On the construction of specifications from requirements

Zhi Li[1,2], Jon G. Hall[3], and Lucia Rapanotti[3]

[1] College of Computer Science and Information Technology, Guangxi Normal University, No.15 Yu Cai Road, Guilin, Guangxi 541004, P. R. China

[2] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, 100871, CHINA

E-mail:zhili@mailbox.gxnu.edu.cn

[3] Centre for Research in Computing, The Open University, Walton Hall, Milton Keynes, Buckinghamshire, MK7 6AA, UK

E-mail:{J.G.Hall,L.Rapanotti}@open.ac.uk

Abstract. Transforming real-world requirements into specifications which are appropriate for subsequent software development is at the heart of Requirements Engineering. Doing it systematically remains an open challenge. In this paper we present a formal approach to systematise the move from requirements to specifications in the context of Jackson's Problem Frames.

Keywords: Problem Frames, Problem Progression, Requirements, Specifications, Communicating Sequential Processes

1 Introduction

One of the contributions of Jackson [1] is a recognition of the distinction between requirements and specifications: requirements pertain to phenomena deeply embedded in the real-world and which are meaningful to stake-holders; specifications relate to phenomena at the interface with the machine. Capturing real-world requirements and deriving appropriate specifications is then at the heart of RE.

Jackson ([2, Page 103]) suggests Problem Progression as a way of moving from requirements to specifications, but gives no technical detail, perhaps because of the lack of a suitable formal underpinning for the Problem Frame framework. Hall, Rapanotti and Jackson started that formalisation in [3]. In this paper, based on the doctoral work of Li ([4], [5]), we describe a problem progression-like construction of specifications from requirements.

This paper is structured as follows: Section 2 recalls Problem Frames and problem progression; in Section 3, we show how to model a problem diagram in CSP, and show how the resultant is related to Lai's Quotient operator; Section 4 illustrates the approach on an example. Section 5 contextualises the work in the literature and Section 6 concludes.

2 Problem Frames and Problem Progression

This work is located within Jackson's Problem Frames framework (PF) [2]. It proposes a formal CSP encoding of Jackson's problem diagrams to which the Lai's Quotient operator can be applied to progress requirements to specifications.

2.1 Problem Diagrams and their meaning

In Problem Frames [2] a software related problem, shortly *problem*, is viewed as a requirement in a real-world context for which a software solution is sought. Problem descriptions are captured and expressed by problem diagrams, which place the computing machine upon which code satisfying the specification will run, the problem (real-world) context, the requirement and the phenomena that relate them in representative juxtaposition.

Briefly, phenomena are at the basis of a problem's descriptions, they being the observable mechanism by which one domain affects others. Descriptions are built, through the cause and effect relationships of phenomena one to another, into domains which are thus sets of related phenomena that are usefully treated as a unit, identifiable within the real-world.

The problem context, as a collection of domains, is converted to be a problem through the addition of a requirement: a constraint on the real-world relationships that are desired through the development of the software. The requirement for a problem provide the real-world effects that should be observed if correctness of the solution specification is to be verified.

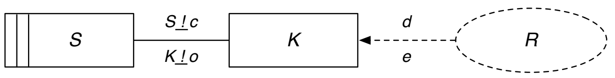


Fig. 1. A generic problem diagram, K may be arbitrarily complex

For illustration of a problem diagram, see Figure 1 in which K is the problem context (with only one domain in this case), R is the requirement and S is the software solution specification. K and S interact through phenomena in set c controlled[4] by S ($S \underline{!}\, c$) and observed by K, and set o controlled by K ($K \underline{!}\, o$) and observed by S. The requirement, R, constrains phenomena set d and refers to set e[5].

According to [3] a problem diagram has, as its formal semantics, the collection of specifications that control those phenomena exposed for control by its environment, that observe all other exposed phenomena and that satisfy the Requirement, i.e.,

$$c, o : [K, R] = \{S : \mathit{Specification} \mid S\ \mathit{controls}\ c \ \wedge\ S\ \mathit{observes}\ o \ \wedge\ K, S\ \mathsf{solves}\ R\}$$

where solves indicates satisfaction[6]. For ease of reference, we will refer to this as the HRJ-semantics. Finding an element of the HRJ-semantics for any particular problem we call the *challenge* for that problem. (For how such challenges may be met in general see, for instance, [8, 9].)

