Reductionist and Antireductionist Perspectives on Dynamics[1]

Catholijn M. Jonker[1], Jan Treur[1,2], and Wouter C.A. Wijngaards[1]

[1]Vrije Universiteit Amsterdam, Department of Artificial Intelligence
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
Email: {jonker,treur}@cs.vu.nl
URL: http://www.cs.vu.nl/{~jonker,~treur}

[2]Universiteit Utrecht, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

Abstract

In this paper reduction and its pragmatics are discussed in the light of the development in Computer Science of languages to describe processes. The design of higher-level description languages within Computer Science has had the aim of allowing for description of the dynamics of processes in the (physical) world on a higher level avoiding all (physical) details of these processes. The higher description levels developed have dramatically increased the complexity of applications that came within reach. The pragmatic attitude of a (scientific) practitioner in this area has become inherently anti-reductionist, but based on well-established reduction relations. The paper discusses how this perspective can be related to reduction in general, and to other domains where description of dynamics plays a main role, in particular, biological and cognitive domains.

1 Introduction

In the philosophical literature on reduction of scientific theories, the advantages of having a reduction relation between two theories for scientific practice are not always addressed explicitly. Often it is implicitly assumed that these advantages are based on the elimination of the higher-level theory. For example, Kim (1996, p. 214-216) emphasizes ontological simplification and having to deal with fewer assumptions about the world. Since the lower-level theory is retained, ontological simplification entails giving up the ontology of the higher-level theory. Such a reductionist perspective provokes resistance from those researchers and philosophers who defend an autonomous status for higher-level theories in the special sciences.

In this paper we distinguish between:

- reduction *in a structural sense* (i.e., a *reduction relation*, already established or being established, between two theories and their ontologies and laws)
  and

[1] Appeared in *Philosophical Psychology Journal*, vol. 15, 2002, pp. 381-409.

- the *pragmatics* related to reduction (i.e., the *use* of an existing or to be achieved reduction relation, by researchers in scientific practice).

Our position is that an actual or envisioned structural reduction is compatible with use and (further) development of the base theory or the theory to be reduced, or both. In practice, thus, the reduced theory is not eliminated.

As a case study, we consider the historical development within Computer Science of languages to describe dynamics of processes. Briefly, the historical pattern is as follows (cf. Tanenbaum, 1976, Knuth 1981, Booch 1991, Kotonya and Sommerville 1998): In the early days of Computer Science, languages were used that described the dynamics of processes by specifying step by step the (physical) transitions, for example. For simple processes this may suffice. However, with a broadening of the scope of applications, these step-by-step descriptions became more and more complex and lacked transparency; therefore higher-level languages that abstract from many of the details of the actual processes became indispensable. Elements were introduced in the higher-level languages by which a description can be structured in terms of increasingly abstract functional units that cover larger parts of the processes (introduced historically in this order: procedures, modules, objects, components, agents, organisations). The result is an increase in the degree of complexity of the phenomena for which there are transparent descriptions.

Each description in one of the high-level languages can be translated into lower-level descriptions, and ultimately into physical processes that also can be simulated within a computer. Translation is automated in a generic manner and hidden from the users of the higher-level languages. In performing simulations in the computer, higher-level descriptions are reduced to lower-level ones. The benefit for programmers is that by working at the more abstract level of the higher-level language, they can keep complexity within the scope of human capabilities; whereas if they tried to work with the lower-level descriptions, the task would quickly become too complex. Thus, the strategy in the areas of Computer Science devoted to scientific modeling, the result is an intrinsically *anti-reductionist* frame of mind in practitioners, but grounded in explicitly defined *reduction* relations.

The paper is structured as follows. First, in Section 2, a brief overview is presented of the development of different levels of language Computer Science. Next, in Section 3 some of the

issues from the literature on reduction are briefly summarized and related to the process description languages case. In Section 4 three different contributions to the literature on scientific explanation and reduction are discussed: from Jackson and Pettit (1988, 1990), Dennett (1987), and Bickle (1998). A common aspect in these three different positions is the inherently anti-reductionist perspective on the pragmatics of explanation. In comparison, the pragmatics of the anti-reductionist perspective on process explanation within practice is discussed. In Section 5, different pragmatic uses of a reduction relation are discussed and compared. Section 6 concludes the paper.

2 Description Levels for Dynamics

Before turning directly to the development of process description languages developed within Computer Science, we consider the role played by dynamics within Cognitive Science and within Computer Science.

2.1 Dynamics within Cognitive Science and within Computer Science

Within Cognitive Science the dynamical perspective, especially Dynamical Systems Theory (DST) is taken as a point of departure; e.g. Kelso (1995), Port and van Gelder (1995)has received much attention. This approach advocates modelling the dynamics of cognitive phenomena using algebraic, difference and differential equations. One of the advantages of DST is that it is able to model the temporal aspects of cognitive events such as recognition time, response time, and time involved in executing motor patterns and locomotion. Many convincing examples have illustrated the usefulness of DST; however, they often only address lower-level cognitive processes such as sensory or motor processing. Some higher-level cognitive processes have been addressed, as in ; Busemeyer and Townsend's (1993) model of decision making. A quantitative approach based on DST has been less successful in describing the dynamics of higher-level processes with a predominately qualitative character, such as reasoning, complex task performance, and certain capabilities of language processing.

One can envision extending the DST repertoire with higher-level description languages that have been developed in Computer Science, thereby increasing their ability to model higher-level cognitive phenomena. These modelling techniques allow high-level expression of temporal relations, i.e., relations between a state of a process at one point in time, and states at

other points in time. Some recent studies on the applicability of such techniques to the dynamics of higher-level cognitive phenomena have shown promising results:

- dynamics of beliefs, desires and intentions: Jonker, Treur, and Vries, (2002); Jonker, Treur, and Wijngaards (2001), and
- dynamics of human reasoning processes: Jonker and Treur (2002a).
- dynamics of complex task performance: Brazier, Jonker, Treur, and Wijngaards (2000); Brazier, Treur, Wijngaards, and Willems (1999); Cornelissen, Jonker, and Treur (2002), see also Section 2.3 below

From the perspective of reduction and its pragmatics, one can ask how descriptions of processes in higher-level languages relate to the processes in the physical world and their descriptions in physical terms, and how these relationships are exploited in practice. The way in which higher-level process description languages have been created and used in Computer Science can be informative.

In Computer Science, the focus is on modelling processes (for example, business processes) in the world (real or being designed) and making a computer perform simulations of these processes. These processes need to be specified precisely. If the behaviour displayed by a computer does not match our expectations, an explanation of this 'fault' is searched for. Therefore, in Computer Science explanation of observed behaviour, both retrospective and prospective, is crucial.

Processes are viewed as evolutions (often called *trajectories* or *traces*) of states over time. States can be states in the world, for example a state within a business process, or states within a computer. For example, within a computer the *central processing unit* (CPU) maintains a physical state in a storage device, based on electrical circuits. If time is taken as discrete, a trace can be described as a sequence S_0, S_1, S_2, … of states. For example, within a computer silicon transistors within the CPU generate such sequences of successive physical states; they are called *simulation traces* or *runs*, the process as a whole is called *computation*.

Describing such sequences of states requires languages for:

(1) *state descriptions*
(2) *descriptions relating states* over time, for example transitions from one state to the other.

At a lowest level of description one can describe states by (combinations of) bits, and transitions of states in a step-by-step manner by modifications of these bits. This is extremely laborious, and the introduction of description languages that allow to specify, for example, that a number of transitions have to take place in a row, or to specify other more complex relationships over time within a trace, has made modeling much easier and extended its power. For example, language elements have been introduced for specifying whole functional units (e.g., procedures, modules, objects, components).

Such languages each have their own specific *semantics*, which specifies howdescriptions refer to processes, independent of how they relate to other, e.g., lower-level, languages. For the languages developed, these semantics usually have been defined in a mathematically precise manner on the basis of mathematically defined (e.g., algebraic) structures that formalise traces of a process: *formal semantics*. Using these languages for more complex phenomena, transparent descriptions can be made. A number of different description levels that have been developed in recent history are briefly discussed; see Figure 1 for an overview of languages at these different description levels.

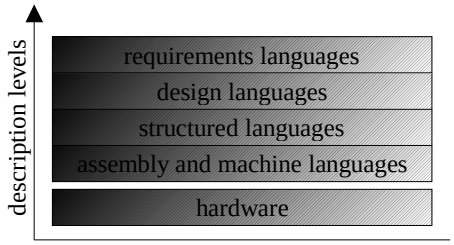

Figure 1. Process Description Languages at Different Levels.

2.2 The Early Programming Language Levels within Computer Science

The first two types of languages briefly discussed are machine languages and assembly languages on the one hand, and structured programming languages on the other hand.

