From Worlds to Machines

Axel van Lamsweerde

Abstract: This paper provides a personal account of some of the foundational contributions made by Michael Jackson in the area of Requirements Engineering. We specifically focus on the relationship between problem worlds and machine solutions, and try to connect each contribution to related efforts while suggesting possible continuations. The anchoring of machine solutions on problem worlds is first considered together with means for delimiting, structuring, and characterizing problem worlds. The elaboration of machine specifications from world requirements is then discussed, including the use of satisfaction arguments, the questioning of requirements and assumptions, and the reuse of problem schemas. A car handbrake control system is used as a running example to illustrate the main ideas.

Keywords: Requirements engineering, requirements satisfaction, domain descriptions, specification derivation, adequacy checking, problem structures, knowledge reuse.

Introduction

Poor requirements were recurrently recognized to be the major cause of software problems such as cost overruns, delivery delays, failure to meet expectations, or degradations in the environment controlled by the software. The early awareness of the so-called requirements problem [2, 3, 4] raised preliminary efforts in developing modeling languages for requirements definition and analysis [1, 34, 12, 10]. With the increasing complexity of software-intensive systems, the research challenges raised by the requirements problem were so significant that an active community emerged in the nineties with dedicated conferences, workshops, working groups, networks, and journals. The term "requirements engineering" (RE) was introduced to refer to the process of eliciting, evaluating, specifying, consolidating, and changing the objectives, functionalities, qualities, and constraints to be achieved by a software-intensive system.

Michael Jackson was involved in requirements engineering research since the early days. His ICSE'78 paper, for example, pointed out that the entities and events to be considered at RE time pertain to the real world surrounding the software to be developed [15]. In spite of this, a profound confusion persisted in software engineering research on the role and real nature of system requirements, software specifications, domain knowledge, and the relationships among them. These notions were quite often considered similar. Driven by bottom-up efforts to abstract away from programming languages, a variety of diagrammatic notations and specification languages flourished. It was however generally not clear what their target was – the software objects, operations and behaviors, or their counterpart in the real world? System prescriptions or domain descriptions? The problem space or the solution space? The intrinsic intertwining of these two spaces did not help clarify such confusions and misunderstandings.

A series of papers by Michael and colleagues, written for different purposes or under different perspectives, brought the much needed clarifications. The one that hit me the most in my perceptions at that time was the first I was aware of, presented at the very first research conference dedicated to the area [19]. Fortunately enough, a marvelous essay appeared soon after [16], elaborating on the main ideas and bringing them in a form more accessible to a wider audience. I still believe that this book is among the very few ones to be included in the reading list for anyone entering the field. Some of the foundational principles introduced there were summarized in an ICSE keynote paper [17], illustrated through a detailed example [20] and made further precise for a more technical audience [38].

Central to this work is the distinction between the problem world and the machine solution. To make sure that a software solution correctly solves some problem, we must first correctly define what problem needs to be solved. The problem is generally rooted in a complex organizational, technical or physical world. The aim of a software development project is to improve this world by building some machine expected to solve the recognized problem. The RE process is concerned with the machine's effect on the surrounding world and the assumptions we make about this world.

A series of fundamental questions then arise quite naturally:

- How should the machine be anchored on the problem world?
- How do we characterize the problem world?
- How do we delimit it and structure it?
- How can we ensure a correct transition from the problem world to a machine solution so that the machine as specified will meet the requirements expressed in the problem world?
- Can we support such transition by reuse of generic problem structures?

In his writings, Michael has provided insightful and elegant answers to these questions. The purpose of this paper is to provide a brief personal account of them, trying to situate them with respect to earlier or parallel efforts while suggesting some issues and perspectives raised by them.

This review is by no means intended to be comprehensive. It is inevitably biased by my research focus and by the ways Michael's work has influenced my own efforts.

Anchoring the Machine on the Problem World

Michael's popular visualization of the overall relationship between the world and the machine is now found in many RE courses and tutorials. Let us briefly recall it to define some important concepts and introduce the running example we will use to illustrate the main points.

The *problem world* is some problematic part of the real world that we want to improve by building some machine solution. It typically consists of components that interact according to certain rules and processes, e.g., car drivers, motors, handbrakes and rules for safe brake control (see Fig. 1). In the problem world, we may apprehend certain phenomena of interest such as a car driver wishing her car to leave some place.

INSERT FIGURE 1

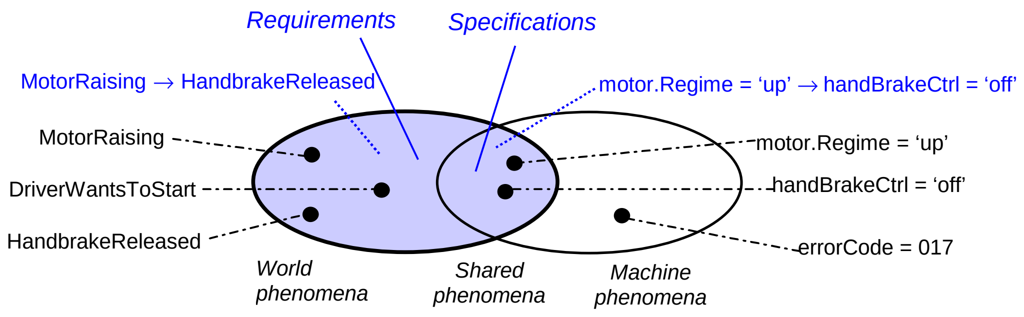

Figure 1. Phenomena and statements in the problem world and the machine solution

The machine solution is expected to solve some problem – in our example, the manual release of the car handbrake proves to be inconvenient or even unsafe in certain situations. The *machine* is an abstraction for what needs to be developed or installed in order to solve the problem. It consists of software to be developed, such as a handbrake controller in our example, possibly together with input/output devices, such as sensors and actuators, and sometimes COTS components to be purchased. In the machine solution, we may apprehend certain phenomena of interest too, such as an error code getting some specific value under certain error conditions (see Fig. 1).

