On Modelling User Observations in the UTP

Michael J. Banks and Jeremy L. Jacob

Department of Computer Science, University of York, UK
{Michael.Banks,Jeremy.Jacob}@cs.york.ac.uk

Abstract. This paper presents an approach for modelling interactions between users and systems in the *Unifying Theories of Programming*. Working in the predicate calculus, we outline generic techniques for calculating a user's observations of a system and, in turn, for identifying the information that a user can deduce about the system's behaviour from those observations. To demonstrate how this approach can be applied in practical software development, we propose some alternative refinement relations that offer greater flexibility than classical refinement by utilising knowledge of the observational abilities of users.
Keywords: UTP, multi-user systems, co-operating and independent refinement, information flow, distributed testing.

1 Introduction

This paper is concerned with software systems whose purpose is to provide a range of services to multiple end-users. This class of *multi-user* systems encompasses a large range of software products, from operating systems and database software to telecommunications networks and cloud computing services.

A multi-user system consists of a central server that receives requests from multiple users (clients) and delivers service in response to those requests. Naturally, the system may offer different services to different users. It is usual to provide each user with its own private interface to the system, to ensure that the system can distinguish between its users and to ensure that users do not interfere with each other's interactions with the system. These interfaces impose a structure on the system's environment and allow the system's designers to model the interactions that users can perform with the system.

When working in process algebras such as CSP, it is usual to model the users of a system as individual processes operating in parallel with a process representing the system [1]. By analysing the synchronisations between the system and the user processes, this approach may be used to verify that a system specification delivers the functionality expected by its users. However, this approach is suitable only for reasoning about systems expressed in the same semantic domain in which the user processes are formulated. This may be problematic if the designers of a system wish to analyse its interactions with users in a more concrete description of the system (such as a program), or a system description that consists of multiple components expressed in different formalisms. In

such circumstances, it may be necessary to adopt a more general approach for reasoning about the interactions between a system and its users.

Our main contribution in this paper is a systematic approach for formally modelling the interactions between users and systems in Hoare and He's Unifying Theories of Programming (UTP) [2]. This approach fits seamlessly within the predicate semantics of the UTP; it can, therefore, be integrated with existing UTP theories to support the analysis of multi-user systems that are specified in languages with a UTP semantics.

This paper is structured as follows. Section 2 provides an overview of the UTP. Section 3 formalises two distinct classes of observations of multi-user systems and introduces our approach for modelling the abilities of users to observe a system's execution. Building on this approach, Section 4 presents a method for calculating the space of observations that individual users can make of a system. This method is extended to the UTP theory of designs in Section 5 and applied to a byte register in Section 6 as a simple worked example.

To demonstrate how our approach can be advantageous in formal software development, Section 7 describes some alternative notions of refinement that are based on the observational abilities of users. These refinement relations offer system designers the ability to carry out refinement steps that do not compromise the system's functionality from the perspective of its users, but which are nevertheless forbidden by classical refinement.

In Section 8, we survey some areas of research addressing multi-user systems in a formal setting and discuss the relevance of our approach to those areas. Finally, we present our conclusions and outline some topics for future work.

2 Unifying Theories of Programming

The semantic model of the UTP is the alphabetised relational calculus. In the UTP, specifications and programs alike are expressed as predicates over an alphabet (set) of observational variables. The purpose of the observational variables is to record all of the information about a program's behaviour that is visible to the program's environment whenever an observation of the program is made.

In the UTP theory of relations, a program operation is expressed as a predicate that relates initial observations of the program state to the corresponding observations of the program state taken after (or during) the execution of the operation. The observational variables representing intermediate observations of a program state are decorated with a prime, to distinguish them from the (undecorated) observational variables of the initial program state.

Central to the UTP is the refinement ordering between predicates, which is characterised by implication: [2]

$$S \sqsubseteq T \triangleq [\, T \Rightarrow S \,] \tag{1}$$

where the square brackets denote universal quantification over all free variables. Hence, $S \sqsubseteq T$ asserts that every observation of T is a possible observation of S.

The UTP is best known as a framework for giving a denotational semantics to various programming paradigms and the features of programming languages. However, the UTP also features a powerful standalone notation for reasoning about program correctness. We adopt the UTP in this capacity as the framework in which we cast our approach for reasoning about multi-user systems.

3 Preliminaries

3.1 System-Level and Interface-Level Observations

We model an observation of a UTP predicate S as a predicate that associates each observational variable in S's alphabet with a single value. For instance, the predicate $x = 42 \wedge y = 99$ records a possible observation of the predicate $x > 0 \wedge y > x$. We distinguish between two separate classes of observations:

- A *system-level* observation records the entirety of the information about the behaviour of a system (expressed as a predicate) that may be acquired by monitoring the whole of the system's environment.
- An *interface-level* observation records the information that a specified user acquires when it observes a system's behaviour through its interface. Hence, each interface-level observation is a *projection* of a system-level observation.

While a system-level observation provides all the information about a system's behaviour that is visible to the environment, an interface-level observation provides only a subset of that information. Throughout this paper, we assume that a user's observation of a system provides the only source of information about the execution of the system that is available to the user. Hence, a user can neither inspect the internal state of the system directly, nor can it monitor the aspects of the system's behaviour that are not visible through its interface. Unless stated otherwise, we require that users are isolated from each other and do not share their observations of the system with other users.

Following the UTP notational conventions, we denote the list of undashed variables $s_1, \ldots, s_i$ of a system-level observation by s and the corresponding list of dashed variables by s'. Likewise, we denote the lists $u_1, \ldots, u_j$ and $u'_1, \ldots, u'_j$ of variables of an interface-level observation by u and u' respectively. In keeping with our assumption that system-level variables are hidden from users, we enforce the condition that the sets of variables in s and u (and s' and u') are disjoint.

3.2 Views

A user's interface to a system is modelled by a predicate known as a *view*. A view defines the mapping between interface-level observational variables and system-level observational variables. Thus, a view determines which aspects of each system-level observation are relayed to the user associated with that view.