2.2.1 Machine and assembly language

A first approach to describe states and traces makes use of state descriptions based on *bits* (binary alternatives for a state aspect, usually indicated by 0 and 1). For a computer these bits can be related to electrical signals within the hardware. Depending on which electrical circuits are causally affected by these physical bit representations, the CPU successively modifies the physical state into another one. Machine language can be used to describe such state transitions, where the states are described in binary form, usually represented by bit strings; for example: `10010110 00110101 11101011`. For processes within a computer, the bits used by a CPU are of two types: the operation-codes (opcodes) and data-values. The opcodes are simply bits that represent an action to perform (a direct state transition) on the data values; e.g., opcode 01 could mean a left shift of data values in the storage, and opcode 02 could mean a right shift. The opcodes form a language (called the *machine language*) in which the CPU can be given commands. Programming in machine language is done by storing bits of type operation-code on a storage device that will cause the CPU to do the desired computations. An example of an execution trace at this level is as follows:

Machine code trace in binary (bit) notation:

Accumulator	N Z C V flags	RAM at 030B	Program Counter
00010000	0 0 0 0	00000000	0000001100000000
00100000	0 0 0 0	00000000	0000001100000001
00100000	0 0 0 0	00100000	0000001100000100
01000000	0 0 0 0	00100000	0000001100000101
10000000	0 0 0 0	00100000	0000001100000110
10000000	0 0 0 0	00100000	0000001100000111
10100000	0 0 0 0	00100000	0000001100001010

Machine code in hexadecimal notation:

Accumulator	N Z C V flags	RAM at 030B	Program Counter
10	0 0 0 0	00	0300
20	0 0 0 0	00	0301
20	0 0 0 0	20	0304
40	0 0 0 0	20	0305
80	0 0 0 0	20	0306
80	0 0 0 0	20	0307
A0	0 0 0 0	20	030A

Note that in both tables, the program counter value is the value of the program counter just before that particular instruction is fetched. The results of that instruction can be seen on the next line.

An example in 6502 machine code is

```
0A 8D 0B 03 0A 0A 18 6D 0B 03 60 00
```

The 6502 microprocessor is an 8-bit processor running at a 1 Mhz clock speed that was used in such computers as the Commodore 64, Atari, NES and the Apple II, but also in appliances such as televisions and alarm clocks. The representations in the above code are in *hexadecimal* notation. Hexadecimal notation is often used in Computer Science, as it is easier to manipulate bits with them. In this notation, 16 digit symbols are used: `0`, `1` … `9`, `A`, `B`, `C`, `D`, `E`, `F`. In the regular notation, the so-called *decimal* notation, that most people use, these would correspond to the values 0, 1 … 9, 10, 11, 12, 13, 14, 15. To convert a hexadecimal (base-16) notation to the decimal notation (base-10), add the value of the rightmost digit to 16 times the value of the digit to the left of it, plus 16^2 times the digit to the left of that, and so on. Thus, the hexadecimal notation `030B` would be $16^3 \cdot 0 + 16^2 \cdot 3 + 16 \cdot 0 + 11 = 779$. The advantage of hexadecimal notation over the decimal notation is that in hexadecimal notation each digit symbol stands for 4 bits (4 symbols in the binary, base-2 notation). Thus, two hexadecimal digits stand for the value of one byte (8 bits).

That the machine code example, in hexadecimal notation, is at memory location `0300` is denoted by:

```
0300: 0A 8D 0B 03 0A 0A 18 6D 0B 03 60 00
```

This is the bytes in memory starting at location `0300` (in hexadecimal, in decimal notation 768, in memory, further representations are all in hexadecimal notation). To understand what it does, take the example that the processor is reading the instructions at this memory location, i.e., the program counter register has the value `0300`. The accumulator register (called register A) has a value `10` (16 in decimal). The processor chip obtains the value of the program counter memory location from the memory chips. The program counter is then increased by 1, to `301`. The byte `0A` is read from memory, this causes the 6502 hardware to shift the bits of the accumulator register, such that the new content is `20`. This shifting is done by actually routing the electrical current representing the accumulator content through wiring on the processor microchip that is slanted to the left, feeding a zero current to the rightmost bit. At the end of this operation, the status flags in the 6502 processor are `N=0, Z=0, C=0`. The operation is finished and again the value of the program counter memory location, `0301`, is obtained. The value `8D` is read. This causes the subsequent values `0B` and `03` to be read. The program counter is increased by 3, to `0304`. The value `030B` is output on the pins of the processor that connect to the bus[2] leading to the RAM memory. The value of register A, `20` is also output on the pins together with a 'write' operation signal. The memory chips detect this signal and store the value `20` at location `030B` (overwriting the old byte `00` there). The program counter is now `0304`. The value `0A` is read, register A's bits are shifted again. A contains `40`, `N=0, Z=0, C=0`. The program counter advances to value `0305`, where the value `0A` is read again. The register A contains value `80`, this bit value causes the status bits to be `N=1, Z=0, C=0`. The program counter is now `0306`, and the value `18` is read. This causes status bit `C` to be forced to from the old value (`0`) to `0`. The program counter is `0307`, the value `6D` is read. This causes the next two values to be read, `0B` and `03`. The value `030B` is output on the pins of the bus to the memory chips, which respond by putting the value of memory location `030B` on the bus. In this manner the value from memory location `030B` is retrieved, value `20`. This value is then added to register A (`80`), also adding status bit `C` (`0`). The result is that register A contains the value `A0`, the status flags are `N=1, Z=0, C=0, V=0`. The program counter is now `030A`, value `60` is read, causing the program counter to be filled with a value from the stack, ending this example.

What happened here? The machine code at address `0300` took the value in register A, `10` (16 in decimal), and returned `A0` (160 in decimal). The code multiplied[3] it by 10 (in decimal, `A` in hexadecimal). In another program, storing a value in register A, and storing the program counter on the stack, then jumping to location `0300` will put $10 \cdot A$ into register A and pull the program counter from the stack. So in a higher-level language this would be seen as a subroutine for multiplying by 10.

Table 1 An example program in machine language

[2] A bus is a long strip of several parallel wires that connects chips so that each can see and modify the electrical state of each wire. The system bus of a computer typically connects the processor with memory and all other devices.

[3] The 6502 microprocessor has no multiplication built in.

The machine language, whose commands are executed by the CPU itself[4], is taken as the lowest programming language in Figure 1 (Tanenbaum, 1976). Table 1 displays a machine language example. The machine code is written for the 6502 microprocessor. Modelling or programming processes in machine language is a tedious task. In order to facilitate this, a process description language called *assembly* was introduced. In assembly, mnemonic *keywords* can be used to specify the operation codes, and symbolic *labels* can be used to refer to data values Table 2 contains assembly code (a description one level higher) for the 6502 processor. The machine code in Table 1 can be automatically generated from the assembly code in Table 2 using an assembler program.

Below the assembly code to multiply the accumulator register on a 6502 processor by 10 is presented:

```
MULT10  ASL         ;multiply by 2
        STA TEMP    ;temp store in TEMP
        ASL         ;again multiply by 2 (*4)
        ASL         ;again multiply by 2 (*8)
        CLC
        ADC TEMP    ;as result, A = x*8 + x*2
        RTS

TEMP    .byte 0
```

In this assembly code, MULT10 is the entrance to a subroutine that multiplies register A by 10, returning the value in register A. Since the 6502 has no multiplication built in, it has to use addition and bitshifts. An intermediate value is stored in the location TEMP, which is identified in the code as a 1-byte segment of memory with the value 0. The text after ; signs is commentary annotation.

Table 2 An example program in assembly language

2.2.2 Structured programming languages

Assembly and machine languages are closely tuned to the brand and type of CPU that is used in a computer, and are still very tedious to use to model processes. On the other hand, descriptions of algorithms have been developed on a conceptual level that do not refer to any computer (hardware) structure. Such languages have their own semantics, i.e., their own independent way of describing processes (in what is called an *algorithmic* manner), independent of any computer type. It turned out to be possible to define higher-level process description languages, the so-called *structured programming* languages that (1) allow one to describe processes, and (2) can be related in a systematic manner to (lower-level) languages; cf. Knuth (1981). The example in Table 3, taken from Kernighan and Ritchie (1978, pp. 108-110), is written in the structured language C. It does sorting (an array of names for example)

[4] Nowadays, often CPUs do not directly execute their own machine language, but are made for compatibility and efficiency reasons out of one or more smaller, simpler cpus. These smaller cpus sometimes run a fixed program

by the quicksort algorithm. The program from Table 3 will work on a variety of processors. The code contains several functions, subroutines, to divide the work into portions.