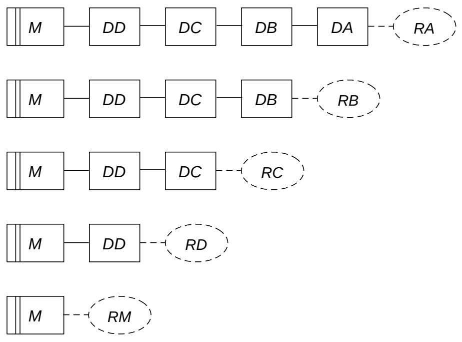
[4] We use $\underline{!}$ instead of ! to distinguish it from the CSP's ! used later on in the paper.

[5] Although, as here, the distinction between constraint and reference is not always distinguishable within PFs.

[6] related to Zave & Jackson's notion of correctness [6], as generalised in Hall and Rapanotti's Problem Oriented Engineering (POE, [7]).

2.2 Problem Progression

The idea of problem progression was briefly introduced in [2], and is illustrated in the following figure adapted from that work. In the words of Jackson:

> “You can think of any problem [expressed in PF] as being somewhere on a progression towards the machine, like this:
>
> 
>
>
> 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 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.”

In the above figure, the solution to each of the problems is represented by the same machine *M* since the problems are transformed in a solution-preserving way: the solution to the transformed problem satisfies the original problem. To see the importance of problem progression, we note that the HRJ-semantics is not constructive: it does not tell us how to determine an element of the collection which constitutes the formal semantics of a problem diagram. We note, however, that if $c, o : [K, R]$ is the semantics of the initial problem at the top of a progression, say $P_{initial}$, and $c', o' : [K', R']$ is that of the progressed problem at the bottom, say $P_{progressed}$, then the challenge for $P_{progressed}$ can be met by applying progression to any element of the semantics of $P_{initial}$.

3 Modelling and Progressing Problem Diagrams with CSP

Problem Frames do not presuppose any one description language for problem diagrams. In this section, we argue that we can use CSP as one such description language. The advantage is that once such a CSP description exists, then we can apply the Lai’s Quotient operator.

3.1 Modelling a Domain as a CSP Process

We argue that there are similarities (in fact, a close match) between Jackson's notion of a domain in Problem Frames and the notion of a process in CSP: they are both self-contained entities that interact with other domains (processes) through shared phenomena (alphabet). Based on this observation, the formalisation becomes then quite straightforward: a domain *D* in Problem Frames is seen a process D, with its set of shared phenomena as the alphabet αD. Individually, a single shared phenomenon (including an instance of shared event, state or role) of domain *D* is formalised as a single external event ev of process D. Note this does not prevent *D* having "internal" phenomena, only that such phenomena should be hidden from its environment through event hiding.

The notion of parallel composition in CSP was introduced to investigate the behaviour of a complete system composed of subsystems that act and interact with each other as they evolve concurrently. For example, when we analyse the combined behaviour of two processes put together, their interactions (if they exist) can be regarded as events that require simultaneous participation of both processes involved. Hoare [10] argues that we can assume that the alphabets of the two processes are the same when analysing their overall behaviour. He uses the notation $P \mid\mid Q$ to denote the process that behaves like the composition of processes P and Q interacting in lock-step synchronisation.

In Problem Frames, the interactions between domains and solution have similar characteristics: each phenomenon they share is considered instantaneous, and both domains are simultaneously engaged in the same phenomenon [2]. To link Problem Frames and CSP, we note that the context K (described in CSP) and solution S (also described in CSP) stand in juxtaposition as $K \mid\mid S$. That it satisfies the requirement R can then be translated into CSP as $K \mid\mid S$ **sat** R[7].