Example 1. Suppose x and y are observational variables of an arbitrary system-level predicate. Consider the following views:

$$A \triangleq x_A = x \land y_A = y$$
$$B \triangleq z_B = \max(x, y)$$
$$C \triangleq x_C = x \land (z_C = 0 \lhd x < y \rhd z_C = 1)$$

View A provides a user with complete knowledge of the values of x and y, since they are a function of the interface-level observational variables x_A and y_A. View B provides a user with the value of the larger of x and y, but no indication of whether $x < y$, $x = y$ or $x > y$. View C provides the value of x, but offers only partial information about y's value in relation to x's value.

When dealing with multiple views, each representing a different interface to a system, we require that each view's set of interface-level observational variables is disjoint from those of the other views. (All of the views listed in Example 1 are pairwise disjoint.)

Definition 1 (Disjoint views). *A pair of views $V_1(s, u_1)$ and $V_2(s, u_2)$ are said to be disjoint if and only if $u_1 \cap u_2 = \emptyset$.*

3.3 Healthiness Conditions for Views

To ensure that all views represent a viable mapping between system-level and interface-level observations, it is necessary to impose some constraints on the structure of views. We capture the space of viable views by defining two healthiness conditions **VH1** and **VH2**.

Definition 2 (VH1). *A view $V(s, u)$ is* **VH1***-healthy if and only if, for every system-level observation over the variables in s and s', there is a complementary interface-level observation over the variables in u and u' that satisfies V:*

$$\mathbf{VH1}(V) \quad \textit{iff} \quad \forall\, s, s' \bullet \exists\, u, u' \bullet V \tag{2}$$

The purpose of **VH1** is to ensure that a view maps each system-level observation to at least one interface-level observation. It follows that predicates such as $x = y$, $x > 0$ and **false** are not **VH1**-healthy views, because they restrict the domain of x.

While **VH1** insists that a view describes a total mapping from system-level to interface-level observations, it does not require that mapping to be functional — i.e. each system-level observation maps to exactly one interface-level observation — although this will often be the case for views in practice. Moreover, **VH1** does not place any restrictions on whether an interface-level observation maps to zero, one or multiple system-level observations.

It is possible to write a "time-travelling" **VH1**-healthy predicate where the values of the initial (undashed) interface-level observational variables depend

upon the values of the intermediate (dashed) system-level observational variables. Such a predicate cannot correspond to a user's interface of a system in reality, since an intermediate observation of a system state can only be made once a system has started, whereas an initial observation describes the system state before execution commences.

Another class of undesirable **VH1**-healthy predicates are those featuring dependencies between the initial and intermediate system-level variables (such as $x' = x + 1$), or the initial and intermediate interface-level variables. These predicates impose functional requirements on the structure of systems and therefore should not be considered to be views. We exclude these predicates — along with the aforementioned "time-travelling" predicates — from the space of views by defining a second healthiness condition **VH2**.

Definition 3 (VH2). *A view $V(s, u)$ is* **VH2***-healthy if and only if it places no dependencies between s and s', u and u', or s' and u:*

$$\mathbf{VH2}(V) \;\; \textit{iff} \;\; ((\exists\, s, u \bullet V) \wedge (\exists\, s', u \bullet V) \wedge (\exists\, s', u' \bullet V)) = V \tag{3}$$

We say that a view that satisfies both **VH1** and **VH2** is **VH**-healthy. All of the views in Example 1 are **VH**-healthy, as are all the views that we consider in the following sections.

The following lemma states that the view obtained by taking the conjunction of disjoint **VH**-healthy views is itself a **VH**-healthy view, which gives at least as much information about a system's behaviour as each view individually.

Lemma 1 (Conjunction of VH-healthy views). *If $V_1(s, u_1)$ and $V_2(s, u_2)$ are disjoint* **VH***-healthy views over the same set of system-level observational variables s, then $(V_1 \wedge V_2)(s, u_1 \cup u_2)$ is also* **VH***-healthy.*

4 Relating Users and Systems

In this section, we define predicate transformers for calculating the aspects of a system's behaviour that are visible to a user through a specified view and, given an interface-level observation of the system, the information that a user can deduce about the system's behaviour.

4.1 Calculating Interface-Level Predicates

Given a system-level predicate S and a view V, it is possible to derive a predicate that encodes the space of all interface-level observations that can be made of S when viewed through V. We define a predicate transformer **P** (for "project") to calculate this predicate.

Definition 4 (P predicate transformer). *The interface-level predicate obtained by substituting $V(s, u)$ into a predicate $S(s)$ is given by:*

$$\mathbf{P}\,(V, S) \;\triangleq\; \exists\, s, s' \bullet V \wedge S \tag{4}$$

The predicate $\mathbf{P}(V, S)$ is the image of S as viewed through V; in other words, $\mathbf{P}(V, S)$ is satisfied by exactly those interface-level observations that can be made by viewing S through V. Notice that applying $\mathbf{P}(V)$ to S hides the system-level observational variables in S.

Example 2. Consider the system-level predicate $E \triangleq x + y = 10$, where $x, y \in \mathbb{N}$. For the views listed in Example 1, the projections of E are:

$$\begin{aligned}
\mathbf{P}(A, E) &= x_A + y_A = 10 \\
\mathbf{P}(B, E) &= z_B \geq 5 \wedge z_B \leq 10 \\
\mathbf{P}(C, E) &= (z_C = 0 \wedge x_C \geq 0 \wedge x_C \leq 4) \vee (z_C = 1 \wedge x_C \geq 5 \wedge x_C \leq 10)
\end{aligned}$$

A view can be regarded as a special kind of linking predicate between two data types, where system-level observational variables are instances of the concrete data type, and interface-level observational variables are instances of the abstract data type. It follows that applying $\mathbf{P}(V)$ to a predicate S may be interpreted as performing data refinement on S *in reverse*, because concrete (system-level) observations are replaced by abstract (interface-level) observations. This suggests that existing techniques for reasoning about data refinement can be applied to identify results concerning the application of $\mathbf{P}(V)$ to predicates. Moreover, the application of views to translate from system-level to interface-level models of systems is related to *abstract interpretation* [3].

We now present some properties of $\mathbf{P}$ that we use later in the paper.

