Generating Counterexamples for Model Checking by Transformation

G.W. Hamilton

School of Computing and Lero
Dublin City University
Ireland

`hamilton@computing.dcu.ie`

Counterexamples explain why a desired temporal logic property fails to hold. The generation of counterexamples is considered to be one of the primary advantages of model checking as a verification technique. Furthermore, when model checking does succeed in verifying a property, there is typically no independently checkable witness that can be used as evidence for the verified property. Previously, we have shown how program transformation techniques can be used for the verification of both safety and liveness properties of reactive systems. However, no counterexamples or witnesses were generated using the described techniques. In this paper, we address this issue. In particular, we show how the program transformation technique *distillation* can be used to facilitate the construction of counterexamples and witnesses for temporal properties of reactive systems. Example systems which are intended to model mutual exclusion are analysed using these techniques with respect to both safety (mutual exclusion) and liveness (non-starvation), with counterexamples being generated for those properties which do not hold.

1 Introduction

Model checking is a well established technique originally developed for the verification of temporal properties of finite state systems [3]. In addition to telling the user whether the desired temporal property holds, it can also generate a *counterexample*, explaining the reason why this property failed. This is considered to be one of the major advantages of model checking when compared to other verification methods. Fold/unfold program transformation techniques have more recently been proposed as an approach to model checking. Many such techniques have been developed for logic programs (e.g. [11, 15, 4, 1, 8]). However, very few such techniques have been developed for functional programs (with the work of Lisitsa and Nemytykh [12, 2] using supercompilation [17] being a notable exception), and these deal only with safety properties. Unfortunately, none of these techniques generate counterexamples when the temporal property does not hold.

In previous work [6], we have shown how a fold/unfold program transformation technique can be used to facilitate the verification of both safety and liveness properties of reactive systems which have been specified using functional programs. These functional programs produce a *trace* of states as their output, and the temporal property specifies the constraints that all output traces from the program should satisfy. However, counterexamples and witnesses were not generated using this approach. In this paper, we address this shortcoming to show how our previous work can be extended to generate a counterexample trace when a temporal property does not hold, and a witness when it does.

The program transformation technique which we use is our own *distillation* [5, 7] which builds on top of positive supercompilation [16], but is much more powerful. Distillation is used to transform the programs defining reactive systems into a simplified form which makes them much easier to analyse. We

then show how temporal properties for this simplified form can be verified, and extend this to generate counterexamples and witnesses. The described techniques are applied to a number of example systems which are intended to model mutually exclusive access to a critical resource by two processes. When a specified temporal property does not hold, we show how our approach can be applied to generate a corresponding counterexample and when the property does hold we show our approach can be applied to generate a corresponding witness.

The remainder of this paper is structured as follows. In Section 2, we introduce the functional language over which our verification techniques are defined. In Section 3, we show how to specify reactive systems in our language, and give a number of example systems which are intended to model mutually exclusive access to a critical resource by two processes. In Section 4, we describe how to specify temporal properties for reactive systems defined in our language, and specify both safety (mutual exclusion) and liveness (non-starvation) for the example systems. In Section 5, we describe our technique for verifying temporal properties of reactive systems and apply this technique to the example systems to verify the previously specified temporal properties. In Section 6, we describe our technique for the generation of counterexamples and witnesses, and apply this technique to the example systems. Section 7 concludes and considers related work.

2 Language

In this section, we describe the syntax and semantics of the higher-order functional language which will be used throughout this paper.

2.1 Syntax

The syntax of our language is given in Figure 1.

$$
\begin{array}{lll}
e ::= & x & \text{Variable} \\
& |\ c\ e_1 \ldots e_k & \text{Constructor Application} \\
& |\ \lambda x.e & \lambda\text{-Abstraction} \\
& |\ f & \text{Function Call} \\
& |\ e_0\ e_1 & \text{Application} \\
& |\ \textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1\ |\cdots| p_k \rightarrow e_k & \text{Case Expression} \\
& |\ \textbf{let}\ x = e_0\ \textbf{in}\ e_1 & \text{Let Expression} \\
& |\ e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n & \text{Local Function Definitions} \\
& & \\
p ::= & c\ x_1 \ldots x_k & \text{Pattern}
\end{array}
$$

Figure 1: Language Grammar

A program is an expression which can be a variable, constructor application, λ-abstraction, function call, application, **case**, **let** or **where**. Variables introduced by λ-abstractions, **let** expressions and **case** patterns are *bound*; all other variables are *free*. An expression which contains no free variables is said to be *closed*.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern and the same constructor cannot appear within

more than one pattern. We assume that the patterns in a **case** expression are exhaustive; we also allow a wildcard pattern _ which always matches if none of the earlier patterns match. Types are defined using algebraic data types, and it is assumed that programs are well-typed. Erroneous terms such as **case** $(\lambda x.e)$ **of** $p_1 \to e_1 \mid \cdots \mid p_k \to e_k$ and $(c\, e_1 \ldots e_n)\, e$ where c is of arity n cannot therefore occur.

2.2 Semantics

The call-by-name operational semantics of our language is standard: we define an evaluation relation $\Downarrow$ between closed expressions and *values*, where values are expressions in *weak head normal form* (i.e. constructor applications or λ-abstractions). We define a one-step reduction relation $\overset{r}{\rightsquigarrow}$ inductively as shown in Figure 2, where the reduction r can be f (unfolding of function f), c (elimination of constructor c) or β (β-substitution).

$$((\lambda x.e_0)\, e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x \mapsto e_1\}) \qquad (\textbf{let } x = e_0 \textbf{ in } e_1) \overset{\beta}{\rightsquigarrow} (e_1\{x \mapsto e_0\})$$

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e} \qquad\qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\, e_1) \overset{r}{\rightsquigarrow} (e_0'\, e_1)}$$

$$\frac{p_i = c\, x_1 \ldots x_n}{(\textbf{case } (c\, e_1 \ldots e_n) \textbf{ of } p_1 : e_1' \mid \ldots \mid p_k : e_k') \overset{c}{\rightsquigarrow} (e_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})}$$

$$\frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots p_k : e_k) \overset{r}{\rightsquigarrow} (\textbf{case } e_0' \textbf{ of } p_1 : e_1 \mid \ldots p_k : e_k)}$$

