Topsy-Turvy Requirements

Michael Jackson

Department of Computing
The Open University
Milton Keynes MK7 6AA
United Kingdom

Abstract. A cyber-physical system comprises an assemblage of given material and human domains together with the computing element or elements introduced to control them. Requirements for such systems can be understood as properties of system behaviours: the central task of system development is to design the overall behaviour of the system to satisfy these requirements. This overall behaviour can be regarded as a structure of simplified independent behaviours, modified and recombined to address their interactions. A widely adopted approach to requirements perversely conceals this structure.

Keywords: arboricide, behaviour, comprehension, problem world, requirement, topsy-turvy.

Introduction

It is a great pleasure to write a paper for this festschrift in celebration of Martin Glinz's 60th birthday. Much of Martin's career has been devoted to the discipline of requirements engineering, and this short paper is intended as a small contribution to that discipline.

The requirements engineering community is a broad church: there is no agreed definition of what counts as a requirement. Instead, requirements take many forms according to industry custom, the nature of the system, the available facilities of requirements management software, and the backgrounds and tastes of the requirement engineers. Some requirements are goals; some express generalised aspirations to such desirable but elusive properties as safety, reliability or convenience. Some are paragraphs of natural language text; some are formal invariants; some are use cases; some are abbreviated accounts of software products or functions. And some are mysteriously isolated state transitions—fragments of a large state machine that may never be revealed in its entirety. It is this last category—requirements expressed as transition fragments—that I want to discuss in this short paper.

In a paper [Miller+06] presented at FME'03 in Pisa, Steve Miller, Alan Tribble and Mats Heimdahl describe how the requirements for the mode logic of a Flight Guidance System, expressed as "shall" statements in natural language, were formalised in CTL and checked against a large formal model of the intended system

behaviour [Heimdahl05, Miller+06]. The requirements were initially expected to be complete, unambiguous and consistent: inevitably, they were not. When errors were found, and corrections chosen, the affected informal requirements were modified. Virtually all the informal requirements had to be modified in this way. The paper gives some detail of one example. This was its original statement:

> *"If Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the Flight Control Panel."*

Before checking, this requirement was regarded as 'well-validated and non-controversial'. Checking showed two errors: the event of pressing the HDG switch could be pre-empted by a simultaneous event of higher priority; and, in any case, it should be ignored unless the side in which it occurs is active. The requirement was therefore modified [Heimdahl05] to give this cumbersome but more accurate version:

> *"If this side is active and Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the FCP (providing no higher priority event occurs at the same time)."*

Another example from the same system, given in the cited papers is:

> *"If this side is active and the mode annunciations are on, the mode annunciations shall be turned off if the onside FD is off, the offside FD is off, and the AP is disengaged."*

Heimdahl recognises that all is not well. He sees some potential advantages in use cases, describing them as 'more structured' than "shall" statements, but questioning whether they can capture all of a system's requirements.

David Harel, in a paper [Harel09] describing the origins of statecharts, also criticises requirements expressed in the fragmented style. He recounts how the voluminous requirements of a chemical manufacturing plant included the following three specifications of a tiny piece of behaviour, buried in three totally different locations:

> *"If the system sends a signal HOT then send a message to the operator"*
>
> *"If the system sends a signal HOT with T>60deg then send a message to the operator"*
>
> *"When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T<60deg."*

Harel points out the redundancy, inconsistency and logical confusion in these three statements, and goes on to explain how, participating in the development of an avionics system, he was led to develop the language and semantics of statecharts. He recounts an anecdote which compellingly demonstrated the intelligibility of the notation to a domain expert, who was able immediately to identify an error in the requirements expressed in a statechart.