Lemma 2 ($\mathbf{P}$ is order-preserving). *Provided that S and T are predicates defined over the same space of system-level observational variables as V:*

$$S \sqsubseteq T \;\Rightarrow\; \mathbf{P}(V, S) \sqsubseteq \mathbf{P}(V, T) \tag{5}$$

If a view V is divided into two parts V_1 and V_2, such that $V = (V_1 \wedge V_2)$ and V_1 and V_2 are disjoint, then the interface-level predicate $\mathbf{P}(V_1, S) \wedge \mathbf{P}(V_2, S)$ that is generated by projecting S through V_1 and V_2 separately is satisfied by a (potentially) larger space of observations than $\mathbf{P}(V, S)$ itself. We generalise this result to an arbitrary number of views in Lemma 3.

Lemma 3 (Splitting V may weaken $\mathbf{P}(V)$). *When $V_1, \ldots, V_n$ are pairwise disjoint views over the same space of system-level observational variables:*

$$\mathbf{P}\left(\left(\bigwedge_{i \in 1..n} V_i\right), S\right) \;\Rightarrow\; \bigwedge_{i \in 1..n} \mathbf{P}(V_i, S) \tag{6}$$

4.2 Calculating System-Level Predicates

Given an interface-level predicate U formed by projecting a system-level predicate S through view V, it is possible to recover some (if not all) knowledge of the definition of S by substituting V back into U. We formalise this process by defining another predicate transformer $\mathbf{R}$ (for "retract").

Definition 5 (R predicate transformer). *The system-level predicate obtained by substituting $V(s, u)$ into an interface-level predicate $U(u)$ is given by:*

$$\mathbf{R}\,(V, U) \;\triangleq\; \forall\, u, u' \bullet V \Rightarrow U \tag{7}$$

$\mathbf{R}\,(V, U)$ recovers the weakest system-level predicate T such that the projection of each system-level observation of T through V matches an interface-level observation of U, and each observation of U corresponds to an observation of T projected through V. It follows from $\mathbf{R}$'s definition that $[\, V \wedge T \Rightarrow U \,]$ [2].

The $\mathbf{P}$ and $\mathbf{R}$ predicate transformers are not inverses of each other, since a view need not define a one-to-one correspondence between system-level and interface-level observations. They do, however, form a Galois connection between system-level and interface-level predicates under the refinement ordering.

Theorem 1 ($\mathbf{P}$ and $\mathbf{R}$ form a Galois connection). *The $\mathbf{P}$ and $\mathbf{R}$ predicate transformers form a Galois connection (axiality) between the spaces of system-level and interface-level predicates linked by a given view. Thus:*

$$U \sqsubseteq \mathbf{P}\,(V, S) \quad \textit{if and only if} \quad \mathbf{R}\,(V, U) \sqsubseteq S \tag{8}$$

Corollary 1. *By substituting $\mathbf{P}\,(V, S)$ in place of U in Theorem 1, we obtain:*

$$\mathbf{R}\,(V, \mathbf{P}\,(V, S)) \sqsubseteq S \tag{9}$$

Corollary 1 implies that every system-level observation of S is also a system-level observation of $\mathbf{R}\,(V, \mathbf{P}\,(V, S))$. This conforms to the intuition that applying $\mathbf{P}\,(V)$ to a system-level predicate S may discard information about the observations permitted by S, which cannot be recovered by applying $\mathbf{R}\,(V)$ to $\mathbf{P}\,(V, S)$.

Example 3. Continuing from Example 2, the system-level predicates recovered by applying $\mathbf{R}\,(V)$ to each interface-level projection of E are as follows:

$$\begin{aligned}
\mathbf{R}\,(A, \mathbf{P}\,(A, E)) &= x + y = 10 \\
\mathbf{R}\,(B, \mathbf{P}\,(B, E)) &= \max(x, y) \geq 5 \wedge \max(x, y) \leq 10 \\
\mathbf{R}\,(C, \mathbf{P}\,(C, E)) &= (x \geq 0 \wedge x \leq 4 \lhd x < y \rhd x \geq 5 \wedge x \leq 10)
\end{aligned}$$

Observe that $\mathbf{R}\,(A, \mathbf{P}\,(A, E)) = E$, because view A preserves the values of x and y in x_A and y_A. However, both $\mathbf{R}\,(B, \mathbf{P}\,(B, E))$ and $\mathbf{R}\,(C, \mathbf{P}\,(C, E))$ are weaker predicates than E, because information about the exact values of x and y is discarded when $\mathbf{P}\,(B)$ and $\mathbf{P}\,(C)$ are applied to E.

4.3 Observations and Deductions

Users can acquire knowledge about a system's behaviour in two ways: by observation and by deduction. While we assume users can only observe a system's behaviour through an interface supplied by the system, there may be nothing to prevent a user from possessing *a priori* knowledge of the design of the system

or its interface. Hence, if a user has knowledge of the system's implementation then, given a projection of a system-level observation through its view, the user may apply this knowledge to rule out potential system-level observations that are incompatible with its own observation and thereby deduce more detailed information about the system behaviour.

When ϕ is an interface-level observation of a predicate S viewed through V, the predicate describing all system-level observations of S that are compatible with ϕ is given by $\mathbf{R}(V, \phi) \wedge S$. Thus, a user that knows the definition of S can deduce more (but not necessarily all) information about the system-level observation of S than it can from $\mathbf{R}(V, \phi)$ alone. In turn, if a user at V has knowledge of the structure of another user's view W, then the user can infer all observations of S through W that are compatible with its own observation ϕ by calculating $\mathbf{P}(W, (\mathbf{R}(V, \phi) \wedge S))$.

We note in passing that, at the level of a system's implementation, a user's interface will exhibit physical and temporal characteristics (such as fluctuations in responsiveness) that are not modelled at the abstract level of a view. By monitoring these properties of its interface, a user may be able to deduce greater knowledge about the internal state of the system than can be calculated from its interface-level observation of the system alone.

5 Reasoning about Multi-User Designs

The UTP theory of designs represents the space of terminating programs with precondition (assumption) P and postcondition (commitment) Q: [2]

$$P \vdash Q \;\triangleq\; ok \wedge P \Rightarrow ok' \wedge Q \tag{10}$$

The Boolean variables ok and ok' facilitate reasoning about termination: ok records that the program has started and ok' signifies that the program has terminated. Thus, if the program is started in an initial state that satisfies P, then the program is guaranteed to terminate in a final state satisfying Q.

