ISSN 1744-1986

The Open University

Technical Report N° 2006/02

Problem Reduction: a systematic technique for deriving Specifications from Requirements

Lucia Rapanotti
Jon G. Hall
Zhi Li

23rd February 2006

Centre for Research in Computing
Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Problem Reduction: a systematic technique for deriving Specifications from Requirements

Lucia Rapanotti Jon G. Hall Zhi Li
Centre for Research in Computing
The Open University
{L.Rapanotti, J.G.Hall, Z.Li}@open.ac.uk

February 23, 2006

Abstract

In this paper we explore the notion of problem reduction as a systematic transformation from requirements to specifications. We adopt the notion of problem as a requirement in a real-world context for which a software solution is sought, and view the process of software development as a problem solving process, leading ultimately, and hopefully, to a solution which satisfies the requirement in its context. In this paper, we focus on how a solution specification can be derived from a requirement, and introduce *problem reduction* as a systematic transformation to achieve this. We reflect on how problem reduction captures requirements engineering practices, express it in the context of Problem Frames and provide a set of rules for its application. The intention of the work is to increase the understanding of the problem solving process as well as to provide techniques to support sound engineering practices.

1 Introduction

The notion of problem in Software Requirements Engineering was introduced by Jackson over a number of years [13, 14, 15]. In this view, a problem is seen as a requirement in a real-world context for which a software solution is sought. The process of software development is then a problem solving process, leading ultimately, and hopefully, to a solution which satisfies the requirement in its context. The approach is gaining popularity (see [6] for an overview of current research and practice), and complements other approaches in Requirements Engineering such as goal-oriented [36, 32] and scenario-based [31, 1, 4] ones.

The problem-oriented approach has many qualities. It has a principled basis [37, 8, 10], which allows for adequacy argumentation and rich traceability [11] from requirements to solutions. It embodies a discipline of descriptions [16] for the capture of the relevant properties of the real-world context, while being sufficiently flexible to allow for informality [10]. It supports reuse of expertise through the identification of recurrent problem classes [12]. These qualities allow for a thorough analysis of single problems, as well as for the validation and verification of solutions. However, structuring the problem solving process for software remains, by and large, a craft rather than an engineering endeavour. By far the most practised approach to solving problem is a divide-and-conquer approach in the form of problem decomposition: various

forms of decomposition have been identified, from projections [12] to architecturally-inspired ones [30]. The related issue of recomposing sub-problems in the presence of conflicting requirements has also been addressed to a certain extent [20, 10].

In this paper we look at *problem reduction* as a technique to complement problem decomposition in solving problems, and in particular as a systematic transformation from problem towards solution. After reflecting on how the notion of reduction captures requirements engineering practices, we express it in the context of problem-oriented engineering, and provide a set of rules for its application. A running example is used for illustration. The intention of the work is to increase the understanding of the problem solving process as well as to provide practical engineering tools for problem solving.

The paper is organised as follows. Section 2 overviews some background and related work. Section 3 introduces problem reduction and applies it a running example. Section 4 discusses the approach, its relation to other techniques and limitations. Section 5 concludes the paper.

2 Background and related work

In this section we provide a brief overview of some background work relevant to our development. In particular, we summarise some relevant elements of problem frames, and look at the notion of requirements traceability, problem progression and causal reasoning. We also review some related work.

2.1 Problem Frames

Problem frames are a concretisation of many of Jackson's ideas on requirements specification into a framework for representing, analysing and classifying software problems. This section only provides a brief overview of the aspects of the framework which are more relevant to our work. A much more complete treatment can be found in [15].

In problem frames, problems are represented through the graphical notation of *problem diagrams*. This defines the context of a problem by capturing the characteristics and interconnections of the parts of the world the problem is concerned with, as well as a requirement to be satisfied in such a context. Figure 1 gives a simple example of a problem diagram. In the figure, the problem is to specify a machine (the solution) to control a device (the problem context) so that a certain work regime (the requirement) is satisfied.

This is, of course, a very simple example, with the problem context consisting of only a single device. The device is an example of *given domain*, that is that part of

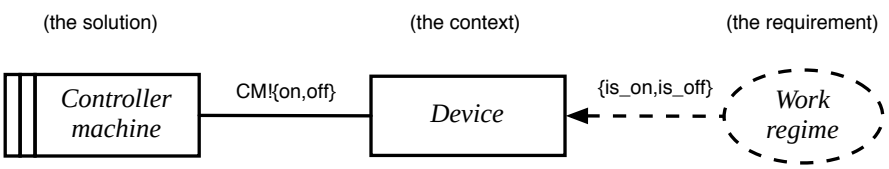


Figure 1: A simple problem diagram

the problem which is given. The solution is known as the *machine domain*, that is the artefact to be specified by the process of solving the problem. As such, there is only one machine domain in each problem. On the other hand, the context may be arbitrarily complex and made up of a number of interconnected given domains. We will see an example of a multi-domain context later on in the paper.

Problem diagrams also indicate the shared phenomena between domains. Such phenomena provide the vocabulary for the problem, and capture a variety of elements of interest to the problem, such as entities, values, events, commands or operations. In Figure 1, the link between device and machine indicates the phenomena that are shared between them. In this example, the shared phenomena are commands that the Controller machine can issue to switch the device on and off. That such phenomena are controlled by the Controller machine is indicated by ! after the abbreviation CM. Provided separately from the diagram are the designations of phenomena, which ground the problem's vocabulary in physical phenomena. (We will return to discuss shared and controlled phenomena later on in the paper.) We will use problem diagrams extensively in this paper for illustration of examples and techniques.

Not included in the problem diagram, but an essential part of the problem definition, are the domain descriptions and the statements of requirement. A notable point is that the problem frames approach does not prescribe the notation to be used for these descriptions and statements; formal and informal descriptions and statements are equally acceptable, provided they are precise enough to capture the characteristics of interest of the various respective domains. For instance, in our example, we could have:

Device: a device that can assume one of the following states: is_on and is_off. Commands on and off can be issued to effect state changes in the device.

Work regime: the device should be switched on at least once a day. Once switched on, it should not be on for longer than 2 hours.

The purpose of analysing a problem with enough rigour and precision is to be able to give a specification that is implementable and can be argued to satisfy the requirement in the given context. As for domain descriptions and requirement statements, the specification need not be formal, only sufficiently precise to allow further design and implementation. A possible specification for this problem might be:

Specification: The Controller machine should issue an on command at least once a day. Once an on command is issued by the Controller machine, it should be followed by an off command within 2 hours.

Note how close the statement of this specification is to that of the requirement: the main difference is that, while the requirement is expressed in terms of the internal states of the Device, the specification is expressed only in terms of phenomena which are shared (and in this case, controlled) by the Controller machine.

Once a specification is derived somehow, an important aspect of requirements analysis is arguing that it satisfies the stated requirement in the given context. We refer to such an argument as the *adequacy argument*[1]. This too needs not be formal: we need to acknowledge that there are parts of the real-world that escape a formal characterisation, but whose properties need to be taken into account nevertheless, and that formal proofs are in many cases unobtainable. In our example, a possible adequacy argument is as follows:

[1] This argument is also known as the correctness argument [15].

Argument: A machine that satisfies the specification will issue an on command at least once a day (specification), which results in the device been switched on (domain description) at least once a day, which satisfies the requirement. Once the machine has issued an on command (specification), it will issue an off command (specification), which results in the device being switched off (domain description), within 2 hours. Therefore, the device will never be in operation for more than 2 hours, which satisfies the requirement.

Note how the adequacy argument brings together domain descriptions and requirement (and specification) statements. Note also, how it relies only on facts which are known from such descriptions and statement. For instance, should the device been switchable on and off manually, the specification might not meet the requirement. However, unless this fact is explicitly stated as part of the problem contet, it cannot be taken into account in the argument.

