Model Checking and UTP Design Verification

Hugh Anderson and Gabriel Ciobanu

National University of Singapore
School of Computing
Department of Computer Science
hugh,gabriel@comp.nus.edu.sg

Abstract We give a different perspective on verification of programs. Our perspective emphasizes the use of design verification in the unified theory of programming. The main idea is that of applying model checking to the verification of programs expressed in the pre and postcondition style of the unified theory of programming, leading to a closer relationship between program development and program verification. In particular, we derive a model using the concept of the UTP *design*, and express state and temporal properties as relations between observations. A model checking relation is derived from a satisfaction relation between the model and its properties.

Keywords: Symbolic Model Checking, Unifying Theories of Programming.

1 Introduction

In their book "Unifying Theories of Programming", Hoare and He develop a consistent theory of computing, based on a relational calculus [5]. The book introduces the relational calculus, and uses it to develop concepts of program design, refinement of programs, and an algebra of programming. The work has been adopted by many researchers working in various areas of computing. It has been used to formally define the semantics of a wide-spectrum programming/specification language in [11]. Woodcock and Hughes develop a refinement calculus for parallel programming in [10]. Sherif and He have explored a time model in [8]. In this paper, we use the unified theory of programming (UTP) for verification of programs in a model checking style.

The approach taken in the book is to formalize and characterize a class of relations useful for program development. One such class of relations is called a "**design**", and encompasses in one notation both *specification* and *implementation*. In UTP both specification and implementation are seen as instances of designs, with a clear link between them. A particular program development may be characterized by a progression of designs from more abstract designs (specifications) to more concrete ones (implementations).

The relation between a specification and an implementation for a specific program development is the *refinement* relation of [1], which is mimicked in UTP by an implication relation between UTP designs.

Proceedings of the 1st South-East European
Workshop on Formal Methods, SEEFM'03
November 2003, Thessaloniki, Greece

In this paper, we outline how to express

- models and states of the implementation which we are verifying;
- properties that we wish to check;
- the *model checking* relation $\models$.

We take the view that a model may be derived directly from a UTP *design*, and that if a property is true for a particular UTP design, then it is also true for the related implementations (derived UTP designs).

In the model checking world, a property is normally expressed as a *state* assertion or a *temporal* assertion. We show how to express state and temporal properties as a relation between observations over a model, and then define a model checking relation between these properties and the model.

We assume the reader is familiar with the ideas and notations used in model checking. Before considering the use of the unifying theories in model checking, we consider in Section 2 how UTP concepts and notations are used to represent some computing structures in a unified manner. Section 3 presents a simple example used in the rest of the paper, and some key points about model checking. In Section 4, model checking is reformulated within the unifying theory framework, showing how model checking in UTP may be done using binary decision diagrams, closely following the approach used for symbolic model checking. Finally we conclude with remarks about the links between traditional model checking and the proposed design verification.

2 Unifying theories of programming

The components of the unified theory of programming are *alphabets*, *signatures* and *healthiness conditions*.

The *alphabet* is just a set of names representing observations that may be made about the program. These names may include relations between program variables, and also between other variables not mentioned in the program text, such as:

- `ok` indicating that the program has started;
- `ok'` indicating that the program has terminated.

In general, the unprimed names indicate the observations *before* the program execution, and the primed ones indicate observations *afterwards*.

The *signature* of a theory is a set of primitive operators and constants of the theory, and the syntax used to combine them. In UTP, we assume the signature of predicates, and extend them with new operators as needed. For example, in developing the concept of a *design* in the unifying theories, the connective $\vdash$ is introduced to indicate the relationship between a pair of predicates p (for *preconditions*) and q (for *postconditions*):

$$p \vdash q \;\widehat{=}\; (\mathrm{ok} \wedge p) \Rightarrow (\mathrm{ok}' \wedge q)$$

The *healthiness conditions* select subtheories from a theory. For example, we may be able to specify all sorts of programs from a particular alphabet and signature, but we

are only interested in programs that may be physically realized. For example, we can make no observations about a program that has not yet started running. This may be expressed as a healthiness condition for any observation o about a program:

$$o = (\mathrm{ok} \Rightarrow o)$$

This condition selects a subtheory of all the possible programs, isolating those that are implementable if the healthiness condition is true.

We have already seen how the concept of a *design* is presented in UTP. Many programming concepts may be easily represented. For example, the Hoare triple $\{p\}\, C\, \{q\}$ representing the relation between a precondition p, a code segment C and a postcondition q, is defined in UTP as a relation between three predicates p, C and q:

$$\{p\}\, C\, \{q\} \;\widehat{=}\; [C \Rightarrow (p \Rightarrow q')]$$

The square brackets are a notational convenience, indicating that the enclosed expression is universally quantified over all variables of its alphabet.

This section gave the flavour of UTP designs, and shows the relational calculus expressing the notion of the Hoare triple. More details may be found in [5]. In the following sections we given an example model, and show the representation of model checking concepts within UTP.

3 Example model and model checking

Verification of software may be viewed as the process of checking some property P against either the software, or a model $\mathcal{M}$ of the software. For software in general, this is a hard problem, as the verification process may involve in-depth reasoning, perhaps requiring theorem provers to confirm parts of the verification.