Figure 2: One-Step Reduction Relation

We use the notation $e \rightsquigarrow$ if the expression e reduces, $e \Uparrow$ if e diverges, $e \Downarrow$ if e converges and $e \Downarrow v$ if e evaluates to the value v. These are defined as follows, where $\overset{r*}{\rightsquigarrow}$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$:

$$e \rightsquigarrow, \text{ iff } \exists e'.e \overset{r}{\rightsquigarrow} e' \qquad e \Downarrow, \text{ iff } \exists v.e \Downarrow v$$
$$e \Downarrow v, \text{ iff } e \overset{r*}{\rightsquigarrow} v \wedge \neg(v \rightsquigarrow) \qquad e \Uparrow, \text{ iff } \forall e'.e \overset{r*}{\rightsquigarrow} e' \Rightarrow e' \rightsquigarrow$$

3 Specifying Reactive Systems

In this section, we show how to specify reactive systems in our programming language. While reactive systems are usually specified using *labelled transitions systems* (LTSs), our specifications can be trivially derived from these. Reactive systems have to react to a series of *external events* by updating their *states*. In order to facilitate this, we make use of a *list* datatype, which is defined as follows for the element type a:

$$List\ a ::= Nil \mid Cons\ a\ (List\ a)$$

We use $[]$ as a shorthand for *Nil*, and $[s_1, \ldots, s_n]$ as a shorthand for a list containing the elements $s_1 \ldots s_n$. We also use $++$ to represent list concatenation. Our programs will map a (potentially infinite) input list of external events and an initial state to a (potentially infinite) output list of *observable states* (a *trace*), which gives the values of a subset of state variables whose properties can be verified.

In this paper, we wish to analyse a number of systems which are intended to implement mutually exclusive access to a critical resource for two processes. In all of these systems, the external events belong to the following datatype:

$$Event ::= Request_1 \mid Request_2 \mid Take_1 \mid Take_2 \mid Release_1 \mid Release_2$$

Each of the two processes can therefore request access to the critical resource, and take and release this resource. Observable states in all of our example systems belong to the following datatype:

$$State ::= ObsState\ ProcState\ ProcState$$

$$ProcState ::= T \mid W \mid U$$

Each process can therefore be thinking (T), waiting for the critical resource (W) or using the critical resource (U).

Each of our example systems is transformed into a simplified form as previously shown in [6] using distillation [5, 7], a powerful program transformation technique which builds on top of the supercompilation transformation [17, 16]. Due to the nature of the programs modelling reactive systems, in which the input is an external event list, and the output is a list of observable states, the programs resulting from this transformation take the form $e^\emptyset$, where e^ρ is defined as shown in Figure 3 where the **let** variables are added to the set ρ, and will not be used as **case** selectors.

$$
\begin{array}{lll}
e^\rho & ::= & Cons\ e_0^\rho\ e_1^\rho \\
& \mid & f\ x_1 \ldots x_n \\
& \mid & \textbf{case}\ x\ \textbf{of}\ p_1 \to e_1^\rho \mid \cdots \mid p_k \to e_n^\rho,\ \text{where}\ x \notin \rho \\
& \mid & x\ e_1^\rho \ldots e_n^\rho,\ \text{where}\ x \in \rho \\
& \mid & \textbf{let}\ x = \lambda x_1 \ldots x_n.e_0^\rho\ \textbf{in}\ e_1^{(\rho \cup \{x\})} \\
& \mid & e_0^\rho\ \textbf{where}\ f_1 = \lambda x_{1_1} \ldots x_{1_k}.e_1^\rho \ldots f_n = \lambda x_{n_1} \ldots x_{n_k}.e_n^\rho
\end{array}
$$

Figure 3: Simplified Form Resulting From Distillation

The crucial syntactic property of this simplified form is that all functions must be tail recursive; this is what allows the resulting programs to be verified more easily. In all of the following examples, the variable *es* represents the external event list.

Example 1 In the first example shown in Figure 4, each process can request access to the critical resource if it is thinking and the other process is not using it, take the critical resource if it is waiting for it, and release the critical resource if it is using it. The LTS representation of this program is shown in Figure 5 (for ease of presentation of this and subsequent LTSs, transitions back into the same state have been omitted).

Example 2 In the second example shown in Figure 6, each process can request access to the critical resource if it is thinking and the other process is not using it, take the critical resource if it is waiting for it and the other process is thinking, and release the critical resource if it is using it. The LTS representation of this program is shown in Figure 7.