The problem world and the machine solution both have their own phenomena while sharing others. The *shared phenomena* define the interface through which the machine interacts with the world. They are in the intersection of the two sets in Fig. 1. The machine *monitors* some of the shared phenomena while *controlling* others. For example, the machine in Fig. 1 monitors the phenomenon of the shared variable "motor.Regime" getting the value "up"; it controls the phenomenon of the shared variable "handBrakeCtrl" getting the value "off". The shared phenomena yield the *boundary* beween the world and the machine; we thereby know what will be automated by the machine and what will not.

In this framework, a *requirement* is a statement about world phenomena, shared or not shared, that the machine should help satisfy (in general, jointly with other components of the problem world). A *specification* is a statement about shared phenomena that the machine must satisfy alone through the phenomena it controls. For example, the statement

the handbrake shall be released if the motor regime is physically raised

in Fig. 1 is a requirement whereas the statement

handBrakeCtrl shall have the value *off* if *motor.Regime* has the value *up*

is a specification. A specification is thus a requirement while the converse is generally not true (see Fig. 1). Requirements are formulated in the vocabulary of stakeholders, in terms of phenomena of the problem world, whereas specifications are formulated in the vocabulary of software developers.

Related efforts. The four-variable model developed independently by Parnas and colleagues is closely related to this framework [31]. The machine here reduces to the software-to-be; the problem world, called *environment*, includes associated input/output devices. A requirement is defined as a mathematical relation between a set of monitored variables and a corresponding set of controlled variables in the environment; a specification is a mathematical relation between a set of software input variables and a corresponding set of output variables. Value changes for the monitored/controlled variables correspond to phenomena owned by the environment whereas value changes for the software input/output variables correspond to shared phenomena. The input/output variables are "images" of the monitored/controlled ones; they define the interface between the software-to-be and its input/output devices. A specification "translates" a corresponding requirement into the vocabulary of the software input/output variables.

In our own work, the terms "system goals" and "software constraints" [5] were originally used for what Michael called "requirements" and "specifications", respectively. The former are under the responsibility of one or more system agents, including people, devices and software, whereas the latter are under the sole responsibility of the software-to-be. Michael's characterization of software specifications in terms of monitored/controlled phenomena led us to a realizability criterion for goal assignment to single agents [26], and the introduction of explicit shared interfaces among interacting agents for inductive synthesis of requirements from scenarios [25].

Earlier system development methodologies such as context analysis [34], definition study [13] or participative analysis [29] also emphasized the need for grounding software requirements into the surrounding world. The important distinction beween requirements and specifications was sometimes recognized among practitioners too, e.g., under terms such as "user requirement" *vs.* "software requirement" or "customer requirement" *vs.* "product requirement".

Issues and perspectives. Michael's introduction of the machine concept as an abstraction for what needs to be developed or installed appears very helpful through the various stages of the RE process. At the earlier stages we may consider the machine to include input/output devices (such as sensors and actuators) together with foreign software compoments; at later stages we may introduce finer-grained responsibilities among components and restrict the machine to the software-to-be only. Such machine refinement should be supported through systematic rules and patterns [22].

In the first RE phases of requirements elicitation and evaluation, we generally need to consider multiple versions of the world together with multiple alternative machines [22]. This makes the actual picture a bit more complicated than what Fig. 1 may suggest.

1. The world-as-is and the world-to-be. In order to understand the problem to be solved and identify the requirements on a machine-based solution, we need to investigate two versions of the world:

- The *world-as-is* is the problematic portion of the real world as it exists before a machine solution is built into it;
- The *world-to-be* is the corresponding relevant portion of the real world as it should be when a machine solution will be built and operate into it.

In our example, the world-*as-is* is a standard car driving system with no automated support for handbrake control. We need to investigate this system in order to fully understand the concepts and components involved in it together with the rules and constraints for safe handbraking. We also need to investigate the world-*to-be* as some new concepts, components, rules and constraints emerge from automated handbrake control.

2. Alternative machine solutions. While investigating the world-to-be we often need to consider and evaluate alternative options for a machine solution, each generally resulting in a different set of shared phenomena. In our handbrake control example, we might consider one option where handbrake release results from a button being pressed by the driver, another where a vocal driver command is transmitted through a dedicated speech recognition component of the machine, and other options where the machine infers through different, alternative means that the driver wants to start. The world-machine boundary will in general vary from one option to the other, with more or less functionality being automated. In the preceding example, the more "intelligent" machine options will result in increased automation. The world-machine boundary is thus rarely fixed a priori when the RE process starts. Assessing alternative boundaries and selecting a most appropriate one is an important aspect of the RE process.

3. World evolutions and variations. In many software engineering projects the picture gets still more complicated.

- For requirements prioritization and evolution, we generally need to envision future versions beyond the world-*to-be*, to which lower-priority requirements and likely changes will be transferred. In other words, we may need to consider worlds-*to-be-next* –for example, what new or modified features might be desirable once drivers will get used to automated handbrake control?
- In product-line projects, we also need to consider multiple variants of the world-to-be and the machine solution, depending on variations among users or usage conditions –e.g., different handbrake control features for different classes of cars from the same manufacturer.

Multiple world-machine versions call for the elicitation and evaluation of multiple machine anchorings on the problem world; shared phenomena will in general vary from one version to the other.