States

At this programming language level, instead of states based on bits as in machine language, the states of a process description (or program) are defined in a more abstract mathematical manner by values that are assigned to the variables used in the process description. Moreover, variables can be indexed (arrays). An example of a state is:

```
v[2] : “Athens”
v[3] : “Paris”
v[4] : “Amsterdam”
v[5] : “London”
i : 3
j : 4
temp : “Paris”
```

Variables are indicated by letters or words freely chosen by the programmer as names. Such a language, on the one hand, provides independent descriptions of states that can be related to states in the world, independent of a computer type, at a conceptually higher level. On the other hand, well-defined relationships between such higher-level states and states defined at the level of bits are assumed, but these relationships are kept hidden.

Relations between states: transitions and grouping

Within a program the lines with an = sign in it denote assignment steps, changing the program’s state by assigning a new value to the variable as indicated. Even though many lines are assignments, a lot of assignments are hidden by the language. For example, a function call, such as swap(v, left, last), entails the assignment of values to the arguments. Most of the actions are thus based on assignments. An example of a trace at this level in which Amsterdam and Paris are swapped is:

```
==============
v[2] : “Athens”
v[3] : “Paris”
v[4] : “Amsterdam”
v[5] : “London”
i : 3
j : 4
temp : “”
==============
v[2] : “Athens”
v[3] : “Paris”
v[4] : “Amsterdam”
v[5] : “London”
i : 3
j : 4
temp : “Paris”
```

---

in *their* machine language – called *microcode* – that will make the collection of smaller CPUs simulate the behaviour of the original machine language.

==========
v[2] : “Athens”
v[3] : “Amsterdam”
v[4] : “Amsterdam”
v[5] : “London”
i : 3
j : 4
temp : “Paris”
==========
v[2] : “Athens”
v[3] : “Amsterdam”
v[4] : “Paris”
v[5] : “London”
i : 3
j : 4
temp : “Paris”

As an example, for a description of this process in the structured language C, see Figure 1. For example, the for-loop (i.e., for i = 1 to 100 do …) defines in one expression a sequence of successive steps, each of which involves a further series of steps, which finally are related to transitions of the program’s state; see Table 3. This is an example of a higher-level concept in the language, referring to a more complex process as a whole. Such concepts entail a more sophisticated and structured higher-level relationship over time in the trace of program states than direct transitions.

Below, an example of a program in C, a structured programming language, to sort using the quicksort algorithm.

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
        int i, last;
        void swap(char *v[], int i, int j);

        if(left >= right)     /* do nothing if array contains */
                return;       /* fewer than two elements */
        swap(v, left, (left+right)/2);
        last = left;
        for(i = left+1; i <= right; i++)
                if(strcmp(v[i], v[left]) < 0)
                        swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1);
        qsort(v, last+1, right);
}

/* swap:  interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
        char *temp;