3.2 Interpreting Problem Progression in terms of Lai's Quotient Operator

To meet the challenge of finding a CSP process specification S such that $K \mid\mid S$ **sat** R, we need a new operator that can perform the opposite calculation of parallel composition. Lai and Sanders [11] extend Hoare and He's notion of "weak inverse" of sequential composition to parallel composition and they have given the *weakest environment* calculus to provide the weakest process X that placed in parallel with an established subcomponent P satisfies their overall specification R:

$$X \mid\mid P \textbf{ sat } R \Leftrightarrow X \textbf{ sat } P \setminus\!\!\setminus R$$

$P \setminus\!\!\setminus R$ is called the weakest environment of a process. Lai and Sanders [11] provide a closed predicate definition for the weakest environment: given specifications P, R and a chosen set $A \subseteq \alpha P$, the weakest environment of P in R, denoted $P \setminus\!\!\setminus R$ with

[7] That **sat** is subsumed in the POE notion of of satisfaction is left as an exercise for the reader.

alphabet $\alpha R \setminus \alpha P \cup A$ as the specification:

$$\begin{aligned} P \mathbin{\backslash\!\!\backslash} R(tr, ref) \triangleq \forall ur : traces(R)\ \forall\ rep \subseteq \alpha P \\ \bullet\ [tr = ur \upharpoonright \alpha\,(P \mathbin{\backslash\!\!\backslash} R) \\ \wedge\ P(ur \upharpoonright \alpha P, rep) \\ \Rightarrow R(ur, rep \cup ref)] \end{aligned}$$

For us, the importance of Lai's Quotient operator is that it provides a (in some sense) canonical solution to a problem challenge, at least when domains are described in the CSP family of notations. Now the HRJ-semantics (at least in the simple case when $c \cup o = d \cup e$) becomes:

$$c, o : [K, R] = \{S : Specification \mid S! = c\ \wedge\ S? = o\ \wedge\ S\ \mathbf{sat}\ K \mathbin{\backslash\!\!\backslash} R\}.$$

Figure 2 illustrates the role of Lai's Quotient operator in problem progression.

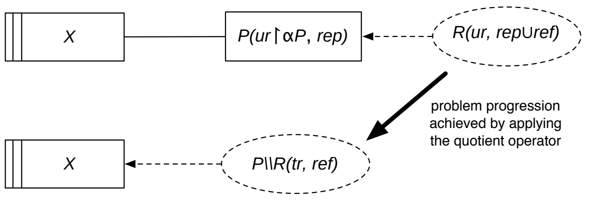

X
P(ur↾αP, rep)
R(ur, rep∪ref)
problem progression achieved by applying the quotient operator
X
P\\R(tr, ref)

Fig. 2. A generic problem progression illustrating the application of Lai's Quotient operator

4 Example

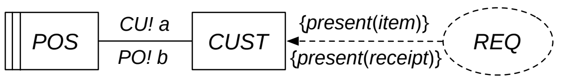
Our illustrative example is a Point-of-Sale (POS) system which allows customers to scan and pay for their shopping without any intervention from supermarket staff. Here is the problem description:

A new point-of-sale (POS) system is needed to process sales for a supermarket shop in the UK. The POS includes both the desired software and some hardware purchased from a third party, including a barcode reader, a cash acceptor and dispenser handler, a touch-screen display, and a receipt printer, etc. The problem is that customers should pay for and receive a receipt for the correct amount on presentation of items to the POS system.

The problem diagram for the POS is illustrated in Figure 3.

Table 1 shows the identified domains and their CSP descriptions. Since we are in the realm of CSP, descriptions for the diagram are written in CSP, followed by a simple narrative for those not familiar with that language.

For brevity of presentation, we use $item$, $notice$, pay, $change$, and $receipt$ as a short form of events $present(item)$, $present(notice)$, $present(payment)$, $present(change)$,



