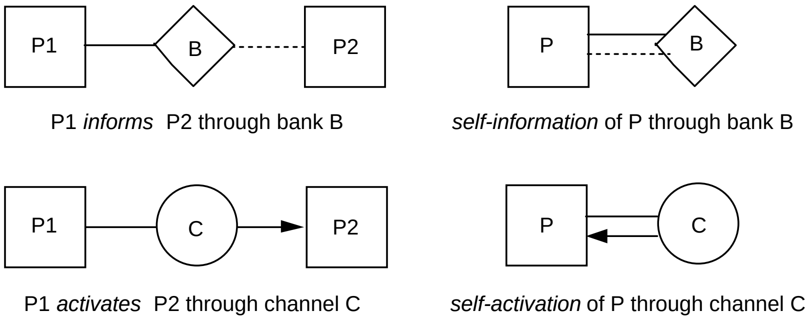


Figure 1 Legend of the smartienet diagram

Figure 1 exhibits the symbolic representations of the components of a smartienet diagram. A *processor* represents the transition mechanism of a system, consisting of its transition base and the operating cycle. Therefore it is also called the kernel of a system. *Banks* serve to communicate facts. To this end, they are able to contain facts. The set of stored facts at some moment, is called the contents of the bank at that moment. The contents of a bank is updated and inspected by processors. A bank is defined by its *storage base*, which is the set of all facts it is able to store. An elementary bank contains facts exactly one fact type, a composite bank contains facts of more than one fact type. The storage bases of the banks in a smartienet are disjoint. *Channels* serve to communicate commands. To this end, channels are able to store commands. The set of stored agenda at some moment, is called the contents of the channel at that moment. The contents of a channel is updated and inspected by processors. When a command becomes current, it is 'emitted'. A channel is defined by its *emission base*, which is the set of all actions it is able to emit. An elementary channel contains commands exactly one action type, a composite bank contains commands of more than one action type. Emission means that the contained action is dealt with by all processors that are connected to the channel through an action link. The emission bases of the channels in a smartienet are disjoint. The *boundary* separates the kernel from the environment. Banks and channels that are shared between processors in the kernel and processors in the environment, are called interface banks and interface channels respectively. They are drawn on the boundary line.

A processor is connected to a bank by means of an inspection link if the storage base of the bank is a sub set of the state base of the system of which the processor is the kernel. Likewise it is connected to a bank by means of a mutation link if the storage base of the bank is a sub set of the system’s mutation base. Next, a processor is connected to a channel by means of an action link if the emission base of the channel is a sub set of the action base of the system of which the processor is the kernel. Likewise it is connected to a channel by means of a reaction link if the emission base of the channel is a sub set of the system’s reaction base. We are now able to define the *elementary processor*. It is a processor that has only one action link. In terms of the SMART model it means that the action base is the extension of exactly one action type. A *composite processor* has multiple action links. Generally it represents a network of elementary processors that are interconnected through banks and channels. Otherwise said, it ‘covers’ a smartienet. In the next section, the notions of elementary and composite processors will be exemplified.



P1 *informs* P2 through bank B

self-information of P through bank B

P1 *activates* P2 through channel C

self-activation of P through channel C

Figure 2. The basic constructs in a smartienet

As examples of the diagrammatical notation of smartienets, the basic constructs in a smartienet are drawn in Figure 2. Bank B is called a *mutation bank* of processor P1 and an *information bank* of processor P2. Likewise, Channel C is called a *reaction channel* of processor P1 and an *action channel* of processor P2. On the right hand side of the figure the self-information and the self-activation of a processor P are exhibited. The bank B is both a mutation bank and an information bank of P. Also, the channel C is both a reaction channel and an action channel of P.

4 The System Design Process

Although almost every systems development method has its own picture of the process of designing and developing a system, they have a common core, as exhibited in Figure 3, be it probably not as articulated as in this figure. The system to be designed is called the object system (OS). It is going to support a system that is called the using system (US). The starting point is the ontology or ontological model of the US. It is the top of a layered set of white-box models (cf. [7]) of the US. Usually, this set of models is extracted from the current US by means of *reverse engineering*. However, it may also have been constructed from scratch. Based on the ontology of the US, the *requirements* concerning the needed support by the OS are determined. The result is a black-box model of the OS, called its functional model. Being a black-box model of the OS means that it is expressed in terms of the white-box models of the US. From this functional model, the *specifications* for the OS are devised. They are expressed in terms of the white-box models of the OS, starting with its ontology. To exemplify this, we take an example from the enterprise domain. A requirement could be that the monthly turnover of an enterprise is provided on the first day of the next month. So, the US is this enterprise. The notion of turnover is meaningful for the actors in the US. In the specifications for the OS, turnover is defined as the outcome of some calculation. Although the same name may be used, the OS 'does not know about turnover', it only 'knows about calculating' (and even that is questionable of course).