$Cons\ (ObsState\ T\ T)\ (f_1\ es)$
where
$f_1 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Request_1 \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es)$
                     $|\ Request_2 \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es)$
$f_2 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Take_1\ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ T)\ (f_4\ es)$
                     $|\ Request_2 \rightarrow Cons\ (ObsState\ W\ W)\ (f_5\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es)$
$f_3 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Request_1 \rightarrow Cons\ (ObsState\ W\ W)\ (f_5\ es)$
                     $|\ Take_2\ \ \ \ \ \rightarrow Cons\ (ObsState\ T\ U)\ (f_6\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es)$
$f_4 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Release_1 \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ T)\ (f_4\ es)$
$f_5 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Take_1 \rightarrow Cons\ (ObsState\ U\ W)\ (f_7\ es)$
                     $|\ Take_2 \rightarrow Cons\ (ObsState\ W\ U)\ (f_8\ es)$
                     $|\ \_\ \ \ \ \ \ \rightarrow Cons\ (ObsState\ W\ W)\ (f_5\ es)$
$f_6 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Release_2 \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ T\ U)\ (f_6\ es)$
$f_7 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Release_1 \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es)$
                     $|\ Take_2\ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ U)\ (f_9\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ W)\ (f_7\ es)$
$f_8 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Release_2 \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es)$
                     $|\ Take_1\ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ U)\ (f_9\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ W\ U)\ (f_8\ es)$
$f_9 = \lambda es.$**case** es **of**
        $Cons\ e\ es\ \rightarrow$ **case** e **of**
                        $Release_1 \rightarrow Cons\ (ObsState\ T\ U)\ (f_6\ es)$
                     $|\ Release_2 \rightarrow Cons\ (ObsState\ U\ T)\ (f_4\ es)$
                     $|\ \_\ \ \ \ \ \ \ \ \ \ \rightarrow Cons\ (ObsState\ U\ U)\ (f_9\ es)$
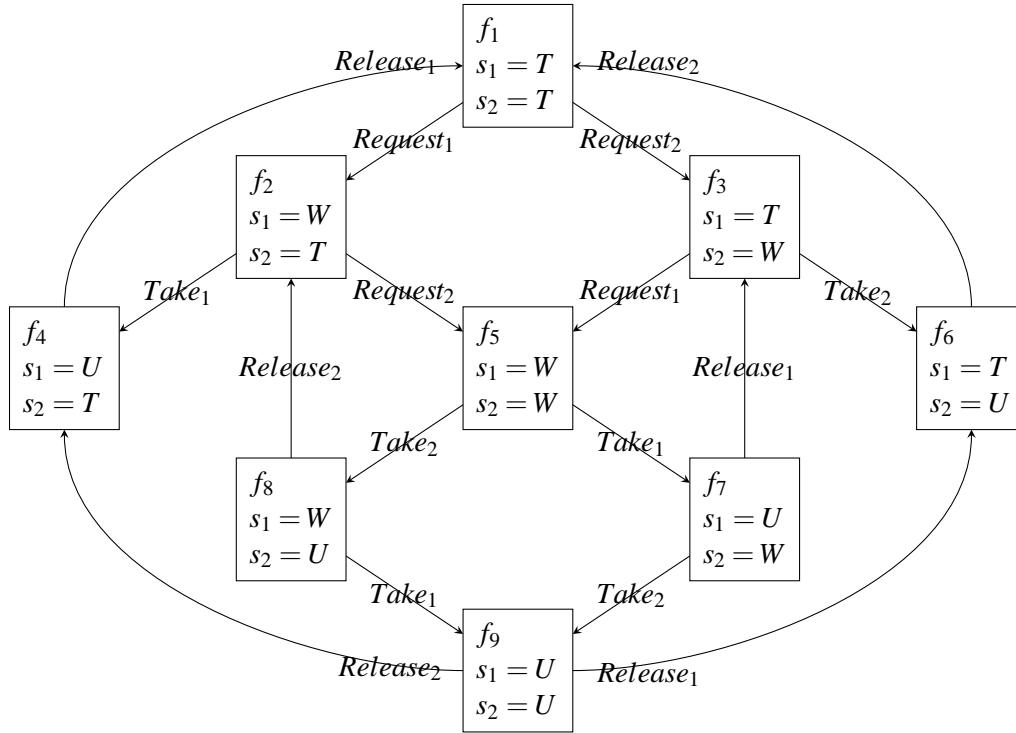
Figure 4: Example 1

Figure 5: LTS Representation of Example 1

Example 3 In the final example shown in Figure 8, each process can request access to the critical re-
source if it is thinking, take the critical resource if it is waiting for it and requested access before the
other process, and release the critical resource if it is using it. Note that this program is the result of
transforming an implementation of Lamport's bakery algorithm [9] for two processes as shown in [6].
Although the original program makes use of numbered tickets and is therefore an infinite state system,
the use of tickets is completely transformed away and the resulting program has a finite number of states.
The LTS representation of this program is shown in Figure 9.

4   Specification of Temporal Properties

In this section, we describe how temporal properties of reactive systems are specified. We use Linear-
time Temporal Logic (LTL), in which the set of well-founded formulae (WFF) are defined inductively as
follows. All atomic propositions p are in WFF; if φ and ψ are in WFF, then so are:

- $\neg \varphi$

- $\varphi \vee \psi$

- $\varphi \wedge \psi$

- $\varphi \Rightarrow \psi$

- $\Box \varphi$

- $\Diamond \varphi$

- $\bigcirc \varphi$

Cons (*ObsState T T*) (*f₁ es*)
where
$f_1 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                Request₁ → *Cons* (*ObsState W T*) (*f₂ es*)
                | *Request₂* → *Cons* (*ObsState T W*) (*f₃ es*)
                | _   → *Cons* (*ObsState T T*) (*f₁ es*)

$f_2 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                Take₁   → *Cons* (*ObsState U T*) (*f₄ es*)
                | *Request₂* → *Cons* (*ObsState W W*) (*f₅ es*)
                | _   → *Cons* (*ObsState W T*) (*f₂ es*)

$f_3 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                Request₁ → *Cons* (*ObsState W W*) (*f₅ es*)
                | *Take₂*   → *Cons* (*ObsState T U*) (*f₆ es*)
                | _   → *Cons* (*ObsState T W*) (*f₃ es*)

$f_4 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                Release₁ → *Cons* (*ObsState T T*) (*f₁ es*)
                | _   → *Cons* (*ObsState U T*) (*f₄ es*)

$f_5 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                _ → *Cons* (*ObsState W W*) (*f₅ es*)

$f_6 = \lambda es.$**case** *es* **of**
        Cons e es → **case** *e* **of**
                Release₂ → *Cons* (*ObsState T T*) (*f₁ es*)
                | _   → *Cons* (*ObsState T U*) (*f₆ es*)

