Behaviours as Design Components of Cyber-Physical Systems

Michael Jackson[✉]

The Open University,
Milton Keynes MK7 6AA, UK
jacksonma@acm.org

Abstract. System behaviour is proposed as the core object of software development. The system comprises both the software machine and the problem world. The behaviour of the problem world is ensured by the combination of its given properties and the interacting behaviour of the machine. The fundamental requirements do not mandate specific system behaviour but demand that the behaviour exhibit certain desirable properties and achieve certain effects. These fundamental requirements therefore include usability, safety, reliability and others commonly regarded as 'non-functional'. A view of behaviour content and structure is presented, based on the Problem Frames approach, leading to a specification in terms of concurrent behaviour instances created and controlled within a tree structure. Development method is not addressed in this short paper; nor is software architecture. For brevity, and clearer visibility of the thread of the paper's theme, much incidental, explanatory, illustrative and detailed material is relegated to end notes. A final section summarises the claimed value of the approach in addressing the characteristic challenges of cyber-physical systems.

1 Introductory Remarks

In a cyber-physical system the software controls a part of the physical and human world. The system comprises both the computing equipment and the parts of the physical world it governs. Examples are vending machines, radio-therapy machines, cars, aircraft, trains, lifts, cranes, heart pacemakers, automatic lathes, ATMs, and countless others.

This paper aims to explain and justify an improved version of the *Problem Frame* approach [1] to the development of such systems. The approach deals with the pre-formal[1] work that creates a bridge from the stakeholders' purposes and desires[2], leading to a detailed software specification, emphasising the centrality of the system behaviour. The main text of the paper is itself no more than an outline: some illustrations, clarifications and additional details are presented in notes, along with appeals to some eminent authorities.

The process of creating the bridge must start from the stakeholders' purposes and desires—that is, from the system requirements. But the satisfaction of those requirements by the running system in operation starts from the other end. The *machine*—the system's computing equipment executing the system's software—interacts with the physical *problem world* to monitor and control what happens there: it is the resulting *system behaviour*[3] that must satisfy the requirements. The system behaviour is the essential product of software development.

The requirements[4] are desired properties of the system behaviour; but they are distinct from it. The development problem, therefore, is to design a behaviour that satisfies the requirements, with an accompanying software specification that can be

---

[1] The work described is *pre-formal* because its desired product is a documented understanding of the system, sufficiently sound and well-structured to justify and guide the subsequent deployment of formal techniques. As von Neumann and Morgenstern wrote [3]:

"There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied. Consequently the initial task is to clarify the knowledge of the matter by further careful descriptive work."

In addition to careful description, software development demands exploration, invention and design. These activities must be open to unexpected discoveries, and should therefore not be constrained by *a priori* commitment to the tightly restricted semantics of a formal language. This does not mean that pre-formal work is condemned to gratuitous vagueness. It means only that for describing each particular topic and aspect that will be encountered the appropriate semantics and appropriate scope and level of abstraction cannot be exactly determined in advance. The freedom to make these choices in an incremental, opportunistic and emergent fashion should not be hampered by premature choice of a formal language.

[2] The stakeholders of a system are those people and organisations who have a legitimate claim to influence the design of the system behaviour. Some stakeholders—for example, the driver of a car or the wearer of a cardiac pacemaker—are themselves participants in the system behaviour. Others—for example, the representative of a regulatory body or of the company paying for the system—are not. Stakeholder purposes and desires may be formally or informally satisfiable, and may be observable in the problem world or outside it.

[3] The word *system* is often used to denote only the machine executing the software. Here, instead, we always use it to denote the machine and the physical problem world together. For a cyber-physical system the execution of the software is merely a means to obtain a desired behaviour in the physical world outside the machine, and has no significance except in that role. We use the word *behaviour* to denote either an assemblage of processes with multiple participants or an instance of the execution of the assemblage; which is meant should be clear from the context in each case.

[4] There are many kinds and forms of requirements. Some are constraints on budgets and delivery dates, on the composition and organisation of the development team, and other such matters of economic or social importance. Here we are concerned only with those requirements whose satisfaction is to be judged solely by the behaviours and effects of the system in operation.

shown to ensure that behaviour. Evaluating whether a proposed behaviour design is satisfactory must be a co-operative task for developers and stakeholders together[5]; producing and validating the accompanying software specification is primarily the developers' responsibility.

2 The World and the Machine

The problem world is, in general, a heterogeneous[6] assemblage of identifiable distinct parts of the material and human world that are of interest to the stakeholders. These parts are called *problem domains*: a vital task in development is investigating and analysing their *given properties* and behaviours on which the design will rely[7]. These properties are given, in the sense that they are independent of the machine[8]; but in combination with the domains' acceptance of the constraints imposed by the machine[9] they will determine the system behaviour[10].

---

[5] A stakeholder criterion of requirement satisfaction may lie far outside the problem world: for example, the system may be required to attract a large number of new, as yet unidentified, customers in new markets. A requirement may be insufficiently exact to allow rigorous validation: for example, that the behaviour of a car should never surprise its driver. Satisfaction of such requirements must be carefully considered by the stakeholders and developers during the design work; but cannot be formally demonstrated and can be convincingly evaluated only by experience with the installed system.

[6] The problem world of an avionics system, for example, includes the airframe, its control surfaces and undercarriage, the engines, the earth's atmosphere, the airport runways, the aviation fuel, the pilots and other crew, the passengers, the gates for embarkation and disembarkation, other aircraft, the air traffic control system, and so on.

[7] We regard the problem domains as given in the sense that the task of software engineering, *per se*, is not to develop or redesign physical artifacts, but to create software that will monitor and control their behaviour. In practice, of course, some projects may demand a degree of co-design of physical and software artifacts, and software engineers will have a central contribution to make to that work.

[8] The given properties and behaviours of a physical problem domain are constrained by the laws of physics, by its designed or otherwise constituted form, and also by its external environment. A domain is potentially capable of exhibiting varying behaviours according to the contexts in which it may be placed.

[9] Constraints on a domain's potential behaviour are applied by its context. In a cyber-physical system the immediate context comprises its physical neighbours—the machine and other domains with which it interacts. A domain that does not interact directly with the machine may be constrained by causal chains involving other domains.