Characterizing the Problem World

Considering now a specific version of the problem world, how do we characterize it? Michael has introduced a fundamental distinction between three types of statements:

- *Descriptive* statements state properties about the world in the indicative mood. Such properties hold typically because of natural laws or physical constraints. For example, the statement "a car's motor regime is raising when the air conditioner starts" is a descriptive statement.about the problem world in Fig. 1.
- *Prescriptive* statements state desirable properties of the world in the optative mood. Such properties need to be enforced. For example, the statement "the handbrake shall be released when the car's motor regime is raising" is a prescriptive statement.
- *Definitions* are statements assigning a precise meaning to concepts or auxiliary terms used in the problem world. For example, the statement "a car's motor regime is said to be raising if it increases by *X* above neutral level" is a definition.

The distinction between these types of statement is essential in the context of engineering requirements. Requirements are prescriptive. Knowledge about the problem world is made explicit through descriptive statements. We may need to negotiate, weaken, or find alternatives to prescriptive statements. We cannot negotiate, weaken, or find alternatives to descriptive statements. Unlike prescriptive or descriptive statements, definitions have no truth value. It makes no sense to say that a definition is satisfied or not. We need, however, to check them for accuracy, completeness, and adequacy –the same way as prescriptive or descriptive statements.

Descriptive statements play a specific role when we reason about requirements. In particular, they are used for deriving specifications from requirements in an arguably correct way [20], see below.

In his work, Michael has also emphasized the need for complementing formalized statements with their *designation*, that is, a precise definition of what the atomic expressions and predicates in such statements mean in terms of objects and phenomena in the problem world. This often neglected aspect of formalization is essential for assertions to be fully precise and interpreted in the same way by different persons. A designation corresponds to the notion of interpretation in classical logic, where the domain of interpretation is here constrained to be the problem world.

Related efforts. The four-variable model introduced a similar distinction between the set REQ of prescriptive constraints between monitored and controlled variables, to be maintained by the machine, and the set NAT of descriptive constraints resulting from physical laws in the environment [31].

The clarification of the difference betwee these types of statements has been been tremendously helpful in subsequent RE research work, including ours. In goal-oriented approaches to RE, *goals* are prescriptive statements of intent about the problem world, under the responsibility of agents structuring this world. We often need descriptive properties to prove the correctness of goal refinements into subgoals [6] or the correctness of goal operationalizations into system operations [27]. We also need descriptive statements for deriving preconditions for goal obstruction during risk analysis [24] and for deriving boundary conditions for conflict among goals during inconsistency management [23].

The role and importance of designations led modeling frameworks to enforce annotations of model items in order to make their interpretation fully precise in terms of world phenomena [22].

Issues and perspectives. Further useful distinctions can be made beyond prescriptive and descriptive statements, see Fig. 2. *Assumptions* about the problem world need frequently be made; they deserve special attention during the RE process:

- they are often involved in satisfaction arguments (see below);
- they are especially subject to scrutiny during risk analysis and adequacy checking;
- they may be used to discard or prefer alternative options;
- they tend to be more volatile as the problem world evolves.

INSERT FIGURE 2

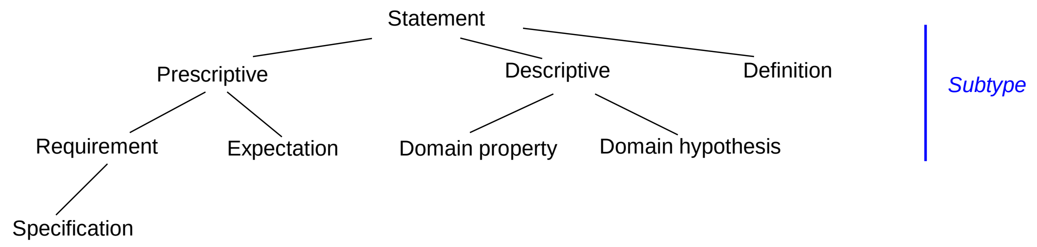

Figure 2. Further distinction among statements in the problem world

Some assumptions are prescriptive; they prescribe specific behaviors of problem world components that the machine cannot enforce. We might call these *expectations* [23, 24]. For example, "the driver shall press the acceleration pedal if he/she wants to start" is an expectation. Other assumptions are descriptive as they do not prescribe behaviors; we might call them *domain hypotheses*. The latter are not expected to hold invariably, unlike descriptive domain properties resulting from natural laws. For example, "handbrakes are never used below temperatures of –50°" is a domain hypothesis.

While both types of assumption deserve special scrutiny, expectations require a special treatment as they may call for specific actions to enforce them, e.g., the definition of dedicated procedures to be documented for people to follow them.

Delimiting and structuring the Problem World

Two obvious questions arise as we elicit statements about the problem world according to those different categories:

- How do we organize such statements for easier evaluation, documentation, and analysis?
- How do we delimit the scope of investigation and determine which statements are relevant and which ones are not?

Michael suggested that the scope be limited to the *subject matter* of the problem world [38]. If the problem is about handbrake control, we should not care for control of damping devices.

He also proposed problem diagrams as a means for structuring the problem world and for defining its scope more precisely [18]. As shown in Fig. 3, a *problem diagram* is a simple graph where nodes represent problem world components and edges represent connections through shared phenomena declared in the labels. In such declarations, an exclamation mark after a component name indicates that this component controls the phenomena in the declared set. A rectangle with a double vertical stripe represents the machine we need to build. A dashed oval represents a requirement. It may be connected to a component through a dashed line, to indicate that the requirement *refers* to it, or by a dashed arrow, to indicate that the requirement *constrains* it. Such connections may be labelled as well to indicate which corresponding phenomena are referenced or constrained by the requirement.