We interpret a design as a system that starts in an initial state consisting of inputs from users and terminates in a final state that yields outputs to users. It is reasonable to expect that users can identify when a system has started and when it has terminated. Since system-level variables are not directly visible to users (Section 3.1), it is necessary to extend the alphabet of a view V by introducing new interface-level observational variables ok_V and ok'_V corresponding to ok and ok'. We require that V guarantees that $ok_V = ok$ and $ok'_V = ok'$. This requirement is encoded by the **OK** healthiness condition.

Definition 6 (OK and VHD). *A view $V(s, u)$ is* **OK***-healthy if and only if $V = \mathbf{OK}(V)$ holds:*

$$\mathbf{OK}(V) \;\triangleq\; V \wedge ok_V = ok \wedge ok'_V = ok' \tag{11}$$

We say a view is **VHD***-healthy if it is both* **VH***-healthy and* **OK***-healthy.*

Since designs are defined in terms of ok and ok', we generalise the definition of a design by substituting ok_V and ok'_V in place of ok and ok'. Definition 7 introduces a shorthand for interface-level projections of designs.

Definition 7 (Interface-level design). *Provided V denotes a view:*

$$P \vdash_V Q \triangleq ok_V \wedge P \Rightarrow ok'_V \wedge Q \tag{12}$$

The $\mathbf{P}$ predicate transformer can be applied to a $\mathbf{VHD}$-healthy view V and a design to obtain an interface-level design that expresses the interface-level projection of the behaviour of the design.

Lemma 4 ($\mathbf{P}$ and designs). *Whenever V is $\mathbf{VHD}$-healthy, then $\mathbf{P}(V, P \vdash Q)$ can always be written in the form of an interface-level design:*

$$\mathbf{P}(V, P \vdash Q) = (\forall s, s' \bullet V \Rightarrow P) \vdash_V \mathbf{P}(V, Q) \tag{13}$$

The precondition of $\mathbf{P}(V, P \vdash Q)$ is the weakest condition over interface-level observations that is sufficient to ensure that, whenever ϕ is an interface-level observation that satisfies that precondition, then *all* system-level observations compatible with ϕ satisfy P. Therefore, if a user's initial observation of $P \vdash Q$ (projected through V) satisfies the precondition of $\mathbf{P}(V, P \vdash Q)$, then the user is guaranteed that P is satisfied. It follows that if the precondition of $\mathbf{P}(V, P \vdash Q)$ is satisfied, then Q will hold upon the termination of $P \vdash Q$, as will the interface-level projection $\mathbf{P}(V, Q)$ of Q.

If no projection of the initial system-level observational variables through V provides sufficient information about the initial system-level observational variables to guarantee that P holds, then the precondition of $\mathbf{P}(V, P \vdash Q)$ collapses to **false** and so nothing is guaranteed about the final observation of $\mathbf{P}(V, P \vdash Q)$.

6 Worked Example: A Byte Register

We now consider the application of the theory developed in the previous sections to a simple multi-user system. The purpose of this example is to demonstrate how the specification and design of multi-user systems may be guided by calculating the interface-level observations of a system and identifying the information that users can deduce from these observations.

Our example focuses on a register capable of storing a single byte. We model the register's value by an integer variable x with domain 0..255. The register also features a Boolean variable y that indicates numeric overflow when set.

Consider an operation that doubles the value stored in x, provided that the initial value of x lies in the range 0..127 and the overflow bit is not set. We model this operation as a UTP design as follows:

$$DBL \triangleq x \in 0..127 \wedge y = 0 \vdash x' = 2x \wedge y' = 0$$

Suppose that two users can observe the register: the first user (with view H) can observe the values of the higher four bits of the value of x, and the second user (with view L) can observe the lower four bits of x. The overflow bit y is visible to both users. The views of these users are given by:

$$H \triangleq \mathbf{OK}\left(x_H = \left\lfloor \frac{x}{16} \right\rfloor \land x'_H = \left\lfloor \frac{x'}{16} \right\rfloor \land y_H = y \land y'_H = y'\right)$$
$$L \triangleq \mathbf{OK}\left(x_L = x \bmod 16 \land x'_L = x' \bmod 16 \land y_L = y \land y'_L = y'\right)$$

Effectively, the H view masks out the lower four bits of the register from the first user's observations, while the L view hides the higher four bits from the second user. Both of these views are **VHD**-healthy.

We now investigate the projections of DBL's behaviour through H and L. The calculation of $\mathbf{P}(H, DBL)$ is simplified by applying Lemma 4:

$$\begin{aligned}\mathbf{P}(H, DBL) &= \begin{pmatrix} \forall\, x, y, x', y' \bullet H \Rightarrow x \in 0..127 \land y = 0 \\ \vdash_H \\ \exists\, x, y, x', y' \bullet H \land x' = 2x \land y' = 0 \end{pmatrix} \\ &= x_H \in 0..7 \land y_H = 0 \vdash_H (x'_H = 2x_H \lor x'_H = 2x_H + 1) \land y'_H = 0\end{aligned}$$

$\mathbf{P}(H, DBL)$ indicates that a user at H can only be certain that DBL's precondition is satisfied when $x_H \in 0..7$, since any other value of x_H corresponds to a value of x that violates the precondition of DBL.

Assuming the precondition of DBL holds, the value of the fifth most significant bit of x determines whether $x'_H = 2x_H$ or $x'_H = 2x_H + 1$. However, since this bit cannot be observed by a user at H, that user can only be certain of the value of x'_H once the operation is complete.