[10] The system behaviour is not to be conceived or expressed as a set of stimulus-response pairs or in any other similarly fragmented form. It extends over time, and is to be understood as a whole. As Poincaré asked [4]:

"Would a naturalist imagine that he had an adequate knowledge of the elephant if he had never studied the animal except through a microscope?"
"It is the same in mathematics. When the logician has resolved each demonstration into a host of elementary operations, all of them correct, he will not yet be in possession of the whole reality; that indefinable something that constitutes the unity of the demonstration will still escape him completely."

The disadvantages of a fragmented view of behaviour are made explicit in another paper elsewhere [5].

The problem world must not exclude human domains. People participate in systems in many different roles—for example: as a casual user of a vending machine, as a plant operator, as the driver of a car or train or the pilot of an aircraft, as a passenger, as the patient in a surgical operation or a radiotherapy system, as the recipient of medication dispensed by an infusion pump, as the wearer of a cardiac pacemaker, as a source of problem world data that is otherwise inaccessible[11] to the machine. In various ways and to various extents the physical and behavioural properties of a human participant contribute to the system behaviour[12]. The development must investigate and understand them in all their relevant aspects.

We speak of the machine in the singular: a development problem has multiple problem domains, but only one machine. This is, of course, a conceptual simplification: the computing equipment of a realistic system may be distributed, and even shared with another system. The simplification is appropriate in an initial view of the problem: while the physical structure of the problem world is largely given, and offers clear enough distinctions among problem domains, the structure of the machine[13], which we must develop, has not yet been designed[14].

3   Challenges of Cyber-Physical Systems

Each system presents its own particular challenges to the developers; but some important challenges are common to all cyber-physical systems that are to any degree critical. Among the most salient are two intertwined challenges: formal description of the ineluctably physical problem world, and dependable design of the complex system behaviour. If the system behaviour resulting from its interactions with the designed software is to be the subject of fully convincing reasoning, the problem world must be

---

[11] For example, to describe the precise layout of a road junction for a traffic control system, and the positions within it of the lights, vehicle sensors and pedestrian crossing request buttons.

[12] For example, the physiology of a recipient of a cardiac pacemaker is crucial to the system design. So too is the physical size of a machine press operator whose safety depends on the limited arm span which prevents the operator from pressing the start button with one hand while the other hand is in the danger area.

[13] The machine specification produced by the development approach presented here is simultaneously physical—being explicitly described in terms of its interfaces to the physical world—and abstract—because it need not necessarily correspond to a software or hardware module of the eventual implementation.

[14] The problem world naturally presents itself to us as populated by distinct entities or domains, whereas the machine does not. The design process, briefly presented in later sections, allows decomposition of what was initially postulated to be one machine into two or more smaller machines.

formally described. But the problem world is not a formal universe[15]: any formalisation is an approximation, at best barely adequate to its particular purpose, context and use. The system behaviour of any realistic system is inevitably complex, and the non-formal nature of the problem world adds greatly to this complexity.

The response to the physicality challenge must itself be twofold. First: it must embody the practice of a sound modelling discipline[16] to minimise uncertainty in the mapping between each description and its physical subject. This is a requirement for any development method, and we will not discuss it further here.

Second: a dependable structuring of the system behaviour must address both the inherent complexity of the system's function and purpose, and the added complexity due to the non-formal physical problem world. Inherent complexity arises from the multiplicity of system functions, features, modes and phases of any realistic system. Further, design of a critical system is likely to demand a wide operating envelope, encompassing a very wide variety of conditions in which the system must behave dependably. This inherent complexity cannot be mastered by *ad hoc* variations within one monolithic behaviour. Instead, multiple behaviours must be specifically designed for specific conditions and functions and activated as conditions and needs demand.

The complexity added by the physicality of the problem world springs from reality's deviations from any one formal model, manifested as equipment failures, unexpected behaviour due to neglected physical effects, operator errors, and other exceptional contingencies. These deviations must be accommodated while maintaining safe and dependable—though possibly degraded—operation. This complexity may require a variety of formalisations of the same physical domains[17], which in turn requires a variety of behaviours that depend on those domains.

---

[15] In an unjustly neglected response [16] to Fred Brooks's acclaimed talk *No Silver Bullet*, Wlad Turski wrote:

"There are two fundamental difficulties involved in dealing with non-formal domains also known as 'the real world':

(1) Properties they enjoy are not necessarily expressible in any single linguistic system.
(2) The notion of mathematical (logical) proof does not apply to them."

This is the salient challenge that physicality presents to dependable system design. It is absent from abstract mathematical problem worlds, such as the world of integers and the problems of finding and dealing with large primes.

[16] Such a discipline would contribute to solving the problem characterised in an illuminating paper [17] by Brian Cantwell Smith as the relationship between the model and the world: "In the end, any adequate theory of action, and, consequently, any adequate theory of correctness, will have to take the model-world relationship into account". A discipline of description should constitute a major topic of research in its own right, but the need has been largely ignored by the software engineering community. Some aspects are touched on informally in a 1992 paper [18] and a 1995 book [19]. Further work is in progress but is not discussed in the present paper.

[17] For example, tolerating faults in physical equipment may demand at least two formalisations. In one, the equipment is assumed faultless, and the associated behaviour relies on that faultless functionality. In the other, the potentiality for fault is acknowledged, and the associated behaviour relies only on residual domain properties that allow faults to be detected, diagnosed, and mitigated. The two behaviours may be concurrently active, and the two—even potentially conflicting—formalisations are relied on simultaneously.

4   Dependability and the Problem World Boundary

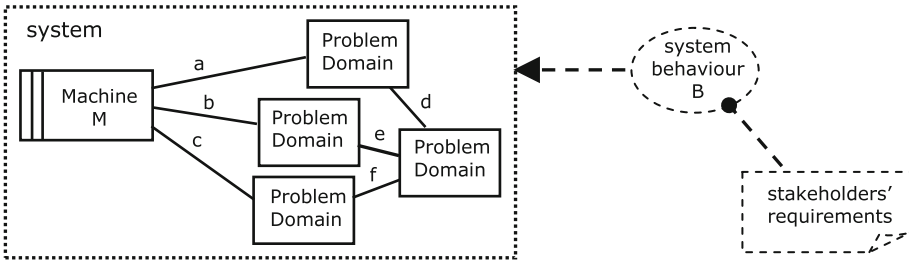The general form of a development problem is shown in Fig. 1.