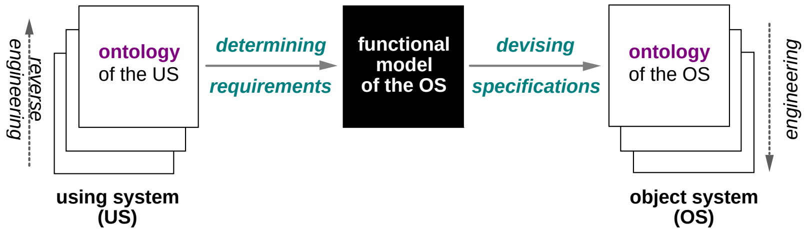


Figure 3. Picture of the design process of a system

The ontology of the OS has to be transformed into more detailed models of which the last one can be implemented directly on a machine. The process of constructing the layered set of white-box models of the OS is called *engineering*. Generally, at least three layers are needed: the ontology, a software construction model (e.g. expressed in UML), and executable specifications (e.g. programs in Java). The importance of the ontological model or specifications is that it provides the basis for identifying the object classes and all other items in the second layer.

To illustrate the design process, we take the traffic at a simple crossing of two roads (called road 1 and road 2) as the US. Let us assume that the responsible municipal officer wants the traffic be controlled and has thought up the next requirements for what (s)he has named the traffic control system (TCS). 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. 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 signaled to stop during an amount of time that is called the stop time. 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 signaled to move on, while the traffic on road 2 is waiting. Figure 4 summarizes these requirements in a graphical way. Of course, the same requirements hold when road 1 and road 2 are exchanged. However, each may have its own standard move time, stop time, and clear time.

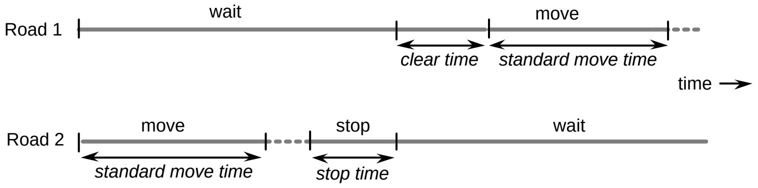


Figure 4. The traffic control cycles and their interdependencies

What we do have now is a *functional model* of the TCS (which is our SS) from which we will devise the ontology of the TCS. Instead of 'road' we decide to use the term 'cycle'. We first present the global ontology of the TCS, in order to clearly set the boundary of the system (cf. Figure 5). There are two interface banks (called 'param' and 'phase'), and one interface channel (called 'let_pass'). Next, two environmental processors are identified. As said earlier, these processors are in principle composite. The processor 'traffic' activates the traffic controller by means of 'let_pass' commands. This processor also inspects the bank 'phase'. The processor called 'supervisor' informs the traffic controller by stating and changing control parameters like e.g. the duration of the "stop" time.

Space limitations prohibit us to discuss the process of constructing the complete ontology of the TCS. We therefore just present the results of this process, being the specification of the components **S**, **M**, **A**, and **R** of the TCS:

S = {phase(Cycle), move_time(Cycle), stop_time(Cycle), clear_time(Cycle)};
M = {phase(Cycle)};
A = {let_pass(Cycle), set_phase(Cycle)};
R = {set_phase(Cycle)};