a: {*present(item)*, *present(payment)*}

b: {*present(notice)*}, {*present(change)*}, {*present(receipt)*}

Fig. 3. Point-of-sale: problem diagram

Name	Description
CUST	$CUST = \sqcap_{i \in \{1,\ldots,100\}} item!i \rightarrow notice?i \rightarrow PAY$, where $PAY = \sqcap_{p \in \{1,2,5,10\}} pay!p \rightarrow (change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST} \;\Box\; notice?\,n \rightarrow PAY)$. *CUST* is a customer who wants to buy an item from the shop. (1). He presents the item he wants (whose price is *i* pence, ranging from 1 to 100) to the *POS* via the bar code scanner. (2). after receiving a notice *n* from the *POS*, he presents, perhaps, part payment in cash *p* pence, a coin of value 1p, 2p, 5p or 10p to the *POS* (e.g., through a cash acceptor). (3). If the payment is sufficient, i.e., *i* ≤ *p*, then the customer will be given the change *c* (e.g., via the dispenser handler), followed by a receipt for *r* = *i* as a proof of purchase (e.g., a printout from the receipt printer); (4). if the payment is insufficient, i.e., *p* < *i* , then further notices displaying the remaining amount of payment are issued to the customer until sufficient payment is presented, after which the customer will be given the change and a receipt. (5). *i,n,p,c,r* are assumed to be in natural numbers, i.e., *i,n,p,c,r* ∈ *N*, and the above payment method is in cash for a single item, and that *i*, *n*, *p*, *c*, *r* are expressed in pence in British money.
POS	*POS=?* The software specifications to be calculated.

Table 1. Domains and their descriptions

and $present(receipt)$ in Figure 3, respectively. These events are also the shared phenomena between domains in Figure 3, whose designations are explained in natural language in Table 2 .

The requirement is informally described as: "*customers should pay for and receive a receipt for the correct amount on presentation of items to the POS system*". According to this statement, requirement REQ only constrains two events: whenever event $item.i$ happens, eventually event $receipt.r$ should happen, and the value of r should be equal to that of i, i.e., $r = i$. Therefore,

$$REQ = \sqcap_{i \in \{1,\ldots,100\}} item.i \rightarrow receipt.i \rightarrow STOP_{\{item, receipt\}}.$$

Note we use $item.i$ and $receipt.i$ to represent that both $CUST$ and POS participate in this event: from $CUST$'s perspective, the event should be denoted as $item!i$, and from POS's perspective, the same event should be denoted as $item?i$, therefore expression $receipt.i$ includes both perspectives of $CUST$ and POS.

The above process expression is not detailed enough for us to construct POS because it does not prescribe all of the interaction behaviours between $CUST$ and POS, i. e., events $notice$, pay and $change$ do not appear in REQ's alphabet. For the problem diagram in Figure 3 we need to find a process POS such that

Name	Designation
present(item)	The event in which the customer presents an item of product s(he) wants to buy to the *POS* system. This event is initiated and controlled by the customer *CUST* domain, thus represented by *CU!* that proceeds it.
present(payment)	The event in which the customer presents the payment for the purchased item to the *POS* system. This event is initiated and controlled by the customer *CUST* domain, thus represented by *CU!* that proceeds it.
present(notice)	The event in which the *POS* system presents a notice to the customer. This event is initiated and controlled by the *POS* domain, thus represented by *PO!* that proceeds it.
present(change)	The event in which the *POS* system presents the change due to the customer. This event is initiated and controlled by the *POS* domain, thus represented by *PO!* that proceeds it.
present(receipt)	The event in which the *POS* system presents a receipt to the customer. This event is initiated and controlled by the *POS* domain, thus represented by *PO!* that proceeds it.

Table 2. Shared phenomena and their designations

$(POS \mid\mid CUST) \setminus [\{item, notice, pay, change, receipt\} \setminus \{item, receipt\}]$ **sat** REQ, and the solution set for the problem diagram is:

$$\begin{aligned}
&\{notice, change, receipt\}, \{item, pay\} : [CUST, REQ] \\
&= \{POS : Specification | POS! = \{notice, change, receipt\} \wedge POS? = \{item, pay\} \\
&\wedge (POS \mid\mid CUST) \setminus \{notice, pay, change\} \ \textbf{sat}\ REQ\}.
\end{aligned}$$

Notice that the problem is to find a POS to satify the above formula. However, Lai's quotient can not directly allow us to calculate POS. As do Lai and Sanders [11], we therefore introduce the above missing events into a more detailed requirement statement which we call $REQC$.