Fig. 1. Problem diagram

The single striped box represents the machine; the plain boxes represent problem domains; the solid lines (labelled *a,b,.. f*) represent interfaces of physical *shared phenomena* such as events and states[18]. Together, the machine, the problem domains, and their mutual interfaces constitute the *system*; the *system boundary* is represented by the dotted box. The ellipse represents the system behaviour[19] resulting from the interaction of the machine with the problem world. The document symbol represents the *requirements*.

Let us suppose that the developers have calculated or otherwise determined that the desired system behaviour is to be B; and that their designed behaviour of the machine at its problem world interfaces *a,b,c* is M[20]. Then, if the given behaviours of the problem domains are {W*i*}, it is necessary for system dependability to demonstrate convincingly the *fundamental entailment*, that

---

[18] Phenomena are shared in the CSP [6] sense that more than one domain participates in the same event, or can observe the same element of a domain state. A shared event or shared mutable state is controlled by exactly one participating domain and observed by the other participants.

[19] In the problem diagram a symbol with a dashed outline represents a symbolic, possibly informal, description. The behaviour ellipse represents a behavioural description of the system. The requirements symbol represents a description of stakeholder desires and purposes. The level of abstraction at which the subject matter is described will, of course, vary according to the context and purpose of the description.

[20] The relationship between machine, problem world properties and system behaviour is complex. It should not be assumed that the machine design can be derived formally, or even systematically, from the other two. In particular, there may be more than one machine that can achieve a chosen behaviour in a given problem world.

$$M, \{Wi\} \mid = B.$$

That is: the designed machine M installed as shown in the problem world whose given domain properties are {Wi}, will ensure the behaviour B[21]. For a critical system this demonstration should be formal[22], to provide the strongest possible assurance of the system behaviour. The domain properties {Wi}—and the machine specification M—must therefore be captured in an adequate formalisation that is sufficiently faithful[23] to the physical realities it approximates.

---

[21] As Harel and Pnueli rightly observe [7]:

"While the design of the system and then its construction are no doubt of paramount importance (they are in fact the only things that ultimately count) they cannot be carried out without a clear understanding of the system's intended behavior. This assertion is not one which can be easily contested, and anyone who has ever had anything to do with a complex system has felt its seriousness. A natural, comprehensive, and understandable description of the behavioral aspects of a system is a must in all stages of the system's development cycle, and, for that matter, after it is completed too."

[22] The intrusion of non-formal concepts and concerns vitiates a formal demonstration. The system boundary is therefore related in its aim, though not in its realisation, to Dijkstra's notion of program specification as a firewall. He wrote [8]:

"The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical 'firewall' between two different concerns. The one is the 'pleasantness problem,' i.e. the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' i.e. the question of how to design an engine meeting the specification…. the two problems are most effectively tackled by… psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

Dijkstra's aim was to achieve complete formality in program specification and construction. Our aim here is to preserve a sufficient degree of formality within the system boundary to achieve dependability of system behaviour. The firewall ensures—*pace* Dijkstra's dismissive characterisation of the 'pleasantness problem'—only that what is inside is sufficiently formal: not that everything outside is informal. Some requirements are formal: for example, the requirement in an electronic purse system that money is conserved in every transaction even if the transaction fails.

[23] All formalisation of the physical world, at the granularity relevant to most software engineering (though not, perhaps, to the engineering of experiments in particle physics) is conscious abstraction. Because the physical world, at this granularity, is not a formal system, a formal model can be only an approximation to the reality. In a formal world, after the instruction sequence

$$"x := P; y := x; x := y"$$

the condition $"x = P"$ will certainly hold. But in a robotic system, after the sequence

$$"x := P; Arm.moveTo(x); x := Arm.currentPosition"$$

the condition $"x = P"$ may not hold. Moving the arm and sensing its position both involve state phenomena of the physical world. Movement of the arm may fail, and will certainly be imprecise; and the resulting position will be imprecisely sensed and further approximated in the machine by a floating-point number.

Unreliability and approximation limit the dependability of any cyber-physical system [9] and the confidence that can be legitimately placed in formal demonstration. A crucial concern in the design of a critical system is achieving acceptable dependability within these limits.

Furthermore, the problem world and machine must be regarded as a closed system: the demonstration of the fundamental entailment must not rely on any assertion that cannot be inferred from the physical laws of nature and the explicitly stated properties[24] of the problem domains[25]. These constraints on the problem world necessitate the distinction[26] shown in Fig. 1 between the system behaviour and the stakeholders' requirements—which are often informal, and often related to parts of the world outside the system boundary.

5   Complexity of System Behaviour

Satisfying the system requirements may demand great behavioural complexity. Some of this complexity springs from normal system operation, which may comprise many necessary functions and features structured in multiple phases and modes[27]. Interaction among these functions (especially when they operate concurrently, but also when they are consecutive, or nested, or alternating) is a source of further complexity[28]. Some complexity is due to the need to detect and mitigate equipment faults, operator errors or other exceptional events. Some springs from the need to coordinate normal operation

---

[24] The given properties of each problem domain must be investigated and explicitly described: together, they provide the $\{Wi\}$ in the entailment $M,\{Wi\} \models B$. It is a mistake to elide these descriptions into a single description encompassing both the machine and the problem domains. A separate description of a domain's given properties clearly distinguishes what the machine relies on from what it must achieve, and allows those potential properties and behaviours to be made explicit that the machine, by its behaviour, suppresses, avoids or neglects.

[25] The system can be closed in the necessary sense by *internalising* external impacts on the problem domains. Suppose, for example, that domains A and B are both vulnerable to failure of a common electrical power supply P. If P is not included as a problem domain, electrical power failure in A must be formalised as a spontaneous and unpredictable internal event of A, and similarly for B. It is then impermissible to assert that power failures of A and B are coordinated, since there is no problem domain to which this co-ordination can be ascribed. Similarly, in an automotive system the driver must be included as a problem domain if the driver's physical capabilities and expected behaviours are relied on to prove the entailment $M,\{Wi\} \models B$.