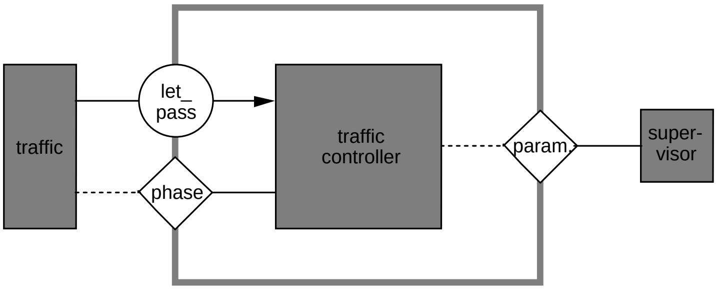


Figure 5. Global ontology of the TCS

Possible facts in the traffic control system are ‘phase(1)=wait’ meaning that cycle 1 is in its wait phase, and ‘move_time(2)=20’ meaning that the standard move time in cycle 2 is 20. The arguments ‘1’ and ‘2’ are values of the variable Cycle. Possible actions are ‘let_pass(1)’ and ‘set_phase(2)’. The transition base **T** can be specified as follows, using a pseudo-algorithmic language that is based on the one in [9]:

```
on let_pass(1) →
        if      phase(1) = wait and phase(2) = move →
                generate set_phase(2,stop) with delay
                        max(0, (move_time(2) - age(phase(2) = move)))
        fi
no

on set_phase(2,stop) →
        if      phase(2) = move →
                generate set_phase(2,wait) with delay stop_time(2);
                phase(2) := stop
        fi
no

on set_phase(2,wait) →
        if      phase(2) = stop →
                generate set_phase(1,move) with delay clear_time(1);
                phase(2) := wait
        fi
no

on set_phase(1,move) →
        if      phase(1) = wait and phase(2) = wait → phase(1) := move
        fi
no
```

The next explanation holds if these rules are executed at time Now. First, between '**on**' and '→' the current action is mentioned to which the system has to respond. Next, the truth value of the condition between '**if**' and '→' is verified. If it is true, the actions between '→' and '**fi**' are performed. If it is not true, nothing will happen. In the first rule, the condition is true if cycle C1 is in its wait phase and cycle C2 is in its move phase. The (single) action consists of generating a command of which the action part is 'set_phase(2,stop)'. By giving the delay the specified number, the phase of C2 will be changed to "stop" exactly after it has been in its "move" phase for the standard move time, provided it is still in its standard "move" phase at the time of executing the rule (cf. Figure 4). If C2 is in its prolonged "move" phase at that time, the delay will be zero. This command will become current at time CT + the delay. There is no change of the state of some system because there is no mutation (M is empty). The effect of the mutation 'phase(2) := stop in the second rule is that the phase of cycle C2 turns from "move" to "stop". More precisely, the fact 'phase(2) = move' becomes nonexistent at time CT, and the fact 'phase(2) = stop' becomes existent at CT. A similar reasoning holds for the mutations in the other rules. The condition 'phase(2) = wait' in the fourth rule is an extra safety condition. Similar rules as the ones above hold for the other cycle, mutatis mutandis.

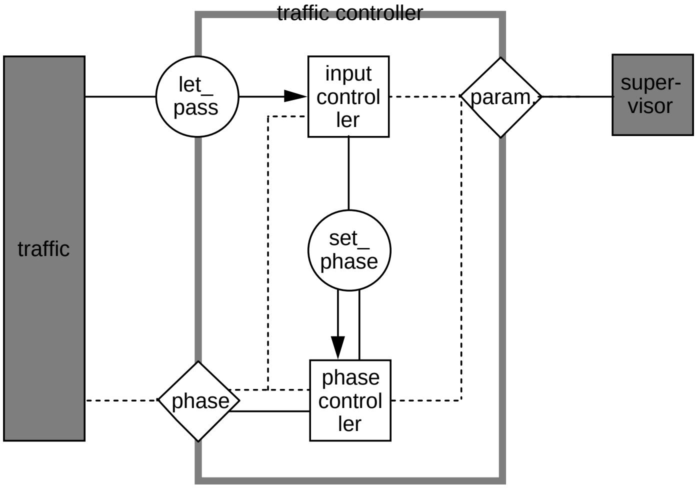


Figure 6. Detailed ontology of the traffic control system

We are now able to show the detailed ontology of the traffic control system (Figure 6). There are two internal elementary processors (called input_controller and phase_controller) and one internal channel (called set_phase). The storage base of the bank 'param' is the union of the extensions of move_time(Cycle), stop_time(Cycle) and clear_time(Cycle). The storage base of the bank 'phase' is the extension of phase(Cycle). The emission base of the channel 'let_pass' is the extension of let_pass(Cycle), and the emission base of the channel 'set_phase' is the extension of set_phase(Cycle). The first action rule above belongs to the processor 'input_controller', the other three to 'phase_controller'. The links between the components of the smartienet follow from the SMART specification of the system.