The model checking approach to verification [3,7] is to abstract out key elements of the software and to verify just these elements. Various techniques and structures have been developed to automatically and efficiently check the abstract elements against specified properties. As an example, we may be interested in checking that a certain program variable v has the value 0 at a certain stage of the execution of our program. In this case, the predicate $v = 0$ is checked against a representation (model) of the program which indicates how such predicates are transformed.

Given the underlying reliance on binary abstractions, it is no surprise that model checking is being used in the analysis of digital electronic circuits, but it has also proved effective in the software domain, particularly in the areas of protocol analysis, the behaviour of reactive systems, and for checking concurrent systems.

Consider the circuit in Figure 1. This circuit has three logic gates (two inverters and an AND gate) connected together. Three probe points in the circuit are identified as x, y and z. In our modelling technique, we ignore considerations such as time delays in the interconnections or indeterminate logic levels, and view these probe points as variables that range over **true**/**false** (or 1/0, or +5V/0V). In the following discussion, we will write **false** as 0, **true** as 1, and we assume that the circuit is synchronous - that is, all changes of state in the circuit occur at the same time.

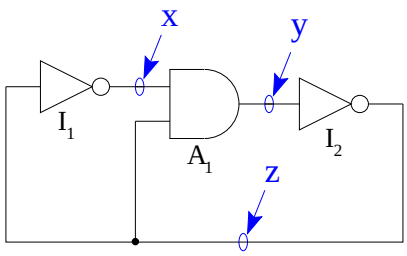

Figure 1. Simple electronic circuit

We can represent a state s_i of this circuit at any time i as a valuation of the probe points (x, y, z). The initial state s_0 for the circuit is defined to be $(1, 0, 0)$. $\mathcal{S}$ denotes the set of all states. If the points change simultaneously according to the logic gates, we would see the following successive states:

State $\mathcal{S}$	x	y	z
s_0	1	0	0
s_1	1	0	1
s_2	0	1	1
s_3	0	0	0
s_4	Cycle repeats, as $s_4 = s_1$		

The logic gates impose a relation between the probe points in successive states, each new value (x', y' and z') being dependant on the previous values of x, y and z. Though there are eight possible combinations of values of the probe points, given the specified starting state, the circuit has only four *reachable* states.

The transition relation may be expressed as a predicate using the variables (x, y, z, x', y' and z') and given in disjunctive normal form:

$$\begin{aligned} t = {} & (x \wedge \bar{y} \wedge \bar{z} \wedge x' \wedge \bar{y}' \wedge z') \\ & \vee (x \wedge \bar{y} \wedge z \wedge \bar{x}' \wedge y' \wedge z') \\ & \vee (\bar{x} \wedge y \wedge z \wedge \bar{x}' \wedge \bar{y}' \wedge \bar{z}') \\ & \vee (\bar{x} \wedge \bar{y} \wedge \bar{z} \wedge x' \wedge \bar{y}' \wedge z') \end{aligned}$$

Any predicate may also be encoded as a binary decision tree (BDT), in which the levels denote the different variables, and paths through the tree represent valuations of the transition relation. Note that if we reorder the variables, we get a different decision tree, but this new tree still represents our transition relation. In other words, the relation is independent of the order of the variables.

The binary decision tree does not scale well, but there are optimizations that may be done. An optimization to exploit repetition on BDTs leads to Binary Decision Diagrams (BDDs) to represent the relation. BDDs provide a canonical form for the BDTs.

In summary, within the traditional presentation, a model is a finite state transition system $\mathcal{M} = (S, \mathcal{R}, V)$, where S represents a finite set of states, $\mathcal{R}$ represents a transition relation given as a set of pairs of states, and V is a valuation function defining the truth values for each state.

3.1 Model checking

In model checking, desired properties are given as state and temporal formulæ in modal logics such as Computation Tree Logic (CTL), or Linear Temporal Logic (LTL). These temporal logics are propositional languages with modal operators and quantifiers related to time, and each is a sub-logic of CTL*, which in turn is a sublogic of the μ-Calculus [4].

A CTL formula may either be a *state* formula or a *path* formula. A *state* formula is one which is true in a particular state, whereas a *path* formula is one which is true along a particular computation path.

There are several different notations used to express CTL expressions, however each notation expresses the same concept. For example in CTL, the path quantifiers $\forall$ (all computation paths) and $\exists$ (at least one computation path) are also found as the letters **A** and **E**. In addition, the path operators $\Diamond$ (at some future time) and $\Box$ (at all future times) are also found as the letters **F** and **G**. The operators **X** (in the next state), **U** (one property holds until another holds) and **R** (the dual of **U**) complete the set. We will use the *letter* notation used in the model checking literature.

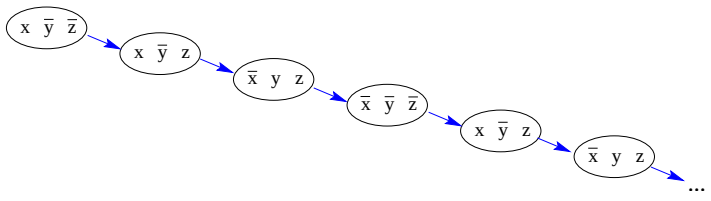

Figure 2. An unfolded state transition diagram

The CTL expressions are used to express properties of unfolded state transition diagrams such as the one for our example, seen in Figure 2. Note that a more complex model will have a possibly infinite *tree* of states, rooted at an initial state.