It is instructive to consider what can be observed of DBL through L:

$$\begin{aligned}\mathbf{P}(L, DBL) &= \begin{pmatrix} \forall\, x, y, x', y' \bullet L \Rightarrow x \in 0..127 \land y = 0 \\ \vdash_L \\ \exists\, x, y, x', y' \bullet L \land x' = 2x \land y' = 0 \end{pmatrix} \\ &= \mathbf{false} \vdash_L x'_L = 2x_L \bmod 16 \land y'_L = 0 \\ &= \mathbf{true}\end{aligned}$$

Since an observation at L provides no information regarding the upper four bits of x, a user at L cannot determine in any circumstances whether the precondition of DBL is satisfied. Thus, from the perspective of L, nothing can be guaranteed about DBL's behaviour, as is reflected by the outcome of the calculation above.

Depending on the context in which the register is used, this limitation may be unacceptable. Hence, we relax the precondition of DBL to cover all values of x that can be stored by the register and, when $x \geq 128$, to assign an arbitrary value from the range 0..255 to x' and set y' to 1:

$$DBL2 \triangleq x \in 0..255 \land y = 0 \vdash \begin{pmatrix} x' = 2x \land y' = 0 \\ \lhd\ x \in 0..127\ \rhd \\ x' \in 0..255 \land y' = 1 \end{pmatrix}$$

Observe that $DBL \sqsubseteq DBL2$, because the postcondition of $DBL2$ reduces to the postcondition of DBL when the precondition of DBL is satisfied.

The interface-level observations of $DBL2$ through L are given by:

$$\mathbf{P}\,(L, DBL2) = x_L \in 0..15 \wedge y_L = 0 \vdash \begin{pmatrix} x'_L = (2x_L) \bmod 16 \wedge y'_L = 0 \\ \lhd\, x_L \in 0..7 \,\rhd \\ x'_L \in 0..15 \wedge y'_L = 1 \end{pmatrix}$$

After the $DBL2$ operation completes, the user at L can determine if the double operation was successful by checking that $y'_L = 0$.

One may now proceed to refine $DBL2$ to the implementation level. Of course, if one carries out data refinement on the variables of $DBL2$ (such as replacing x with eight Boolean variables to represent the bits of the register), then the corresponding data refinements must also be made to the H and L views.

7 Refinement of Multi-User Systems

$S \sqsubseteq T$ mandates that every system-level observation of T is a system-level observation of S. This condition is sufficient to ensure that a concrete design T satisfies all the functionality properties of its abstract predecessor S. However, this condition is sometimes too strong for stepwise developments of multi-user systems, since it forbids some classes of reasonable refinement steps that do not impair functionality. This point is illustrated by the following example.

Example 4. Consider an operation on the aforementioned register that doubles the lower four bits of x and the upper four bits of x in isolation:

$$INDBL2 \triangleq x \in 0..255 \wedge y = 0 \vdash \begin{pmatrix} x' \bmod 16 = (2x) \bmod 16 \\ \wedge \begin{pmatrix} \left\lfloor \frac{x'}{16} \right\rfloor = \left(2 \left\lfloor \frac{x}{16} \right\rfloor\right) \wedge y' = 0 \\ \lhd\, x \in 0..127 \,\rhd \\ x' \in 0..255 \wedge y' = 1 \end{pmatrix} \end{pmatrix}$$

Notice that $DBL2 \not\sqsubseteq INDBL2$, because when $x \in 0..127 \wedge y = 0$, $INDBL2$ always sets the fourth most significant bit of x' to 0 regardless of the value of the fifth most significant bit of x, unlike $DBL2$. However, the users at H and L cannot individually tell $INDBL2$ apart from $DBL2$, since each interface-level observation of $INDBL2$ matches an interface-level observation of $DBL2$.

The proposed transition from $DBL2$ to $INDBL2$ indicates that, when developing a multi-user system, it may be more appropriate (in some cases) to carry out refinement w.r.t. the interface-level observations of the system, rather than the system-level observations of the system as a whole.

7.1 Interface-Level Refinement

This section outlines some alternative notions of refinement that are defined in terms of the interface-level observations of systems, instead of system-level observations. These refinement relations are more flexible than the classical notion

of refinement, because they allow particular refinement steps to introduce new behaviours into a system in a controlled manner, while preserving the correctness of the system from the perspective of its users.

First, we introduce an notion of refinement which we call *user refinement*. We say that T is a user refinement of S w.r.t. a view V if and only if every interface-level observation of T made through V corresponds to an observation of S made through V. Intuitively, user refinement allows new system-level observations to be added to a predicate, provided that no interface-level observations are introduced to the projection of the predicate through V.

Definition 8 (User refinement). *For a given view V, T is a user refinement of S — denoted by $S \sqsubseteq_V T$ — if and only if:*

$$S \sqsubseteq_V T \triangleq \mathbf{P}(V, S) \sqsubseteq \mathbf{P}(V, T) \tag{14}$$

It follows from Lemma 2 that $S \sqsubseteq T$ implies $S \sqsubseteq_V T$. Unlike $\sqsubseteq$, $\sqsubseteq_V$ is not a partial order in the general case, although it is always a pre-order.

7.2 Co-operating and Independent Refinement

Jacob [4,5] proposed two notions of refinement — known as *co-operating refinement* and *independent refinement* — intended for application in the development of multi-user systems.

Co-operating refinement allows users to exchange their observations of a system after its execution has terminated. Hence, the users may potentially reconstruct more information about the system behaviour than they could obtain from their individual observations alone.

Independent refinement assumes users cannot communicate with each other; instead, the only information that each user can obtain about the behaviour of a system is their own interface-level observation of the system.

We express co-operating and independent refinement in the UTP by extending the definition of user refinement to a set of disjoint views $\mathcal{W}$ (representing multiple users) in different ways.

Definition 9 (Co-operating and independent refinement). *Co-operating and independent refinement generalise $\sqsubseteq$ as follows:*

$$S \sqsubseteq^{co}_{\mathcal{W}} T \triangleq S \sqsubseteq_{\bigwedge \mathcal{W}} T \tag{15}$$

$$S \sqsubseteq^{ind}_{\mathcal{W}} T \triangleq \bigwedge_{V \in \mathcal{W}} S \sqsubseteq_V T \tag{16}$$

As with $\sqsubseteq_V$, the $\sqsubseteq^{co}_{\mathcal{W}}$ and $\sqsubseteq^{ind}_{\mathcal{W}}$ orderings are pre-orders but not partial orders [4]. When $\mathcal{W}$ contains only a single view V, the $\sqsubseteq^{co}_{\mathcal{W}}$ and $\sqsubseteq^{ind}_{\mathcal{W}}$ relations reduce to the definition of $\sqsubseteq_V$.