This example is intended for illustration only, so it has been oversimplified. For instance, we haven't taken into consideration what happens when the controller is first introduced in its context (this is known as the initialisation concern [15]) or whether the machine should monitor the current state of the device before attempting to change its state.

As a final remark, the reader may note the subtle difference in our expression of domain descriptions vs. requirement and specification statements. The former are in the *indicative* mode, that is they express how things are. The latter are in the *optative* mode, that is they express how we would like things to be. This is an important distinction in problem frames, which we emphasise in this paper by a careful separation of the term 'description', which we only use in the indicative mode, and 'statement', which we only use in the optative mode.

2.2 Requirements traceability

Requirements traceability refers to the ability to track requirements throughout a product's life-cycle [7], to establish and understand links across requirements, design and implementation of software [27], and more generally, to manage requirements in the face of change and system evolution. Ramesh and Jarke [29] make a useful distinction between 'low-end' and 'high-end' uses of traceability, informed by a large empirical study into current industrial practice. Low-end use is primarily intended to establish simple dependencies between requirements, links between requirements and system components that satisfy those requirement, and the corresponding compliance verification procedures. High-end use also attempts to capture the complex relationships between requirements and their context, e.g, organisations with their strategic and operational goals, the diverse stakeholders, their viewpoints and any resulting conflicts. It also attempts to capture and track the rationale behind choices or conflict resolutions. [11] introduces the term *rich traceability* to indicate the practice of adding rationale or argumentation to traceability links in order to clarify the meaning of such links: given a particular statement of requirement, rich traceability will add an argument to explain which artefacts combine to satisfy it, and how this is achieved. Importantly, augmentation in rich traceability is not necessarily in the form of mathematical reasoning: modelling, simulation or engineering calculations and practices are considered as adequate forms of argumentation.

In this paper, we consider rich traceability in the context of the sequence of problems resulting from the requirement transformations induced by problem reduction.

2.3 Causal reasoning

The philosophical concept of causality or causation refers to the set of all particular "causal" or "cause-and-effect" relations. Causality can be used as the basis of reasoning in situations when we can assume that events of one sort (the causes) are systematically related to events of some other sort (the effects). Indeed, causality has been used extensively in inductive reasoning [2].

[25] observes how causal reasoning is ubiquitous in daily life and a useful tool for describing mechanisms, problems and solutions. It proposes a formal causal language for requirements specification aiming at filling the gap between natural language and formal reasoning. From that work, we adopt some basic notation, which we summarise in this section, and which will be used in the development of problem reduction.

Firstly, we distinguish between an event and the occurrences of that event. For instance, the single event 'bell rings' will typically occur many times in the lifetime of the bell.

Secondly, we consider cause as a relationship between events which induces a relationship between occurrences of those events. Notationally, we use:

- $\rightsquigarrow$ to indicate direct cause: given two events $ev1$ and $ev2$, $ev1 \rightsquigarrow ev2$ indicates that an occurrence of $ev1$ is the immediate cause of an occurrence of $ev2$; and
- $\rightsquigarrow^+$ to indicate the transitive closure of $\rightsquigarrow$: given two events $ev1$ and $ev2$, $ev1 \rightsquigarrow^+ ev2$ indicates that an occurrence of $ev1$ eventually leads to an occurrence of $ev2$, possibly through a chain of other event occurrences.

For example, 'bell rings' $\rightsquigarrow$ 'dog barks' is an example of the first form, while 'bell rings' $\rightsquigarrow^+$ 'Tom opens door' is an example of the second form, assuming that 'bell rings' $\rightsquigarrow$ 'dog barks' and 'dog barks' $\rightsquigarrow$ 'Tom opens door' (for some reason, Tom opens the door when his dog barks).

Of course, the distinction between the first and second form of causality depends on the level of analysis: if we can abstract 'dog barks' away, then 'bell rings' $\rightsquigarrow$ 'Tom opens door'.

2.4 Progression in problem solving

Problem frames acknowledge that early requirements are typically 'deep' in the world, i.e., distant from the interface of the machine. This is reflected in the fact that a problem diagram's context can be arbitrarily complex, with the requirement expressed in terms of phenomena which are not shared with the machine. When faced with this sort of problem, the search for a solution may be facilitated by some sort of progression [12] from requirements deep in the world towards a machine specification.

This view is not limited to problem frames, but shared by other approaches to requirements. For instance, in goal-oriented approaches [36, 32] high-level goals, which could be regarded as expressions of requirements deep in the world, are successively decomposed and refined until technical requirements are specified for a machine to satisfy. Similarly, in scenario-based approaches [31, 1, 4] some process of refinement is often required from high-level scenarios, often capturing business processes within an organisation, down to low-level scenarios expressing the direct interaction between a software system and its actors.

Although a reflection of some useful practice, progression has yet to become accepted as an engineering technique. This is the gap this work and its sequels proposes to fill.

2.5 Related work

While the move from specification to design and implementation of software is well understood, having been the subject of study in software development and formal methods for many years, the systematic derivation of specifications from requirements remains, by and large, an open problem.

Work has been carried out on the subject in the goal-oriented requirements engineering community on the derivation of specifications from goal-oriented models of requirements [34, 22]. In those approaches, requirements are expressed as goals seen as statements of intents, which can be organised in AND/OR refinement structures. Goals may range from high-level, strategic concerns (e.g, business goals in an organisation), down to low-level, technical concerns (e.g, specific constraints on the software to be). The requirements elaboration process is one which transforms high-level goals into operational specifications of software services. In [22] this process of elaboration is made systematic through the association of the goal-model with a small set of related models, which capture structural and behavioural aspects of the system to be developed. For soundness, the construction of such models is subject to a set of consistency rules. The resulting specifications are expressed in a real-time temporal logic [23]. [34] follows similar lines, although it exploits scenarios [1] for the elaboration of behavioural models. Also from the goal-oriented community, comes the TROPOS project [3], an effort into the definition of a software development methodology driven by requirements expressed as goals. The basic idea behind TROPOS is to combine intentional goal models with a set of UML [26] techniques for modelling processes and object structures. As the move from UML to code is well-understood (see, e.g., [21]), what needs to be explored is the development of UML models consistent with the goal model. Work on TROPOS is still in progress.

Johnson [19] proposes some automatic support for the process of transforming requirements into specifications, based on the idea of exploiting operational descriptions of requirements and environmental properties. Tool support allows for such descriptions to be written (in a language defined by the author for this purpose [18]) by the modeller, from which simulations of the combined system and environment behaviour can be derived, and for semi-automatic transformations to be applied in order to transform the requirements descriptions into specifications of system behaviour. Operational descriptions are claimed to be favoured by analysts, and to lead to a more accurate requirements analysis process thanks to the automatic simulations they allow.

In [17], Jackson and Zave state some principled elements of a method for deriving specifications from requirements, and illustrate them on a simple example. Indeed, that work shares with our approach much of its principled basis, including the separation of indicative and optative descriptions, the designation of phenomena, the view of a specification as a particular form of requirement. We will expand on such a principled basis in the next section when we enter the details of our approach. The intent of our work, however, is to embody such principles in practical techniques for deriving specifications from requirements in the context of a problem-based requirements analysis.

3 Problem Reduction

The technique we develop in this section takes its inspiration from a discussion and illustration which appear in [15], reflected in Figure 2.