Figure 6: Example 2
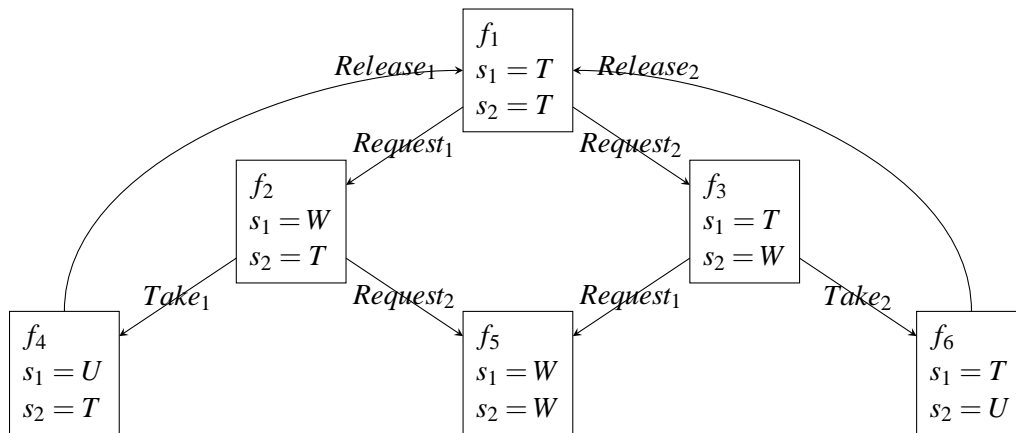
Figure 7: LTS Representation of Example 2

Cons (*ObsState T T*) (*f₁ es*)
where
$f_1 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Request₁* $\rightarrow$ *Cons* (*ObsState W T*) (*f₂ es*)
$\qquad\qquad\qquad$ | *Request₂* $\rightarrow$ *Cons* (*ObsState T W*) (*f₃ es*)
$\qquad\qquad\qquad$ | _ $\qquad\quad$ $\rightarrow$ *Cons* (*ObsState T T*) (*f₁ es*)
$f_2 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Take₁* $\qquad$ $\rightarrow$ *Cons* (*ObsState U T*) (*f₄ es*)
$\qquad\qquad\qquad$ | *Request₂* $\rightarrow$ *Cons* (*ObsState W W*) (*f₆ es*)
$\qquad\qquad\qquad$ | _ $\qquad\qquad$ $\rightarrow$ *Cons* (*ObsState W T*) (*f₂ es*)
$f_3 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Take₂* $\qquad$ $\rightarrow$ *Cons* (*ObsState T U*) (*f₅ es*)
$\qquad\qquad\qquad$ | *Request₁* $\rightarrow$ *Cons* (*ObsState W W*) (*f₇ es*)
$\qquad\qquad\qquad$ | _ $\qquad\qquad$ $\rightarrow$ *Cons* (*ObsState T W*) (*f₃ es*)
$f_4 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Release₁* $\rightarrow$ *Cons* (*ObsState T T*) (*f₁ es*)
$\qquad\qquad\qquad$ | *Request₂* $\rightarrow$ *Cons* (*ObsState U W*) (*f₈ es*)
$\qquad\qquad\qquad$ | _ $\qquad\qquad$ $\rightarrow$ *Cons* (*ObsState U T*) (*f₄ es*)
$f_5 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Release₂* $\rightarrow$ *Cons* (*ObsState T T*) (*f₁ es*)
$\qquad\qquad\qquad$ | *Request₁* $\rightarrow$ *Cons* (*ObsState W U*) (*f₉ es*)
$\qquad\qquad\qquad$ | _ $\qquad\qquad$ $\rightarrow$ *Cons* (*ObsState T U*) (*f₅ es*)
$f_6 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Take₁* $\rightarrow$ *Cons* (*ObsState U W*) (*f₈ es*)
$\qquad\qquad\qquad$ | _ $\quad$ $\rightarrow$ *Cons* (*ObsState W W*) (*f₆ es*)
$f_7 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Take₂* $\rightarrow$ *Cons* (*ObsState W U*) (*f₉ es*)
$\qquad\qquad\qquad$ | _ $\quad$ $\rightarrow$ *Cons* (*ObsState W W*) (*f₇ es*)
$f_8 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Release₁* $\rightarrow$ *Cons* (*ObsState T W*) (*f₃ es*)
$\qquad\qquad\qquad$ | _ $\qquad\quad$ $\rightarrow$ *Cons* (*ObsState U W*) (*f₈ es*)
$f_9 = \lambda es.$**case** *es* **of**
$\qquad$ *Cons e es* $\rightarrow$ **case** *e* **of**
$\qquad\qquad\qquad$ *Release₂* $\rightarrow$ *Cons* (*ObsState W T*) (*f₂ es*)
$\qquad\qquad\qquad$ | _ $\qquad\quad$ $\rightarrow$ *Cons* (*ObsState W U*) (*f₉ es*)
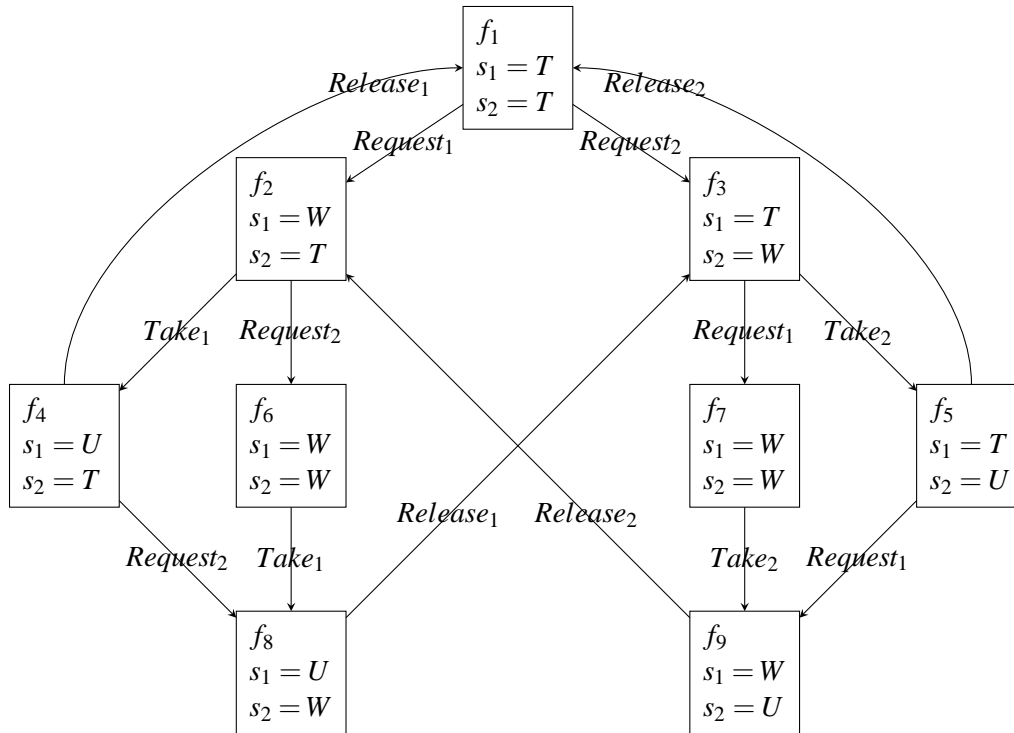
Figure 8: Example 3