Theorem 2 (Ordering on refinement relations). *Standard refinement is a stronger ordering than co-operating refinement which, in turn, is a stronger ordering than independent refinement:*

$$S \sqsubseteq T \;\Rightarrow\; S \sqsubseteq^{co}_{\mathcal{W}} T \tag{17}$$

$$S \sqsubseteq^{co}_{\mathcal{W}} T \;\Rightarrow\; S \sqsubseteq^{ind}_{\mathcal{W}} T \tag{18}$$

As established by Theorem 2, co-operating and independent refinement are weaker than the conventional definition of refinement, because they allow non-determinism to be *added* to a specification, so long as no new interface-level observations of the specification are possible. However, these notions of refinement are strong enough to preserve the functionality inherent in a system's specification from the perspectives of the users of the system.

Example 5. Returning to Example 4, it is the case that $DBL2 \sqsubseteq^{ind}_{\{H,L\}} INDBL2$, since $DBL2 \sqsubseteq_H INDBL2$ and $DBL2 \sqsubseteq_L INDBL2$ both hold. This means that *INDBL2* can safely substitute for *DBL2*, provided that the users at H and L are unable to communicate with each other. However, we do not have $DBL2 \sqsubseteq^{co}_{\{H,L\}} INDBL2$, because if the users at H and L combine their observations, they can identify behaviours of *INDBL2* that are not behaviours of *DBL2*.

Relaxing the notion of refinement to co-operating or independent refinement provides an extra degree of flexibility when making design choices for a system. In particular, *INDBL2* allows the users at H and L to access separate registers without needing to keep those registers synchronised, which means that an implementation of *INDBL2* may provide each user with their own local instance of the register. More generally, these refinement relations offer the opportunity to distribute a system's workload across multiple processors, provided the refined system produces the same results to its users.

A spectrum of refinement relations may be constructed from the $\sqsubseteq^{co}$ relation. A set of views $\mathcal{W}$ may be partitioned into subsets $\mathcal{W}_1, \ldots, \mathcal{W}_n$ to represent groups of users, whereby users associated with views in the same group may co-operate but users associated with views in different groups are isolated from each other. Thus, a multi-user refinement relation which accounts for the boundaries separating these users is given by:

$$S \sqsubseteq^{grp}_{\mathcal{W}} T \;\triangleq\; S \sqsubseteq^{co}_{\mathcal{W}_1} T \wedge \ldots \wedge S \sqsubseteq^{co}_{\mathcal{W}_n} T \tag{19}$$

The notion of co-operating refinement may also be applicable in the development of distributed systems composed of multiple interacting components. For instance, it may be desirable to replace one component Z of the system with another component Z'. This replacement can be carried out with the assurance that other components of the system are not affected by the change, if it can be shown that $Z \sqsubseteq^{co}_{\mathcal{W}} Z'$ holds, where $\mathcal{W}$ is the set of views representing the channels through which the other components interact with Z. The replacement is justified because these other components cannot distinguish Z' from Z by their interactions with Z', even if they share information about those interactions with each other.

7.3 Reasoning About Information Flow

An important topic in theoretical studies of computer security is to measure the information that a low-level (unprivileged) user of a system can learn about the activities of other high-level users by observing the system. If the high-level interactions are associated with sensitive or intrinsically valuable data, then it is imperative that the low-level user is unable to deduce confidential information about this data by monitoring the system's execution [6].

In Section 4.3, we described how one user can deduce information about the observations of another user. We are now able to define an ordering for comparing systems according to the amount of information about high-level observations that can "flow" to a low-level user.

Definition 10 (Security ordering). *Let H and L denote the (disjoint) views of a high-level user and a low-level user respectively. Then, a system T provides no more information flow from H to L than a system S if and only if:*

$$S \preceq_L^H T \triangleq S \sqsubseteq_L T \wedge (\mathbf{P}(L \wedge H, T) \sqsubseteq \mathbf{P}(L \wedge H, S) \wedge \mathbf{P}(L, T)) \tag{20}$$

The $\preceq_L^H$ relation encodes a *security ordering* on predicates [5,7]. The first condition ensures that, from the perspective of a low-level user at L, every observation of T is an observation of S. The second condition requires that, for every observation ϕ of T viewed through L, the set of H observations of S that are compatible with ϕ is a subset of the H observations of T compatible with ϕ. These conditions together guarantee that the low-level user can deduce no more information about the activities at H from an observation of T as it can from the equivalent observation of S. Hence, if $S \preceq_L^H T$ holds, then we say that T provides no more information flow about activities at H to the low-level user as does S. (Of course, this assertion applies only so far as the semantic model of S and T, as it excludes from consideration factors such as the probability distribution or the timing characteristics of system behaviours.)

8 Related Work

Our approach for reasoning about multi-user systems in the UTP opens up several new avenues of investigation. We briefly review two areas of research in which we believe our approach is particularly relevant.

Information Flow Security. A multitude of techniques for measuring and restricting information flow within systems — to guarantee the secrecy of confidential data — have been defined within frameworks based on trace semantics [5,8,9,10]. In these frameworks, a user's observation is given by applying a projection function to the system trace. Our notion of a view is a generalisation of these projection functions, because a view may cover other observational variables besides the trace variables, such as the variables recording the refusal set associated with a trace (when working in the UTP theory of reactive designs.)

Refining a specification may introduce new paths of information flow into the specification, thus enabling a low-level user to deduce more detailed information about the activities of a high-level user and potentially violating security requirements [7]. Various techniques have been proposed to resolve this problem, such as ensuring that the system appears deterministic from the low-level user's viewpoint [11] or strengthening the definition of refinement to preserve information flow security properties [12]. The refinement relation obtained by intersecting the $\sqsubseteq$ and $\preceq$ orderings is an example of the latter approach, but it may be too strong a refinement relation for practical use. It is perhaps more appropriate to employ a weaker notion of refinement (such as co-operating or independent refinement) together with the $\preceq$ ordering in the development of multi-user systems where information security is a priority.