From [15]:

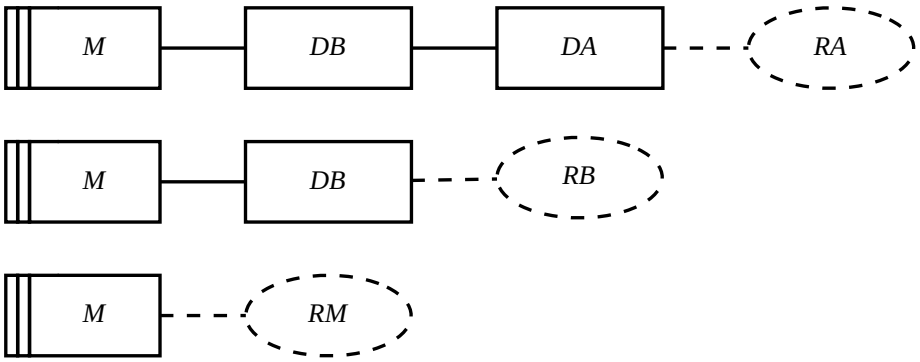


Figure 2: The basic idea of progression (adapted from [15])

> "The top problem is deepest into the world. Its requirement RA refers to domain DA. By analysis of the requirement RA and the domain DA, a requirement RB can be found that refers only to domain DB, and guarantees the satisfaction of RA. This is the requirement of the next problem down. Eventually, at the bottom, is a pure programming problem whose requirement refers just to the machine and completely ignores all problem domains."

No systematisation of progression is offered in [15]. In this paper, we propose problem reduction as a systematic transformation for progressing a problem closer to the machine, as described above, that is for allowing the systematic derivation of RB from RA and DA, in a way which guarantees the satisfaction of RA.

We present our approach to progression in the context of a process of problem analysis, in which we identify and separate[2] the following activities:

- Identifying relevant domains;
- Identifying relevant phenomena and their designation;
- Identify phenomena sharing and control;
- Describing domain behaviour;
- Stating the requirement;
- Moving the requirement closer to a specification;
- Formulating an adequacy argument.

The substantive contribution of this paper is the provision of techniques for the activity of moving the requirement closer to a specification. The other activities are presented as necessary support activities in the process of analysis.

A running example will be used for illustration of the techniques, and relevant heuristics will be provided at each step for the benefit of the practitioner.

[2]This separation is entirely for presentation purposes. Indeed, the process of analysis of any realistic problem will require repeated iterations among these steps.

3.1 Running example

For illustration of the technique, in the sequel we will use the following example problem. We consider the development of a Point of Sale (POS) system for a shop. The new POS software system is to be used to process all sales within the shop. The system is to include a controller, to be designed, and some hardware, purchased from a third party. The new POS hardware includes a barcode reader, a credit card reader, a keyboard and display, and a cash drawer. A possible scenario of use for the new system (as might have been elicited from stakeholders) is as follows:

> "Tom, a customer, takes an item he wishes to purchase to the till and gives it to cashier Mary. Mary uses the barcode reader to read the barcode on the item. The POS display indicates the amount due to purchase the item. Mary takes a credit card payment from Tom: she scans Tom's card using the card reader and waits for the sale to be completed and a receipt printed. She then returns the credit card and items to Tom, together with the receipt as proof of purchase."

3.2 Activity: Identifying relevant domains

Similar to other approaches to requirements analysis [31, 5], an important step in a problem definition is the identification of the problem context and the system boundary, as this allows us to start identifying what is already in the world, and what is the subject of design. From a problem frames' viewpoint, this corresponds to identifying all relevant physical domains in the problem.

Considering the example in the previous section, the domains are summarised in the following table, where for each of them, a brief description is given explaining which part of the world they represent:

POS h/w	The new POS hardware, including a barcode reader, a card reader, a keyboard and display, and a cash drawer.
Cashier	A shop employee authorised to perform sales.
Customer	A person wanting to buy an item from the shop.
Controller machine	The solution to be designed.

Here is a summary of our approach to identifying domains:

Identifying domains Given domains should be identified based on any available problem statement and/or in consultation with relevant stakeholders. Given domains are physical: they represent parts of the real world whose behaviour and properties can be observed. Each domain should be given a name and a brief description indicating which part of the real-world the domain represents. There is only one machine domain in each problem, which is the subject of design.

3.3 Activity: Identifying relevant phenomena and their designation

In problem frames, interaction occurs through physical phenomena which are shared between domains. Physical phenomena may include physical entities, physical events

or physical representations of states and values.

Problem frames do not prescribe how phenomena should be chosen and represented. In our development, however, we will be a little more prescriptive in the way we use the various types of phenomena. For simplicity, we will focus only on those physical phenomena which are entities or events, while we will consider states and values as representing entities' attributes. This is a common approach to identifying elements in an application domain (see, e.g., [21]). For instance, we might consider a physical entity such as "book", with a related event "give(book)" between two (or more) parties, while the book's ISBN will be considered as an attribute of the book and represented as "book.ISBN"[3].

The choice of phenomena is important as they constitute the basic vocabulary for building problem descriptions, such as behavioural domain descriptions, requirement and specification statements, or adequacy arguments. As such it plays an important role in problem analysis in general, and problem reduction in particular. Phenomena correspond to aspects of the real-world that can be observed, hence have to be grounded therein. This is achieved through their *designation*, that is some description that communicates which parts of the real-world the phenomena are representative of. Note that such a description is about the (informal) real-world and has to be significant to the problem's stakeholders, hence is stated informally.

For our example, the table in Figure 3 summarises the entities and events we have identified, together with their designation. Note that our choice, although sensibly based on the problem description, is still arbitrary to a large extent. In a real-problem analysis relevant stakeholders would be consulted to compile an appropriate list.

The following summarises our approach to identifying phenomena:

Identifying phenomena Phenomena are ground terms used for problem descriptions. They correspond to aspects of the real world you can observe. The following types of phenomena could be identified: entities, together their attributes, and events. Entities correspond to physical objects in the problem domain. Events usually involve entities and these are often shared among the given domains and/or the machine in the problem context. For each phenomenon, provide a designation that grounds it in the problem world.

3.4 Activity: Identifying sharing and control

In problem frames, the sharing of phenomena is something all sharing participants are part of simultaneously. As [15] states: "the participation in a shared event is like a hammer hitting a nail: there is only one event, and the hammer and the nail both take part in it simultaneously". In the case of a shared entity, value or state, both sharing parties have access to it. As explained in the previous section, phenomena that are shared are represented in a problem diagram as annotations on links between domains. Not all phenomena are shared, however. Some pertain to single domains only, e.g., the internal state of a device. We call such phenomena *internal*.

All internal phenomena of a domain are by default controlled by that domain. For shared phenomena, however, the problem frames approach assumes that only one sharing domain can control that phenomenon. In terms of notation, given a phenomenon ph shared between domains D and D', we will write $D!ph$ to indicate that D controls ph. For brevity, when a domain D controls all phenomena in a set a, we will write

[3]This notation is reminiscent of that of object-oriented approaches, although there is no intention of taking that perspective here.

item	entity	An item available for sale in the shop. It has an attribute *info*, representing relevant item information.
payment	entity	The payment a Customer makes for the purchase of an item. It has an attribute *info* representing relevant payment details.
receipt	entity	A slip of paper containing details of the purchase of an item and its payment, which is used as proof of purchase. It has an attribute *info* representing the purchase details.
present(item)	event	The exchange of an item between Customer and Cashier
present(payment)	event	The exchange of a payment between Customer and Cashier
present(receipt)	event	The exchange of a receipt between Customer and Cashier
enter(item.info)	event	The action of the Cashier of entering item information using the POS h/w, e.g., scanning the item's barcode using the POS h/w barcode reader
enter(payment.info)	event	The action of the Cashier of entering payment information using the POS h/w, e.g. scanning a credit card using the POS h/w card reader
transfer(item.info)	event	The action of the POS h/w of transferring item information to the Controller
transfer(payment.info)	event	The action of the POS h/w of transferring payment information to the Controller
generate(receipt.info)	event	The action of the Controller of making receipt information available to the POS h/w
print(receipt.info)	event	The action of the POS h/w of printing a receipt

Figure 3: Phenomena

$D!a$. The notion of control has slightly different connotations depending on the type of phenomenon. In particular:

if the phenomenon is an event ev, that D controls ev means that D makes an occurrence of that event happen. If ev is shared between D and D', only D can make the shared event occurrence happen;

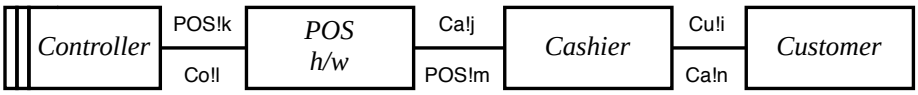
if the phenomenon is an entity en shared between two domains D and D', that D controls en means that only D can change its state or make the entity en available for sharing.