Figure 9: LTS Representation of Example 3

The temporal operator $\Box\varphi$ means that φ is *always* true; this is used to express *safety* properties. The temporal operator $\Diamond\varphi$ means that φ will *eventually* be true; this is used to express *liveness* properties. The temporal operator $\bigcirc\varphi$ means that φ is true in the *next* state. These modalities can be combined to obtain new modalities; for example, $\Box\Diamond\varphi$ means that φ is true infinitely often, and $\Diamond\Box\varphi$ means that φ is eventually true forever. Fairness constraints can also be specified for some external events (those belonging to the set F) which require that they occur infinitely often. For the examples given in this paper, it is assumed that all external events belong to F.

Here, propositional models for linear-time temporal formulas consist of a list of observable states $\pi = [s_0, s_1, \ldots]$. The satisfaction relation is extended to formulas in LTL for a model π and position i as follows.

$$
\begin{array}{lll}
\pi, i \vDash p & \text{iff} & p \in s_i \\
\pi, i \vDash \neg\varphi & \text{iff} & \pi, i \nvDash \varphi \\
\pi, i \vDash \varphi \lor \psi & \text{iff} & \pi, i \vDash \varphi \text{ or } \pi, i \vDash \psi \\
\pi, i \vDash \varphi \land \psi & \text{iff} & \pi, i \vDash \varphi \text{ and } \pi, i \vDash \psi \\
\pi, i \vDash \varphi \Rightarrow \psi & \text{iff} & \pi, i \nvDash \varphi \text{ or } \pi, i \vDash \psi \\
\pi, i \vDash \Box\varphi & \text{iff} & \forall j \geq i.\pi, j \vDash \varphi \\
\pi, i \vDash \Diamond\varphi & \text{iff} & \exists j \geq i.\pi, j \vDash \varphi \\
\pi, i \vDash \bigcirc\varphi & \text{iff} & \pi, i+1 \vDash \varphi
\end{array}
$$

A formula φ holds in model π if it holds at position 0 i.e. $\pi, 0 \vDash \varphi$.

The atomic propositions of these temporal formulae can be trivially translated into our functional language. For our verification rules, we define the following datatype for truth values:

$$TruthVal ::= True \mid False \mid Undefined$$

We use a Kleene three-valued logic because our verification rules must always return an answer, but some of the properties to be verified may give an undefined outcome. For our example programs which attempt to implement mutual exclusion, the following two properties are defined. Within these temporal properties, we use the variable s to denote the current observable state whose properties are being specified.

Property 1 (Mutual Exclusion) This is a safety property which specifies that both processes cannot be using the critical resource at the same time. This can be specified as follows:

$$\Box(\textbf{case } s \textbf{ of}$$
$$ObsState \; s_1 \; s_2 \; \rightarrow \; \textbf{case } s_1 \textbf{ of}$$
$$U \rightarrow \textbf{case } s_2 \textbf{ of}$$
$$U \rightarrow False$$
$$| \; \_ \; \rightarrow \; True$$
$$| \; \_ \; \rightarrow \; True)$$

Property 2 (Non-Starvation) This is a liveness property which specifies that each process must eventually get to use the critical resource if they are waiting for it. This can be specified for process 1 as follows (the specification of this property for process 2 is similar):

$$\Box((\textbf{case } s \textbf{ of}$$
$$ObsState \; s_1 \; s_2 \; \rightarrow \; \textbf{case } s_1 \textbf{ of}$$
$$W \rightarrow True$$
$$| \; \_ \; \rightarrow \; False) \Rightarrow \Diamond(\textbf{case } s \textbf{ of}$$
$$ObsState \; s_1 \; s_2 \; \rightarrow \; \textbf{case } s_1 \textbf{ of}$$
$$U \rightarrow True$$
$$| \; \_ \; \rightarrow \; False))$$

5   Verification of Temporal Properties

In this section, we show how temporal properties of reactive systems defined in our functional language can be verified. We define our verification rules on the restricted form of program defined in Figure 3 as shown in Figure 10. The parameter φ denotes the property to be verified and ϕ denotes the function variable environment. ρ denotes the set of function calls previously encountered; this is used for the detection of loops to ensure termination. ρ is also used in the verification of the $\Box$ operator (which evaluates to *True* on encountering a loop), and the verification of the $\Diamond$ operator (which evaluates to *False* on encountering a loop); ρ is reset to empty when the verification moves inside these temporal operators. For all other temporal formulae, the value *Undefined* is returned on encountering a loop.