CTL expressions require every temporal operator to be preceded by a quantifier. Since there are five temporal operators, and two quantifiers, we have ten base expression types, but all of these may be expressed in terms of just three expressions:

- **EX** p : For one computation path, property p holds in the next state;
- **EG** p : For one computation path, property p holds at every state;
- **E**[p **U** q] : For one computation path, property p holds until q holds.

Model checking is commonly expressed as a ternary relation ($\models$):

$$\mathcal{M}, s \models P$$

The relation is true when the property P holds in state s for a given model $\mathcal{M}$. The relation is normally defined inductively, with a set of interlocking rules for state and path formulæ. A labelling algorithm may then be used to establish the set of states satisfying the relation. However, this approach is not particularly efficient, in terms of the size of the structure used.

A more efficient technique relies on representing the relation as a BDD, and constructing a checking procedure for the CTL. The checking procedure returns a BDD structure which represents the states that satisfy the formula. This technique is efficient as operations on BDDs are relatively efficient [2].

4 Model checking in UTP

In the UTP theory of model checking explored here, we begin by defining the notions of *model* and *state*, and the *property* to be checked for that model. We show how UTP designs may be used to create an appropriate model, and work through the same circuit diagram used in Section 3.

Model checking has been successfully integrated with other formalisms [6], where CTL models are derived directly from object specifications. A feature of the UTP model checking approach is that the checking is performed using a transition relation derived *directly* from the UTP *design*.

4.1 UTP notions of model and state

In this section we show how the model is derived directly from a UTP *design*. A UTP design expresses the relation between a pair of predicates representing the preconditions (assumptions) and postconditions (commitments) for a program. This relation is expressed as a predicate with unprimed state variables representing key observations over the program before the program starts, and primed variables standing for the values when the program terminates.

Since a UTP design is already expressed as a predicate it is relatively easy to derive the transition relation t of the previous section. If we consider our electronic circuit, we might express an initial design for this circuit in UTP terms as the parallel composition of three components:

$$\begin{aligned} I_1 &\mathrel{\widehat{=}} \text{true} \vdash x' = \bar{z} \\ I_2 &\mathrel{\widehat{=}} \text{true} \vdash z' = \bar{y} \\ A_1 &\mathrel{\widehat{=}} \text{true} \vdash y' = x \wedge z \\ \text{PPTransformer} &\mathrel{\widehat{=}} I_1 \parallel I_2 \parallel A_1 \end{aligned}$$

Since the output alphabets of each of the components are disjoint, the parallel composition of the components is just the conjunction of the pre and post-conditions:

$$\text{PPTransformer} \mathrel{\widehat{=}} \text{true} \vdash (x' = \bar{z}) \wedge (z' = \bar{y}) \wedge (y' = x \wedge z)$$

This design corresponds to the transition system for the circuit, and expresses a *probe-point* transformer which, when given values for x, y and z, returns the new primed values. However, we are more interested in the sequence of states when our circuit runs continuously, and so we define a recursive design which mimics the circuit precisely:

$$\text{Circuit} \mathrel{\widehat{=}} \text{PPTransformer}; \text{Circuit}$$

We can view this design in an operational sense as a predicate transformer, transforming an *observation consistent with the assumptions of the design*, into a new *observation consistent with the commitments of the design*. If we begin with an initial observation over the state variables $(x \wedge \bar{y} \wedge \bar{z})$ matching state s_0 from Section 3, then the PPTransformer asserts that the commitment will be $x \wedge \bar{y} \wedge z$, corresponding to the primed observations. Operationally, we could view this as a transition from $s_0 = x \wedge \bar{y} \wedge \bar{z}$ to $s_1 = x \wedge \bar{y} \wedge z$.

In the following presentation, the set S of program states represents a set of valuations of all observations of the program. Formally speaking, we start with an alphabet A of observation variables. In our case $A = \{x, y, z\}$. These observation variables are evaluated according to a valuation function $v : A \to \{\text{true}, \text{false}\}$. We use $\overline{x}$ to express that $v(x) = \text{false}$, and x for $v(x) = \text{true}$.

An observation $o \in O$ is a conjunction of valuation values expressed as x or $\bar{x} \wedge z$, for instance. The observation $\bar{x} \wedge z$ is sometimes written as $\bar{x}z$ for short, and so we may write expressions such as $\bar{x}z \wedge \bar{x}y\bar{z}$, or $(o \wedge s_0) = o$, where o is an observation, and s_0 is a state. Each state is an observation, but there are observations that are not states. Later in this presentation, when we consider a relation consisting of pairs of observations, this is a more general, and larger set than a transition relation which is a set of pairs of states.

The set S of states is given by all possible valuations over all observation variables. We write individual states in S as s_n or in shorthand as a string of observation values. In our case, $S = \{xyz, xy\bar{z}, x\bar{y}z, x\bar{y}\bar{z}, \bar{x}yz, \bar{x}y\bar{z}, \bar{x}\bar{y}z, \bar{x}\bar{y}\bar{z}\}$.