Although the distinction is an important one, separating events and their occurrences in all descriptions can be cumbersome at times. Therefore, in the following, we will often use the shorthand 'domain D does ev' to signify 'domain D makes an occurrence of ev happen'.

As we are interested in causality between events, our main focus will be the sharing and control of events. The table below summarises the events controlled by the various domains in our example.

Customer	Cashier	POS h/w	Controller
present(item) *present(payment)*	*enter(item.info)* *enter(payment.info)* *present(receipt)*	*transfer(item.info)* *transfer(payment.info)* *print(receipt.info)*	*generate(receipt.info)*

Control and sharing of those events is indicated in the diagram of Figure 4. (This type of diagram is called a *context diagram* [15]: it is a problem diagram without a requirement.). For instance, *present(item)* is shared between Customer and Cashier and is controlled by Customer.



i	{*present(item)*, *present(payment)*}
j	{*enter(item.info)*, *enter(payment.info)*}
k	{*transfer(item.info)*, *transfer(payment.info)*}
l	{*generate(receipt.info)*}
m	{*print(receipt.info)*}
n	{*present(receipt)*}

Figure 4: Phenomena sharing and control

Identifying sharing and control For each phenomenon, establish whether it is internal or shared, and the domain which controls it. Represent shared phenomena on a problem diagram as annotations of the links between the sharing domains. Indicate that domain D controls phenomenon ph as $D!ph$.

3.5 Activity: Describing domain behaviour

While identifying shared phenomena allows us to reason about the interaction of domains, identifying causal relations allows us to reason about the behaviour of domains. We will exploit the two together to reason about chains of behaviour within the problem context.

We will use causal relations to capture behaviour as part of domain descriptions. In particular, given two events $ev1$ and $ev2$ either shared by or internal to a domain D, we will adopt the notation of Section 2.3 and write as part of the description of D that $ev1 \rightsquigarrow ev2$ to indicate that an occurrence of $ev1$ is the immediate cause of an occurrence of $ev2$, and $ev1 \rightsquigarrow^{+} ev2$ to indicate that an occurrence of $ev1$ is the eventual cause of an occurrence of $ev2$, possibly through a chain of other event occurrences.

In reasoning about a domain's behaviour, we also need to take into consideration the particular nature of the domain. [15] makes a useful distinction between *causal* and *biddable* domains[4].

A causal domain is one whose properties include predictable causal relationships among its phenomena. For instance, the POS hardware in our example is a causal domain, built so that the following causal relations hold (at least at some level of abstraction):

enter(item.info) $\rightsquigarrow$ *transfer(item.info)*
enter(payment.info) $\rightsquigarrow$ *transfer(payment.info)*
generate(receipt.info) $\rightsquigarrow$ *print(receipt.info)*

A biddable domain lacks in predictable internal causality, but can still, to a limited extent, be assumed to behave in a particular way. Biddable domains model people. For instance, the Customer in our example is biddable: there is no predictable causal relation between its internal and shared phenomena. However, based on some domain knowledge of customers' behaviour, we can assume that when a Customer wishes to purchase an item, they are likely to present the item and its payment to the Cashier. The Cashier is also biddable. Presumably, the Cashier has been trained in order to work in the shop, and we can assume some level of predicability in their behaviour. So, a trained Cashier could be treated as a causal domain, for certain parts of their behaviour – those associated with the operation of the POS h/w, where we can assume the following causal relations will hold:

present(item) $\rightsquigarrow$ *enter(item.info)*
present(payment) $\rightsquigarrow$ *enter(payment.info)*
print(receipt.info) $\rightsquigarrow$ *present(receipt)*

Given the above causal relations, examples of behaviours consistent with them are [5]:

$\langle$ *present(item)*, *enter(item.info)*, *transfer(item.info)* $\rangle$

$\langle$ *present(payment)*, *enter(payment.info)*, *transfer(payment.info)* $\rangle$

$\langle$ *generate(receipt.info)*, *print(receipt.info)*, *present(receipt)* $\rangle$

Describing domain behaviour Consider the nature of each given domain, whether causal or biddable. For each causal domain, express any causal relationship between its phenomena. For each biddable domain, express the causal aspects of its behaviour.

[4] There are other types of domains which we do not consider in our discussion.

[5] Each pair of angle brackets delimits a sequence of event occurrences; for convenience, in each sequence, we have labelled an event occurrence with the event name.

Note that the form of causal relations we have considered so far is very simple. In particular, there are no choices to be made. Although this is appropriate in our example, in general, the treatment of real-world problems will require choices to be considered. We do not give a full treatment of conditional causality in this paper, but will address some of the issues in Section 4.

3.6 Activity: Stating the requirement

In our example, the system to be designed will serve the business need of allowing customers to purchase items. A possible expression of this requirement is:

Purchase The Customer having shared a number of *present(item)* followed by one *present(payment)*, if *payment* is for the correct amount, should share a *present(receipt)*, where *receipt.info* should list *item.info*, for all the items, and *payment.info*.

The following observations can be made of this description. Firstly, the requirement is expressed in terms of the identified phenomena. This is consistent with our view that such phenomena constitute a basic vocabulary for precise descriptions: that the phenomena have a clear designation, means that we will know what the requirement means. (Note that a typical stakeholder is unlikely to express a requirement so precisely or formally; however this expression is not so far from an informal expression a stakeholder may be more comfortable with, once the phenomena designation is taken into account.) Secondly, the requirement is expressed from the Customer's viewpoint, i.e., it is expressed only in terms of phenomena the Customer has access to, whether internal or shared by the Customer. This is consistent with the view that requirements reflect needs deep in the world. Thirdly, the requirements expresses some constraints on phenomena, that is the conditions that must be true of them. In our example the constraints express the particular ordering in which occurrences of *present(payment)*, *present(item)* and *present(receipt)* should happen, and the particular correspondence that must exist between their entities, e.g, that the *payment* must of the correct amount for the *item*, etc. Finally, and consistently with the guidelines of [31], the requirement can be validated, that is, it is possible to determine whether a solution satisfies it. This is because the constraints it expresses are measurable: once a solution is provided, one will be able to check whether it allows only permissible sequences of event occurrences.

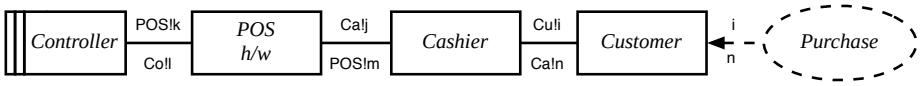
The properties of this requirement description are captured by the following:

Stating Requirements Consider deep-in-the world requirements to start the process of analysis, and use identified phenomena as the basic vocabulary for your requirement statements. A requirement should be stated in the optative mode, and should express some measurable constraints on the identified phenomena.