The verification rules can be explained as follows. Rules (1-4) deal with the logical connectives $\wedge$, $\vee$, $\Rightarrow$ and $\neg$. These are implemented in our language in the usual way for a Kleene three-valued logic using the corresponding operators $\wedge_3$, $\vee_3$, $\Rightarrow_3$ and $\neg_3$. Rules (5a-d) deal with a constructed stream of states. In rule (5a), if we are trying to verify that a property is always true, then we verify that it is true for the first state (with ρ reset to empty) and is always true in all remaining states. In rule (5b), if we are trying to verify that a property is eventually true, then we verify that it is either true for the first state (with ρ reset to empty) or is eventually true in all remaining states. In rule (5c), if we are trying to verify that a property is true in the next state then we verify that the property is true for the next state. In rule (5d), if we are trying to verify that a property is true in the current state then we verify that the property is true for the current state by evaluating the property using the value of the current state for the state variable s. Rules (6a-c) deal with function calls. In rule (6a), if we are trying to verify that a property is

$$
\begin{aligned}
&(1)\quad \mathscr{P}[\![e]\!]\ (\varphi \wedge \psi)\ \phi\ \rho && = (\mathscr{P}[\![e]\!]\ \varphi\ \phi\ \rho) \wedge_3 (\mathscr{P}[\![e]\!]\ \psi\ \phi\ \rho)\\
&(2)\quad \mathscr{P}[\![e]\!]\ (\varphi \vee \psi)\ \phi\ \rho && = (\mathscr{P}[\![e]\!]\ \varphi\ \phi\ \rho) \vee_3 (\mathscr{P}[\![e]\!]\ \psi\ \phi\ \rho)\\
&(3)\quad \mathscr{P}[\![e]\!]\ (\varphi \Rightarrow \psi)\ \phi\ \rho && = (\mathscr{P}[\![e]\!]\ \varphi\ \phi\ \rho) \Rightarrow_3 (\mathscr{P}[\![e]\!]\ \psi\ \phi\ \rho)\\
&(4)\quad \mathscr{P}[\![e]\!]\ (\neg\varphi)\ \phi\ \rho && = \neg_3(\mathscr{P}[\![e]\!]\ \varphi\ \phi\ \rho)\\
&(5a)\ \mathscr{P}[\![Cons\ e_0\ e_1]\!]\ (\Box\varphi)\ \phi\ \rho && = (\mathscr{P}[\![Cons\ e_0\ e_1]\!]\ \varphi\ \phi\ \emptyset) \wedge_3 (\mathscr{P}[\![e_1]\!]\ (\Box\varphi)\ \phi\ \rho)\\
&(5b)\ \mathscr{P}[\![Cons\ e_0\ e_1]\!]\ (\Diamond\varphi)\ \phi\ \rho && = (\mathscr{P}[\![Cons\ e_0\ e_1]\!]\ \varphi\ \phi\ \emptyset) \vee_3 (\mathscr{P}[\![e_1]\!]\ (\Diamond\varphi)\ \phi\ \rho)\\
&(5c)\ \mathscr{P}[\![Cons\ e_0\ e_1]\!]\ (\bigcirc\varphi)\ \phi\ \rho && = \mathscr{P}[\![e_1]\!]\ \varphi\ \phi\ \rho\\
&(5d)\ \mathscr{P}[\![Cons\ e_0\ e_1]\!]\ \varphi\ \phi\ \rho && = v,\ \text{where}\ \varphi[e_0/s] \Downarrow v
\end{aligned}
$$

$$
(6a)\ \mathscr{P}[\![f\ x_1\ldots x_n]\!]\ (\Box\varphi)\ \phi\ \rho =
\begin{cases}
True, & \text{if } f \in \rho\\
\mathscr{P}[\![e[x_1/x_1',\ldots,x_n/x_n']]\!]\ (\Box\varphi)\ \phi\ (\rho \cup \{f\}), & \text{otherwise}
\end{cases}
$$
$$
\text{where } \phi(f) = \lambda x_1'\ldots x_n'.e
$$

$$
(6b)\ \mathscr{P}[\![f\ x_1\ldots x_n]\!]\ (\Diamond\varphi)\ \phi\ \rho =
\begin{cases}
False, & \text{if } f \in \rho\\
\mathscr{P}[\![e[x_1/x_1',\ldots,x_n/x_n']]\!]\ (\Diamond\varphi)\ \phi\ (\rho \cup \{f\}), & \text{otherwise}
\end{cases}
$$
$$
\text{where } \phi(f) = \lambda x_1'\ldots x_n'.e
$$

$$
(6c)\ \mathscr{P}[\![f\ x_1\ldots x_n]\!]\ \varphi\ \phi\ \rho =
\begin{cases}
Undefined, & \text{if } f \in \rho\\
\mathscr{P}[\![e[x_1/x_1',\ldots,x_n/x_n']]\!]\ \varphi\ \phi\ (\rho \cup \{f\}), & \text{otherwise}
\end{cases}
$$
$$
\text{where } \phi(f) = \lambda x_1'\ldots x_n'.e
$$

$$
(7a)\ \mathscr{P}[\![\mathbf{case}\ x\ \mathbf{of}\ p_1 \to e_1\ |\cdots|\ p_n \to e_n]\!]\ (\Diamond\varphi)\ \phi\ \rho
$$
$$
= (\bigvee_{p_i \in F} \mathscr{P}[\![e_i]\!]\ (\Diamond\varphi)\ \phi\ \rho) \vee_3 (\bigwedge_{i=1}^{n} \mathscr{P}[\![e_i]\!]\ (\Diamond\varphi)\ \phi\ \rho)
$$

$$
(7b)\ \mathscr{P}[\![\mathbf{case}\ x\ \mathbf{of}\ p_1 \to e_1\ |\cdots|\ p_n \to e_n]\!]\ \varphi\ \phi\ \rho
$$
$$
= \bigwedge_{i=1}^{n} \mathscr{P}[\![e_i]\!]\ \varphi\ \phi\ \rho
$$

$$
\begin{aligned}
&(8)\quad \mathscr{P}[\![x\ e_1\ldots e_n]\!]\ \varphi\ \phi\ \rho && = Undefined\\
&(9)\quad \mathscr{P}[\![\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \varphi\ \phi\ \rho = \mathscr{P}[\![e_1]\!]\ \varphi\ \phi\ \rho\\
&(10)\ \mathscr{P}[\![e_0\ \mathbf{where}\ f_1 = e_1\ldots f_n = e_n]\!]\ \varphi\ \phi\ \rho\\
&\qquad\qquad\qquad\qquad = \mathscr{P}[\![e_0]\!]\ \varphi\ (\phi \cup \{f_1 \mapsto e_1,\ldots,f_n \mapsto e_n\})\ \rho
\end{aligned}
$$