        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}
```

Table 3 An example program in a structured programming language

Descriptions in a structured language such as the language C can be (many-to-many) mapped onto descriptions in assembly language, which can be quite complex in general. Programs

called *compilers* translate from a structured language to machine language (sometimes by intermediate translation to assembly, then invoking an assembler). Descriptions in these structured languages can be far removed from the actual machine language translation. Such a translation is not unique: different compilers that can be used (for example, a Sun cc compiler or a GNU gcc compiler) map the same structured language description onto different lower-level descriptions.

2.3 Higher-level Design and Requirement Specification Languages

Still higher-level languages to describe dynamics have been created, see Figure 1. These languages typically are not procedurally oriented (as are most lower-level languages) around assignment actions and sequences of assignment actions, but instead have a much more abstract view on the states or on the order of state transitions. An example of a language at this level for a specific type of application is the query language SQL, which operates on the domain of databases. This language allows for declarative expression of queries for a database, abstracting from the way in which the actual check of the query within a given database environment is performed. Another example is Prolog, a language in which logical implications can be expressed, and when given a query, the answer to the query can be determined by inferencing. The precise inference procedure used is not specified within the language itself, but contributed by an implementation environment.

Usually the higher-level languages are interpreted by a lower-level program or compiled by a lower-level program. Being interpreted means that during execution another program, written in a lower-level language, reads and executes expressions in the language. Two types of higher-level languages will be discussed in a bit more detail: design languages and requirements languages.

2.3.1 Design Languages

In design languages, the dynamics of a process is specified without specifying a number of the more technical details needed to obtain the behaviour of a process; e.g., Booch, (1991). Component-based design languages are currently considered important; e.g., Brown (1996). By Brazier, Jonker and Treur (2002), a simple system modelling a diagnostic task, is described, an example of using descriptions in the component-based design language DESIRE; see Table 4a. Diagnostic reasoning processes aim at the identification of the cause of a disturbed situation (a fault). In most of these situations not all relevant observational facts

are known in advance. The process of acquisition of additional (observation) information is an essential part of most diagnostic processes. Therefore, dynamics play an important role in diagnosis. In general, diagnostic reasoning consists of a number of sub-processes (performed by specific components of the task) such as the determination of *hypothesis*, the choice of applicable *tests*, the performance of tests and the interpretation of the test results. Strategic information such as the suitability of a test, likeliness of a hypothesis being true and the cost and effect of a test play an important role.

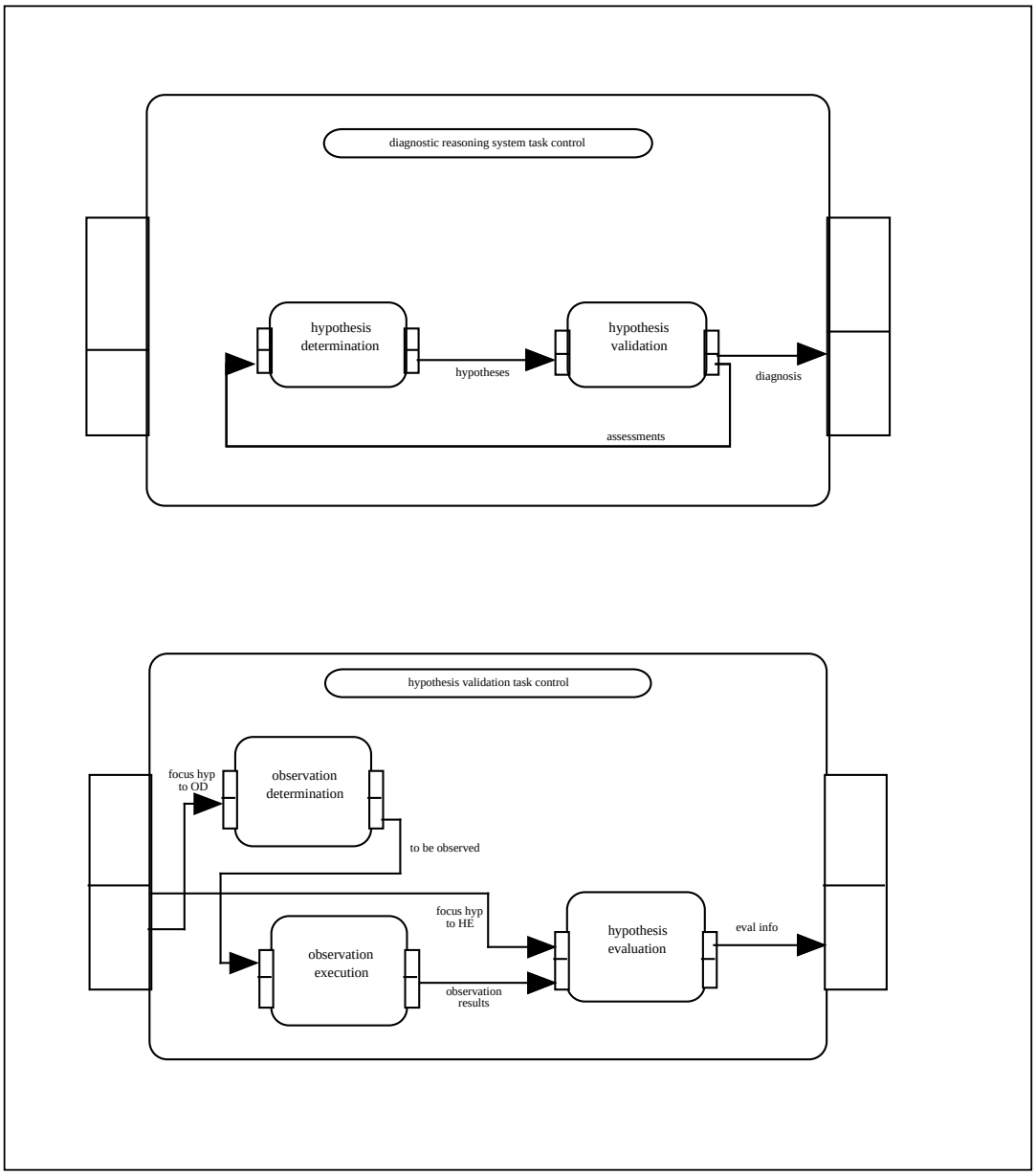

Table 4a An Example Component-Based Design Description in Graphical Form

The overall behaviour of this component-based system can be explained from the behaviours of its components, and the way in which they are composed; cf. Cummins (1975)'s notion of componential analysis. The first picture shows the component structure at the top level, the second picture the component structure within the component Hypothesis Validation. The component Hypothesis Determination generates hypotheses that are validated by the component Hypothesis Validation. A component-based design description of this overall process can be seen in the first figure. The arrows indicate information exchange. They connect the output of one component (small box at the right hand side of a box) to the input of another component (small box at the left hand side of a box). The second figure shows how the component Hypothesis Validation is composed of the components Observation Determination, Observation Execution, and Hypothesis Evaluation. Descriptions in a design language as the example above are a communication device for humans--produced by system designers, and handed over to programmers to be implemented.

For design languages often automated implementation generators exist, translating a design description into a descriptions in a lower-level language, for example a description in the structured language C. Traces for a component-based design as depicted in Figure 4a are high level sequences of states that can be represented by linguistic-like structures (in this case in a relational language), for example of the form depicted in Table 4b. In this trace each transition is a step from the outcome of one component to an outcome of another (next active) component. Compared to the small steps in the form of the transitions to which machine code refers, these transitions are huge steps. One specifies the relations between the states in such a trace on the basis of component-based pictures as in Table 4a, until the components are reached that are not composed: primitive components. These components can have high level specifications based on knowledge bases defining the logical relations between their input and output.

time	Component: Atom
1	OE: observation_result(car_starts, neg)
2	HD: assumed(battery_empty, pos)
3	OD: predicted(lights_work), neg)
4	OE: to_be_observed(lights_work)
5	OE: observation_result(light_works, pos)
6	HE: rejected(battery_empty, pos)
7	HD: assumed(battery_empty, neg)

Table 4b High level trace

2.3.2 Requirement Languages

Requirements languages are used, for example, in determining what requirements a system for a customer should fulfill (see, e.g., Kotonya and Sommerville, 1998). They are often put in natural language, but requirement languages can also be formal. As an example, consider how only the input of a function, and the output related to the input can be specified, such as 'the output has the elements of the input in a sorted order', or 'for input a, the output x satisfies the equation $x^7 + x^2 - ax + 5 = 0$'. Requirements can be specified for a system as a whole, but also for components within a system. An example of a specification in a requirement language of the example diagnostic system discussed in Section 2.2.1 is shown in Table 5.

Below, an example of a requirements specification for a diagnostic system in TTL, a behavioural requirement specification language.

R1	If the system has terminated, then there is at least one hypothesis that has been evaluated and not rejected.
R2	As long as there are hypotheses that have not been rejected and all hypotheses that have been generated have been rejected, the system will keep generating new hypotheses.
R4	Every hypothesis that is generated will be validated which provides an indication of being rejected or confirmed.
R5	For each hypothesis that is generated, all implied relevant predictions about the observable part of the world state are generated.
R6	For each prediction that regards the observable part of the world state, the appropriate observation is made.
R7	Each hypothesis for which there is a prediction that does not match the corresponding observation result, is rejected.

Table 5 Description in a Requirement Specification Language

These requirements express dynamic properties of high level traces and the states of the same type as depicted in Table 4b. Note that in such a requirement states within a trace at arbitrary time points can be related, which allows for specification of dynamic properties completely different from step-like properties, i.e., such expressions define at a high level of description which system states in a trace have to be related over time. Automated translation of these higher-level descriptions to lower levels is possible in specific cases, for example if a commitment is made to a specific type of design.

2.4 Multiple Realisation Relations between Descriptions at Different Levels

In summary, languages have been developed to describe processes within the world or within a computer at different levels. Each of these languages has its own way of referring to process instances or traces in the world, which allows empirical validation of a description. Moreover, this relationship between descriptions and actual process instances is formalised in the semantics of such a language. This picture shows that the different language levels have an independent and autonomous status; they do not depend on each other. Yet, relationships between descriptions in the different languages exist.

When translating a higher-level description into a lower-level description, different translations can be made that have exactly the same effect. Therefore every description level is *multiply realizable* in lower levels. An example: when a high level description of a sorting operation, which only specifies how the output should relate to the input, is translated, this translation could take the form of any of the known sorting algorithms, many of which would be represented by a completely different execution pattern by the hardware, but the result is the same. Descriptions of processes are also *multiply supervenient* in the sense that one description at some level can be related to multiple descriptions at higher levels; cf. Gasper (1992), p. 668. For example, the same behaviour of specific hardware, could have been specified in different manners in a structured programming language, for example, by a Pascal program or by a C program description, with the same results.

If a choice for a particular implementation environment is made, descriptions in higher-level languages can be translated into lower-level ones in an automatic manner. For example by using a compiler or assembler the lower-level code can be automatically obtained, thus choosing one of the possible realisation relations.

To a certain extent, sometimes an automatic translation upwards to a high level description for the lower-level code is possible using special programs such as disassemblers and decompilers (these are typically used in the reverse engineering of competitors' software). The upward translation is of much less quality, in general, as it cannot reproduce the symbolic labellings, when these where not used by the lower level: the decompiling software is not that intelligent in the sense that it can recognize the more abstract patterns.

3 Addressing Multiple Realizability

Both in Computer Science and Cognitive Science multiple realisation occurs. What role do reduction relations between higher-level descriptions and a lower-level descriptions play in such cases?

3.1 The Classical Perspective

Nagel's classical definition of reduction of a theory T_2 (the theory to be reduced) to a theory T_1 (the base theory or reducing theory) is as follows:

a) A *bridge principle* or *bridge law* is a definitional or empirical principle or law connecting an expression of T_2 to an expression of T_1. A bridge principle is *biconditional* if it has the form $a \leftrightarrow b$ where a is an expression of T_2 and b an expression of T_1.

b) A theory T_2 is *Nagel-reducible* to T_1 if and only if all laws of T_2 are logically derivable from the laws of T_1 augmented with appropriate bridge principles connecting the expressions of T_2 with expressions of T_1.