[26] Unfortunately, in many development projects this distinction is elided, and requirements are stated as explicit direct descriptions—albeit often fragmented descriptions—of system behaviour. This is a mistake, exactly parallel to the classic mistake of specifying a program by giving a procedural description of its behaviour in execution.

[27] For example, an avionics system must support the normal sequence of flight phases: gate departure, taxiing, take-off, climbing, cruising, and so on. A radiotherapy system must support the normal prescription and treatment protocols: prescription specification and checking, patient positioning, position adjustment, beam focusing, dose delivery, beam shutoff, and so on.

[28] In telephone systems of the late 20th century such features as call forwarding, call blocking and voicemail proliferated. The complexity resulting from their interactions caused ever-increasing difficulty in the development of those systems, and often produced inconvenient and disagreeable surprises for users. This *feature interaction* problem [10, 11] became widely known: it was soon recognised as a serious problem in most realistic systems.

with equipment maintenance and repair. Some is due to the need to maintain safe operation even in extreme adverse conditions.

In general, dependable system behaviour means behaviour that varies in a comprehensibly dependable[29] way, according to what is possible and desirable in different circumstances. Fault tolerance, for example, does not demand that normal functional behaviour continue in the presence of a major fault, but permits it to be replaced by a different behaviour, functionally degraded but dependable and preserving safety[30]. Complex system behaviour is understood as a combination of simpler *constituent behaviours*. A constituent behaviour has the same structure and pattern as the complete system behaviour pictured in Fig. 1: it has a machine, an assemblage of problem domains, and a number of relevant requirements. The development problem, both for the complete behaviour and for each constituent behaviour, has two interrelated aspects: to design a behaviour to satisfy the requirements; and to specify an associated machine that will ensure that behaviour[31].

An ill-chosen constituent can complicate the task of developing the whole to which it belongs—either by itself presenting excessive difficulty in its own development, or by leaving a misshapen residual complement[32]. Identifying suitable constituent behaviours is the primary task in structuring the system behaviour. An essential guide in this

---

[29] 'Comprehensibly dependable' does not imply 'predictable'. A realistic system has problem domains—notably its human participants—that exhibit non-deterministic behaviour. In general, therefore, prediction of system behaviour is always contingent. What matters is that neither the developers nor the human participants should be surprised by unexpected occurrences of anomalous behaviour.

[30] For example, if the main power supply fails in a passenger lift system the car is to be moved, under auxiliary power, to the nearest floor for the passengers to disembark. If the hoist cable breaks a more radical solution is necessary: the lift car is locked in the shaft to prevent free fall, and the passengers must then wait to be rescued by an engineering crew.

[31] The development problem for a constituent behaviour is spoken of as a *subproblem*. Initially the constituent behaviour is considered in isolation from other behaviours, ignoring both its interactions at common problem domains and its interaction with its controlling behaviour. (Behaviour control is discussed in Sect. 8.)

[32] The second of Descartes's famous rules of thought [12] was:

"Divide each problem that you examine into as many parts as you can and as you need to solve them more easily."

Leibniz rightly observed in response [13]:

"This rule of Descartes is of little use as long as the art of dividing remains unexplained… By dividing his problem into unsuitable parts, the inexperienced problem-solver may increase his difficulty."

Any discipline that aims to master complexity by decomposition must identify and apply criteria of component simplicity.

task is a set of explicit criteria of behavioural simplicity. These criteria include: regularity of the machine's software structure[33]; constancy of given domain properties during the behaviour[34]; a clear and tersely explicable purpose for the behaviour[35]; and simplicity of the causal pattern by which the machine ensures the behaviour in the problem world[36].

6 Large Behaviour Structure: Principles

This structuring of complex system behaviour, in terms of constituent behaviours, aims above all at producing an understandable and verifiable[37] specification of system behaviour. It is not the immediate aim of this structuring to produce a modular software structure capable of translation into efficient executable code. If a specification is executable, that is an advantage. But the primary aim is intelligibility of the system behaviour, demanding a correspondence between constituent behaviours and functional purposes that can be understood and evaluated by the stakeholders[38].

A constituent behaviour should initially be analysed in a simplified form in which its possible interactions with other behaviours are ignored: consideration of those

---

[33] A machine's software structure is regular if there is no *structure clash* [14]. That is: the dynamic structure of the software clearly composes the dynamic structures at its interfaces to problem domains.

[34] Reasoning about the relationship between the machine and the system behaviour is greatly complicated if the given domain properties are not constant. For example, they may vary with environmental conditions or with varying loads imposed by varying requirements on the system behaviour.

[35] Both top-down and bottom-up design of the system behaviour are used as necessary. If—as is the case for any realistic system—no tersely explicable purpose of the whole system behaviour can be identified, bottom-up design must be used: the purpose of the whole will then emerge from the designed combination of the constituents.

[36] The causal pattern by which the machine ensures the problem world behaviour is what Polanyi [15] calls the *operational principle* of a *contrivance*—and a system is a contrivance in his sense. Simplicity of this causal pattern is one important characteristic of a simple behaviour.

[37] Formal verification of a specification proves the entailment M, $\{Wi\} \models B$. Some additional formal and informal verification is needed to demonstrate the quasi-entailment $\{Wi\}, B \mid\sim R$—that is, that the requirements are satisfied. Demonstrating that the formalisation of the given problem world is sufficiently faithful to the physical reality is an entirely distinct task: it is inherently non-formal, and is typically both the hardest and the most vital.

[38] For example, an automotive feature such as Cruise Control or Stop-Start must correspond to an identifiable part or projection of the system behaviour specification, not to a collection of stimulus-response pairs distributed among many parts of the whole specification.

interactions and the complexities they introduce should be postponed until a later point at which the behaviours are to be combined[39]. The same strategy—an incremental approach to inescapable complexity[40]—can be applied to other complexities due to exceptional conditions that can arise within the behaviour itself.