INSERT FIGURE 3

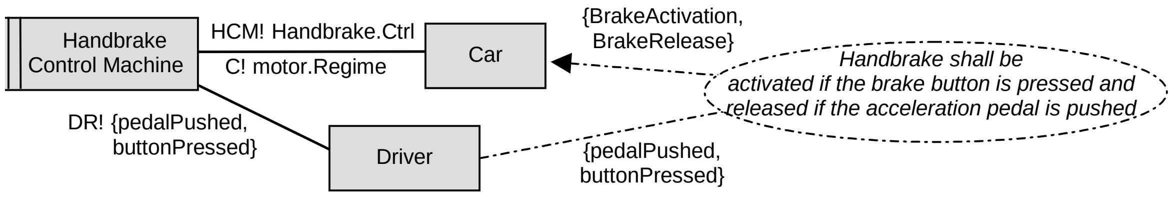
Handbrake Control Machine
HCM! Handbrake.Ctrl
C! motor.Regime
Car
{BrakeActivation, BrakeRelease}
Handbrake shall be activated if the brake button is pressed and released if the acceleration pedal is pushed
DR! {pedalPushed, buttonPressed}
Driver
{pedalPushed, buttonPressed}

Figure 3. Problem diagram for structuring the problem world

For example, Fig. 3 shows a problem world structured into three components: the handbrake control machine, the car, and the car driver. The handbrake control machine *HCM* controls the changes of value of the state variable *Handbrake.Ctrl* whereas the car component *C* monitors them. The latter component controls the changes of value of the state variable *motor.Regime*. The driver component *DR* controls the events *pedalPushed* and *buttonPressed*. These events are referenced in the requirement shown in the dashed oval. The latter requirement constrains the world phenomena *BrakeActivation* and *BrakeRelease*.

Related efforts. The context diagrams in *Structured Analysis* were a rudimentary form of problem diagram showing the connections between the software-to-be and environment components through shared interfaces [8]. In KAOS, the problem world is structured along complementary views: the active elements composing it, called *agents*; the *goals* these agents must satisfy; the *operations* they need to perform in order to operationalize the goals; and the *objects* which the goals and operations refer to [5]. Each view is captured through dedicated diagrams. In particular, agent diagrams correspond to problem diagrams; they capture the various system agents together with their shared variables (monitored or controlled) and their assigned goals [22]. The problem scope is delimited by the top-level and bottom-level goals in the goal refinement graph; the former must be satisfiable through cooperation of the system's agents only whereas the latter must be satisfiable by single agents assigned to them. In *i**, the problem world is structured into agents interconnected through *dependency* links of various types; an agent may depend on another for a goal to be achieved, a task to be performed, or a resource to be made available [36].

Issues and perspectives. The scope of the problem world is not always as easy to delimit as it might appear at first sight.

- Feature interaction problems may call for extending the normal scope to aspects apparently unrelated to the subject matter of the problem world [23]. In our simple example, air conditioning control should be included in the scope of investigation (see below) even though it seems totally unrelated to handbrake control.
- The elaboration of security requirements calls for analyzing potential threats against the system in order to anticipate appropriate countermeasures [21]. The scope of the problem world should therefore be extended with malicious components. How should the normal scope be thereby extended and how far? For example, should the handbrake control world care for the possibility of handbrake release by a car robber?
- In open systems such as service-based systems, foreign components appear, disappear, or evolve without any control from the machine. How should the problem world be bounded accordingly?

Identifying the "right" components for structuring the problem world is not necessarily an obvious task. A variety of heuristics may be used to support this [22]. Early architectural choices can also facilitate component identification and interconnection [32].

Another interesting issue concerns the further structuring of problem diagrams. For systems of significant size and constrained by a large number of requirements, we may need to compose/decompose some components in a problem diagram as well as the requirements on them. Structuring mechanisms such as aggregation, specialization, refinement and enrichment should therefore be supported together with corresponding proof obligations.

Chaining Satisfaction Arguments

Our job as requirements engineers is to elicit, make precise, and consolidate requirements, assumptions, and domain properties. In particular, we need to ensure that the requirements will be satisfied whenever the specifications are met and provided the assumptions and domain properties hold [16, 11]. This can be achieved through entailments called *satisfaction arguments*, taking the form:

$$\{SPEC, \mathrm{ASM}, \mathrm{DOM}\} \models Req$$

which reads:

> **if** the specifications in set *SPEC* are satisfied by the machine, the assumptions in set *ASM* are satisfied in the problem world, the domain properties in set *DOM* hold, and all those statements are consistent with each other,
>
> **then** the requirements *Req* are satisfied in the problem world.

The use of such arguments was thoroughly discussed and illustrated in [16, 20]. In our simple example it might look like this:

(*Req*:) MotorRaising → HandbrakeReleased

(*Spec*:) motor.Regime = 'up' → handBrakeCtrl = 'off'

(*Asm*:) motor.Regime = 'up' ↔ MotorRaising

handBrakeCtrl = 'off' ↔ HandbrakeReleased

To ensure that the specification *Spec* entails the requirement *Req*, we need to identify the assumptions *Asm* and make sure that the latter will be satisfied (some aren't in this example, see below). Assuming for the moment that these assumptions are adequate, we can obtain *Req* from *Spec* by the following rewriting: (a) we replace motor.Regime = 'up' in *Spec* by MotorRaising thanks to the first equivalence in the assumptions *Asm*; and (b) we replace handBrakeCtrl = 'off' in *Spec* by HandbrakeReleased thanks to the second equivalence in *Asm*.

Beyond establishing the correctness of specifications with respect to requirements, satisfaction arguments play an important role in managing the traceability among requirements, specifications and assumptions for requirements evolution [22]. If an assumption becomes no longer valid, for example, the specifications linked to it by satisfaction arguments must change accordingly.