By adding the requirement to the context diagram of Figure 4, we arrive at the problem diagram of Figure 5. The figure emphasises that the requirement constrains phenomena in sets i and n.

3.7 Activity: Moving the requirement closer to a specification

The technique we propose for the reduction of the requirement to a specification is based on solution preserving requirements transformations, and it is achieved through



i	{*present(item)*, *present(payment)*}
j	{*enter(item.info)*, *enter(payment.info)*}
k	{*transfer(item.info)*, *transfer(payment.info)*}
l	{*generate(receipt.info)*}
m	{*print(receipt.info)*}
n	{*present(receipt)*}

Figure 5: The problem diagram

the repeated applications of a small set of rules. The rules are used to derive new statements of requirements by progressively replacing expressions concerning phenomena, until a statement is reached in which only phenomena shared with the machine are mentioned - such a statement satisfying our notion of specification.

In basing our notion of causality on that of [25], we limit ourselves to rules that can be applied to the treatment of causal relations without choice and in which each event has exactly one cause and/or effect. We will discuss more general forms of causality in Section 4.

In the following, we denote by D *does* ev any part of a requirement statement that implies an occurrence of event ev controlled by D, and by D *observes* ev any part of a requirement statement that implies an occurrence of event ev observed by D. In any requirement statement we will always try to emphasise and separate expressions of types D *does* ev and D *observes* ev. For example, we will rewrite Purchase as:

Purchase The Customer

- having done: a number of *present(item)* followed by one *present(payment)*,
- if *payment* is for the correct amount, should observe: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

Note that this is a syntactic rewrite of convenience: it emphases a pattern that we can exploit in the expression of our rules.

3.7.1 Changing viewpoint

Our first rule has to do with generating a new requirement statement based on the two perspectives of domains sharing an event: that of controlling and that of observing the event. The rule is as follows:

R1: Changing viewpoint

a) Consider domains D and D' that share event ev (see Figure 6.a), and requirement statement R, containing D *does* ev. Then a requirement statement R', can be derived from R by replacing D *does* ev with D' *observes* ev.

b) Consider domains D and D' that share event ev (see Figure 6.b), and requirement statement R, containing D *observes* ev. Then a requirement statement R', can be derived from R by replacing D *observes* ev with D' *does* ev.

Rationale Expressions in R on phenomena other than events are untouched by this rule, and remain the same in R'. Hence, any resulting constrains are the same in the two requirement statements, which are then satisfied under the same conditions. We can reason about constaints on events in terms of behaviours, i.e., sequences of event occurrences. Assume B is the set of permissible behaviours under R, then B includes all permissible occurrences of ev. These are the same regardless of whether they are considered controlled or observed by a domain. Therefore B is also the set of permissible behaviour under R'. It follows that R is satisfied whenever R' is satisfied.

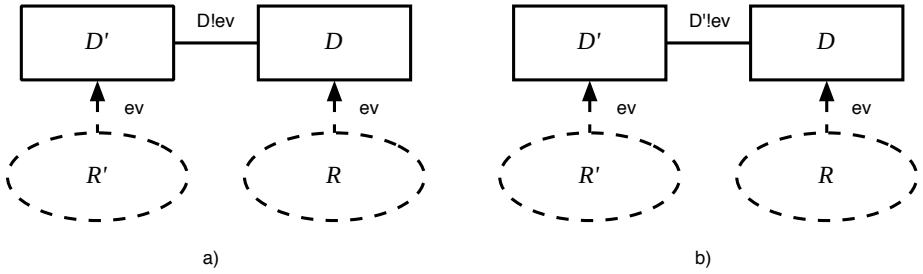


Figure 6: Changing viewpoint

By repeated applications of this rule to our example, from the **Purchase** requirement statement (expressed from the Customer's viewpoint) we derive the following statement (from the Cashier's viewpoint):

Sale The Cashier

- having observed: a number of *present(item)* followed by one *present(payment)*,
- if *payment* is for the correct amount, should do: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

3.7.2 Reducing through causes and effects

Our second rule generates a new requirement statement by replacing effects with causes, or causes with effects, according to the causal relations existing among events in domain descriptions. Once again, there are two cases. The first case corresponds to a situation in which a requirement expression regarding an event ev is replaced with an expression regarding the event ev' of which ev is the cause. The second case corresponds to a situation in which a requirement expression regarding an event ev is replaced with an expression regarding the event ev' which is the cause of ev.

The rule is as follows:

R2: Reducing through causes and effects

a) Consider domains D, D' and D'' that share events ev and ev' (see Figure 7.a), and requirement statement R, containing D' *observes* ev. Assume $ev \rightsquigarrow ev'$. Then a requirement statement R', can be derived from R by replacing D' *observes* ev with D' *does* ev'.

b) Consider domains D, D' and D'' that share events ev and ev' (see Figure 7.b), and requirement statement R, containing D' *does* ev. Assume $ev' \rightsquigarrow ev$. Then a requirement statement R', can be derived from R by replacing D' *does* ev with D' *observes* ev'.

Assumption There is a one-to-one correspondence between causes and effects.

Rationale Expressions in R on phenomena other than events are untouched by this rule, and remain the same in R'. Hence, any resulting constrains are the same in the two requirement statements, which are then satisfied under the same conditions. We can reason about constaints on events in terms of behaviours, i.e., sequences of event occurrences. Assume B is the set of permissible behaviours under R, then B includes all permissible occurrences of ev. In the first case of the rule, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only effect of ev. In the second case, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only cause of ev. Because of the one-to-one correspondence between causes and effects we have assumed, there exists a one-to-one correspondence between B and B', hence R is satisfied whenever R' is satisfied under such a correspondence.

Time concern This rule can be applied safely only under the assumption that the time elapsed between the occurrence of an event and that of its effect is negligible in the problem under consideration, i.e., event occurrence orderings in behaviours is not affected.

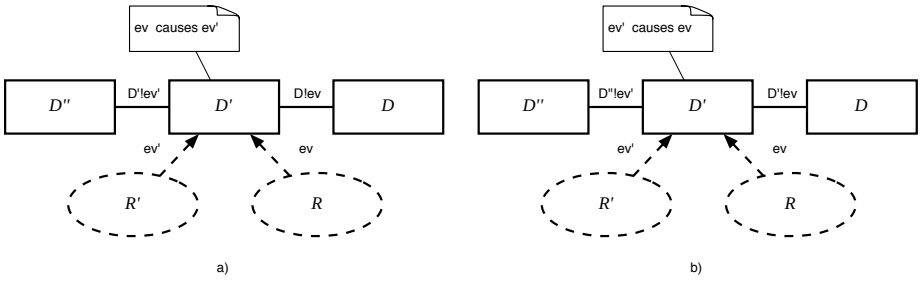


Figure 7: Reducing through causes and effects

By applying this rule repeatedly to the events in the Sale requirement statement we can derive the following statement:

Sale' The Cashier

- having done: a number of *enter(item.info)* followed by one *enter(payment.info)*,
- if *payment* is for the correct amount, should observe: *print(receipt.info)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

In the statement, the condition on the entities has remained unchanged, while events' causes and effects have been swapped in a move towards a specification - i.e., a requirement statement closer to the machine. Note how, in this example, the time elapsed between causes and effects has been disregarded as not of relevance. Later on, we will discuss an example when this is not the case.