We construct $REQC$ in a way that relates to $CUST$'s behaviour, meanwhile still satisfying REQ after hiding events $notice$, pay and $change$. Based on our work in [5], we can now construct $REQC$[8]:

```
$REQC = \sqcap_{i \in \{1,\dots,100\}} item.i \rightarrow notice.i \rightarrow REQCPAY(i, i)$, where
 $REQCPAY(i, remain) =$
 $\sqcap_{p \in \{1,2,5,10\}} pay.p \rightarrow$
            $if\ p < remain$
            $then\ (notice.(remain - p) \rightarrow REQCPAY(i, remain - p))$
            $else\ (change.(p - remain)$
            $\rightarrow receipt.i \rightarrow STOP_{\alpha REQC})$
```

Applying the hiding operator $\setminus$ to $REQC$, we get

$$\begin{aligned}
&REQC \setminus \{notice, pay, change\} \\
&= (\sqcap_{i \in \{1,\dots,100\}} item.i \rightarrow notice.i \rightarrow REQCPAY(i, i)) \setminus \{notice, pay, change\} \\
&= \sqcap_{i \in \{1,\dots,100\}} item.i \rightarrow (REQCPAY(i, i) \setminus \{notice, pay, change\}) \\
&= \sqcap_{i \in \{1,\dots,100\}} item.i \rightarrow receipt.i \rightarrow STOP_{\{item, receipt\}} \\
&\textbf{sat}\ REQ.
\end{aligned}$$

[8] $REQC$ is constructed from an abstract $REQC_A$, see pages 79-82 in [5] (URL: http://www.scm.keele.ac.uk/staff/z_li/PhD_Thesis.pdf)for details

Thus, if POS is such that
$(POS \parallel CUST)$ **sat** $REQC$,
then
$(POS \parallel CUST)\backslash\{notice, pay, change\}$ **sat** $REQC\backslash\{notice, pay, change\}$ **sat** REQ.
From the properties of Lai's quotient, any POS **sat** $CUST \setminus\!\!\setminus REQC$ will solve the problem, though in general Lai's quotient may not always lead to a process [11].

4.1 Solving the Problem Using Lai's Quotient

In this problem, $CUST$ and POS synchronise on all their communication channels, namely, $item, notice, pay, change, receipt$. Recall that in Lai's definition of the quotient, set A is the alphabet of chosen communication channels between the two subprocesses X and P. Therefore, $CUST \setminus\!\!\setminus REQC$'s alphabet should be calculated as $(\alpha REQC \setminus \alpha CUST) \cup A$. We choose the entire alphabet of $CUST$ as the set A because it is assumed that all of $CUST$'s alphabet are synchronised communications with POS, and is constrained or referred to by $REQC$. In our model, we ignore any other irrelevant behaviours of $CUST$ in this formal analysis. Therefore,
$A = \{item, notice, pay, change, receipt\}$
$\alpha REQC = \{item, notice, pay, change, receipt\}$,
$\alpha CUST = \{item, notice, pay, change, receipt\}$,
$\alpha(CUST \setminus\!\!\setminus REQC) = (\alpha REQC \setminus \alpha CUST) \cup A$
$= \{item, notice, pay, change, receipt\}$.
We will solve the problem by constructing:
$POS = (CUST \setminus\!\!\setminus REQC)$.
The predicate expressions for $CUST$ and $REQC$, as needed in Lai's quotient, are derived according to the predicative semantics introduced by Lai and Sanders [11]. For ease of presentation, we express their predicate expressions in tabular forms, as shown below.

Predicates on $CUST$'s tr and $accept$ (its meaning is given below) expressed in a tabular form:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of tr	$\langle\rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i-p_1)$	...	$p.p_n$	$c.(\Sigma_{x=1}^{n} p_x - i)$	$r.i$
$accept$	$\{i\}$	$\{n\}$	$\{p\}$	$\{c, n\}$	...	$\{p\}$	$\{c, n\}$	$\{r\}$	$\{\}$

Predicates on $REQC$'s tr and $accept$ expressed in a tabular form:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of tr	$\langle\rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i-p_1)$	...	$p.p_n$	$c.(\Sigma_{x=1}^{n} p_x - i)$	$r.i$
$accept$	$\{i\}$	$\{n\}$	$\{p\}$	$\{n\}, \{c\}$	...	$\{p\}$	$\{n\}, \{c\}$	$\{r\}$	$\{\}$

In the above tables, we have abbreviated events to their first letters, and shown all possible behaviours of $CUST$ and $REQC$ that are associated with an item that costs i. An item of cost i will lead to a trace of no longer than $2i + 3$ events: each time the customer pays, it must be with a coin of value greater than 1 $pence$, so that the amount remaining is at most one less. As i is finite, all traces of the system are finite.