Related efforts. Satisfaction arguments have been known for some time in programming methodology. When we build a logic program *P* in some environment *E*, the program has to satisfy its specification *S*. Therefore we need to argue that $P, E \models S$ [14]. The use of satisfaction arguments for RE was also suggested in [37]. Such arguments were made explicit in terms of goal refinement and operationalization in [5]. The role of assumptions and domain properties in argumentation trees became however apparent only after Michael's work.

Issues and perspectives. In practice, we often need to *chain* multiple satisfaction arguments in order to establish that a set of specifications ensures some higher-level concern. The specifications are shown to entail corresponding requirements; the latter are then shown to entail higher-level requirements, and the argumentation proceeds recursively until the higher-level concern is reached [22]. Domain properties and assumptions may be used all along the argumentation chain. Fig. 4 shows how such chaining of satisfaction arguments yields an *argumentation tree*. The tree shows how each parent node is satisfied by AND-satisfaction of its child nodes –the latter being specifications, requirements, domain properties, or assumptions.

INSERT FIGURE 4

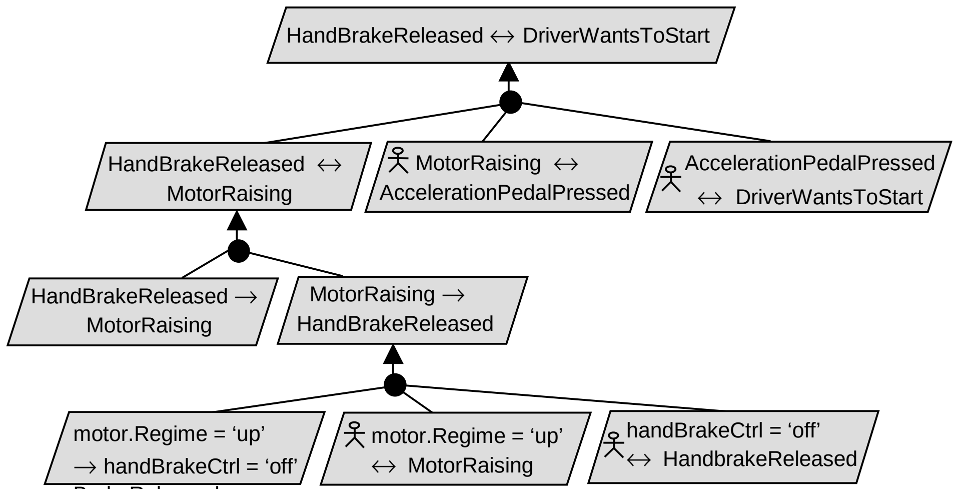
HandBrakeReleased ↔ DriverWantsToStart
HandBrakeReleased ↔ MotorRaising
MotorRaising ↔ AccelerationPedalPressed
AccelerationPedalPressed ↔ DriverWantsToStart
HandBrakeReleased → MotorRaising
MotorRaising → HandBrakeReleased
motor.Regime = 'up' → handBrakeCtrl = 'off'
motor.Regime = 'up' ↔ MotorRaising
handBrakeCtrl = 'off' ↔ HandbrakeReleased

Figure 4. Chaining satisfaction arguments: an argumentation tree

Note that the bottom subtree in Fig. 4, consisting of the three bottom leaf nodes and their parent node, corresponds to the satisfaction argument at the beginning of this section. This parent node and its left companion entail a new parent node in the tree, and so on. The "fellow" icon in Fig. 4 is used to indicate assumptions –in this case, expectations on problem world components such as the driver, motor sensors and actuators, and the brake actuator.

Deriving Specifications From Requirements

The turnstile control example in [20] provided a convincing illustration of how requirements can be systematically refined into specifications. Such refinement consists of incrementally replacing the non-shared phenomena referenced in the requirements by shared "images". As suggested in the previous section, such replacement relies on the use of domain properties and assumptions together with corresponding satisfaction arguments.

Related efforts. At about the same time we were exploring a pattern-based approach to provide systematic guidance in the requirements refinement process [6]. Refinement patterns encode common refinement tactics as goal-subgoal AND-trees –e.g., decomposition-by-cases, decomposition-by-milestones, divide-and-conquer, guard-introduction, etc. Such patterns can be formalized and organized in a pattern catalogue for easy retrieval. They are proved formally correct and complete once for all, e.g., using some available theorem prover. The patterns are then reused in matching situations through instantiation of their meta-variables.

Michael's work led us to extend our catalogue with patterns specifically dedicated to the resolution of problems of unmonitorability or uncontrollability by the machine [26]. Fig. 5 shows two such patterns together with their instantiation in our running example. The *accuracy* statements appearing as right child node in the refinement tree are often domain properties or environment assumptions.