Miller, Heimdahl and Harel all acknowledge in different ways that requirement statements written in this style as fragmented transitions suffer from fundamental deficiencies. Miller and Heimdahl demonstrate that a collection of such statements may misrepresent the actual intended behaviour of the system and may even be self-contradictory. Heimdahl observes [Heimdahl05] that these deep defects are mitigated in practice by what he calls "collateral validation"—that is, by the processes of software design, coding and testing. Developers, as they perform these processes, come to recognise many of the logical and other errors which abound in the stated requirements. Heimdahl rightly warns that a 'model-driven' approach in which code

is automatically derived from formally stated requirements elides these development processes and with them the opportunity to identify and correct the requirement errors.

Harel recognises a deficiency in the essential nature of fragmented transition requirements. They capture individual transitions of a large state machine, but do not exhibit the machine itself in its entirety. Because a realistic system has many levels of detail, and many overlapping modes and states, the representation of the complete state machine must be structured. The structuring must allow hierarchical and orthogonal relationships among component state machines, and seems to demand complex interactions among the events and transitions of the component machines. Harel's development of the statechart notation aimed specifically to address these needs.

Topsy-Turvy Requirements

Surprisingly, large sets of fragmented transition requirements are the norm in some industries. They are supported by requirements management software that is widely accepted and used. They are eagerly mined in an effort to trace the implementation elements that satisfy each individual requirement and, conversely, the requirement elements that justify each element in the implementation. The resulting development tasks are on a heroic scale. In a recent workshop on requirements engineering a member of an automotive software development group proudly announced that the software for one particular car model had 200,000 requirements.

This is a self-inflicted wound. In a word, we can see that fragmented transition requirements are *topsy-turvy*. They are upside down, reversed, inverted: the top is at the bottom and the bottom is at the top. Their creators are like perversely optimistic makers of jigsaws who make collections of independently created jigsaw pieces in the hope that they can be assembled to give pictures which the creators themselves have never seen and have only dimly conceived. We should not be surprised if the resulting collections of jigsaw pieces cannot be fitted together to give meaningful pictures. They present an unwanted challenge to comprehension—a gratuitous *arboricide* puzzle for the reader to solve [Jackson95]. For instance, in Heimdahl's first example, many questions invite our attention. What if Heading Select mode is already selected when the HDG switch is pressed? How and when is Heading Select mode cleared? If the press event is pre-empted will it be ignored, or will it take its desired effect as soon as it reaches the top of the priority list? What if at that moment the HDG switch has been pressed again, or this side is no longer active? Can Heading Select mode be selected by any other event? What makes the side active? Can it become inactive between the press event and the associated response? Which are the events of higher priority? Is the priority ordering static or dynamic? Not even correct answers to these questions would allow us to make sense of this isolated fragment of the jigsaw puzzle.

The essential point is that the requirements are fragments of larger behaviours—often of multiple overlapping behaviours. For those who design the requirements, for those who read and analyse them, and for the stakeholders whose needs and desires

they must embody, it is these larger behaviours that demand design, comprehension and assent. The mathematician Poincaré expressed the point succinctly [Poincaré08]:

> "Would a naturalist imagine that he had an adequate knowledge of the elephant if he had never studied the animal except through a microscope?"

The large behaviours of a system, evoked by the interactions of the relevant parts of the human and material problem world with each other and with the system software, provide the properties, functionalities, constraints and affordances that must satisfy the needs of all the stakeholders. We cannot hope to understand them if we express the system requirements as a large collection of fragmented transitions.

Why?

Why do so many development groups continue to suffer this self-inflicted wound? There are several reasons, and not all of them are spurious. One obvious reason is that a widespread practice, once deeply entrenched in a community, is very hard to dislodge. For researchers, it becomes an accepted baseline from which further progress can be attempted without lengthy preliminaries to introduce and justify unfamiliar assumptions: a major departure carries the risks and difficulties of a paradigm shift or even a scientific revolution. For many development organisations it is the established default: it is even claimed [Miller+06] that "The very commonality of use of shall statements indicates they are a natural and intuitive way for designers to put their first thoughts on paper." Practitioners are already inured to its use. Managers are naturally reluctant to adopt a risky novelty in place of a commonly accepted technique for whose use they know they will not be criticised.