Figure 10: Verification Rules

always true, then if the function call has been encountered before while trying to verify the same property we can return the value *True*; this corresponds to the standard greatest fixed point calculation normally used for the $\Box$ operator in which the property is initially assumed to be *True* for all states. Otherwise, the function is unfolded and added to the set of previously encountered function calls for this property. In rule (6b), if we are trying to verify that a property is eventually true, then if the function call has been encountered before while trying to verify the same property we can return the value *False*; this corresponds to the standard least fixed point calculation normally used for the $\Diamond$ property in which the property is initially assumed to be *False* for all states. Otherwise, the function is unfolded and added to the set of previously encountered function calls for this property. In rule (6c), if we are trying to verify that any other property is true, then if the function call has been encountered before we can return the value *Undefined* since a loop has been detected. Otherwise, the function is unfolded and added to the set of previously encountered function calls. Rules (7a-b) deal with **case** expressions. In rule (7a), if we are trying to verify that a property is eventually true, then we verify that it is either eventually true for at least one of the branches for which there is a fairness assumption (since these branches must be selected eventually), or that it is eventually true for all branches. In Rule (7b), if we are trying to verify that any

other property is true, then we verify that it is true for all branches. In rule (8), if we encounter a free variable, then we return the value *Undefined* since we cannot determine the value of the variable; this must be a **let** variable which has been abstracted, so no information can be determined for it. In rule (9), in order to verify that a property is true for a **let** expression, we verify that it is true for the **let** body; this is where we perform abstraction of the extracted sub-expression. In rule (10), for a **where** expression, the function definitions are added to the environment ϕ.

Theorem 5.1 (Soundness) $\forall e \in Prog, es \in List\ Event, \pi \in List\ State, \varphi \in$ WFF:
$(e\ es \overset{r*}{\leadsto} \pi) \wedge (\mathscr{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = True \Rightarrow \pi, 0 \vDash \varphi) \wedge (\mathscr{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = False \Rightarrow \pi, 0 \nvDash \varphi)$

Proof. The proof of this is by structural induction on the program *e*. $\qquad\qquad\square$

Theorem 5.2 (Termination) $\forall e \in$ Prog, $\varphi \in$ WFF: $\mathscr{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset$ always terminates.

Proof. Proof of termination is quite straightforward since there will be a finite number of functions and uses of the temporal operators $\square$ and $\diamondsuit$, and verification of each of these temporal operators will terminate when a function is re-encountered. $\qquad\qquad\square$

Using these rules, we try to verify the two properties (mutual exclusion and non-starvation) for the example programs for mutual exclusion given in Section 3. Firstly, distillation is applied to each of the programs.

Example 1 For the program shown in Figure 4, Property 2 (non-starvation) holds. The verification of Property 1 (mutual exclusion) is shown below where we represent Property 1 by $\square\varphi$ and the function environment by ϕ.

$\quad \mathscr{P}[\![Cons\ (ObsState\ T\ T)\ (f_1\ es)]\!]\ (\square\varphi)\ \emptyset\ \emptyset$
$= \{5a\}$
$\quad (\mathscr{P}[\![Cons\ (ObsState\ T\ T)\ (f_1\ es)]\!]\ \varphi\ \emptyset\ \emptyset) \wedge_3 (\mathscr{P}[\![f_1\ es]\!]\ (\square\varphi)\ \emptyset\ \emptyset)$
$= \{5d\}$
$\quad (\varphi[(ObsState\ T\ T)/s]) \wedge_3 (\mathscr{P}[\![f_1\ es]\!]\ (\square\varphi)\ \emptyset\ \emptyset)$
$= \{\text{calculation, 6a, 7b, 5a, 5d}\}$
$\quad (\mathscr{P}[\![f_1\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\}) \wedge_3 (\mathscr{P}[\![f_2\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{6a\}$
$\quad (\mathscr{P}[\![f_2\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{\text{calculation, 6a, 7b, 5a, 5d}\}$
$\quad (\mathscr{P}[\![f_2\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\}) \wedge_3 (\mathscr{P}[\![f_4\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\}) \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\})$
$\quad \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{6a\}$
$\quad (\mathscr{P}[\![f_4\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\}) \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{\text{calculation, 6a, 7b, 5a, 5d}\}$
$\quad (\mathscr{P}[\![f_1\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2, f_4\}) \wedge_3 (\mathscr{P}[\![f_4\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2, f_4\}) \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\})$
$\quad \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{6a\}$
$\quad (\mathscr{P}[\![f_5\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$
$= \{\text{calculation, 6a, 7b, 5a, 5d}\}$
$\quad (\mathscr{P}[\![f_5\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2, f_5\}) \wedge_3 (\mathscr{P}[\![f_7\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2, f_5\}) \wedge_3 (\mathscr{P}[\![f_8\ es]\!]\ (\square\varphi)\ \phi\ \{f_1, f_2, f_5\})$
$\quad \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\square\varphi)\ \phi\ \{f_1\})$

$= \{6a\}$

$\quad (\mathscr{P}[\![f_7\ es]\!]\ (\Box\varphi)\ \phi\ \{f_1,f_2,f_5\}) \wedge_3 (\mathscr{P}[\![f_8\ es]\!]\ (\Box\varphi)\ \phi\ \{f_1,f_2,f_5\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box\varphi)\ \phi\ \{f_1\})$

$= \{\text{calculation, 6a, 7b, 5a, 5d}\}$

$\quad \textit{False}$

Example 2 For the program shown in Figure 6, Property 1 (mutual exclusion) holds. The verification of Property 2 (non-starvation) is shown below where we represent Property 2 by $\Box(\varphi \Rightarrow \Diamond\psi)$ and the function environment by ϕ.