To derive a transition relation for the design, we can apply the PPTransformer to the initial state $s_0 = x\bar{y}\bar{z}$, collecting the original and transformed state variables as a transition, and repeating with the new transformed variables until no new transitions are returned:

Transition	**Original**	**Transformed**
r_0	$s_0 = x\bar{y}\bar{z}$	$s_1 = x\bar{y}z$
r_1	$s_1 = x\bar{y}z$	$s_2 = \bar{x}yz$
r_2	$s_2 = \bar{x}yz$	$s_3 = \bar{x}\bar{y}\bar{z}$
r_3	$s_3 = \bar{x}\bar{y}\bar{z}$	$s_1 = x\bar{y}z$
r_4	No new transitions, as $r_4 = r_1$	

The transition relation for the Circuit design is expressed as a set of pairs of states in the usual way:

$$t = \{(x\bar{y}\bar{z}, x\bar{y}z), (x\bar{y}z, \bar{x}yz), (\bar{x}yz, \bar{x}\bar{y}\bar{z}), (\bar{x}\bar{y}\bar{z}, x\bar{y}z)\}$$

Note that this corresponds exactly to the relation in Section 3. This *set-of-pairs* notation will be used in Section 4.3 where we define model checking in UTP.

4.2 Specification of properties in UTP

In UTP terms, a property P is an expression whose elements are observations, and whose connectives are simple state ones ($\wedge$, $\vee$ and $\neg$) or the more complex temporal ones such as **G**, **X** and **U**.

For example, we may be interested in the property "x will be true until $\bar{y}$". This property would be written as x **U** $\bar{y}$.

4.3 Model checking in UTP

We begin with some auxiliary functions which are used to build a satisfaction function. The function $\mathbf{map} : 2^{\mathcal{R}} \times S \rightarrow 2^{\mathcal{R}}$ (where $2^{\mathcal{R}}$ is the set of all subsets of $\mathcal{R}$) takes as arguments a transition relation r and a state s and returns a subrelation in which each element has the state s as a first component of the transition relation pair:

$$\mathbf{map}(r, s) \mathrel{\widehat{=}} \{(s_1, s_2) \in r \mid s_1 = s\}$$

The function $\mathbf{tmap} : 2^{\mathcal{R}} \times O \rightarrow 2^{\mathcal{R}}$ takes as arguments a transition relation r and an observation o and returns a subrelation of r in which the observation o is "included" in the first component of the transition relation pair:

$$\mathbf{tmap}(r, o) \mathrel{\widehat{=}} \{(s_1, s_2) \in r \mid (o \wedge s_1) = o\}$$

In this definition $(o \wedge s_1) = o$ expresses the fact that the observation o is part of the conjunction provided by s_1. This is why we use the phrase "o is included in the first component". For instance,

$$\mathbf{tmap}(\{(x\bar{y}\bar{z}, x\bar{y}z), (x\bar{y}z, \bar{x}yz), (\bar{x}yz, \bar{x}\bar{y}\bar{z}), (\bar{x}\bar{y}\bar{z}, x\bar{y}z)\}, y) = \{(\bar{x}yz, \bar{x}\bar{y}\bar{z})\}$$

The function $\mathbf{fmap} : 2^{\mathcal{R}} \times O \rightarrow 2^{\mathcal{R}}$ takes a transition relation r and an observation o and returns a subrelation of the complement $\bar{r}$ of r, in which the observation o is included in the first component of the transition relation pair:

$$\mathbf{fmap}(r, o) \mathrel{\widehat{=}} \{(s_1, s_2) \in \bar{r} \mid (o \wedge s_1) = o\}$$

An example of **fmap** using our relation t is large, as the complement of t has 60 pairs, but the interested reader may wish to confirm that

$$\begin{aligned}\mathbf{fmap}(t, \bar{x}\bar{y}\bar{z}) = \{&(\bar{x}\bar{y}\bar{z}, \bar{x}\bar{y}\bar{z}), (\bar{x}\bar{y}\bar{z}, \bar{x}\bar{y}z), (\bar{x}\bar{y}\bar{z}, \bar{x}y\bar{z}), (\bar{x}\bar{y}\bar{z}, \bar{x}yz),\\ &(\bar{x}\bar{y}\bar{z}, x\bar{y}\bar{z}), (\bar{x}\bar{y}\bar{z}, xy\bar{z}), (\bar{x}\bar{y}\bar{z}, xyz)\}\end{aligned}$$

Given two pairs of observations $o = (o_1, o_2)$, and $o' = (o'_1, o'_2)$, we define

$$o \lhd o' \mathrel{\widehat{=}} ((o_1 \wedge o'_1) = o_1) \wedge ((o_2 \wedge o'_2) = o_2)$$

We now define a matching function $\blacktriangleleft$: $2^{O \times O} \times 2^{O \times O} \rightarrow 2^{O \times O}$ which takes two sets of pairs of observations and returns a subset of the second one. The sets of pairs of

observations may be used to express transition relations, in which the observations correspond exactly to the states. Given two such relations r_1 and r_2, we define

$$r_1 \blacktriangleleft r_2 \mathrel{\widehat{=}} \{o \in r_2 \mid \exists o' \in r_1 : o' \lhd o\}$$

For example:

$$\{(xz, yz), (\bar{y}z, y)\} \blacktriangleleft \{(x\bar{y}\bar{z}, x\bar{y}z), (x\bar{y}z, \bar{x}yz), (\bar{x}\bar{y}\bar{z}, x\bar{y}z)\} = \{(x\bar{y}z, \bar{x}yz)\}$$

We observe that the first pair of r_1 matches the second pair of r_2, and that the second pair of r_1 also matches the second pair of r_2. As a final result, we get the second pair of r_2.