The key concept here is the existence of bridge principles. In practice, these bridge principles have to be biconditional to permit the possibility of deriving nontrivial T_2-laws from T_1 laws, thereby satisfying b).

How are these concepts related to process description languages developed within Computer Science? Suppose two descriptions are given, a higher-level description and a lower-level description. For the theory to be reduced, T_2, the higher-level description is taken; for the base theory T_1 the lower-level description. Bridge principles relate expressions that are part of the higher-level description to expressions in the lower-level description. At first sight it may seem that such principles indeed exist. They are well-defined in a mathematically precise manner and even automated in compilers that translate any description in terms of the higher-level language to a description in terms of the lower-level language. However, as discussed in Section 2.4, one higher-level description can be translated into many lower-level descriptions: each description in one of the language levels depicted in Figure 1 is multiply realizable in the lower levels.

This situation is quite similar to the situation in Cognitive Science, where it has also been argued that multiple realizability occurs so that there is not one unique set of bridge

principles. Kim (1996, Ch. 9) outlines three alternatives approaches for coping with a multiple realizability in Cognitive Science:

- Reduction using a set of bridge principles based on disjunctions of the lower-level properties specified in the different realisations
- Supervenience
- Local reduction, based on multiple sets of context-specific bridge principles

Each of these options will be briefly discussed.

3.2 Bridge Principles based on Disjunctions

The first option posits a set of bridge principles of the form

$$M \leftrightarrow P^d$$
$$N \leftrightarrow Q^d$$
$$\ldots.$$

where

P^d is a disjunction $P_1 \vee P_2 \vee ...$

Q^d is a disjunction $Q_1 \vee Q_2 \vee ...$

….

where, P_1, P_2, ... are the different realisers (indicated by expressions in T_1-ontology) of higher-level property M (in T_2-ontology). The problem is that there may be an indeterminate (possibly infinite) number of 'wildly heterogeneous' possible realizers . One cannot, using the vocabulary of the lower level, specify the set of disjunctive realizers. A similar problem arises with respect to process description languages in Computer Science since there to there is an endless variation of lower-level programs that do essentially doing the same, viewed from a higher-level perspective.

3.3 Supervenience

Kim (1996, 1998) proposed that the principle of superveneince provided a nonreductionist way to cope with multiple realisation from a physicalist perspective. He explains supervenience as follows:

> Mental properties supervene over physical properties in that for every mental property M that occurs at some point in time t, there exists some physical property P that also occurs at t, such that always if P occurs at some point in time t', also M occurs at t'.

This seems to cover the situation for process description languages from Computer Science at different levels well. However, a disadvantage of this abstract notion of supervenience is that it does not give specify how higher-level descriptions relate to lower-level descriptions. To overcome this disadvantage Kim introduces the notion of local or context-specific reduction.

3.4 Local or Context-Specific Reduction

In a context-specific reduction (Kim, 1996, pp. 233-236) the aim is not to find *one set* of bridge principles, but to accept *multiple sets* of context-specific bridge principles. In this case at each instance of time, each higher-level description can be related to a lower-level description based on an *appropriately chosen* context-specific set of bridge principles. The contexts are chosen in such a manner that all situations in which a specific type of realisation occurs are grouped together, and are jointly described by one set of bridge principles. Thus, within the formulation of supervenience given above, the set of all P (the variable of the existential quantifier) that satisfy the clause 'that also occurs at t, such that always if P occurs at some point in time t', also M occurs at t' ' is partitioned into subsets each of which defines a context. In Cognitive Science such a grouping could be based on species, i.e., groups of organisms with (more or less) the same architecture, although objections may be put forward against this granularity of grouping; it might well be the case that certain mental properties have different realisations over organisms of the same species, or even different realisations within one organism over time.

In the context of an organism or system with structure or architecture description S, biconditional bridge principles can be stated in a conditional manner as follows; cf. Kim (1996), p. 233:

$$S \rightarrow (M \leftrightarrow P)$$

This means that for all systems with structure S the bridge principle $M \leftrightarrow P$ applies. For systems with another structure, other bridge principles apply. This also generates a disjunctive form, but logically different from the one above; namely instead of

a set of disjunctive bridge principles (as discussed in Section 3.2)

this time in the form of

a disjunction of sets of (non-disjunctive) bridge principles:

(case of S_1)	(case of S_2)	(case of S_3)	…..
[$M \leftrightarrow P_1$	$\vee$	$\vee$	$\vee$
$N \leftrightarrow Q_1$	[$M \leftrightarrow P_2$	[$M \leftrightarrow P_3$	[….
….]	$N \leftrightarrow Q_2$	$N \leftrightarrow Q_3$	
	….]	….]	]

For the case of process description languages within Computer Science, an implementation environment (including a compiler) of the person's choice can play this role architecture S, and its context-specific set of bridge principles. A different choice of implementation environment will come up with a different set of bridge principles and, hence, with a different lower-level description for the same higher-level description. To be more precise, in the case of process description languages from Computer Science the role of S above is played by the specification of the compiler that is used to relate high level descriptions to low level descriptions; such a compiler is the core of an implementation environment. Indeed, committing oneself to one specific compiler can be described as the situation where only one set of bridge principles is used. Putting aside the difference in interpretation that in one case bridge principles are involved that may be considered definitional and in the other case as empirical correlation laws, (Kim, 1996, p. 213), the situation for process description languages within Computer Science is, in a structural sense, similar to the situation that one psychological theory of pain T_2 can be related to different realisations in the form of physiological theories T_1 in different species (or even within different animals within one species), as put forward by Kim (1996), Ch. 9, pp. 233-236. A choice for a particular implementation environment for a high level program can be compared to the choice of a particular species for developing a higher-level cognitive theory.

4 Explanation and Levels of Description

In this section we discuss why explanations at different levels are useful for practice, from the angle of the pragmatics of explanation. First the perspective developed by Jackson and Pettit (1988, 1990) is discussed (their 'program explanation' as opposed to causal explanation). They argue that a program explanation is more useful because it has a wider scope of applicability (in possible worlds) than a causal explanation at the physical level (in the actual

world). Subsequently, Dennett (1987)'s view on the use of intentional stance versus physical stance explanations is addressed. Dennett emphasizes that intentional stance explanations are tractable in cases where physical stance are not. Next, Bickle (1998)'s perspective on folkpsychological explanations versus neurobiological explanations is discussed. He emphasizes the value it can have to keep and improve the higher-level theory as a basis for explanation, in addition to a neurobiological theory. Finally, the use of explanations in Computer Science is discussed and compared.

Like us, Jackson and Pettit (1988, 1990) exploit a metaphor from Computer Science is exploited. They develop a notion of higher-level explanation, meant to be suitable for special sciences such as Biology, Cognitive Science and Social Sciences: *program explanation*. According to this type of explanation, 'G occurred because F occurred' for higher-level properties F and G, can be an adequate explanation in the following way: F ensures ('programs for') some lower-level property P, which causes G. Or: F ensures ('programs for') some lower-level property P, which causes a lower-level property Q for which G is a higher-level description. For example, the question 'Why was the vase breaking ?' can be answered by: 'Because it was fragile'. Here the higher-level property of being fragile ensures or programs for the lower-level property of having a specific molecular structure. They explain the name 'program explanation' as follows:

> 'The property-instance does not figure in the productive process leading to the event but it more or less ensures that a property-instance which is required for that process does figure. A useful metaphor for describing the role of the property is to say that its realization programs for the appearance of the productive property and, under a certain description, for the event produced. The analogy is with a computer program which ensures that certain things will happen – things satisfying certain descriptions – though all the work of producing those things goes on at a lower, mechanical level' (Jackson and Pettit, 1990), p. 114

They discuss the value of such a higher-level explanation, using the example of explaining radiation from the decay of atoms, as follows:

> 'According to (Lewis, 1988), to explain something is to provide information on its causal history . . . A program explanation provides a different sort of information . . . A program account tells us what the history might have been. It gives modal information about the history, telling us for example that in any relevantly similar situation, as in the original situation itself, the fact that some atoms are decaying means

> that there will be a property realized - that involving the decay of such and such particular atoms - which is sufficient in the circumstances to produce radiation. In the actual world it was this, that and the other atom which decayed and led to radiation, but in possible worlds where their place is taken by other atoms, the radiation still occurs. ' (Jackson and Pettit, 1990), p. 117.

Jackson and Pettit emphasize that such a form of higher-level explanation, based on some theory T_2 has advantages over causal explanation, based on a basic theory T_1, in the sense that other information is provided, which implies *increased genericity*: it not only applies to the actual world, but also to other possible worlds.

As opposed to explanations from a direct physical perspective (the *physical stance*), Denett, (1987, 1991) advocates use of the *intentional stance*. He points to different description levels with ontologies for emerging patterns in the simulation environment Life to explain the advantage of explanations using such higher-level descriptions; cf. Dennett (1987), pp. 37-39; Dennett (1991), pp. 37-42. In addition, he uses different description levels in a computer system (actually of a chess computer), embedded (and hence visualised) in the two-dimensional Life environment as a metaphor to explain the advantage of intentional stance explanations for mental phenomena over physical stance explanations:

> 'The scale of compression when one adopts the intentional stance toward the two-dimensional chess-playing computer galaxy is stupendous: it is the difference between figuring out in your head what white's most likely (best) move is versus calculating the state of a few trillion pixels through a few hundred thousand generations. But the scale of savings is really no greater in the Life world than in our own. Predicting that someone will duck if you throw a brick at him is easy from the folk-psychological stance; it is and will always be intractable if you have to trace the protons from brick to eyeball, the neurotransmitters from optic nerve to motor nerver, and so forth. ' (Dennett, 1991), p. 42

Dennett puts the emphasis on tractability. To explain more complex phenomena in our real world, higher-level explanations are tractable, whereas lower-level explanations are not.

Bickle (1998, pp. 199-211) discusses an approach to reduction, called *revisionary reduction*. Three conditions are put forward that separate this account from retentive and replacement reduction (pp. 200-201):

1. Explanations generated by T_R approximate the actual events causing the observable phenomena
2. Key explanatory concepts of T_R fragment into several distinct concepts of T_B, and often each of the latter appropriate candidates for cross-theoretic "identification" lie at appropriate limits or within particular domains of application of T_B.
3. Revisionary cases of reduction display mutual evolutionary feedback between T_R and T_B, where developments within each theory serve to mutually constrain and promote fruitful development within the other, but especially within T_R.

In a more detailed manner, Bickle (1998, pp. 205-208), illustrates this account for the higher-level (e.g., folk psychological) and lower-level (e.g., neurobiological) explanation in the context of Hawkin and Kandel's (1984a,b) case:

> 'Of course, the functional profiles assigned to cognitive states on Hawkin and Kandel's neurobiological account are much more fine-grained and detailed, for that account recognizes distinctions and connections that folk psychology either lumps together or leaves extremely vague . . . Here again, however, we can expect that injection of some neurobiological details back into folk psychology would fruitfully enrich the latter, and thus allow development of a more fine-grained folk-psychological account that better matches the detailed functional profiles that neurobiology assigns to its representational states. There is no principled reason against such enrichment.' (Bickle, 1998), p. 207-208

Here Bickle proposes that by relating a folk psychological explanation to a neurobiological account, a decision can be made to enrich the former by introducing some new intermediary states, based on the more detailed path provided by the latter. He proposes to use a reduction relation in scientific practice not to actually reduce the theory T_2 to a theory T_1 so as to eliminate T_2, but to extend or improve the theory T_2 on the basis of theory T_1. Therefore, abandoning explanation on the basis of T_2 and replacing such explanation by explanation on the basis of T_1 is not at issue; on the contrary, the explanatory value of T_2 is strengthened by the process of co-evolution of T_2 and T_1.

4.4 Explanation and Levels of Description in Computer Science

Working with process descriptions within Computer Science, two perspectives occur: the perspective on a process to be designed, before it came into existence, and the perspective in retrospect, after the process has been described. Accordingly, two sorts of explanation are performed: explanation in advance and explanation in retrospect. For an explanation *in*

advance (i.e., a prediction), the abstract view on the behaviour of the process aimed for (i.e., intended by the modeller or programmer) is specified in some higher-level language. By using an implementation environment based on an automated compiler, machine language for the hardware to simulate this behaviour is generated. So, if you were to ask for an explanation why a particular computer system will display a particular behaviour, the answer would be in one of the higher levels of description in Figure 1. It can be said that the computer will do something because some electrical current passed some electrical circuitry, or, it can be said that the computer will do something, because in some description language a particular expression had been given. In this sense, the descriptions in the different languages are an explanation of why the computer subsequently will perform a particular behaviour. However, for the programmer the higher-level explanation is feasible and useful, whereas the lower-level explanation usually is not.

Providing an *explanation in retrospect* is typically observed during simulation (or execution) of a process description (program) on a computer, for example in the context of the search for *bugs*. A bug is an error in a process description, resulting in a deviation from the behaviour that the modeller (or programmer) aimed for. A bug will thus display behaviour that deviates from the expectations of the modeller. This behaviour, however, is still in accordance with the description of the behaviour in the language that the modeller used, and all subsequent lower languages that it had been translated into. The modeller will want an explanation of why the computer system shows this unexpected behaviour. The modeller needs to know why the bug happened, so as to fix all possible occurrences of the bug. In principle, several types of explanations are possible for a bug.

The lowest level explanation is that the wrong electrical currents went through the circuitry. The electrons inside the hardware went the wrong way. In order to fix this, the modeller would need to change the circuitry. However, this would only stop the bug from happening on that machine. Using the same not corrected higher-level description will result in similar deviating behaviour in all other cases that the higher-level process description is used. Next, an explanation can be given in machine code, for example that a certain machine register contained the wrong value. Fixing the machine code would fix the particular occurrence of the bug, but only for that particular type of cpu. The bug could still manifest on other types of CPU.

The deviating behaviour can also be explained in the higher-level language that the modeller used to specify the behaviour. There, a certain expression can be deemed responsible: execution of the program results in the deviating behaviour, because the expression is not adequate. This is the type of explanation the modeller wants, since (1) it is much easier to understand for the human at this level what is wrong, and (2) by correcting this higher-level description, the behaviours for all (automatically generated) realizations of the description are corrected.

In Computer Science, the explanation at a higher description level is considered more valuable. Not only will it allow for fixing the problem, but also more abstract languages typically specify more of the behaviour per expression used. As such, in high level descriptions a larger subset of behaviour is specified. When a bug occurs, the most abstract description is the most useful, as it covers the exact set of occurrences that could happen in any implementation environment.

One minor comment can be made here. A modeller assumes that the implementation environments she or he uses are adequate. In other words, the context-specific bridge principles are assumed correct. An observed deviation in behaviour is not attributed to this environment and these bridge principles, but rather to the higher-level program; in 99.9 % of the cases this is an adequate strategy. In exceptional cases, however, it is possible that a bug is present in the compiler software, especially when features are used that were not extensively tested or verified earlier. A modeller may consider this possibility if a bug is displayed whereas the higher-level process description seems adequate. For the sake of the argument we exclude this situation.

In conclusion, the different Computer Science process description languages can be considered useful for explaining the behaviour of processes in the world or in computer systems. They are used to explain what a process is intended to do, and they are used to explain what a process has done. Different levels of description in these process description languages occur. A description at one level is multiply realizable in several possible lower-level descriptions. Depending on a specific implementation environment chosen, a description at a higher level can be translated into a specific description at a lower level in an automatic manner. Each level of description can be used to explain a particular behaviour. In practice, explanations are considered more valuable when expressed in terms of a higher level of

description. Due to the fact that reductions to lower levels can be completely automated, the programmer need not have any knowledge of the lower levels.

To increase the scope of applicability, the strategy in the scientific area of Computer Science has been to increase the distance between the relevant complex physical processes and the conceptual explanations of them. The success of this strategy was only possible because of the development of high-level modelling languages and supporting software environments (relating the high level languages to the lower-level ones in a hidden manner) that enabled practitioners to work on a high level of description without the need of technical expertise of the lower-level languages: these details are hidden from them by the software environments used.

5 Pragmatic Strategies

If reduction relations between theories have been established, still choices can be made on how to exploit these relationships in scientific practice. A number of possible choices are the following.

(1) One possible choice is to *actually reduce* theory T_2 to theory T_1, with elimination of theory T_2, including its ontology and its laws. For example, scientific texts will only include terms of T_1, not of T_2.

(2) Another possible choice is to *identify* theory T_2's ontology with expressions of theory T_1, without actually eliminating theory T_2, but still using its ontology and its laws, knowing that they are identical to certain expressions of T_1. In this case, scientific texts can include terms of both of T_1 and T_2 and their relationships, where the terms of T_2 are considered synonymous to expressions of T_1. The availability of a reduction relation can be considered as *additional knowledge* that provides a better foundation and embedding for T_2, thus increasing the value of T_2. Therefore motivation to use T_2 and its relation to T_1 has increased.

(3) A third choice may be that, if it has been established that T_2 has a solid foundation in terms of T_1, in scientific practice *more emphasis* is put on – whenever applicable - using T_2 instead of T_1; in a sense this choice in a certain domain of scientific practice can be considered as eliminating or ignoring or abstracting away T_1. Scientific texts on certain topics will only include terms of T_2 , not of T_1.