Distributed Testing. When testing distributed systems, it is customary to place an isolated tester (user) at each interface of the system. For the results of test runs to be useful, it should be possible to combine the observations of multiple testers in order to reconstruct the exact trace of a system. However, if testers are physically separated and no global clock is present, then it may be difficult (or impossible) to rule out alternative behaviours of the system.

Recent work has identified conditions under which tests of distributed systems (modelled as finite state machines) can be designed and controlled to ensure that the exact system behaviour can always be identified, without requiring testers to synchronise their actions externally of the system under test [13,14]. This problem can be stated using our terminology as follows: for every set of interface-level observations $\{\phi_1, \ldots, \phi_n\}$ of a system S projected through views $V_1, \ldots V_n$, there must exist a unique system-level observation Φ of S such that $\mathbf{P}\,(V_i, \Phi) = \phi_i$ for each $i \in 1..n$.

9 Conclusions

We have presented an approach for studying multi-user systems in the UTP. We have described how the observational abilities of users can be modelled as UTP predicates (views) and have identified predicate transformers for calculating the projection of a system's behaviour to its users. The novelty of our approach is that, by codifying these concepts in the UTP explicitly, we obtain a semantically appealing method for reasoning about specifications of multi-user systems across the spectrum of UTP theories.

We have also investigated some alternative notions of refinement that are based on what users can deduce about the system's behaviour from their observations. These refinement relations afford the implementer of a system greater flexibility in making particular design decisions that would be prohibited by classical refinement.

The emphasis of this paper is on generality. Our approach is sufficiently abstract to be used in combination with a variety of UTP theories; for instance, we have demonstrated its application within the theory of designs in Section 5.

Moreover, our definitions of co-operating and independent refinement are not tied to the semantics of particular UTP theories. Indeed, these refinement relations may potentially be applied in practice to the stepwise development of multi-user systems, wherever the observational abilities of users are known.

A drawback of our approach is that applying the **P** and **R** predicate transformers can be tedious and error-prone if carried out manually, because a view may define a complex relation between interface-level and system-level observational variables. With the emergence of tool support for the UTP, there is potential for overcoming this difficulty by mechanising our approach. This would enable some of the techniques described in this paper — such as reasoning about information flow between users, or verifying co-operating or independent refinements over systems — to be carried out with machine assistance.

The focus of our current research is to integrate our approach with the UTP semantics of the *Circus* formalism [15], in order to reason about the observational abilities of users of systems modelled in *Circus*. We envisage the work presented in this paper will provide the foundations for a comprehensive platform for analysing *Circus* systems from the perspective of their users.

Acknowledgements

Michael Banks is supported by a UK Engineering and Physical Sciences Research Council DTA studentship.

We are grateful to Ana Cavalcanti, Frank Zeyda and the anonymous referees for offering insightful comments on this paper, and to Chris Poskitt for proofreading. We also thank Jim Woodcock and the members of the Programming Languages and Systems research group at York for their feedback on an seminar talk on the ideas described in this paper, which helped us to clarify our ideas and to simplify the presentation of technical details.

References

1. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
2. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science. Prentice Hall Inc. (1998)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, New York, NY, USA, ACM (1977) 238–252
4. Jacob, J.L.: Refinement of shared systems. In McDermid, J.A., ed.: The Theory and Practice of Refinement: Approaches to the Development of Large-Scale Software Systems. Butterworths (1989) 27–36
5. Jacob, J.L.: Basic theorems about security. Journal of Computer Security **1**(4) (1992) 385–411
6. Denning, D.E.: Cryptography and Data Security. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA (1982)

7. Jacob, J.L.: On the derivation of secure components. In: Proceedings of the 1989 IEEE Symposium on Security and Privacy, IEEE Computer Society (1989) 242–247
8. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Proceedings of the 1994 IEEE Symposium on Security and Privacy. (1994) 79–93
9. Mantel, H.: A Uniform Framework for the Formal Specification and Verification of Information Flow Security. PhD thesis, Universität Saarbrücken (July 2003)
10. Seehusen, F., Stølen, K.: Information flow security, abstraction and composition. IET Information Security **3**(1) (2009) 9–33
11. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through determinism. In: ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security. Volume 875 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (1994) 33–53
12. Morgan, C.: The shadow knows: Refinement and security in sequential programs. Science of Computer Programming **74**(8) (June 2009) 629–653
13. Chen, J., Hierons, R.M., Ural, H.: Conditions for resolving observability problems in distributed testing. In: Formal Techniques for Networked and Distributed Systems – FORTE 2004. Volume 3235 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 229–242
14. Chen, J., Hierons, R.M., Ural, H.: Overcoming observability problems in distributed test architectures. Information Processing Letters **98**(5) (June 2006) 177–182
15. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing **21**(1) (February 2009) 3–32

A Proofs

Lemma 1

Proof. We assume that V_1 and V_2 are **VH**-healthy individually. To prove that $V_1 \wedge V_2$ is **VH**-healthy, it is sufficient to show that $V_1 \wedge V_2$ is **VH1**-healthy and **VH2**-healthy separately.

$$
\begin{array}{ll}
& \mathbf{VH1}(V_1) \wedge \mathbf{VH1}(V_2) \\
\Leftrightarrow & \quad \langle \text{ definition of } \mathbf{VH1} \rangle \\
& (\forall\, s, s' \bullet \exists\, u_1, u_1' \bullet V_1) \wedge (\forall\, s, s' \bullet \exists\, u_2, u_2' \bullet V_2) \\
\Leftrightarrow & \quad \langle \text{ predicate calculus} \rangle \\
& \forall\, s, s' \bullet (\exists\, u_1, u_1' \bullet V_1) \wedge (\exists\, u_2, u_2' \bullet V_2) \\
\Leftrightarrow & \quad \langle \text{ assumption: } V_1 \text{ and } V_2 \text{ are disjoint} \rangle \\
& \forall\, s, s' \bullet \exists\, u_1, u_1', u_2, u_2' \bullet V_1 \wedge V_2 \\
\Leftrightarrow & \quad \langle \text{ definition of } \mathbf{VH1} \rangle \\
& \mathbf{VH1}(V_1 \wedge V_2)
\end{array}
$$

The proof that $V_1 \wedge V_2$ is **VH2**-healthy follows similarly. □

Lemma 2

$S \sqsubseteq T \Rightarrow \mathbf{P}(V, S) \sqsubseteq \mathbf{P}(V, T)$

Proof.