State formulæ in UTP. We go on to show how to express state formulæ in UTP. When model checking a UTP design, a property P is given as a CTL formula. For state formulæ in UTP, the atomic components are observations, and we use only the non-temporal connectives $\wedge$, $\vee$ and $\neg$.

Given arbitrary properties $p, q \in \mathcal{P}$, and a transition relation $r \in \mathcal{R}$ for the design $\mathcal{D}$, we define a satisfaction mapping function $\mathbf{satmap} : 2^{\mathcal{R}} \times \mathcal{P} \to 2^{\mathcal{R}}$ by:

$$\begin{aligned}
\mathbf{satmap}(r, p) &\mathrel{\widehat{=}} \mathbf{tmap}(r, p) \blacktriangleleft r \\
\mathbf{satmap}(r, \neg p) &\mathrel{\widehat{=}} \mathbf{fmap}(r, p) \blacktriangleleft r \\
\mathbf{satmap}(r, p \wedge q) &\mathrel{\widehat{=}} \mathbf{tmap}(r, p) \blacktriangleleft r \ \cap \ \mathbf{tmap}(r, q) \blacktriangleleft r \\
\mathbf{satmap}(r, p \vee q) &\mathrel{\widehat{=}} \mathbf{tmap}(r, p) \blacktriangleleft r \ \cup \ \mathbf{tmap}(r, q) \blacktriangleleft r
\end{aligned}$$

This function returns a set of "satisfied pairs", a subset of r satisfying the property.

Given a transition relation $r \in \mathcal{R}$ for the design $\mathcal{D}$, the model checking relation $\mathcal{D}, s \models P$ for a property $P \in \mathcal{P}$ in state s is defined by

$$\mathcal{D}, s \models P \mathrel{\widehat{=}} (\mathbf{map}(r, s) \blacktriangleleft \mathbf{satmap}(r, P)) \neq \emptyset$$

That is, we check that the state s belongs to the set of "satisfied pairs". As a worked example of some checks on our design, let us try to see which pairs of t satisfy $x \vee y$ using the approach. We can calculate the satisfaction mapping function:

$$\begin{aligned}
\mathbf{satmap}(t, x \vee y) &\mathrel{\widehat{=}} \mathbf{tmap}(t, x) \blacktriangleleft t \ \cup \ \mathbf{tmap}(t, y) \blacktriangleleft t \\
&= \{(x\bar{y}\bar{z}, x\bar{y}z), (x\bar{y}z, \bar{x}yz)\} \cup \{(\bar{x}yz, \bar{x}\bar{y}\bar{z})\} \\
&= \{(x\bar{y}\bar{z}, x\bar{y}z), (x\bar{y}z, \bar{x}yz), (\bar{x}yz, \bar{x}\bar{y}\bar{z})\}
\end{aligned}$$

Now, for each of the states in the table of Section 4.1, there is a corresponding set of mapping functions:

$$\begin{aligned}
\mathbf{map}(t, s_0) &= \{(x\bar{y}\bar{z}, x\bar{y}z)\} \\
\mathbf{map}(t, s_1) &= \{(x\bar{y}z, \bar{x}yz)\} \\
\mathbf{map}(t, s_2) &= \{(\bar{x}yz, \bar{x}\bar{y}\bar{z})\} \\
\mathbf{map}(t, s_3) &= \{(\bar{x}\bar{y}\bar{z}, x\bar{y}z)\}
\end{aligned}$$

We can now calculate the model checking relation in each state:

$$\mathcal{D}, s_0 \models x \vee y = (\mathbf{map}(t, s_0) \blacktriangleleft \mathbf{satmap}(t, x \vee y)) \neq \emptyset = \mathbf{true}$$
$$\mathcal{D}, s_1 \models x \vee y = (\mathbf{map}(t, s_1) \blacktriangleleft \mathbf{satmap}(t, x \vee y)) \neq \emptyset = \mathbf{true}$$
$$\mathcal{D}, s_2 \models x \vee y = (\mathbf{map}(t, s_2) \blacktriangleleft \mathbf{satmap}(t, x \vee y)) \neq \emptyset = \mathbf{true}$$
$$\mathcal{D}, s_3 \models x \vee y = (\mathbf{map}(t, s_3) \blacktriangleleft \mathbf{satmap}(t, x \vee y)) \neq \emptyset = \mathbf{false}$$

This may be confirmed by examining the table in Section 4.1, and noting that only state s_3 has neither x nor y.