INSERT FIGURE 5

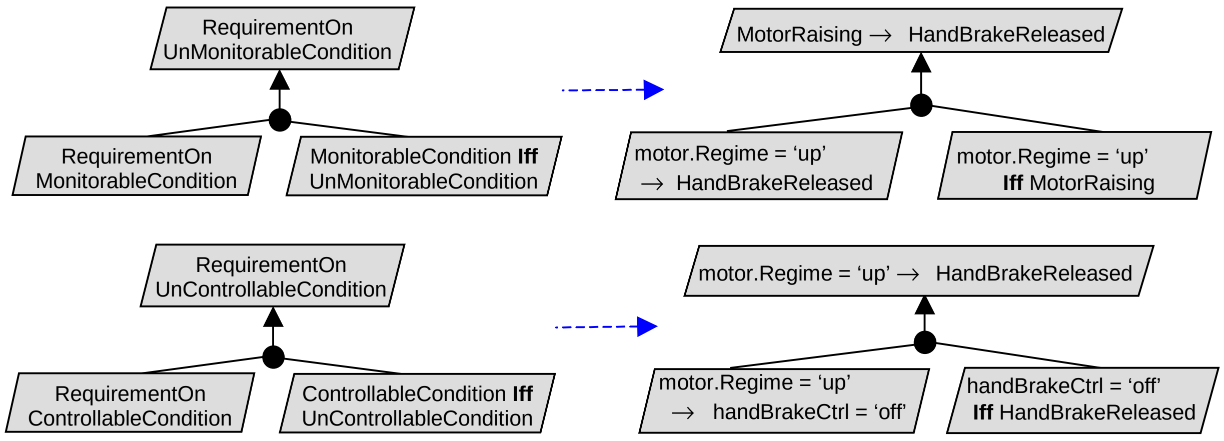
RequirementOn UnMonitorableCondition
RequirementOn MonitorableCondition
MonitorableCondition **Iff** UnMonitorableCondition
MotorRaising → HandBrakeReleased
motor.Regime = 'up' → HandBrakeReleased
motor.Regime = 'up' **Iff** MotorRaising
RequirementOn UnControllableCondition
RequirementOn ControllableCondition
ControllableCondition **Iff** UnControllableCondition
motor.Regime = 'up' → HandBrakeReleased
motor.Regime = 'up' → handBrakeCtrl = 'off'
handBrakeCtrl = 'off' **Iff** HandBrakeReleased

Figure 5. Unrealizability-driven refinement patterns

Issues and perspectives. The derivation of specifications from requirements gets more complicated as the requirements language and the specification language are based on different paradigms. For example, the requirements language might be natural language or some temporal logic; the specification language might be an event-based language such as *SCR* or a state-based language such as *Alloy*, *Z* or *B*. Different languages often rely on different semantic frameworks and assumptions –e..g., frame conditions, concurrency model (maximal vs. lazy progress, synchrony hypothesis on environment-software interactions), etc. The derivation process must then apply additional transformations so that the semantic assumptions on the target language are met [27, 7].

As mentioned before, the derivation process may also be faced with alternative options among which to choose. Systematic support should then be available for evaluating such options [30].

Questioning Statements

When domain properties or assumptions are used in an argumentation-based derivation, it is essential to check whether these are *adequate*. Fatal errors may originate from wrong properties or unrealistic assumptions used in correct derivations. Michael has often made this important point through convincing examples, including the A320 braking logic example [16]. The serious incident during an A320 landing on a rainy day at Warsaw airport might have resulted from the implicit use of a wrong domain property, namely, "the plane wheels are turning **iff** the plane is moving on the runway", not satisfied in case of aquaplaning. We must therefore systematically question the adequacy of critical statements used implicitly or explicitly in the RE process.

Related efforts. Obstacle analysis may help making such questioning more systematic [24]. For any questionable statement, we take its negation and look for alternative preconditions to make this negation true by use of domain properties. (A formal calculus is available for this.) The latter preconditions are then similarly refined until we reach fine-grained causes of obstruction of the root statement whose likelihood and criticality can be assessed by domain experts. This amounts to building a risk AND/OR tree rooted on the questioned statement. When we find fine-grained preconditions that are likely and critical, we must modify the questionable statement accordingly or add new requirements to overcome or mitigate the problem.

Fig. 6 outlines such analysis in the context of the argumentation tree built in Fig. 4. (The forked arrow there denotes an AND-refinement whereas multiple incoming arrows denote an OR-refinement.) The derived obstruction precondition *AirConditioningRaising* turns to be critical and very likely on hot summerdays. The corresponding set of requirements and assumptions should therefore be changed, leading to corrected requirements/assumptions –e.g., the weakening "HandBrakeReleased **Iff** MotorRaising **And Not** AirCo", or an alternative refinement of the root requirement that rules out brake release based on motor regime (e.g., a dedicated button to be pressed, a vocal driver command, etc.)

INSERT FIGURE 6

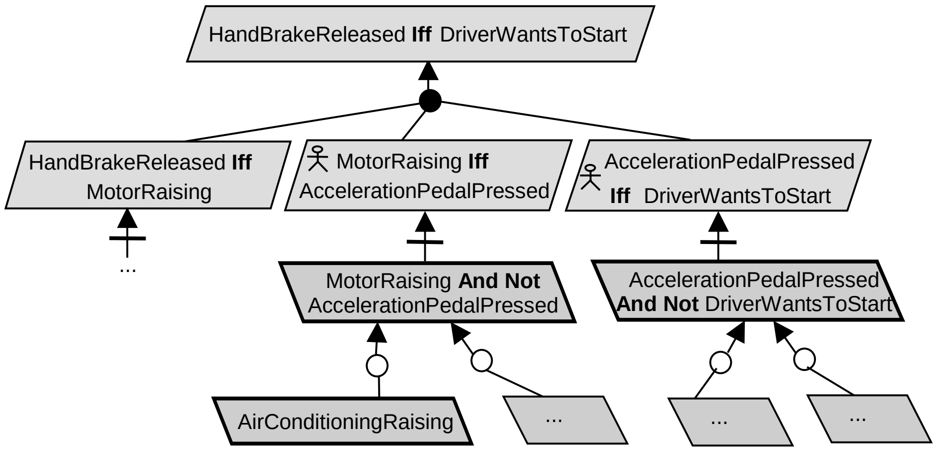
HandBrakeReleased **Iff** DriverWantsToStart
HandBrakeReleased **Iff** MotorRaising
MotorRaising **Iff** AccelerationPedalPressed
AccelerationPedalPressed **Iff** DriverWantsToStart
...
MotorRaising **And Not** AccelerationPedalPressed
AccelerationPedalPressed **And Not** DriverWantsToStart
AirConditioningRaising
...
...
...