Initially considering constituent behaviours—more exactly, *candidate* constituent behaviours[41]—in isolation encourages an important separation of concerns. To the greatest extent possible, the functional content of each behaviour should be separated from the control of its initiation and, in certain circumstances, of its termination. Since the concept of a constituent behaviour rests on a relationship of inclusion of an instance of one behaviour in an instance of another, this separation cannot be complete. But its aim remains valid: to maintain modularity in the explicit structure of the behaviour specification[42]. This modularity supports the understanding of each constituent behaviour as an independent unit of system functionality: it must be allowed to persist as long and as fully as possible, throughout the development and use of the specification[43].

---

[39] It makes obvious sense to understand the components before addressing the task of their composition. Neglect of this principle is the Achilles heel of top-down decomposition and of its cousin stepwise refinement.

[40] The third of Descartes's famous rules of thought [12] was:

"… to conduct my thoughts in such order that, by commencing with objects the simplest and easiest to know, I might ascend by little and little, and, as it were, step by step, to the knowledge of the more complex; assigning in thought a certain order even to those objects which in their own nature do not stand in a relation of antecedence and sequence."

[41] Candidate constituent behaviours arise both in *top-down decomposition*, as briefly illustrated in Sect. 7, and in *bottom-up development*, in which candidate constituents are identified piecemeal. In both cases each candidate constituent must be analysed, and its simplicity evaluated, before it can be definitely accepted as a component in the system behaviour design.

[42] Traditional block-structured programming establishes frame conditions for modules based on scope rules. In a cyber-physical system such frame conditions are frustrated by the connectedness of the physical problem world: behaviours interact unavoidably at physical domains that are common—directly or indirectly—to their problem worlds.

[43] Eagerness to rush to design a software architecture is usually misplaced. One freedom that software—unlike hardware—allows to its developers is the freedom of malleability of their material. Many structural transformations are possible that can preserve chosen specification properties of the source while endowing the target with a completely new property suited to efficient implementation for program code construction and execution. Knowing that such transformations are available, developers should resist the temptation to cast behaviour specifications in the form of an architecture of software modules. The machine associated with the behaviour in each subproblem should be regarded as a projection, not a component, of the complete software.

7 Large Behaviour Structure: Designed Domains

Large behaviour structure is defined in terms of a designed structure of the machines that ensure the constituent behaviours[44]. One—but not the only—design motivation for structuring is decomposition[45]: a behaviour which is proving less simple than originally expected, is restructured with two or more constituent behaviours.

Figure 2 shows, in general terms, a trivial case. In behaviour B0, machine MB0 monitors Domain X and controls Domain Y so that its behaviour is kept in some required relationship with Domain X. The Stop Switch allows behaviour B0 to be terminated on command. On analysis, we are supposing, the tasks of monitoring X and controlling Y prove too complex to be combined in a single undecomposed behaviour B0[46]: so B0 is decomposed into B0' and two constituent behaviours BX and BY.

In B0, the information obtained from Domain X and needed for proper control of Domain Y would be represented and maintained in a data structure in the local store of machine MB0. In B0' this data structure has been 'promoted' from a machine local variable to a problem domain, X-to-Y, common to the two constituent behaviours BX and BY. A problem domain originated in this way is represented by a box with a single stripe: it is a *designed domain* because it is a design artifact of the software development, not a given part of the physical problem world of the original undecomposed problem[47].

In general, the function of a designed domain is to communicate information across time and space between parts of the system behaviour which we want—or are compelled—to separate. The X-to-Y domain allows the constituent behaviours BX and BY to be separated for reasons of behaviour simplicity, while allowing each to conform to

---

[44] Associated with each machine, from its expression as a problem in the pattern of Fig. 1, are the documented descriptions: M of the machine; {Wi} of the problem domains' given properties and behaviours; and B of the system behaviour. The machine is also associated with the relevant requirements {Rj}. It is this assemblage of descriptions that define the behaviour: the machine is the designed means of realising each of its necessary instances.

[45] This is *top-down* structuring. It starts from a firm conception of the function of the whole behaviour to be developed, and, level by level, identifies constituent parts that for any reason should be regarded as separate components. In a realistic cyber-physical system the proliferation of functions and features demands extensive use of *bottom-up* structuring, in which initially there is no firm conception of the whole behaviour: it emerges only gradually from the piecemeal identification and combination of constituents. Bottom-up structuring is briefly discussed later, in Sect. 8.

[46] For example, because there is a structure clash [14]: the process structures of BX and BY are incompatible, and the simplicity criterion that stipulates regular process structure cannot be satisfied in a single undecomposed behaviour B0.

[47] It may seem paradoxical—or, at least, inconsistent—to promote a designed domain, which was merely a local data structure in the software of a machine, as a legitimate problem domain on all fours with the physical domains Domain X and Domain Y. But of course the unpromoted local variable was physically realised in the store of the machine MB0. Its promotion merely makes visible and explicit what was previously hidden and implicit. From the point of view of MBX and MBY it is a problem domain, external to those machines, to be respectively controlled and monitored.
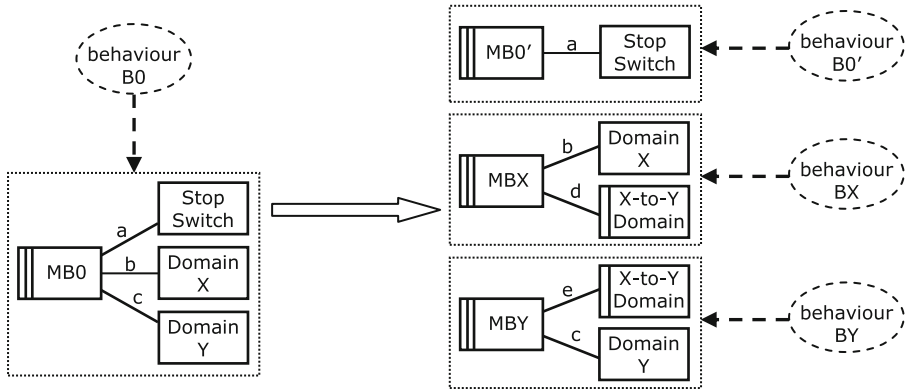
Fig. 2. Decomposition of a simple behaviour

the general pattern and structure of a development problem as shown in Fig. 1, and to benefit from the attendant advantages. The concept of a designed domain is very versatile: its ubiquitous utility reflects the ubiquitous utility of program variables. Some examples of designed domains are: a database in a bank accounts system; a data structure in a road traffic control system, representing the road layout and the positions of traffic light units, pedestrian buttons and vehicle sensors; a train operating timetable; the seat configuration data maintained for each driver by the software of a luxury car; the content and format of a document in an editing and presentation system; and countless others[48].