Temporal formulæ in UTP. For temporal formulæ in UTP, we use the temporal connectives **EX**, **EG** and **EU**. We begin with a pair of temporal functions (F, P) forming a Galois connection, and representing *future* and *past* respectively. In [9], Karger and Hoare demonstrate how these functions may be used to express temporal relations. The interested reader is also directed to the section on Galois connections in [5]. F_r represents the *future* function for the transition relation r, which returns a relation r' with r in its immediate future. Using the running example of transition relation t, we have that

$$F_t(\{(\bar{x}yz, \bar{x}\bar{y}\bar{z})\}) = \{(x\bar{y}z, \bar{x}yz)\}$$
$$F_t(\{(x\bar{y}z, \bar{x}yz)\}) = \{(x\bar{y}\bar{z}, x\bar{y}z), (\bar{x}\bar{y}\bar{z}, x\bar{y}z)\}$$

In the first example, the pair $(\bar{x}yz, \bar{x}\bar{y}\bar{z})$ represents the transition at the lower right from state s_2 to state s_3, and the result of the F_t function is the transition from state s_1 to s_2. The second example returns the two transitions from state s_0 to s_1 and from state s_3 to s_1.

P_{rq} represents the *past* function for the relation r, which returns a relation r' with r in its past, and which satisfies **EX** q. Using the running example of transition relation t, and using the observation $q = \bar{y}$ we have that

$$P_{tq}(t) = \{(x\bar{y}z, \bar{x}yz), (\bar{x}yz, \bar{x}\bar{y}\bar{z})\}$$

The results of the P_{tq} function consist of the transition leading from state s_1 to s_2, and the transition leading from state s_2 to s_3. In each case, if we take the *next* transition, the observation q will be true.

The CTL temporal connectives may be characterized as the fixed point of the temporal functions F_r and P_{rq} [7]. We define the following two fixpoint operators:

$$F^{\Diamond}(X) \mathrel{\widehat{=}} (\nu Y \bullet X \sqcap F(Y))$$
$$P^{\Box}(X) \mathrel{\widehat{=}} (\mu Y \bullet X \sqcup P(Y))$$

Given properties $p, q \in \mathcal{P}$, and a transition relation $r \in \mathcal{R}$ for a design $\mathcal{D}$, we can define the satisfaction mapping function **satmap** for CTL *temporal* formulæ by induction. For a start, let us express a satisfaction function for the required temporal formulæ of Section 3.1 in terms of the temporal functions and their fixpoints:

$$\mathbf{satmap}(r, \mathbf{EX}\ p) \mathrel{\widehat{=}} F_r(\mathbf{satmap}(r, p))$$
$$\mathbf{satmap}(r, \mathbf{EG}\ p) \mathrel{\widehat{=}} F_r^{\Diamond}(\mathbf{satmap}(r, p))\,\{r\}$$
$$\mathbf{satmap}(r, \mathbf{E}[p\ \mathbf{U}\ q]) \mathrel{\widehat{=}} P_{rq}^{\Box}(\mathbf{satmap}(r, p))\,\emptyset$$

The satisfaction function for the $\mathbf{EG}\ p$ temporal formula uses the *future* function iterator $F_r^{\Diamond}$, the greatest fixed point of F_r. The satisfaction function for the $\mathbf{E}[p\ \mathbf{U}\ q]$ temporal formula uses the *past* function iterator $P_{rq}^{\Box}$, the least fixed point of P_{rq}. These functions provide a high-level description of the satisfaction function for the temporal formulæ, but not many clues in how to implement the functions. A lower level description, closer to our approach, requires an appropriate function **backstep** to implement the core of the F_r function:

$$\mathbf{backstep}(r, s) \mathrel{\widehat{=}} \{(s_1, s_2) \in r \mid \forall a \in s : a = s_2\}$$

The function $\mathbf{backstep} : 2^{\mathcal{R}} \times 2^S \to 2^{\mathcal{R}}$ takes a transition relation r together with a set of states, and returns a subrelation of r in which each pair has its second element drawn from the set of states. We can view this as a backwards step in the transition relation. If we have a set of states s which satisfy a property, then we can take a backwards step, and determine the transitions leading to those states. The iterate $P_{rq}^{\Box}$ may be expressed without requiring a new function.

Proposition 1. *The satisfaction function for our temporal formulæ may be expressed in terms of the lower-level operators in the following way:*

$$\mathbf{satmap}(r, \mathbf{EX}\ p) \equiv \{(s_1, s_2) \in r \mid \mathbf{satmap}(\mathbf{backstep}(r, s_1), p) \neq \emptyset\}$$

$$\begin{aligned}\mathbf{satmap}(r, \mathbf{EG}\ p) &\equiv \mathbf{satmap}(r, p)\ \cap\ \mathbf{satmap}(r, \mathbf{EX}\ \mathbf{EG}\ p) \\ &\equiv \textstyle\bigcap_i (\lambda y.(\mathbf{satmap}(r, p)\ \cap\ \mathbf{satmap}(y, \mathbf{EX}\ p)))^i\, \{r\}\end{aligned}$$

$$\begin{aligned}&\mathbf{satmap}(r, \mathbf{E}[p\ \mathbf{U}\ q]) \equiv \\ &\quad \equiv \mathbf{satmap}(r, q)\ \cup\ (\mathbf{satmap}(r, p)\ \cap\ \mathbf{satmap}(r, \mathbf{EX}\ \mathbf{E}[p\ \mathbf{U}\ q])) \\ &\quad \equiv \textstyle\bigcup_i (\lambda y.(\mathbf{satmap}(r, q)\ \cup\ (\mathbf{satmap}(r, p) \cap \mathbf{satmap}(y, \mathbf{EX}\ q))))^i\, \emptyset\end{aligned}$$