The purchase example is simple enough that a repeated application of the two rules we have introduced is sufficient to derive a specification. A succession of derived statements, illustrated in Figure 8, is as follows:

Transfer The POS h/w

- having observed: a number of *enter(item.info)* followed by one *enter(payment.info)*,
- if *payment* is for the correct amount, should do: *print(receipt.info)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

Transfer' The POS h/w

- having done: a number of *transfer(item.info)* followed by one *transfer(payment.info)*,
- if *payment* is for the correct amount, should observe: *generate(receipt.info)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

Process The Controller

- having observed: a number of *transfer(item.info)* followed by one *transfer(payment.info)*,
- if *payment* is for the correct amount, should do: *generate(receipt.info)*

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

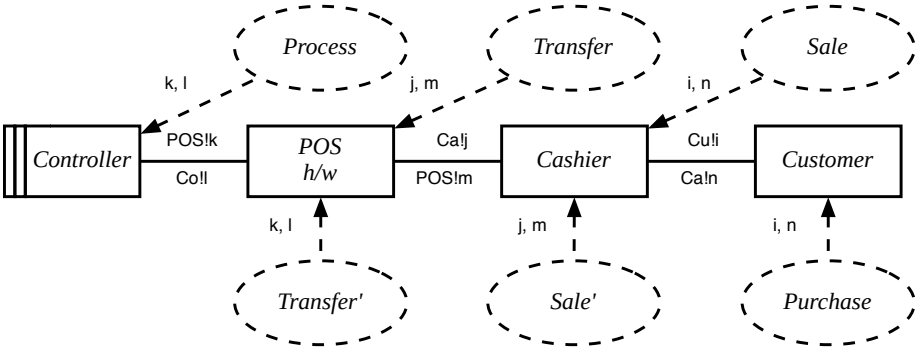


Figure 8: Reduction of requirement to specification

3.7.3 Dealing with internal events

Our example is very simple. In particular, it assumes that all events are shared. Here we consider a rule which deals with internal events.

Assume, for argument sake, that in our example we can access the Customer's thought processes and identify the internal events *intentionToBuy(item)* and *intentionToMake(payment)* with the following causal relations:

intentionToBuy(item) $\leadsto$ *present(item)*

intentionToMake(payment) $\leadsto$ *present(payment)*

and that the requirement is expressed as:

Purchase' The Customer

- having done: a number of *intentionToBuy(item)* followed by one *intentionToMake(payment)*
- if *payment* is for the correct amount, should observe: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

In this case, neither of our previous rules apply, and we need to find a way to express the requirement in terms of shared events. Our next rule has to do with generating new requirement statements by replacing effects with causes, or causes with effects, when some of the events involved are internal. The rule is a variant of R2. Once again, there are two cases. The first case corresponds to a situation in which a requirement expression regarding an internal event ev is replaced with an expression regarding the shared event ev' of which ev is the cause. The second case corresponds to a situation in which a requirement expression regarding an internal event ev is replaced with an expression regarding the shared event ev' which is the cause of ev.

The rule is as follows:

R3: Reducing through causes and effects, in the presence of internal events

a) Consider domains D and D' that share events ev and ev' (see Figure 9.a), and requirement statement R, containing D *does* ev. Assume $ev \leadsto ev'$. Then a requirement statement R', can be derived from R by replacing D *does* ev with D *does* ev'.

b) Consider domains D and D' that share events ev and ev' (see Figure 9.b), and requirement statement R, containing D *does* ev. Assume $ev' \leadsto ev$. Then a requirement statement R', can be derived from R by replacing D *does* ev with D *observes* ev'.

Assumption There is a one-to-one correspondence between causes and effects.

Rationale Expressions in R on phenomena other than events are untouched by this rule, and remain the same in R'. Hence, any resulting constrains are the same in the two requirement statements, which are then satisfied under the same conditions. We can reason about constaints on events in terms of behaviours, i.e., sequences of event occurrences. Assume B is the set of permissible behaviours under R, then B includes all permissible occurrences of ev. In the first case of the rule, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only effect of ev. In the

second case, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only cause of ev. Because of the one-to-one correspondence between causes and effects we have assumed, there exists a one-to-one correspondence between B and B', hence R is satisfied whenever R' is satisfied under such a correspondence.

Time concern This rule can be applied safely only under the assumption that the time elapsed between the occurrence of an event and that of its effect is negligible in the problem under consideration, i.e., event occurrence orderings in behaviours is not affected.

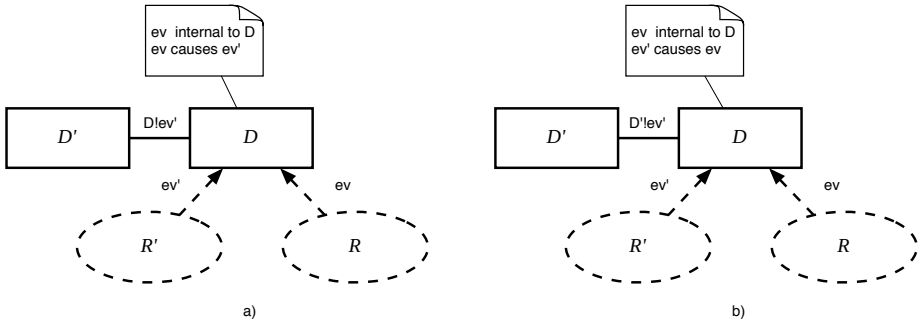


Figure 9: Reducing through causes and effects, in the presence of internal events

The application of this rule to **Purchase'** will result in the **Purchase** statement (see previous section).

3.7.4 Transitive closure

The last rule we introduce addresses the situation in which there exist a causality chain among internal events of a domain.

The rule is as follows:

R4: Transitive closure

a) Consider domain D with internal events ev and ev' (see Figure 10.a), and requirement statement R, containing D *does* ev. Assume $ev \rightsquigarrow^{+} ev'$. Then a requirement statement R', can be derived from R by replacing D *does* ev with D *does* ev'.

b) Consider domains D with internal events ev and ev' (see Figure 10.b), and requirement statement R, containing D *does* ev. Assume $ev' \rightsquigarrow^{+} ev$. Then a requirement statement R', can be derived from R by replacing D *does* ev with D *observes* ev'.

Assumption There is a one-to-one correspondence between causes and effects.

Rationale Expressions in R on phenomena other than events are untouched by this rule, and remain the same in R'. Hence, any resulting constrains are the same in the two requirement statements, which are then satisfied under the same conditions. We can reason about constaints on events in terms of behaviours, i.e., sequences of event occurrences. Assume B is the set of permissible behaviours

under R, then B includes all permissible occurrences of ev. In the first case of the rule, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only eventual effect of ev. In the second case, a set B' of permissible behaviours under R' is derived by replacing each ev occurrence by an ev' occurrence, ev' being the only eventual cause of ev. Because of the one-to-one correspondence between causes and effects we have assumed, there exists a one-to-one correspondence between B and B', hence R is satisfied whenever R' is satisfied under such a correspondence.

Time concern This rule can be applied safely only under the assumption that the time elapsed between the occurrence of an event and that of its effect, for all pair of events involved, is negligible in the problem under consideration, i.e., event occurrence orderings in behaviours is not affected.