---

[48] A designed domain, once identified in a proposed or existing system, raises many important questions about its purpose, use and realisation. Between which behaviours does the domain provide communication? Of which behaviour's machine is the domain a local variable? Can the domain be instantiated more than once? How long does each instance persist? By which behaviours are the values of the domain state initialised and mutated? The reader may wish to ponder these questions for the examples mentioned in the text. Consider, for instance, the road layout domain in the traffic system. It is a designed domain for the traffic control behaviour. In which other behaviour is it a designed domain? Of which machine is it a local variable? Considering these questions can identify important large-scale concerns in system design. For example: a database associated with the operating parameters and constraints of a chemical process plant or a power station can be regarded as a designed domain. Safety demands that update access to this database must be explicitly controlled by the machine of which it is a promoted local variable. Apparent absence of such a machine from the behaviour specification indicates a severe safety exposure.

In some of these examples the designed domain is clearly an *analogical model* of the problem world—dynamic in the accounts system, and static in the traffic system[49]. In other examples it is less obviously a model of the physical problem world. In the editing system, for example, it would be very contrived to regard the document domain as a model of the editing events by which it was created. But in all cases the purpose of a designed domain, as of a program variable, is to communicate information between separate behaviours performed or evoked by the machine[50].

8 Large Behaviour Structure: Control Mechanism

The original machine of a decomposed behaviour controls the machines corresponding to its immediate constituent behaviours. This control relationship induces a rooted tree whose nodes are machines and designed domains, represented in a *behaviour control diagram* as shown in Fig. 3:
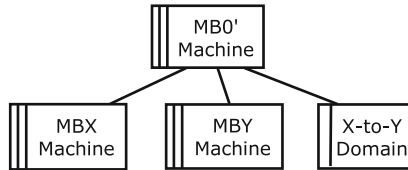
Fig. 3. Behaviour control diagram: controlling and controlled machines and a designed domain

---

[49] A *model* is an artifact providing information about its *subject*. We may distinguish *analogic* from *symbolic* models. A symbolic model—for example, a set of equations or a state transition diagram—is entirely abstract. The notational expression of a symbolic model itself carries no information about the subject: essentially, the model is simply a description that allows formal reasoning in the hope of revealing or proving some implied property or behaviour of its subject. An analogic model—for example, a system of water pipes demonstrating the flow of electricity in a circuit—is a physical object whose physical characteristics are analogues of those of the subject: water flow is analogous to electric current, pipe cross-section to the inverse of electrical resistance, a tank to a battery, and so on.

Often, a software model such as a database or an assemblage of objects is an analogic model of its subject. Each subject entity is analogous to a certain type of record or object; relationships between entities are analogous to pointers or record keys, and so on. The motivation for an analogic model is clear: the model is a surrogate, immediately available to the software, for historical or current aspects of the subject that are not readily accessible to direct inspection.

The danger of an analogic model is, of course, confusion of properties peculiar to the model with those belonging also—albeit by analogy—to the subject. Breaking a water pipe causes water to spill out; but breaking a wire in an electric circuit causes no analogous effect. A well-known example of such confusion in software engineering is the common uncertainty about the meaning of a *null* value in a cell of a relational database table.

[50] In the word processing system, for example, the document designed domain communicates information between the editing behaviour and other behaviours—storage, printing, transformation, and others—in which the document participates.

A machine node can be either a leaf or the root of the tree or of a subtree; a designed domain node can only be a leaf. A machine node represents possible instantiations of the machine: each instantiation starts the machine's execution and hence starts the corresponding problem world behaviour. The machine root node of the tree represents an instantiation of the whole system by an external agent; in any other case a machine is instantiated by its immediately controlling machine[51]. Machine execution instances may be specialised by arguments bound to instances of problem domains[52].

Temporal relationships among machine instances, including concurrency, are determined by the controlling machine[53]. For example, the execution pattern in Fig. 3 may be:

```
begin
instantiate X-to-Y;
instantiate MBX; instantiate MBY;
terminate MBX; terminate MBY
end.
```

in which MBX and MBY execute concurrently.

Machine execution may be terminated in several ways. First: a terminating behaviour may come to an autonomous *designed halt*—that is, a halt corresponding to a particular final outcome foreseen in the design of the machine[54]. Second, although the instantiating agent of a machine is responsible for ensuring that the relevant assumed conditions are present at instantiation, changing environment conditions and other vicissitudes may require *pre-emptive abortion* of the instantiated behaviour by the instantiating agent[55]. Third: it may be necessary in the ordinary course of system execution to bring a non-terminating behaviour, or a terminating behaviour that has not yet reached a designed halt, to an *orderly stop*—for example to cease operation of a vending machine when next there is no customer interacting with the machine. This, too, must be commanded by the instantiating agent.

---

[51] The behaviour control diagram shows only the parent-child relationship. The dynamic rules and patterns of instantiations are not shown in the diagram but only in the specification or program text of the controlling machine. Although designed domains appear in a behaviour control diagram, their associations with individual behaviours by membership of their problem worlds are not represented.

[52] Where a problem domain is populated by multiple individual entities there will be behaviours whose instantiations must be specialised in this way. In a library system, for example, a *loan* behaviour must be specialised to the borrowed book and the borrowing member.

[53] Instances of distinct behaviours, and distinct instances of the same behaviour, suitably specialised, may be temporally related by concurrency or in any other way governed by the controlling behaviour.

[54] A designed *halt* may occur when the goal of the behaviour been attained or has become unattainable. The associated failure condition is within the envisaged results of execution, and must be clearly distinguished from a failure of the assumed environment conditions—which, by definition, is not addressed within the behaviour's own design.

[55] Pre-emptive abortion is typically needed only in emergency conditions. In a lift system, for example, the normal lift service behaviour must be pre-emptively aborted if the hoist cable breaks; in an automotive system, the cruise control behaviour must be aborted if a crash impact is detected. Abortion is, of course, not represented as a behaviour state in Fig. 4. Pre-emptive abortion destroys the behaviour instance, which therefore no longer has any state.