$S \sqsubseteq T$
$\Leftrightarrow$ $\langle$ definition of $\sqsubseteq$ $\rangle$
$[\forall s, s' \bullet T \Rightarrow S]$
$\Rightarrow$ $\langle$ predicate calculus $\rangle$
$[\forall s, s' \bullet (V \wedge T) \Rightarrow (V \wedge S)]$
$\Rightarrow$ $\langle$ predicate calculus $\rangle$
$[(\exists s, s' \bullet V \wedge T) \Rightarrow (\forall s, s' \bullet V \wedge S)]$
$\Rightarrow$ $\langle$ predicate calculus $\rangle$
$[(\exists s, s' \bullet V \wedge T) \Rightarrow (\exists s, s' \bullet V \wedge S)]$
$\Leftrightarrow$ $\langle$ definition of $\mathbf{P}$ $\rangle$
$[\mathbf{P}(V, T) \Rightarrow \mathbf{P}(V, S)]$
$\Leftrightarrow$ $\langle$ definition of $\sqsubseteq$ $\rangle$
$\mathbf{P}(V, S) \sqsubseteq \mathbf{P}(V, T)$ $\quad \Box$

Lemma 3

$\mathbf{P}\left(\left(\bigwedge_{i \in 1..n} V_i\right), S\right) \Rightarrow \bigwedge_{i \in 1..n} \mathbf{P}(V_i, S)$

Proof.

$\mathbf{P}\left(\left(\bigwedge_{i \in 1..n} V_i\right), S\right)$
$\Leftrightarrow$ $\langle$ definition of $\mathbf{P}$ $\rangle$
$\exists s, s' \bullet \left(\bigwedge_{i \in 1..n} V_i\right) \wedge S$
$\Rightarrow$ $\langle$ predicate calculus $\rangle$
$\bigwedge_{i \in 1..n} \exists s, s' \bullet V_i \wedge S$
$\Leftrightarrow$ $\langle$ definition of $\mathbf{P}$ $\rangle$
$\bigwedge_{i \in 1..n} \mathbf{P}(V_i, S)$ $\quad \Box$

Theorem 1

$U \sqsubseteq \mathbf{P}(V, S) \Leftrightarrow \mathbf{R}(V, U) \sqsubseteq S$

Proof.

$U \sqsubseteq \mathbf{P}(V, S)$
$\Leftrightarrow$ $\langle$ definition of $\sqsubseteq$ and $\mathbf{P}$ $\rangle$
$[(\exists s, s' \bullet V \wedge S) \Rightarrow U]$

$\Leftrightarrow$ $\langle$ predicate calculus $\rangle$

$[\,(\forall\, s, s' \bullet \neg\ V \vee \neg\ S) \vee U\,]$

$\Leftrightarrow$ $\langle$ s, s' not free in U and s, s' covered by universal closure $\rangle$

$[\,\neg\ V \vee \neg\ S \vee U\,]$

$\Leftrightarrow$ $\langle$ u, u' not free in S and u, u' covered by universal closure $\rangle$

$[\,\neg\ S \vee (\forall\, u, u' \bullet \neg\ V \vee U)\,]$

$\Leftrightarrow$ $\langle$ predicate calculus $\rangle$

$[\,S \Rightarrow (\forall\, u, u' \bullet\ V \Rightarrow U)\,]$

$\Leftrightarrow$ $\langle$ definition of $\sqsubseteq$ and $\mathbf{R}$ $\rangle$

$\mathbf{R}\,(V, U) \sqsubseteq S$ $\quad \Box$

Lemma 4

$\mathbf{P}\,(V, P \vdash Q) = (\forall\, s, s' \bullet\ V \Rightarrow P) \vdash_V \mathbf{P}\,(V, Q)$

Proof.

$\mathbf{P}\,(V, P \vdash Q)$

$\Leftrightarrow$ $\langle$ definition of $\mathbf{P}$ and $\vdash$ $\rangle$

$\exists\, s, s' \bullet\ V \wedge (ok \wedge P \Rightarrow ok' \wedge Q)$

$\Leftrightarrow$ $\langle$ unfold implication $\rangle$

$\exists\, s, s' \bullet\ V \wedge (\neg\ ok \vee \neg\ P \vee (ok' \wedge Q))$

$\Leftrightarrow$ $\langle$ distributivity, twice $\rangle$

$(\exists\, s, s' \bullet\ V \wedge (\neg\ ok \vee \neg\ P)) \vee (\exists\, s, s' \bullet\ ok' \wedge\ V \wedge\ Q)$

$\Leftrightarrow$ $\langle$ V is **VHD**-healthy $\rangle$

$(\neg\ ok_V \vee \exists\, s, s' \bullet\ V \wedge \neg\ P) \vee (ok'_V \wedge \exists\, s, s' \bullet\ V \wedge\ Q)$

$\Leftrightarrow$ $\langle$ de Morgan, twice $\rangle$

$\neg\ (ok_V \wedge \forall\, s, s' \bullet \neg\ (V \wedge \neg\ P)) \vee (ok'_V \wedge \exists\, s, s' \bullet\ V \wedge\ Q)$

$\Leftrightarrow$ $\langle$ predicate calculus $\rangle$

$(ok_V \wedge (\forall\, s, s' \bullet\ V \Rightarrow P)) \Rightarrow (ok'_V \wedge \exists\, s, s' \bullet\ V \wedge\ Q)$

$\Leftrightarrow$ $\langle$ definition of $\mathbf{P}$ and $\vdash_V$ $\rangle$

$(\forall\, s, s' \bullet\ V \Rightarrow P) \vdash_V \mathbf{P}\,(V, Q)$ $\quad \Box$

Theorem 2

Proof. Equation 17 is a consequence of Lemma 2 and Definition 8. Equation 18 follows from Lemma 3. $\Box$