Kim (1996) puts a number of advantages of reduction relations between a theory T_2 and a theory T_1. These and some other advantages will be discussed in the context of these choices on how to exploit reduction relations; see also Table 7, where choices (1) and (2) are compared.

5.1 Ontological Simplification

Kim (1996, pp. 214-216), proposes ontological simplification as an advantage in the case where theory T_2 is actually reduced to theory T_1: choice (1) above. Having two theories T_1 and T_2 with their own ontologies indeed means having to learn and being aware of more terms than if only the terms of T_1 are sufficient, because the terms of T_2 are replaced by those of T_1. For example, in a case of full reduction between two theories with an approximately equal number of basic terms, the savings is a factor 2. However, in a case of context-specific reduction, say with 10 different theories $T_1^{(i)}$ with number of basic terms approximately equal to the number of terms as T_2, the advantage can be estimated as 11/10, i.e. a factor of 1.1. So in this case, numerically the advantage cannot be made very clear. Other non-numerical arguments might be put forward not directly related to numbers of basic ontological terms; however the above estimations make clear that the value of the advantage of ontological simplification is not so self-evident as may seem at first sight. It is clear that ontological simplification does not take place when both T_2 and T_1 are kept in use, as in choice (2) above. In choice (3), however, ontological simplification may take place in the sense that in scientific practice T_1's ontology is used less extensively. In this case the numerical advantage for 10 theories $T_1^{(i)}$ is approximately 11/1, a quite drastic simplification. As discussed in Section 4.4, this actually has happened in Computer Science, where developers no longer need to know the ontologies and knowledge underlying all the different platforms and implementation environments developed in the past (and often still in use). Instead they can work with a much more restricted high level ontology.

5.2 Less Assumptions on the World

A second advantage of reduction Kim (1996, pp. 214-216), puts forward is that the number of assumptions about the world is lower. By reducing T_2-laws to statements that can be derived from T_1-laws, logical dependencies are identified. This can be considered a substantial advantage, since if some assumptions are logically implied by other assumptions, it is good to know about these relationships. This is independent on the number of assumptions being lower in a numerical sense (the same numerical considerations as before

apply). This advantage arises both if T_2 is actually reduced to T_1 in the sense of being eliminated (choice (1)), and if T_2 is not actually reduced (choices (2) or (3)).

5.3 Insight in Underlying Mechanisms

A third advantage of reduction of a theory T_2 to a theory T_1 Kim (1996, pp. 214-216) advances is that reduction gives more insight in underlying lower-level mechanisms. As an example, the discovery of the structure of DNA and of how segments on DNA play a causal role in the inheritance of properties, gave an extended insight in how things work in the world. The biological notion of a gene, which could be defined in a functional manner within biology, now could be related to a realisation in the form of chemical structures and mechanisms. It is a question whether this advantage is only an advantage of actually reducing one (biological) theory into another (chemical) theory, or that the advantage is that as yet unknown relationships will be found between two theories that both had and will have their independent value (choice (2)), or that this insight is the basis for a decision to take choice (3): using more exclusively T_2 with increased confidence. This is the question whether advantages of reduction are introduced by actually eliminating one of the theories, or by obtaining additional knowledge about relationships between two theories which both will remain to have their - maybe even increased - value. It seems that the advantage of insight in underlying mechanisms can count in all four choices as considered. The example put forward by Bickle in Section 4.3 above, suggests how by going back and forth between a folkpsychological T_2 and a neurobiological theory T_1, the theory T_2 can be improved. Within Computer Science this can be compared with a process of improving the effectivity and efficiency of a higher-level description by going back and forth to lower levels.

5.4 Increased Empirical Test Possibilities

A fourth advantage of reduction is that reduction provides new empirical possibilities. Knowing, for example, that a specific type of perceptual processing relates to activation within a certain brain area makes it possible for brain imaging to test theories about perceptual processes. This advantage counts if T_2 is not actually reduced to T_1, since the empirical possibilities for T_2 are increased, not for T_1. In a case of actual reduction (choice (1)), this advantage does not count, as T_1's empirical possibilities remain the same. For choice (2) this may count as a substantial advantage, but not for choice (3), as in that case T_1 is more or less 'abstracted away', which throws out (with the bathwater) the additional empirical possibilities that T_1 could provide.

5.5 Transparency

Another contribution of reduction to scientific practice is increased human understanding due to the tractability, or transparency of theories. Scientific practitioners often consider this quite important. On this criterion, choice (1) usually has the disadvantage that more complex descriptions are involved in an explanation. As a higher-level theory often is more transparent and understandable than a lower-level one, elimination of T_2 and its basic ontology typically decreases transparency. The danger is not seeing the forest anymore because of the trees. Dennett's explanation of why you dive when I throw a brick (discussed in Section 4.2 above) makes this vivid. Choices (2) and (3) have the advantage that transparency is kept, or even increased if a shift to T_2 is made, thereby leaving behind T_1, as in choice (3).

	Why actually reduce higher-level descriptions to lower-level descriptions: choice (1)	*Why use higher-level descriptions in addition to lower-level descriptions: choice (2)*
ontology	Ontological simplification: less terms to describe phenomena in the world.	Adding additional ontologies to improve understanding; simplification if restricting to higher-level ontologies. Knowledge of how higher-level terms relate to lower-level terms is valuable in addition.
assumptions	Less assumptions on the world.	Assumptions about the world made more understandable. Relations between assumptions on different levels of description are identified.
insight	More insight in underlying lower-level mechanisms for higher-level phenomena.	More insight from a conceptually higher level perspective. By going back and forth between higher-level and lower-level descriptions, a lower-level theory T_1 can have impact on progress in the development of the higher-level theory T_2
validity	Empirical test possibilities at the lower level.	Possibilities for verification and validation at different levels: on the lower level, and on a higher level of description.
transparency	----	Analysis and design of more complex systems possible.
genericity	Context and scope of application narrowed down to the context of the local reduction.	Wider scope of application; in computer system terms: less implementation-environment dependent.

Table 7 Comparison in Brief between two Different Choices in Pragmatics

5.6 Genericity

Jackson and Pettit (1988, 1990), as discussed in Section 4.1, raise the issue of genericity. A higher-level explanation applies to more application contexts than a lower-level explanation. This was also emphasized in the context of process description languages from Computer

Science in Section 4.4. If choice (1) is made, this entails a choice for lower-level explanations with a more limited scope of application. For choices (2) and (3), also higher-level explanations remain or even become more extensively available.

6 Conclusion

In this paper we have discussed scientific strategies related to reduction and its pragmatics. In this conclusion we summarise the lessons that can be learned from our case study of the development of process description languages within Computer Science.

6.1 Why Reduction Can Reinforce a Higher Level Theory

The design of higher-level description languages enabled programmers or modelers to describe processes in the world on a high level and thus avoid addressing physical details directly. This meant that more complex applications could be covered. Although the pragmatic strategy of a practitioner is anti-reductionist, well-established context-specific reduction relations, automated and hidden in specific implementation (software) environments, assure that a solid relation with a physical reality is maintained.

This situation has some similarities to the perspective of local (species-specific) reduction within Cognitive Science, as put forward, for example, by Kim. The case study is also compared to other positions on explanation that emphasize the advantages of taking some distance to physical reality, such as Jackson and Pettit (’s program explanation, Dennett’s intentional stance explanation and folk-psychological explanation in the context of ‘new wave reduction’ as put forward by Bickle. Some aspects of the discussion on reduction can be clarified by distinguishing between (1) the availablity of reduction relations (possibly local) between two scientific theories, and (2) the actual use of these reduction relations by researchers in scientific practice. In some part of the literature, by leaving this distinction implicit, the suggestion may be created that the availability of a reduction relation, in the pragmatic sense, is used to actually reduce and eliminate the higher-level theory. Indeed, the main arguments put forward aiming at protection of the automous status of the higher-level theory criticizes the existence of systematic reduction relations. Such an argument strongly suggests the silent assumption that if such systematic reduction relations exist, in a pragmatic the higher-level theory is to be given up. In contrast, the position put forward here is that, to protect the higher-level theory the availability of a reduction relation is not bad at all; it can go hand in hand with an antireductionist pragmatic strategy reinforcing the higher-level theory

by giving more status to and putting more emphasis on the higher-level theory instead of less. The case of the development of process description languages within Computer Science supports this claim.

The process description language case studies from Computer Science reveal that much progress can be made by, (1) starting from an already given base theory, creating a not yet existing higher-level theory and (context-specific) reduction relations between this new theory and the base theory, and (2) concentrate the emphasis in further scientific development on this new higher-level theory instead of the existing base theory. This is a pragmatic strategy opposite to that of reducing and eliminating a higher-level theory. This strategy may have value in other domains as well.