The first row of the table shows a trace of length l ($0 \leq l \leq 2n + 3$). In the second row of the table we give the events of the trace; in the third row, we indicate the refusal set after that trace. We name this set $accept$ to represent those entries that the process cannot refuse. For example, in the first table, the entry for $l = 3$ is $p.p_1, \{c, n\}$, indicating that the failure is $(\langle i.i, n.i, p.p_1\rangle, \alpha CUST \setminus \{c, n\})$. [9]

We can check that the representation of the table interpreted in this way provide the predicative semantics for the represented terms.

In $CUST$'s table, from

$CUST = \bigsqcap_{i\in\{1,\ldots,100\}} item!i \rightarrow notice?i \rightarrow PAY$, where

$$PAY = \bigsqcap_{p\in\{1,2,5,10\}} pay!p \rightarrow (change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST} \\ \Box\ notice?\ n \rightarrow PAY).$$

we give the following explanations of two representative entries in the table:

- When the trace length is 0, which means $tr = \langle\rangle$, then according to the semantics of event prefix in section 4, $item.i$ can not be refused, $item.i \notin ref \Leftrightarrow ref \subseteq \alpha CUST \setminus \{item.i\}$, that is, $accept = \{i\}$; also according to the semantics, the next event in tr must be the head of $CUST$ which is $item.i$ whose shorthand is $i.i$ in the table;
- $CUST$'s refusal set after the trace $\langle i.i, n.i, p.p_1\rangle$ is derived according to the semantics of external choice in section 4, as follows:
  before $(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST}\ \Box\ notice?\ n \rightarrow PAY)$ is executed, that is, its trace is empty, its behaviour is defined to be
  $(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST})(tr, ref) \wedge (notice?\ n \rightarrow PAY)(tr, ref)$,
  again, according to the semantics of event prefix, $change.c \notin ref \wedge notice.n \notin ref$ holds, which means $ref \subseteq \alpha CUST \setminus \{change, notice\}$, which explains the entry $accept = \{c, n\}$ (notice the shorthand) in $CUST$'s table.

The rest of the entry can be similarly derived accordingly.

Different from $CUST$, the choice is internal after the trace $\langle i.i, n.i, p.p_1\rangle$, i.e.,

$(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha REQC}\ \sqcap\ notice?\ n \rightarrow REQCPAY)$

$REQC$'s refusal set after the trace $\langle i.i, n.i, p.p_1\rangle$ is derived according to the semantics of internal choice, as follows:

the above internal choice's behaviour is defined to be

$$(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha REQC})(tr, ref) \vee (notice?\ n \rightarrow REQCPAY)(tr, ref)$$

according to the semantics of event prefix, $change.c \notin ref \vee notice.n \notin ref$ holds, which means $ref \subseteq \alpha CUST \setminus \{change\}, \{notice\}$, which explains the entry $accept = \{c\}, \{n\}$ in $REQC$'s table. Note that we use "," to represent "exclusive or", which means that $REQC$ can refuse either c or n, but not both.

Lai's quotient is defined as:

$$\begin{aligned}
&CUST \setminus\!\!\setminus REQC(tr, ref) = \\
&\forall ur : traces(REQC)\ \forall\ rep \subseteq \alpha CUST \bullet\ [tr = ur \upharpoonright \alpha\,(CUST \setminus\!\!\setminus REQC) \\
&\wedge\ CUST(ur \upharpoonright \alpha CUST, rep)\ \Rightarrow REQC(ur, rep \cup ref)] \\
&\qquad\qquad (since\ \alpha REQC = \alpha CUST = \alpha(CUST \setminus\!\!\setminus REQC),\ thus\ tr = ur) \\
&\Leftrightarrow \forall rep \subseteq \alpha CUST \bullet [CUST(tr, rep) \Rightarrow REQC(tr, rep \cup ref)]
\end{aligned}$$

[9] We use $accept$ to stand for the intuitive meaning of acceptance, rather than a strictly formal meaning of acceptance, as in [12].