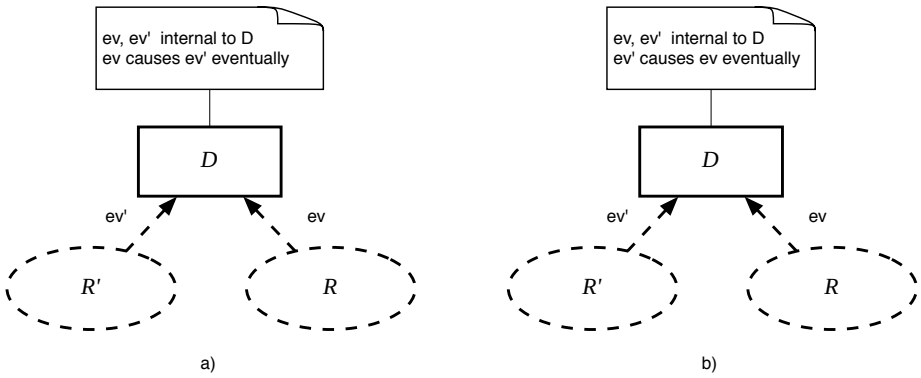


Figure 10: Transitive closure

3.8 Activity: Formulating an adequacy argument

In order to complete the process of problem analysis we need to argue the adequacy of the derived specification in the context of the original problem. Namely, we need to argue that Process satisfies Purchase, given domains POS h/w, Cashier and Customer, and their properties. The fact that Process was derived from Purchase through rule applications, means that Process satisfies Purchase *by construction*. This is because each rule guarantees that if R' is derived from R in a certain context under rule application, thenR' is satisfied whenever R is satisfied in that context. Moreover, the sequence of rule applications provides a means to trace the specification back to the original requirement, with each rule's rationale supporting rich traceability between successive statements.

4 Discussion

4.1 When time is a concern

Many of the rules in the previous section come with a warning they can be safely applied only if the time elapsed between causes and effects can be abstracted away without any harm, that is without inadvertently excluding situations in which a time

delay may cause the requirement not to be satisfied. To understand why this may be the case, here is another example. The problem is to build an automatic mop that cleans up the carpet when Pavlov's dog [28] drools on it. The problem is depicted in Figure 11. In the figure, event *rings* is shared between three domains: the bell, which controls it, and the dog and mop, which observes it. The requirement is:

Clean up Mop should do *cleans* after Dog *drools*.

and Pavlov's dog's behaviour, due to conditioned reflex, is captured by $rings \rightsquigarrow drools$.

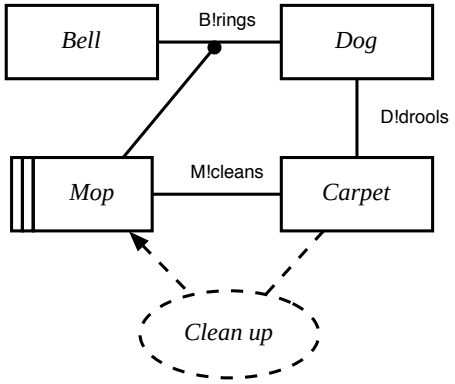


Figure 11: Cleaning up after Pavlov's dog

By applying rule R2, we can derive the specification:

Clean up' Mop should do *cleans* after Bell *rings*.

However, satisfaction of this specification does not guarantees that the original requirement is satisfied in all cases. For instance, suppose the Dog takes one full minute to drool after hearing the bell, whereas the Mop reacts instantly and cleans up the carpet before it becomes necessary.

To summarise, the time concern is a reminder that when dealing with causal relations between events, time is an important consideration. In cases in which it cannot be disregarded in the analysis of a problem, it is likely to give rise to further requirements (say, that Mop should wait 15 minutes after the bell rings and before cleaning up) or further explicit assumptions on how the environment behaves (say, Dog drools immediately). Note that such conditions can easily be added to the requirements by analsyis of the domain's behaviour, hence making the rule applicable again.

4.2 Conditional behaviour

In the treatment of causal behaviour in this paper, we have assumed that simple causal relationships exist between events. For instance, that whenever a Cashier shares a *present(payment)*, this will always cause the Cashier to perform an *enter(payment.info)*. In general, in real-world problems, other conditions may affect a domain's behaviour for which a more sophisticated treatment of causality may be required. For instance, in our example, let us assume that both credit card and cash payments may be acceptable to complete a sale. Let us also assume that the Cashier is supposed to check that any

cash payment does indeed correspond to the amount which is due, while credit card payments are dealt with entirely by the Controller. Then, on sharing a *present(payment)* the Cashier will:

- if the payment is by cash, do a *checkCash()*, and if amount is correct, an *enter(payment.info)*;
- if the payment is by credit card, do an *enter(payment.info)*.

This could be captured by the following relation, where we adopt the convention that conditions appear in square brackets[6] and the relation only holds if the conditions are satisfied:

present(payment) $\rightsquigarrow$ [*payment* is cash] *checkCash()*
checkCash() $\rightsquigarrow$ [*payment* is correct] *enter(payment.info)*
present(payment) $\rightsquigarrow$ [*payment* is by credit card] *enter(payment.info)*

The rewriting of the requirement under conditional behaviour is not as straightforward as in the rules we presented earlier. In this case, we need to separate the two cases of credit card and cash payments as checking that a payment is for the correct amount has a different meaning in each of them. In particular, we expect the controller to check the correctness of credit card payments, but not of cash payments, a responsibility which remains with the Cashier. Therefore, rather than a simple rewriting of the requirement we are faced with the two distinct sub-problems of Figure 12, whose requirements are as follows:

Purchase by cc The Customer

- having done: a number of *present(item)* followed by *present(payment)*, where *payment* is by credit card,
- if *payment* is for the correct amount, should observe: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

Purchase by cash The Customer

- having done: a number of *present(item)* followed by one *present(payment)*, where *payment* is by cash,
- if *payment* is for the correct amount, should observe: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

The progression of the 'Purchase by cc' problems is then analogous of our progression in the previous section. The 'Purchase by cash' problem, instead, deserves further analysis as we discuss in the following section.

4.3 Factoring in domain behaviour

In this section, we are going to discuss ways in which domain properties influence problem reduction and requirement transformation. Once again, let us look at our example and consider the 'Purchase by cash' sub-problem. By applying R2 repeatedly we obtain:

[6]This notation is reminiscent of guards on statemachines [26].

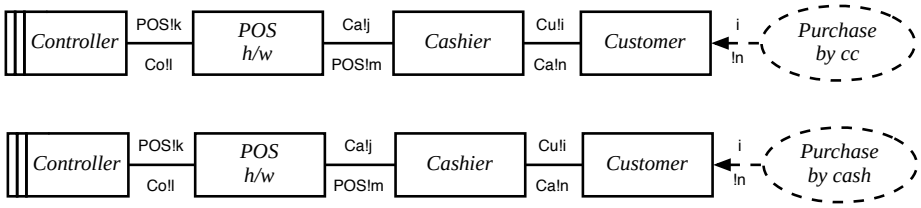


Figure 12: Two sub-problems

Sale by cash The Cashier

- having observed: a number of *present(item)* followed by one *present(payment)*, where *payment* is by cash,
- if *payment* is for the correct amount, should do: *present(receipt)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

As discussed previously, checking the correctness of a cash payment is the responsibility of the Cashier, as reflected in the conditional causal relation defined to capture the Cashier's behaviour. From such a behaviour, it follows that only correct cash payments will give rise to *enter(payment.info)* occurrences. This fact needs to be taken into account in the rewriting of the requirement, when R2 is applied. The new requirement should look something like the following:

Sale by cash' The Cashier

- having done: a number of *enter(item.info)* followed by one *enter(payment.info)*, where *payment* is by cash,
- **assuming** the *payment* is for the correct amount, should observe: *print(receipt.info)*,

where *receipt.info* should correspond to *item.info*, for all the items, and *payment.info*.

In other words, that the payment is for the correct amount is now a working assumption, rather than a condition to be enforced; hence the POS h/w, and ultimately the Controller, will operate under such an assumption from this point forward.