Proof. We provide here the intuition behind the technical details of the proof. Firstly, the $\mathbf{satmap}(r, \mathbf{EX}\ p)$ function definition just given returns those transitions from the transition relation which lead to a state in which the property p holds. Secondly, the intuition behind the $\mathbf{satmap}(r, \mathbf{EG}\ p)$ function definition is that it returns those transitions from the transition relation which always involve a state in which the property p holds. This is done by iteration, starting from the entire transition relation until we reach the greatest fixed point. Finally, the intuition behind the $\mathbf{satmap}(r, \mathbf{E}[p\ \mathbf{U}\ q])$ function definition is that it returns those transitions from the transition relation which have p holding until the property q holds. Again, this is done by iteration, starting from an empty transition relation until we reach the least fixed point.

As a worked example to check a temporal formula against our model, we calculate which states satisfy $\mathbf{EG}\ \bar{x} \vee z$. A particular set of transitions appears over and over again in the evaluation of $\mathbf{EG}\ \bar{x} \vee z$, so in order to keep things within the margins of the page, we will define r_1 as:

$$r_1 = \{(x\bar{y}z, \bar{x}yz), (\bar{x}yz, \bar{x}\bar{y}\bar{z}), (\bar{x}\bar{y}\bar{z}, x\bar{y}z)\}$$

We begin by calculating the satisfaction function $\mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z)$.

$$
\begin{aligned}
\mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z) &\mathrel{\widehat{=}} F_t^{\Diamond}(\mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z))\,\{t\} \\
&\equiv \textstyle\bigcap_i (\lambda y.(\mathbf{satmap}(t, \bar{x} \vee z) \cap \mathbf{satmap}(y, \mathbf{EX}\ \bar{x} \vee z)))^i \{t\} \\
&= (r_1 \cap \mathbf{satmap}(t, \mathbf{EX}\ \bar{x} \vee z)) \cap \ldots \\
&= r_1 \cap (r_1 \cap \mathbf{satmap}(r_1, \mathbf{EX}\ \bar{x} \vee z)) \cap \ldots \\
&= r_1 \cap r_1 \cap (r_1 \cap \mathbf{satmap}(r_1, \mathbf{EX}\ \bar{x} \vee z)) \cap \ldots \\
&= r_1
\end{aligned}
$$

Again, for each of our states, we can calculate the model checking relation:

$$
\begin{aligned}
\mathcal{D}, s_0 \models \mathbf{EG}\ \bar{x} \vee z &= (\mathbf{map}(t, s_0) \blacktriangleleft \mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z) \neq \emptyset = \mathbf{false} \\
\mathcal{D}, s_1 \models \mathbf{EG}\ \bar{x} \vee z &= (\mathbf{map}(t, s_1) \blacktriangleleft \mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z) \neq \emptyset = \mathbf{true} \\
\mathcal{D}, s_2 \models \mathbf{EG}\ \bar{x} \vee z &= (\mathbf{map}(t, s_2) \blacktriangleleft \mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z) \neq \emptyset = \mathbf{true} \\
\mathcal{D}, s_3 \models \mathbf{EG}\ \bar{x} \vee z &= (\mathbf{map}(t, s_3) \blacktriangleleft \mathbf{satmap}(t, \mathbf{EG}\ \bar{x} \vee z) \neq \emptyset = \mathbf{true}
\end{aligned}
$$

This may be confirmed by examining the table in Section 4.1, and noting that $\bar{x} \vee z$ is true only if we begin in s_1, s_2 or s_3.

4.4 Model checking with BDDs

The presentation of a UTP model checking theory in this section is more program oriented than the traditional presentation of BDD-style model checking. Our presentations of transition relations, state and properties do not rely on any particular ordering of the variables, but it is easy to extend the approach by imposing a discipline on the representation of the model and state. This discipline does not amount to a healthiness condition for the theory. It does not produce a subtheory: the UTP theory of model checking is equivalent to the UTP theory of BDD-style model checking.

The discipline is imposed by using strings to represent observations, states and transition relations. Each element in the string corresponds to an observation, and must occur in a specific order. By using corresponding $\mathbf{map}_{\mathrm{BD}}$ and $\mathbf{tmap}_{\mathrm{BD}}$ functions, and with a new matching relation $\blacktriangleleft_{\mathrm{BD}}$ operating over strings, we can closely follow the structures and algorithms used for symbolic model checking.