Another, technical, justification can be argued from the desire to reason formally about system behaviour. Fragmented transition requirements can be regarded as defining operations on the system state. To that extent they fall into the conceptual pattern of action systems, with a long and successful record in computer science; the same pattern underlies the well-known specification languages VDM, Z and Event-B. Fragmentation makes behaviours far more tractable for purposes of finding logical inconsistencies and proving theorems—especially when the proofs are to be mechanised. If the result is incomprehensible, that is merely an inconvenient but unintended by-product.

Further, comprehensibility of individual behaviours alone would not solve the whole problem. The behavioural complexity of a realistic cyber-physical system presents a major intellectual challenge that we have been slow to meet. The complete system behaviour is an ensemble of many constituent behaviours that must satisfy many stakeholder needs and purposes. A varying subset of these behaviours is active during system operation, and their interactions are complex. Expressing requirements as fragmented transitions seems attractive because it appears to offer an escape route from this difficult problem: complexity of system state is substituted for complexity of behaviour, and we are free to solve the problem piecemeal. The complex behaviour becomes a collection of operations on the system state, and we may now consider each operation independently. Unfortunately, this escape route from behavioural complexity is often illusory.

The Challenge

The escape route from complexity is illusory because the data structure of the system state cannot provide an adequate substitute for the dynamic structures of the system behaviour. Dijkstra explained the reason 45 years ago in the famous *go to* letter [Dijkstra68]. First, he gave the well-known and convincing justification of structured programming.:

> "My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost best to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

For a comprehensible program, progress through the static text must correspond closely to progress through the process it evokes. This correspondence demands a coordinate system in which textual and executional progress can be related. The textual index—with some elaboration to handle iterations and procedure calls—provides such a coordinate system to describe the progress: reading the program text we can follow the evolving process. At this point in his exposition Dijkstra asks, and answers, a key question: Why can the program's local variables not provide such a coordinate system? His uncompromising answer is that

> "we can interpret the value of a variable only with respect to the progress of the process."

This is a subtle but vital point. The program state, understood as an element of the cross-product of the values of its explicitly declared variables, cannot provide the map on which we can mark "where we have got to" or "what is going on" or "what are we trying to do now". Dijkstra gives the tiniest possible illustration. In a process to count in a variable *n* the number of people who have entered a room, the value *n* at any moment may be either the number who have entered or that number minus one, depending on whether the process has yet incremented *n* to count the most recent arrival. Looking at *n* alone is not enough: we can understand its meaning only by reference to the process text that clearly shows the ordering of *arrival* followed by *increment n*.

A sequential program corresponds to one comparatively simple process. The behaviour of a realistic cyber-physical system is hugely more complex. There are many interfaces between parts of the complete system—the software and the problem world—at which behaviours must satisfy operational constraints. For example: a drive motor must not be reversed unless it has been at rest for a minimum time. There are human participants whose interactions with the system must be easily understood and convenient. For example: the user of a passenger lift, the operator of a radiation therapy system, or the driver of a car. There are larger-scale behaviours of the system that must satisfy requirements for efficiency and avoidance of starvation. For example: the travel of a lift in its shaft, alternately up and down, must minimise wasted movement while ensuring that every user request is answered by the arrival of the lift at the requested floor. The larger-scale behaviours must afford the desired smaller-scale interaction behaviours of users. Different versions of behaviours are

appropriate to different circumstances. For example: passenger lift service under firefighter control differs from normal lift service. Behaviour in the presence of a fault or failure may be entirely different from normal behaviour or merely a minor variation on it. For example: a slightly degraded lift service can be provided by skipping a floor at which the doors fail to open; but a major failure may require service to be abandoned altogether.

This challenge presented by this multiplicity of relevant behaviours is threefold. First, each behaviour must be represented in a way that makes it comprehensible to all those who must comprehend it and accept that it satisfies their purposes and needs. Some behaviours must be accepted by domain experts; some by requirement engineers; some by program designers; some by safety experts of a regulatory authority; some by operators; and some by representative lay users. There is no reason to suppose that a single formalism will be suited to all of these.