4.4 Progression and decomposition of problems

The discussion in Section 4.2 hinted that there exists a relation between progressing and decomposing [15] problems, in the sense that the former may highlight the need for the latter. In this section we revisit such a relation by paying particular attention to the parts of a problem description which may become the subject of design. Let us consider, once again, our example and the 'Purchase' requirement. As discussed previously, when both cash and credit card payments are considered, checking the correctness of a payment is a joint responsibility of the Cashier and the Controller. This may lead to consider both domains as part of a socio-technical [24] solution to the problem, as represented in Figure 13. Here, the notation of [9] applies, indicating that both Controller and Cashier are the subject of design (marked by double vertical lines) as

part of a socio-technical solution (bounded by the dotted line). In other words, both the Controller and Cashier are the subject of optative descriptions, i.e., properties that we would like to be true of their behaviour, to be brought about by the process of design: say, by construction, in the case of the Controller, and by training in the case of the Cashier.

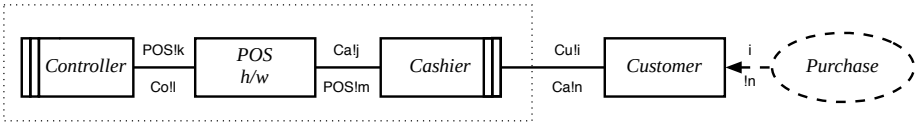


Figure 13: A socio-technical problem

The consideration of such a problem would give rise to the sub-problems decomposition of Figure 14 in which correctness of payment and payment processing are seen as separate concerns.

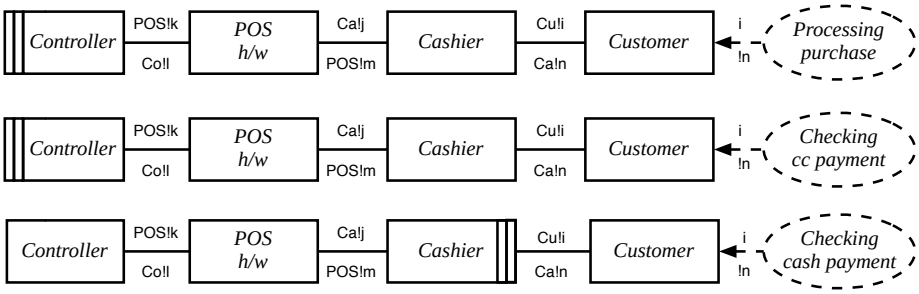


Figure 14: Alternative decomposition

The progression of each such sub-problem can then be obtained through the application of the reduction rules presented in this paper. That both technical and social components should be considered as part of a solution, although a departure from classical problem frames [15], in which only software problems are in scope, is necessary in considering complex real-world problems and their requirements. This is indeed recognised by other approaches to requirements engineering, primarily, goal-oriented ones [35, 33], in which during the process of analysis, responsibilities are assigned both to technical and human 'agents'. Those approaches, however, fail to distinguish indicative and optative properties, i.e, the distinction between the assumptions that can be made of the environment, and properties that have to be brought about by the design process, and the consequent reasoning that this separation affords. What are the consequences, for instance, of assuming the Cashier will always check the correctness of all cash payments, from identifying this issue as an essential aspect of a Cashier's training?

5 Conclusion

The paper has presented an approach to progressing problems expressed as problem diagrams in the context of requirements analysis. The essence of the approach is the ability to reduce requirements to specifications through the systematic application of

a small set of rules. The approach is predicated on a problem-based view of requirements, with the separation of a solution from its context, the definition and designation of relevant phenomena, and the identification of control and causality relations. The paper has focussed on the systematic reduction of simple forms of problems. In particular, formal rules have been proposed only to one-to-one causal relations with no conditionals. The consideration of conditional causality and the discharge of part of the requirement through properties of the problem context were also discussed to a certain extent, together with their relation to problem progression and decomposition. An systematic treatment of these cases, however, is beyond the scope of this paper, and the subject of ongoing investigation.

6 Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants, and the EPSRC, Grant number EP/C007719/1. Thanks also go to Michael Jackson and our colleagues in the Centre for Research in Computing in The Open University, particularly Michael Jackson and Bashar Nuseibeh.

References

[1] I. Alexander and N. Maiden, editors. *Scenarios, stories, use cases through the systems development life-cycle*. Wiley, 2004.

[2] S. Barker. *The Elements of Logic*. McGraw-Hill, 1989.

[3] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information Systems*, 27(6):365–389, 2002.

[4] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[5] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice-Hall., 1994.

[6] K. Cox, J. G. Hall, and L. Rapanotti. Editorial: A roadmap of problem frames research. *Information Science and Technology*, 2005.

[7] O. Gotel and A. Finklestein. An analysis of the requirements traceability problem. In *First International Conference on Requirements Engineering*, pages 94–101. IEEE Computer Society Press, 1994.

[8] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[9] J. G. Hall and L. Rapanotti. Problem Frames for Socio-Technical Systems. In J. Mate and A. Silva, editors, *Requirements Engineering for Socio-Technical Systems*. Idea Group, Inc., 2004.

[10] J. G. Hall and L. Rapanotti. A framework for software problem analysis. Technical Report 2005/05, Department of Computing, April 2005.

[11] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, 2001.

[12] M. Jackson. *Problem Frames*. Addison-Wesley, 2001.

[13] M. A. Jackson. The world and the machine (keynote). In *17th Int. Conf. on Software Engineering (ICSE'95)*. IEEE/ACM, 1995.

[14] M. A. Jackson. Problem analysis using small problem frames. *South African Computer Journal 22: Special Issue on WOFACS98*, pages 47–60, 1998.

[15] M. A. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problem*. Addison-Wesley Publishing Company, 1st edition, 2001.

[16] M. A. Jackson. Some basic tenets of description. *Software and Systems Modeling*, 1(1):59, 2002.

[17] M. A. Jackson and P. Zave. Deriving specifications from requirements: An example. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*. ACM Press, 1995.

[18] W. Johnson. Overview of the knowledge-based specification assistant. In *Proceedings of the second Knowledge-Based Software Assistant Conference*.

[19] W. Johnson. Deriving specifications from requirements. In *Proceedings of ICSE-10*, pages 428–438. IEEE CS Press, 1988.

[20] R. Laney, L. Barroca, M. A. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of the 12th Int. Conf. on Requirements Engineering (RE'04)*, pages 113–122, Kyoto, Japan, 2004. IEEE Computer Society Press.

[21] C. Larman. *Applying UML and patters*. Prentice Hall, 2nd edition, 2002.

[22] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. *ACM SIGSOFT Software Engineering Notes*, 27(6), 2002.

[23] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[24] J. L. Mate and A. Silva, editors. *Requirements Engineering for Socio-Technical Systems*. Information Science Publishing, 2005.

[25] J. D. Moffett, J. G. Hall, A. Coombes, and J. A. McDermid. A Model for a Causal Logic for Requirements Engineering. *Journal of Requirements Engineering*, 1(1):27–46, 1996.

[26] OMG. Unified Modeling Language (UML), version 2.0. http://www.omg.org/technology/documents/formal/uml.htm.

[27] J. Palmer. Traceability. In R. Thayer and M. Dorfman, editors, *Software Requirements Engineering*. 1997.

[28] I. Pavlov. Pavlov' dog. http://evolution.massey.ac.nz/assign2/KD/finalpavlov.html.

[29] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1), 2001.

[30] L. Rapanotti, J. G. Hall, M. A. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*, pages 73–82, Kyoto, Japan, 2004. IEEE.

[31] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison Wesley, Harlow, England., 1999.

[32] A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *International Conference on Software Engineering*, pages 5–19, 2000.

[33] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. on Sofware. Engineering, Special Issue on Inconsistency Management in Software Development*, 1998.

[34] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

[35] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of RE97: 3rd International Symposium on Requirements Engineering*, pages 226–235, 1997.

[36] E. S. Yu. Modeling organizations for information systems requirements engineering. In *Proceedings 1st IEEE International Symposium on Requirements Engineering*, pages 34–41, 1993.

[37] P. Zave and M. A. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.