Figure 6. Systematic questioning of statements

Issues and perspectives. The RE process may involve probabilistic requirements. They take the form "this target condition must be achieved in at least X% of the cases" or "this good condition must be maintained in at least Y% of the cases". Specification derivation, satisfaction arguments and systematic statement questioning get much more complicated then. Little support is available for handling such requirements.

Reusing Problem Schemas

Reusable RE processes and products may significantly help elicit and consolidate requirements. Beyond the reuse of derivation patterns discussed before, we may reuse problem structures. Instead of writing problem diagrams from scratch for every problem world we are faced with, we might predefine a number of frequent problem patterns. A specific problem diagram is then be obtained in matching situations by instantiating the corresponding pattern [18]. A *frame diagram* is a generic problem diagram capturing such problem pattern (called *frame*). The interface labels are now typed parameters; they are prefixed by "C", "E", or "Y" dependent on whether they are to be instantiated to causal, event, or symbolic phenomena, respectively. A generic component in a frame diagram can be further annotated by its type:

- A *causal* component, marked by a "C", has some internal causality that can be enforced –e.g., it reacts predictably in response to external stimuli. The machine component is intrinsically causal.

- A *biddable* component, marked by a "B", has no such enforceable causality –e.g., it consists of people. Constraints on them correspond to expectations (as introduced before).
- A *lexical* component, marked by a "X", is a symbolic representation of data.

Fig. 7 shows a frame diagram for the *commanded behavior* frame. It captures a class of problems where the machine must control the behavior of problem world components in accordance with commands issued by an operator [18].. The frame diagram states that the ControlMachine and the CommandedWorldComponent are causal components sharing the causal phenomena *C1* and *C2*; the former controls *C1* whereas the latter controls *C2*. On the other hand, the biddable component Operator controls command events *E4* that are monitored by the ControlMachine. The requirements constrain a set of world phenomena *C3*, connected to *C1* and *C2* through domain properties or assumptions to be provided, according to control rules refering to the commands *E4*. These rules state how the CommandedWorldComponent should behave in response to the operator's commands *E4*.

INSERT FIGURE 7

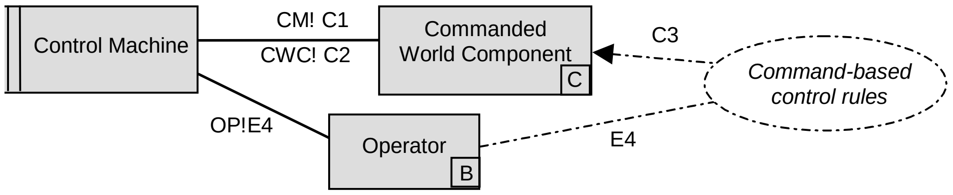

Figure 7. Problem frame diagram: commanded behavior frame

The problem diagram in Fig. 3 illustrates an instantiation of the commanded behavior frame in Fig. 7. This instantiation needs to be compatible with the types of the matching components and interfaces.

Related efforts. Various approaches were explored for the reuse of domain-specific abstractions for RE. Requirements clichés [33], analysis patterns [9], analog requirements frameworks [28] and domain theories [35] provide concept, task, and/or goal models of generic problem worlds together with their prescriptive and descriptive properties. The reuse mechanisms include specialization with single or multiple inheritance, traversal of a specialization hierarchy of domains, or structural and semantic matching based on analogical reasoning techniques.

Issues and perspectives. Many reuse approaches, including problem frames, are faced with similar challenges.

- The description of reusable problem schemas should be sufficiently rich to favor reuse beyond structural information such as component interfaces.
- A reusable schema should be sufficiently specific of a particular class of problems to enable transfer of useful discriminating features.
- The catalog of reusable schemas should be comprehensive for wide applicability in diverse situations and well-organized for easy retrieval.
- The ideal situation of perfect match is not that frequent; some support should therefore be available for validation and adaptation of the instantiated schemas.
- Last but not least, complex problem worlds generally combine multiple types of problems. Precise mechanisms are needed for composing multiple problem schemas and their instantiations.

Conclusion

A reference model for RE should provide a clear, precise, general, and orthogonal set of concepts and relationships on which RE processes and techniques can rely. Michael Jackson has significantly contributed to the elaboration of such model. The complex relationships between problem worlds and machine solutions, and the important role played by domain properties, assumptions and satisfaction arguments are now much better understood. The reference model is increasingly used in RE research, education, and practice. It opens multiple windows for further understanding and investigation of interconnections between problem definition and solution exploration.

Beyond the conceptual and technical contributions outlined in this paper, the RE community owes much to Michael for the clarity and elegance of his thinking and his so enjoyable style of writing.

References

[1] Alford, M., *A Requirements Engineering Methodology for Real-Time Processing Requirements*, IEEE Transactions on Software Engineering, Vol. 3, No. 1, January 1977, 60-69.

[2] Bell, T.E. and Thayer, T.A. *Software Requirements: Are They Really a Problem?*, Proc. ICSE-2: 2nd International Conference on Software Enginering, San Francisco, 1976, 61-68.

[3] Boehm, B.W. *Software Engineering Economics*. Prentice-Hall, 1981.

[4] Brooks, F.P., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, Vol. 20 No. 4, April 1987, pp. 10-19.

[5] Dardenne, A., van Lamsweerde, A. and Fickas, S., *Goal-Directed Requirements Acquisition*, Science of Computer Programming Vol. 20, 1993, 3-50.

[6] Darimont, R. and van Lamsweerde, A., *Formal Refinement Patterns for Goal-Driven Requirements Elaboration*, Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering, October 1996, 179-190.

[7] De Landtsheer, R., Letier, E. and van Lamsweerde, A., *Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models*, Requirements Engineering Journal Vol.9 No. 2, 2004, 104-120.