From the above step of derivation based on Lai's quotient definition, we know that $tr = ur$, which means $POS = CUST \mathbin{\backslash\!\backslash} REQC$'s trace tr is always equal to that of $REQC$, due to the fact that $\alpha REQC = \alpha CUST = \alpha(CUST \mathbin{\backslash\!\backslash} REQC)$ holds. Therefore, all the entries of trace events in POS's table is exactly the same as those in $CUST$'s table.

Next, let us look at the $accept$ entries in POS's tables. We derive some representative $accept$ entries in POS's table from the given entries in $CUST$ and $REQC$'s tables.

In the first trace event, given that $CUST(\langle\rangle, \{n, p, c, r\})$ and $REQC(\langle\rangle, \{n, p, c, r\})$ are true (it is a fact, as shown in the tables),

$$\begin{aligned}&CUST \mathbin{\backslash\!\backslash} REQC(\langle\rangle, \mathit{ref})\\ &= \forall \mathit{rep} \subseteq \{i, n, p, c, r\} \bullet [CUST(\langle\rangle, \mathit{rep}) \Rightarrow REQC(\langle\rangle, \mathit{rep} \cup \mathit{ref})]\end{aligned}$$

That $\mathit{rep} = \{i\}$ contradicts with the fact $CUST(\langle\rangle, \{n, p, c, r\})$ holds. When $\mathit{rep} \subseteq \{n, p, c, r\}$, we know for a fact that the antecedent is always $true$, and in order to make the consequent $true$ so that the entire predicate holds, $\{n, p, c, r\} \cup \mathit{ref} = \{n, p, c, r\}$ must hold, therefore we can derive that $\mathit{ref} \subseteq \{n, p, c, r\}$, which means $\mathit{ref} \subseteq \alpha POS \setminus \{i\}$, which allows us to derive the $accept$ entry in POS's table as $\{i\}$.

The derivations of the other entries in POS's table are similar (see [5] for details).

The constructed table shows POS's behaviour in terms of tr and $accept$:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of tr	$\langle\rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i - p_1)$	...	$p.p_n$	$c.(\Sigma_{x=1}^{n} p_x - i)$	$r.i$
$accept$	$\{i\}$	$\{n\}$	$\{p\}$	$\{n\}, \{c\}$	...	$\{p\}$	$\{n\}, \{c\}$	$\{r\}$	$\{\}$

Note that entries in POS's table correspond to $REQC$'s entries, which leads us to derive POS's expression in a process form based on the correspondence, as follows:

$$\begin{aligned}&POS = \textstyle\bigsqcap_{i \in \{1,\ldots,100\}}\ item?i \rightarrow notice!i \rightarrow POSPAY(i, i), \text{ where}\\ &POSPAY(i, remain) =\\ &\textstyle\bigsqcap_{p \in \{1,2,5,10\}}\ pay?p \rightarrow\ \mathit{if}\ p < remain\ \mathit{then}\ (notice!(remain - p)\\ &\qquad\qquad \rightarrow POSPAY(i, remain - p))\ \mathit{else}\ (change!(p - remain)\\ &\qquad\qquad \rightarrow receipt!i \rightarrow STOP_{\alpha POS})\end{aligned}$$

Note that $POSPAY$ involves the communication of at least two values, value i for the first $receipt$ event, and a variable value $remain$ for later $notice$ event representing the remaining amount of payment needed; the choice is chosen by a conditional: if the payment $remain \leq p$, then a change and a receipt will be given out by POS; if $p < remain$ then a notice for the need of further payment will be given by POS. These elaborated details can be implemented quite easily in a programming language as a function with two parameters, which will be shown in our FDR script later.

With this derivation of POS, we have solved the problem constructively.

5 Related Work

Few authors have considered the formal transformation of requirements into specifications.

Seater et al. [13] have done some related work on deriving specifications from requirements in the context of Problem Frames, in which the requirement is transformed into a specification, and, as a by-product of the transformation, a record of domain assumptions, which they call ‘breadcrumbs’, are produced as justification for the progression: domain assumptions are added manually by the expert analyst. The focus of their transformation is on rephrasing the requirement progressively until it is expressed as a machine specification, while manually adding domain assumptions which make the requirement transformation sound. Their work is based on Alloy [14], a first-order logic modelling language, which is used to express requirements and domain assumptions, and to check the soundness of their requirements transformations. In contrast, our approach systematically progresses whole problems which are expressed in CSP through application of the Lai’s Quotient operator, and solely relying on its logical deductions and proofs of equivalence.