5 Discussion and conclusions

The notion of ontology, as presented and elaborated in this paper, is a rigorous and fully systemic notion, heavily relying on the work of Mario Bunge [2, 3]. We think that such a rigorously defined notion is necessary for the (re)designing and (re)engineering of technical systems and software systems. From many example cases in several courses at Delft University of Technology it has become apparent that ontologies improve the understanding of requirements and of specifications by system developers. Although the traffic control system is a rather simple system, it showed how hard it is to comprehend a system completely and deeply, but still independent from its (possible) implementation. This is in line with the findings in [14]. The obvious benefits of starting from the ontology of a system (both of the US and of the OS, cf. Figure 3) is the not being hampered by technological issues but instead having the right design freedom to choose among possible implementations.

Moreover, an ontology of a system according to the SMART model can rightly be said to possess the properties that we proposed in the introduction: constructional, coherent, comprehensive, concise, and essential. As discussed, these quality criteria can only partly be met by current modeling techniques, like the UML and OMG's MDA.

References

1. Berners-Lee, T., J. Hendler, O. Lasilla, The Semantic Web, *Scientific American*, May 2001.
2. Bunge, M.A., *Treatise on Basic Philosophy*, vol.3, *The Furniture of the World*, D. Reidel Publishing Company, Dordrecht, The Netherlands, 1977
3. Bunge, M.A., Treatise on Basic Philosophy, vol.4, A World of Systems, D. Reidel Publishing Company, Dordrecht, The Netherlands, 1979
4. Dietz, J.L.G., K.M. van Hee, A framework for the conceptual modelling of discrete dynamic systems. In: C. Rolland, F. Bodart, M.leonard (eds.) *Temporal Aspects in Information Systems*, North-Holland, Amsterdam, 1988, pp. 61-76.
5. Dietz, J.L.G., A Communication Oriented Approach to Conceptual Systems Modelling, in: H.G. Sol, K.M. van Hee (eds.) *Dynamic Modelling of Information Systems*, North-Holland, Amsterdam, 1991, pp 37-60
6. Dietz J L G, J.A. Barjis; Petri net expressions of DEMO process models as a rigid foundation for requirements engineering. In: *Proceedings. 2nd International Conference on Enterprise Information Systems*, Escola Superior de Tecnologia do Instituto Politécnico, Setúbal, Portugal, 2000, p. 267-274. ISBN: 972-98050-1-6
7. Dietz, J.L.G., The Atoms, Molecules and Fibers of Organizations, *Data and Knowledge Engineering*, vol. 47, pp 301-325, 2003
8. Dietz, J.L.G. and N. Habing. *A meta Ontology for Organizations*. In *Workshop on Modeling Inter-Organizational Systems (MIOS), LNCS 3292*. 2004. Larnaca, Cyprus: Springer Verlag.

9. Dijkstra, E.W., A Discipline of Programming, Prentice-Hall Inc. New Jersey, 1976.
10. Falkenberg, E.D., (ed.), *A Framework for Information Systems Concepts*, IFIP 1998 (available as web edition from www.wi.leidenuniv.nl/~verrynst/frisco.html)
11. Gruber, T.R., A translation approach to portable ontologies, *Knowledge Acquisition*, 5(2): 199-220, 1993.
12. Hee, K.M. Van, G-J Houben, J.L.G. Dietz, Modeling of Discrete Dynamic Systems - Framework and Examples. In: *Information Systems*, vol. 14, no.4, pp 277-289. Pergamon Press 1989.
13. Houben, G-J, J.L.G. Dietz, K.M. van Hee, The SMARTIE framework for modelling discrete dynamic systems. In: P. Varaiya, A.B. Kurzhanski (eds.), *Discrete Event Systems: Models and Applications*, Lecture Notes in Control and Information Science 103, Springer-Verlag, New York, 1988.
14. Meyer, B., On Formalism in Specifications, in *IEEE Software*, vol. 3, no. 1, January 1985.
15. OMG, *MDA Guide V1.0.1*, http://www.omg.org/cgi-bin/doc?omg/03-06-01
16. Special report, *The Economist*, November 27th 2004, page 75.
17. W3C, *OWL, Web Ontology Language Overview*, http://www.w3.org/TR/2004/REC-owl-features-20040210/