[8] DeMarco, T. *Structured Analysis and System Specification*, Yourdon Press, 1978.

[9] Fowler, M., *Analysis patterns: Reusable Object Models.* Addison-Wesley, 1997.

[10] Greenspan, S.J., Mylopoulos, J. and Borgida, A., *Capturing More World Knowledge in the Requirements Specification*, Proc. ICSE-6: 6th International Conference on Software Enginering, Tokyo, 1982.

[11] Hammond, J., Rawlings, R. and Hall, A., *Will it Work?*, Proc. RE'01 – 5th Intl. IEEE Symp. on Requirements Engineering, Toronto, IEEE, 102-109.

[12] Heninger, K.L., *Specifying Software Requirements for Complex Systems: New Techniques and their Application*, IEEE Transactions on Software Engineering Vol. 6 No. 1, January 1980, 2-13.

[13] Hice, G.F., Turner, W.S., and Cashwell, L.F., *System Development Methodology*. North Holland, 1974.

[14] C.J. Hogger, *Introduction to Logic Programming*, APIC Studies in Data Processing Nr. 21, Prentical Hall, 1984.

[15] Jackson, M., *Information Systems: Modeling, Sequencing, and Transformation*, Proc. ICSE-3: 3rd International Conference on Software Enginering, Munich, 1978, 72-81.

[16] Jackson, M., *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices*, ACM Press, Addison-Wesley, 1995.

[17] Jackson, M., *The World and the Machine*, Proc. ICSE'95: 17th International Conference on Software Engineering, ACM-IEEE, 1995, pp. 283-292.

[18] Jackson, M., *Problem Frames – Analyzing and Structuring Software Development Problems*, ACM Press, Addison-Wesley, 2001.

[19] Jackson, M. and Zave, P., *Domain Descriptions*, Proc. RE'93 – 1st Intl. IEEE Symp. on Requirements Engineering, January 1993, 56-64.

[20] Jackson, M. and Zave, P., *Deriving Specifications from Requirements: an Example*, Proc. ICSE'95: 17th Intl. Conf. on Software Engineering, ACM-IEEE, May 1995, 15-24.

[21] van Lamsweerde, A., *Elaborating Security Requirements by Construction of Intentional Anti-Models*, Proc. ICSE'04, 26th International Conference on Software Engineering, Edinburgh, May, ACM-IEEE, 148-157.

[22] van Lamsweerde, A., *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, January 2009.

[23] van Lamsweerde, A., Darimont, R. and Letier, E., *Managing Conflicts in Goal-Driven Requirements Engineering*, IEEE Trans. on Sofware. Engineering, Vol. 24 No. 11, Nov. 1998, 908-926.

[24] van Lamsweerde, A. and Letier, E., *Handling Obstacles in Goal-Oriented Requirements Engineering*, IEEE Transactions on Software Engineering Vol. 26 No. 10, October 2000, 978-1005.

[25] van Lamsweerde, A. and Willemet, L., *Inferring Declarative Requirements Specifications from Operational Scenarios*, IEEE Trans. on Sofware. Engineering, Vol. 24 No. 12, Dec. 1998, 1089-1114.

[26] Letier, E. and van Lamsweerde, A., *Agent-Based Tactics for Goal-Oriented Requirements Elaboration*, Proc. ICSE'02: 24th Intl. Conf. on Software Engineering, ACM-IEEE, May 2002.

[27] Letier, E. and van Lamsweerde, A., *Deriving Operational Software Specifications from System Goals*, Proc. FSE'10: 10th ACM Symp. Foundations of Software Engineering, Charleston, Nov. 2002.

[28] Massonet, P. and van Lamsweerde, A., *Analogical Reuse of Requirements Frameworks*, Proc. RE-97 - 3rd Int. Symp. on Requirements Engineering, 1997, 26-37.

[29] Munford, E., *Participative Systems Design: Structure and Method*, Systems, Objectives, Solutions, Vol. 1, North-Holland, 1981, 5-19.

[30] Mylopoulos, J., Chung, L. and Nixon, B., *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*, IEEE Trans. on Sofware. Engineering, Vol. 18 No. 6, June 1992, pp. 483-497.

[31] Parnas, D.L. and Madey, J., *Functional Documents for Computer Systems*, Science of Computer Programming, Vol. 25, 1995, 41-61.

[32] Rapanotti, L., Hall, J.G., Jackson, M., and Nuseibeh, B., *Architecture-Driven Problem Decomposition*, Proc. RE'04, 12th IEEE Joint Intl Requirements Engineering Conference, Kyoto, September 2004.

[33] Reubenstein, H.B. and Waters, R.C., *The Requirements Apprentice: Automated Assistance for Requirements Acquisition*, IEEE Transactions on Software Engineering Vol. 17 No. 3, March 1991, 226-240.

[34] Ross, D.T and Schoman, K.E., *Structured Analysis for Requirements Definition*, IEEE Transactions on Software Engineering, Vol. 3, No. 1, January 1977, 6-15.

[35] Sutcliffe, A. and Maiden, N., *The Domain Theory for Requirements Engineering*, IEEE Trans. on Sofware. Engineering, Vol. 24 No. 3, March 1998, 174-196.

[36] Yu, E.S.K., *Modelling Organizations for Information Systems Requirements Engineering*, Proc. RE'93 - 1st Intl Symp. on Requirements Engineering, 1993, 34-41.

[37] Yue, K., *What Does It Mean to Say that a Specification is Complete?*, Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design, Monterey, IEEE Press, 1987.

[38] Zave P. and Jackson, M., *Four Dark Corners of Requirements Engineering*, ACM Transactions on Software Engineering and Methodology Vol. 6 No. 1, January 1997, 1-30.