The presentation in this section is necessarily brief, and we appeal to the use of analogy rather than redefining at length all the new functions. Each function or operator in this presentation works in an analogous way to those in Section 4.3, except that they now operate over strings and sets of strings, rather than over observations, pairs of observations, and sets of pairs of observations:

$$
\begin{aligned}
\mathbf{satmap}_{\mathrm{BD}}(r, p) &\mathrel{\widehat{=}} \mathbf{tmap}_{\mathrm{BD}}(r, p) \blacktriangleleft_{\mathrm{BD}} r \\
\mathbf{satmap}_{\mathrm{BD}}(r, \neg p) &\mathrel{\widehat{=}} \mathbf{fmap}_{\mathrm{BD}}(r, p) \blacktriangleleft_{\mathrm{BD}} r \\
\mathbf{satmap}_{\mathrm{BD}}(r, p \wedge q) &\mathrel{\widehat{=}} \mathbf{tmap}_{\mathrm{BD}}(r, p) \blacktriangleleft_{\mathrm{BD}} r \ \cap\ \mathbf{tmap}_{\mathrm{BD}}(r, q) \blacktriangleleft_{\mathrm{BD}} r \\
\mathbf{satmap}_{\mathrm{BD}}(r, p \vee q) &\mathrel{\widehat{=}} \mathbf{tmap}_{\mathrm{BD}}(r, p) \blacktriangleleft_{\mathrm{BD}} r \ \cup\ \mathbf{tmap}_{\mathrm{BD}}(r, q) \blacktriangleleft_{\mathrm{BD}} r \\
\mathbf{satmap}_{\mathrm{BD}}(r, \mathbf{EX}\ p) &\mathrel{\widehat{=}} F_r(\mathbf{satmap}_{\mathrm{BD}}(r, p)) \\
\mathbf{satmap}_{\mathrm{BD}}(r, \mathbf{EG}\ p) &\mathrel{\widehat{=}} F_r^{\Diamond}(\mathbf{satmap}_{\mathrm{BD}}(r, p))\,\{r\} \\
\mathbf{satmap}_{\mathrm{BD}}(r, \mathbf{E}[p\ \mathbf{U}\ q]) &\mathrel{\widehat{=}} P_{rq}^{\Box}(\mathbf{satmap}_{\mathrm{BD}}(r, p))\,\emptyset
\end{aligned}
$$

Finally, we can specify the model checking relation $\models_{\mathrm{BD}}$. Given a transition relation $r \in \mathcal{R}$ for the design $\mathcal{D}$, the model checking relation $\mathcal{D}, s \models P$ for property $P \in \mathcal{P}$ in

state s is defined by

$$\mathcal{D}, s \models_{\mathrm{BD}} P \mathrel{\widehat{=}} (\mathbf{map}_{\mathrm{BD}}(r, s) \blacktriangleleft_{\mathrm{BD}} \mathbf{satmap}_{\mathrm{BD}}(r, P)) \neq \emptyset$$

In this specification of $\models_{\mathrm{BD}}$, the interpretation is different from that in Section 4.3. In particular, the operations over the binary diagram structure closely follow the operations used in traditional symbolic model checking using BDDs.

5 Conclusion

The presentation of programming topics typically encompasses a wide range of diverse notations and concepts. UTP provides a unification of these presentations, and presents each programming topic in a unified and coherent notation.

We have presented an encoding of design verification for UTP along the lines of the traditional model checking paradigm. The UTP theory of designs provides a basis for program development, embodying concepts of program specification and refinement. UTP designs are structured around predicates representing pre and post-conditions. These predicates provide an appropriate basis for creating the transition relation needed for model checking, and we have shown how to automatically generate this transition relation from the pre and postcondition design.

In traditional model checking, the models are constructed independently of the software intended to implement the model. An advantage of the UTP style is that there is a direct relation between the UTP design and its *model*. The central idea in this paper is that of applying model checking to the verification of programs expressed in the UTP style, leading to a closer relationship between program development and program verification. By contrast, model checking is commonly done by the construction of checkable models independently of the construction of the software, leading to a gap between the verified part of a software project and the delivered (code) part of a software project.

The current approach can serve as a starting platform for further theoretical and practical steps.

Acknowledgements.

The authors would like to thank the anonymous referees for their helpful comments.

References

1. R-J. Back and J. von Wright. *Refinement Calculus, A Systematic Introduction* Springer, 1998.
2. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
4. R. Goldblatt. Modal Logics of Programs. Research Report 94-146, Victoria University of Wellington, 1994.

5. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
6. D. Lucanu and G. Ciobanu. Bisimulation and CTL Models for Hidden Algebraic Specification. Technical Report 03-03, "A.I.Cuza" University of Iaşi, 2003.
7. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.
8. A. Sherif and J. He. Towards a Time Model for Circus. In C. George and H. Miao, editors, *4th International Conference on Formal Engineering Methods, ICFEM 2002* volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer-Verlag, 2002.
9. B. von Karger and C.A.R. Hoare. Sequential Calculus. *Information Processing Letters*, 53(3):123–30, 1995.
10. J.C.P. Woodcock. Unifying Theories of Parallel Programming. In *Logic and Algebra for Engineering Software*. IOS Press, 2002. Also Keynote speech at ICFEM 2002: 4th International Conference on Formal Engineering Methods, Shanghai. IEEE Computer Society Press.
11. J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.