Second, defining the relationships among the behaviours and assembling them into the overall system behaviour is itself a major design task. Which behaviours are disjoint in time? Which overlap? Which are nested one within another? Which interact at common parts of the problem world, much as processes may interact by shared variables? When two behaviours place conflicting demands on the world, which it to take precedence? In what circumstances can a particular behaviour be pre-emptively terminated? When two behaviours are consecutive, can the second follow the first immediately, or is an additional intermediate behaviour necessary to ensure the right initial conditions for the second? How is the simplified ‘main-line’ version of a behaviour to be combined and reconciled with the variations to deal with exceptional conditions that may arise? In effect, this recognition of the importance of behaviours leads to a view of the requirements phase that we might call ‘designing with behaviours’. Behaviours must often be designed rather than merely elicited from stakeholders who—we might mistakenly have imagined—hold them fully-formed in their minds ready to be handed over to the system developers. The resulting behaviours, whether designed or elicited, must be reconciled and combined like the components of a large program.

Third, it must be recognised that the resulting structure of behaviours will demand substantial transformation before software design can even begin. Some behaviour descriptions will include problem world events that are distant from the software machine and may find no representation there. Some behaviours may be implementable as software processes, but some may not. For these latter behaviours it may be necessary to fragment their descriptive texts in the software, adding the text pointer to the set of local variables. Some problem world events may be relevant to more than one behaviour, stimulating a response in each one: in the software design it will be necessary to combine these responses into one software-controlled action. In general, any aspiration to seamless development of a realistic cyber-physical system, in which the same structure is carried through all development phases from requirements to software architecture and design, must be regarded with deep suspicion.

Envoi

A fragmented transition requirement is obviously topsy-turvy because it presents a tiny piece of a complete behaviour that has not been explicitly recognised. But the damage is more than this. Topsy-turvy requirements make it hard to see a core aspect of the large development task with any clarity. This aspect is concerned with designing the multitude of behaviours, making them comprehensible to those who must assent to the designs, and reconciling and combining them into the complex overall behaviour of a realistic system. Reorienting the topsy-turvy behaviours is only the necessary beginning.

Acknowledgements

I thank David Harel, Mats Heimdahl and Daniel Jackson for helpful comments on an earlier draft of this paper.

References

[Dijkstra68] E W Dijkstra; *A Case Against the Go To Statement*; EWD 215, published as a letter to the Editor (Go To Statement Considered Harmful): Communications of the ACM Volume 11 Number 3, pages 147-148, March 1968.

[Harel09] David Harel; *Statecharts in the Making: A Personal Account*; Communications of the ACM Volume 52 Number 3 pages 67-75, March 2009.

[Heimdahl05] Mats P E Heimdahl; *Let's Not Forget Validation*; In Proceedings of VSTTE Workshop, ETH Zurich, 2005, Bertrand Meyer and Jim Woodcock eds; Springer LNCS 4171, 2008.

[Heimdahl07] Mats P E Heimdahl; *Safety and Software Intensive Systems: Challenges Old and New*; in Proceedings of FOSE'07 Workshop, pages 137-152, IEEE, 2007.

[Jackson95] Michael Jackson; *Software Requirements and Specifications: a lexicon of principle, practices and prejudices*; Addison-Wesley, 1995.

[Miller+06] Steven Miller, Alan Tribble, Michael Whalen and Mats Heimdahl; *Proving the Shalls*; International Journal on Software Tools for Technology Transfer (STTT) 2006.

[Poincaré08] Henri Poincaré; *Science et Méthode*; Flammarion, 1908; translated by Francis Maitland, Nelson, 1914, Dover, 1952.

Festschrift for Martin Glinz, Norbert Seyff and Anne Koziolek eds, Verlagshaus Monsenstein und Vannerdat, 2012.