Within goal-oriented approaches to RE, [15] propose a way to derive software specifications from high-level goal-based requirements models expressed in real-time linear temporal logic; the approach defines formal rules for mapping real-time temporal logic specifications to sets of pre, post and trigger conditions of functional specifications, with the application of the rules guided by a catalogue of goal specification patterns. This work share a similar objective as our work, but is located in a different RE paradigm.

6 Conclusions and Future Work

We have proposed a problem-based approach for the systematic transformation of requirements into specifications, through a CSP encoding of problem diagrams and the application of the Lai’s Quotient operator, and we have demonstrated its feasibility through its successful application to a small example, where the problem transformation process is supported by constructing proofs of correctness, with CSP expressions of problem diagrams as its inputs and the derived CSP trace expressions of the machine specifications as its outputs. The technique has been found useful for underpinning requirements analysis in software engineering [16]. For bigger case studies, we have successfully underpinned problem transformation with causal domain knowledge and an associated set of transformation rules [5]. Similarly, once domain knowledge about causal relationships are augmented and formalised as premises of our proofs, the Quotient operator can still be applied to bigger case studies.

The main problem with the approach is the difficulty of having requirements expressed in CSP in the first place, since the *de facto* language used in describing requirements is natural language. Hence, future work will investigate the boundary of its applicability, and any preliminary work needed before applying our technique.

On the other hand, a main virtue of the approach is that, once such a CSP problem model is established and causal relationships are formally represented, problem progression is the outcome of applying Lai’s Quotient operator, making automation of the transformation viable without human actors’ intervention. Future work will aim at providing such automation, which may be useful for solving problems for critical systems for which unambiguous specifications and strong validation are mandatory.

References

1. Jackson, M.: The world and the machine (keynote). In: 17th Int. Conf. on Software Engineering (ICSE'95), Seatle, USA, IEEE/ACM (April 1995) 282–292
2. Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley Publishing Company (2001)
3. Hall, J.G., Rapanotti, L., Jackson, M.: Problem frame semantics for software development. Software and Systems Modeling **4**(2) (May 2005) 189–198
4. Rapanotti, L., Hall, J.G., Li, Z.: Deriving specifications from requirements through problem reduction. IEE Proceedings - Software **153**(5) (October 2006) 183–198
5. Li, Z.: Progressing problems from requirements to specifications in problem frames. Phd thesis, Department of Computing, The Open University, Walton Hall, Milton Keynes, UK (September 2007)
6. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology **6**(1) (January 1997) 1–30
7. Hall, J.G., Rapanotti, L.: Assurance-driven design in problem oriented engineering. International Journal On Advances in Systems and Measurements **2**(1) (2009) 119–130
8. Hall, J.G., Rapanotti, L., Jackson, M.: Problem oriented software engineering: A design-theoretic framework for software engineering. In: Proceedings of 5th IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society Press (2007) 15–24 doi:10.1109/SEFM.2007.29.
9. Hall, J.G., Rapanotti, L., Jackson, M.: Problem-oriented software engineering: solving the package router control problem. IEEE Trans. Software Eng. (2008) doi:10.1109/TSE.2007.70769.
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
11. Lai, L., Sanders, J.W.: A weakest-environment calculus for communicating processes. Research report PRG-TR-12-95, Programming Research Group, Oxford University Computing Laboratory (03-1995 1995)
12. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)
13. Seater, R., Jackson, D., Gheyi, R.: Requirement progression in problem frames: Deriving specifications from requirements. Requirements Engineering Journal (REJ'07) **12**(2) (April 2007) 77–102
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge, MA, USA (April 2006)
15. Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: SIGSOFT 2002/FSE-10, Charleston, SC, USA (November 2002)
16. Li, Z., Hall, J.G., Rapanotti, L.: Modeling domain knowledge in support of requirements analysis in software engineering. In: International Conference on Power and Energy Systems (ICPES2010), IEEE (2010)