In Section 2.1 we argued that the dynamics of higher cognitive processes such as complex task performance and reasoning demand higher-level description languages. In another discipline, biology a similar situation occurs. The attempt to understand the behaviour of a cell such as *E. coli*, and the dynamics of its intracellular processes in terms of the underlying biochemistry leads to hundreds of differential equations with parameters for which reliable estimations are rarely known. Given that two coupled differential equations already can show complex behaviour, even if all parameters were known, this type of description may be intractable and add no understanding. Even taking into account that in this case the biochemistry is by and large known, this situation may be considered similar to explaining from the physical stance why a man dives when a brick is thrown, and Dennett's analysis (see Section 4.2) may apply here accordingly.

Higher-level descriptions may be more adequate for scientific practice than the lower-level biochemistry descriptions. One approach recognizes that some conglomerates of biochemical processes act as functional units such as "metabolic pathway", "catabolism", "transcriptome" and "regulon". Some of these concepts have been or are being defined formally (Kahn & Westerhoff, 1991; Rohwer et al., 1996; Schilling et al., 2000). Viewed from a more high-level, functional perspective, the cell effectively makes decisions regarding its internal dynamics and externally observable behaviour, given its environmental circumstances, and implements these decisions into appropriate actions. This behaviour, viewed from this high-level perspective is less complex than the hundreds of differential equations. This suggests that considering a cell from the perspective of an agent sensing the environment, integrating

that information within its internal state, and then choosing behavioural patterns of action, may provide the basis of an alternative modelling approach; cf. Jonker and Treur (2002b). Some first steps from such a high-level perspective show promising results: cf. Jonker, Snoep, Treur, Westerhoff, and Wijngaards (2002).

6.2 Interlevel Relations and Multiple Realizability

Within Computer Science a pluralist landscape of (often competing) platforms has been developed, i.e., different processors and operating systems such as the PC with Windows, Macintosh with Mac OS, and Sun with UNIX or Solaris. The higher-level languages play a unifying role over these various platforms; their descriptions are multi-realisable within the different platforms. To deal with multiple realisability three options have been evaluated.

The first option, making use of heterogeneous disjunctions to define (global) bridge principles is not appropriate for the case of process description languages within Computer Science, since such a disjunction is not expressible in a lower-level language in an adequate manner.

The second option, supervenience, does apply, but is rather abstract; it does not given detailed information on how the interlevel relations constrain the descriptions at the two levels.

As a refinement of supervenience, the third option is to consider context-specific local reduction relations. This option seems appropriate to describe the case of process description languages within Computer Science. For example, each platform (i.e., different processors and operating systems such as PC with Windows, Macintosh with Mac OS, and Sun with UNIX or Solaris) and implementation environment defines a context in which a practitioner can work, just as a biologist can choose a model species to work with. In this way interlevel relations can play an important role, even given the multi-realisability that occurs also in Computer Science.

6.3 The Empirical Status of Reduction Relations

If the development of Computer Science is viewed as a singular process that takes place in isolation from the rest of the world, it may seem that the (interlevel) reduction relations between descriptions at different levels are purely definitional: they define the higher-level concepts in terms of (a multitude of) lower-level concepts. However, this is not the case for the following reasons. There is and there has been an intimate relationship between the

development of process description languages within Computer Science and applications in society. Today Computer Science as a science probably has by far the largest number of professionals working in society, and computer scientists have much interaction with these professionals. The higher-level languages often have been developed in order to obtain adequate means to describe processes that take place in the real world, for example work flows in factories. A descripion of processes within a specific application context has its own independent validation against this practice, independent of whatever lower-level concepts can be used to realise an automated form of such processes. This semantic relationship between a description and a process to which this description refers, can be formalised mathematically by defining the *formal semantics* of such a description, independent of any relation to lower-level languages.

This reference to actual processes provides empirical content to a higher-level description. In a similar manner (or, for the lowest level, by relating it to a computer's processor), lower-level descriptions can be assigned empirical content. A reduction relation between a higher-level and a lower-level description then can be considered an empirical relationship that can be empirically tested.

Acknowledgements

This paper has benefit much from discussions in the context of the multi-disciplinary project 'Reductionism' funded by and at the Vrije Universiteit Amsterdam. In particular, the authors are grateful for contributions to these discussions from the project members from the three participating faculties (Biology, Sciences and Psychology): Fred Boogerd, Frank Bruggeman, Huib Looren de Jong, Allard Tamminga, and Hans Westerhoff. Additionally, the authors are grateful to the editor William Bechtel for his detailed and insightful comments that have further improved the text.

References

Beer, R.D. (1995). Computational and dynamical languages for autonomous agents. In: (Port and van Gelder, 1995), pp. 121-148

Bickle, J. (1998). *Psychoneural Reduction: The New Wave*. MIT Press, Cambridge, Massachusetts.

Booch, G. (1991). *Object oriented design with applications*. Benjamins Cummins Publishing Company, Redwood City.

Brazier, F.M.T., Jonker, C.M., and Treur, J. (2002). Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, vol. 41, 2002, pp. 1-28.

Brazier, F.M.T., Jonker, C.M., Treur, J., and Wijngaards, N.J.E., (2000). On the Use of Shared Task Models in Knowledge Acquisition, Strategic User Interaction and Clarification Agents. *International Journal of Human-Computer Studies*, vol. 52, 2000, pp. 77-110.

Brazier, F.M.T., Treur, J., Wijngaards, N.J.E., and Willems, M., (1999). Temporal semantics of compositional task models and problem solving methods. *Data and Knowledge Engineering*, vol. 29(1), 1999, pp. 17-42.

Brown, A.W. (ed.) (1996). *Component-Based Software Engineering*. IEEE Computer Society Press, 1996.

Busemeyer, J., and Townsend, J.T. (1993). Decision field theory: a dynamic-cognitive approach to decision making in an uncertain environment. Psychological Review, vol. 100, pp. 432-459.

Cornelissen, F., Jonker, C.M., and Treur, J., (2002). Compositional Verification of Knowledge-Based Task Models and Problem Solving Methods. *Knowledge and Information Systems Journal*. In press

Cummins, R. (1975). Functional Analysis. *The Journal of Philosophy*, vol. 72, pp. 741-760

Dennett, D.C. (1978). *Brainstorms: Philosophical Essays on Mind and Psychology*. Hassocks: Harvester Press, 1978.

Dennett, D.C. (1987). *The Intentional Stance*. MIT Press. Cambridge Mass, 1987.

Dennett, D.C. (1991). Real Patterns. *The Journal of Philosophy*, vol. 88, 1991, pp. 27-51.

Gasper,P. Reduction and instrumentalism in genetics, Philosophy of Science,1992, pp. 655-670.

Hawkins, R.D., and Kandel, E.R. (1984a). Is There a Cell-Biological Alphabet for Simple Forms of Learning? *Psychological Review*, vol. 91, pp. 375-391

Hawkins, R.D., and Kandel, E.R. (1984b). Steps Toward a Cell-Biological Alphabet for Elementary Forms of Learning. In: G. Lynch, J.L. McGaugh, and N.M. Weinberger (eds.), *Neurobiology of Learning and Memory*, Guilford Press, New York, pp. 385-404, vol. 91, pp. 375-391

Jackson, F., and Pettit, P. (1988). Functionalism and Broad Content. *Mind*, vol. 97, pp. 381-400.

Jackson, F., and Pettit, P. (1990). Program Explanation: A General Perspective. *Analysis*, vol. 50, pp. 107-117.

Jonker, C.M., Snoep, J.L., Treur, J., Westerhoff, H.V., and Wijngaards, W.C.A. (2002). Putting Intentions into Cell Biochemistry: An Artificial Intelligence Perspective. *Journal of Theoretical Biology*, vol. 214, pp. 105-134.

Jonker, C.M., and Treur, J., (2002a). Analysis of the Dynamics of Reasoning Using Multiple Representations. In: W.D. Gray and C.D. Schunn (eds.), *Proceedings of the 24th Annual Conference of the Cognitive Science Society, CogSci 2002*. Mahwah, NJ: Lawrence Erlbaum Associates, Inc., 2002, pp. 512-517.

Jonker, C.M. and Treur, J. (2002b). Agent-Oriented Modeling of the Dynamics of Complex Biological Processes I: Single Agent Models. *BioComplexity Journal*, vol. 1. In press, 2002.

Jonker, C.M., Treur, J., and Vries, W. de, (2002). Temporal Analysis of the Dynamics of Beliefs, Desires, and Intentions. *Cognitive Science Quarterly* (Special Issue on Desires, Goals, Intentions, and Values: Computational Architectures), vol. 2, 2002, pp. 471-494.

Jonker, C.M., Treur, J., and Wijngaards, W.C.A., (2001). Temporal Modelling of Intentional Dynamics. In: B. Dunin-Keplicz and E. Nawarecki (eds.), *From Theory to Practice in Multi-Agent Systems, Proc. of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems, CEEMAS'01*, 2001. Lecture Notes in AI, vol. 2296, Springer Verlag, 2002, pp. 141-150. Extended abstract in: *Proceedings of the Third International Conference on Cognitive Science, ICCS 2001*. USTC Press, Beijing, 2001, pp. 344-349. Extended version to appear in *Cognitive Systems Research Journal*.

Kahn D., Westerhoff H.V., (1991). Control theory of regulatory cascades. *Journal of Theoretical Biology*, vol. 153, pp. 255-85.

Kelso, J.A.S. (1995). *Dynamic Patterns: the Self-Organisation of Brain and Behaviour*. MIT Press, Cambridge, Mass.

Kernighan, B.W. & Ritchie, D.M. (1978, 1988), *The C Programming Language*, Prentice Hall, New Jersey.

Kim, J., (1996). *Philosophy of Mind*, Westview Press.

Kim, J. (1998). *Mind in a Physical world: an Essay on the Mind-Body Problem and Mental Causation*. MIT Press, Cambridge, Mass.

Knuth, D.E. (1981). *The Art of Computer Programming*, Addison-Wesley.

Kontonya, G., and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, New York.

Lewis, D. (1988). Causal Explanation. In: *Philosophical Papers*, vol. 2. Oxford: Basil Blackwell

Nagel, E. (1961). *The Structure of Science*. Harcourt, Brace & World, New York.

Port, R.F., Gelder, T. van (eds.) (1995). *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge, Mass.

Rao, A.S. and Georgeff, M.P. (1991). Modelling Rational Agents within a BDI-Architecture. In: J. Allen, R. Fikes and E. Sandewall, (eds.), *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, (KR'91), Morgan Kaufmann, 1991, pp. 473-484.

Rohwer JM, Schuster S, Westerhoff HV. (1996). How to recognize monofunctional units in a metabolic system. *Journal of Theoretical Biology*, vol. 179, pp. 213-28.

Schilling, C.H., Letscher, D., and Palsson, B. Ø. (2000). Theory for the Systemic definition of Metabolic Pathways and their use in Interpreting Metabolic Function from a Pathway-Oriented Perspective. *Journal of Theoretical Biology*, vol. 203, pp. 229-248.

Tanenbaum, A.S. (1976). *Structured Computer Organisation*. Prentice-Hall, London.