Where two constituent behaviours fulfil closely related functions, their controlling behaviour becomes responsible for orderly termination of the whole. For example, in a system to control paid admissions to a zoo, there may be two constituent behaviours: one to manage payments, the other to manage admissions. On terminating the system behaviour when the zoo closes it is necessary for the controlling behaviour to command an orderly stop of the payments, followed by an orderly stop of admissions when the payments received have been exhausted—or, perhaps when it is clear that there are no more visitors to be admitted.

The controlled states of a machine are shown in Fig. 4.

On instantiation the machine enters its initialising state. If initialisation completes successfully the machine enters its running state; alternatively, it may first reach a *designed halt*, or receive a *stop* command from its controller. A *stop* command causes the machine to halt at the next occurrence of a stable problem world state satisfying a defined condition: for example, orderly termination of a lift behaviour might require the lift car to be stationary at a floor with the doors open[56]. While stopping, the machine may reach a state [*term*] in which it is designed to terminate unconditionally, or it may first reach a stable [*non-term*] state at which termination is not unconditional.

The controller can observe—but not modify—the states of the controlled behaviour shown in Fig. 4, along with more specific state details explicitly exported by the controlled behaviour. For example, the *Halted* state may be accompanied by an indication *failed* or *OK*.
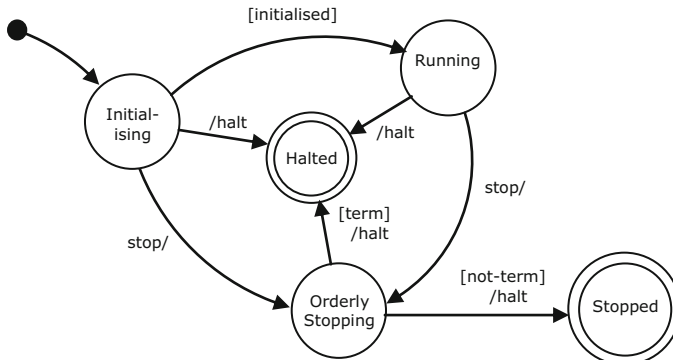

Fig. 4. Standard interface for behaviour control

---

[56] An orderly stop of lift service might take two forms. The fast form brings the lift car to the nearest floor to allow passengers to disembark because the normal power supply has failed and the lift is moving under emergency power; the slower form brings the car to the ground floor under normal power to allow lift service to be suspended without inconveniencing users.

9   Bottom-up Structuring of Large Behaviour

The starting point for the problem depicted in Fig. 2 was a broad, somewhat abstract, understanding of the functional goal of B0: maintaining a certain stated relationship between domains X and Y, constraining Y but not X. Understanding of this *abstract goal behaviour* B0 led to the problem diagram on the left in Fig. 2. On closer examination of the problem, simplicity criteria demanded the decomposition to behaviours B0', BX and BY. Understanding of the desired abstract goal behaviour B0, and the analysis which revealed its complexity, anchored the decomposition and provided a first view of the constituent behaviours BX and BY. B0 might even be imagined as the initial step in a refinement process of which the decomposition, and the design of the three behaviours and the designed domain X-to-Y, are the product of imagined subsequent steps. The end effect of these refinement steps is that the original causal chain $X \rightarrow MB0 \rightarrow Y$ has been refined to $X \rightarrow MBX \rightarrow X\text{-to-}Y \rightarrow MBY \rightarrow Y$[57].

For a complete system of realistic size and complexity no abstract goal behaviour can be identified that captures the overall behaviour as a whole[58]. Instead, the many modes, features, functions, phases, fault mitigations and exception handlers present an almost chaotic population of foreseeable constituent behaviours. Some candidates may be motivated by apparently discrete features or by particular requirements. Others may be hard to identify clearly or even to think of. The eventual interactions and temporal relationships among these candidate behaviours are largely unknown when development is begun. Study and design of these interactions and relationships cannot begin until the constituent behaviours have themselves been brought under some degree of intellectual control by identifying and analysing the subproblems that will define them.

When some candidate constituent behaviours have been identified and analysed in their simplest forms, it becomes possible to consider their relationships and interactions. The whole behavioural structure of the system is progressively[59] built up as a behaviour control tree of accepted candidates. Some constituent behaviours are identified from needs arising only in the process of constructing the tree. Some constituent

---

[57] This refinement process is imaginary because formal refinement cannot be a reliable technique in a non-formal world: the more concrete models may vitiate unacceptable or impractical simplifications in their more abstract predecessors. For example, in Fig. 2 the interposition of the designed domain may introduce sources of latency or error that were implicitly excluded in behaviour B0. When development has been completed it may be possible to retrofit the complexities of the concrete reality to an elaborated abstraction; but this exercise would belong to *ex post facto* rationalisation and formal verification, not to development method.

[58] In the absence of an identified and broadly understood abstract goal behaviour that comprehensibly includes all its constituent behaviours, the overall behaviour must emerge eventually from work on the constituent behaviours at lower levels. No starting point for a refinement process can be identified, because nothing definitive can be said of the overall behaviour while it has not yet emerged.

[59] The bottom-up construction of the behaviour tree is progressive only in the sense that constituent behaviours are gradually pieced together as their individual designs and interactions become progressively clearer. In general, the intermediate products of the construction process will constitute a forest rather than a tree. It is too optimistic to conceive of this forest as an ordered structure, similar to a layered hierarchy to be built up in successive layers from the bottom upwards.

behaviours and designed domains are introduced by local applications of top-down design. Very often these designed domains will be analogic models of parts of the system. Behaviour control may be exerted to terminate one behaviour in order to allow another, incompatible, behaviour to be initiated.

10   From Behaviour Specification to Software Execution

The distinction between the problem and subproblem view, as sketched in Figs. 1 and 2, and the behaviour control diagram, as exemplified in Fig. 3, is essentially the distinction between a set of system behaviour views and a unified software view[60].

The software specification, in terms of machine executions, resulting from the approach described here can be regarded as a large concurrent program, in which each instantiated machine execution corresponds to a process execution. (We note, however, that if two concurrent behaviours produce identical machine responses to the same instance of a problem world event or state change, duplication of the response may be assumed to be harmful. It is then necessary to ensure that the response occurs only once in the problem world for each occurrence of the stimulus.)

If the problem world descriptions $\{Wi\}$ are expressed in a suitable form they may form the basis for a simulation of the physical problem world, as is normal practice for some systems (an avionics example is presented by O'Halloran [2]). Alternatively, if actual instances of the problem world are conveniently, cheaply and safely available, they may provide a test environment for the software.

Finally, for some purposes—including formal verification of pre-formal reasoning— it may prove expedient to fragment the machine behaviour specification into a set of stimulus-response pairs. In this transformation the structure and dynamic state of the behaviour control tree, the designed domains, and the text pointers of the machine instances must all be faithfully represented.

11   System Behaviour and the Salient Challenges

This paper has sketched an approach to development whose central theme and constant concern is the structuring and design of the system behaviour. The system behaviour is the visible intended effect of software execution, and is therefore the true end product of software development. The identification of the stakeholders' role as stating and validating desired properties of the system behaviour—rather than mandating behavioural details—is entirely appropriate: responsibility for designing a feasible behaviour that exhibits those properties must lie with the developers.

---

[60] An obvious possible extension is a third view. The problem diagrams show the relationships between machines and problem domains at the level of each constituent behaviour; the behaviour control diagram shows the relationships among machines. A third view would show the relationships among the problem domains induced by their interfaces of shared phenomena, including interfaces to the machines. The form and representation of such an extension is a topic of further work.

The focus on system behaviour provides an intellectual tool for addressing the salient development challenges of cyber-physical systems. The behaviour control tree, of which a trivial example is shown in Fig. 3, allows the complex overall behaviour to be comprehended in a nested structure of potentially concurrent constituent behaviour components. There is a parallel here with the advantages of classical structured programming. The nested behaviour structure establishes a tree of regions of the system's operating envelope[61], each with the accompanying assumptions that justify the chosen formalisations of the problem world properties. Construction of this tree must proceed largely bottom-up. Until a good understanding has been achieved of its components at every level, it is impossible to determine the scope either of proposed invariant requirements or of the validity of proposed formalisations of given problem domain properties.

At the level of a constituent behaviour the approach to complexity is incremental. In a subproblem the behaviour is initially considered in isolation as a complete system in itself. This initial treatment clarifies the simplest case of the behaviour, in which nothing goes wrong and there is no interference from interaction with other behaviours at common problem domains. At later stages the subproblem is revisited to address these and other sources of complexity. For example: deviation from the simplest case to handle a minor exception; modification to ensure compatibility with other behaviours; and interaction with the controlling parent or child behaviours. It may then become necessary to distinguish multiple versions of the behaviour according to the complexities of the context.

The fundamental thesis of this paper is that behaviour is an indispensable concept for developing dependable cyber-physical systems. The approach briefly presented here is still very much a work in progress, aiming to address, directly and effectively, the salient challenges of behavioural design and structuring.

References

1. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Boston (2001)
2. O'Halloran, C.: Nose-Gear velocity—a challenge problem for software safety. In: Proceedings of System Safety Society Australian Chapter Meeting (2014)

---

[61] The structure of environment conditions assumed by the subproblems naturally follows the structure of the behaviour control tree and the activation choices of controlling behaviours. The environment conditions of a controlled behaviour imply those of its controlling behaviour. The environment conditions assumed by the machine at the tree root are those of the system's complete operating envelope.

3. von Neumann, J., Morgenstern, O.: The Theory of Games and Economic Behaviour. Princeton University Press, Princeton (1944)
4. Poincaré, H.: Science et Methode; Flammarion 1908; tr Francis Maitland, p. 126, Nelson 1914, Dover 1952, 2003
5. Jackson, M.: Topsy-Turvy requirements. In: Seyff, N., Koziolek, A. (eds.) Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday. Verlagshaus Monsenstein und Vannerdat, Muenster (2012)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, Upper Saddle River (1985)
7. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems, pp. 477–498. Springer, New York (1985)
8. Dijkstra, E.W.: On the cruelty of really teaching computer science. Commun. ACM **32**(12), 1398–1414 (1989). (With responses from David Parnas, W L Scherlis, M H van Emden, Jacques Cohen, R W Hamming, Richard M Karp and Terry Winograd, and a reply from Dijkstra)
9. Smith, B.C.: The limits of correctness. In: Prepared for the Symposium on Unintentional Nuclear War; Fifth Congress of the International Physicians for the Prevention of Nuclear War 1985, Budapest, Hungary, 28 June–1 July. ACM SIGCAS Computers and Society, vol. 14,15, Issue 1,2,3,4, pp. 18–26, January 1985
10. Zave, P.: FAQ Sheet on Feature Interaction. AT&T (1999). http://www.research.att.com/~pamela/faq.html
11. Calder, M., Magill, E. (eds.): Feature Interactions in Telecommunications and Software Systems VI. IOS Press, Amsterdam (2000)
12. Descartes, R.: Discourse on Method, Part II; Works, vol. VI (1637)
13. Leibnitz, G.W.: Philosophical Writings (Die Philosophischen Schriften), Gerhardt, C.I. (ed.), vol. IV, p. 331 (1857–1890)
14. Jackson, M.A.: Principles of Program Design. Academic Press, Orlando (1975)
15. Polanyi, M.: Personal Knowledge: Towards a Post-Critical Philosophy. Routledge and Kegan Paul, London (1958). (University of Chicago Press, 1974)
16. Turski, W.M.: And no philosopher's stone either. In: Kugler, H.J. (ed.) Proceedings of the IFIP Congress. World Computer Congress, Dublin (1986)
17. Smith, B.C.: The limits of correctness. In: Prepared for the Symposium on Unintentional Nuclear War, Fifth Congress of the International Physicians for the Prevention of Nuclear War, Budapest, Hungary, 28 June–1 July (1985)
18. Jackson, M., Zave, P.: Domain descriptions. In: Proceedings of IEEE International Symposium on Requirements Engineering, January 1993, pp. 56–64. IEEE CS Press (1992)
19. Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices. Addison Wesley/ACM, New York (1995)