$\quad \mathscr{P}[\![Cons\ (ObsState\ T\ T)\ (f_1\ es)]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \emptyset\ \emptyset$

$= \{5a\}$

$\quad (\mathscr{P}[\![Cons\ (ObsState\ T\ T)\ (f_1\ es)]\!]\ (\varphi \Rightarrow \Diamond\psi)\ \emptyset\ \emptyset) \wedge_3 (\mathscr{P}[\![f_1\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \emptyset\ \emptyset)$

$= \{5d\}$

$\quad ((\varphi \Rightarrow \Diamond\psi)[(ObsState\ T\ T)/s]) \wedge_3 (\mathscr{P}[\![f_1\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \emptyset\ \emptyset)$

$= \{\text{calculation, 3, 6a, 7b, 5a, 5d}\}$

$\quad (\mathscr{P}[\![f_1\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\}) \wedge_3 (\mathscr{P}[\![f_2\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$\quad \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{6a\}$

$\quad (\mathscr{P}[\![f_2\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{\text{calculation, 3, 6a, 7b, 5a, 5d}\}$

$\quad (\mathscr{P}[\![f_2\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\}) \wedge_3 (\mathscr{P}[\![f_4\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\})$

$\quad \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{6a\}$

$\quad (\mathscr{P}[\![f_4\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\}) \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\})$

$\quad \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{\text{calculation, 3, 6a, 7b, 5a, 5d}\}$

$\quad (\mathscr{P}[\![f_1\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2,f_4\}) \wedge_3 (\mathscr{P}[\![f_4\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2,f_4\})$

$\quad \wedge_3 (\mathscr{P}[\![f_5\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{6a\}$

$\quad (\mathscr{P}[\![f_5\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1,f_2\}) \wedge_3 (\mathscr{P}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\})$

$= \{\text{calculation, 6a, 7b, 5a, 3, 5b, 6b}\}$

$\quad \textit{False}$

Example 3 For the program shown in Figure 8 both Property 1 (mutual exclusion) and Property 2 (non-starvation) hold.

6 Construction of Counterexamples and Witnesses

In this section, we show how counterexamples and witnesses for temporal properties of reactive systems defined in our functional language can be constructed. We augment the verification rules from the previous section to generate a *verdict* which consists of a trace (a list of observable states) along with a truth value and belongs to the following datatype:

$$Verdict ::= TruthVal \times List\ State$$

The trace will give a counterexample if the associated truth value is *False*, and a witness if the corresponding truth value is *True*. The logical connectives $\wedge_v, \vee_v, \Rightarrow_v$ and $\neg_v$ are extended to this datatype as $\wedge_v, \vee_v, \Rightarrow_v$ and $\neg_v$, which are defined as follows.

$$(b_1, t_1) \wedge_v (b_2, t_2) \quad = \quad (b, t)$$
$$\text{where}$$
$$b = b_1 \wedge_3 b_2$$
$$t = min\{t_i | t_i \in \{t_1, t_2\} \wedge b_i = b\}$$

$$(b_1, t_1) \vee_v (b_2, t_2) \quad = \quad (b, t)$$
$$\text{where}$$
$$b = b_1 \vee_3 b_2$$
$$t = min\{t_i | t_i \in \{t_1, t_2\} \wedge b_i = b\}$$

$$(b_1, t_1) \Rightarrow_v (b_2, t_2) \quad = \quad (\neg_v(b_1, t_1)) \vee_v (b_2, t_2)$$

$$\neg_v(b, t) \quad = \quad (\neg_3 b, t)$$

If there is more than one counterexample or witness, the function *min* is used to ensure that the *shortest* one is always returned. The rules for the construction of counterexamples and witnesses for the simplified form of program defined in Figure 3 are as shown in Figure 11.

(1) $\mathscr{C}[\![e]\!] (\varphi \wedge \psi) \phi \rho \pi \qquad = (\mathscr{C}[\![e]\!] \varphi \phi \rho \pi) \wedge_v (\mathscr{C}[\![e]\!] \psi \phi \rho \pi)$

(2) $\mathscr{C}[\![e]\!] (\varphi \vee \psi) \phi \rho \pi \qquad = (\mathscr{C}[\![e]\!] \varphi \phi \rho \pi) \vee_v (\mathscr{C}[\![e]\!] \psi \phi \rho \pi)$

(3) $\mathscr{C}[\![e]\!] (\varphi \Rightarrow \psi) \phi \rho \pi \qquad = (\mathscr{C}[\![e]\!] \varphi \phi \rho \pi) \Rightarrow_v (\mathscr{C}[\![e]\!] \psi \phi \rho \pi)$

(4) $\mathscr{C}[\![e]\!] (\neg\varphi) \phi \rho \pi \qquad = \neg_v(\mathscr{C}[\![e]\!] \varphi \phi \rho \pi)$

(5a) $\mathscr{C}[\![Cons\ e_0\ e_1]\!] (\Box\varphi) \phi \rho \pi = (\mathscr{C}[\![Cons\ e_0\ e_1]\!] \varphi \phi \emptyset \pi) \wedge_v (\mathscr{C}[\![e_1]\!] (\Box\varphi) \phi \rho (\pi{+}{+}[e_0]))$

(5b) $\mathscr{C}[\![Cons\ e_0\ e_1]\!] (\Diamond\varphi) \phi \rho \pi = (\mathscr{C}[\![Cons\ e_0\ e_1]\!] \varphi \phi \emptyset \pi) \vee_v (\mathscr{C}[\![e_1]\!] (\Diamond\varphi) \phi \rho (\pi{+}{+}[e_0]))$

(5c) $\mathscr{C}[\![Cons\ e_0\ e_1]\!] (\bigcirc\varphi) \phi \rho \pi = \mathscr{C}[\![e_1]\!] \varphi \phi \rho (\pi{+}{+}[e_0])$

(5d) $\mathscr{C}[\![Cons\ e_0\ e_1]\!] \varphi \phi \rho \pi \quad = (v, \pi{+}{+}[e_0]),$ where $\varphi[e_0/s] \Downarrow v$

(6a) $\mathscr{C}[\![f\ x_1 \ldots x_n]\!] (\Box\varphi) \phi \rho \pi \quad = \begin{cases} (True, \pi), & \text{if } f \in \rho \\ \mathscr{C}[\![e[x_1/x'_1, \ldots, x_n/x'_n]]\!] (\Box\varphi) \phi (\rho \cup \{f\}) \pi, & \text{otherwise} \end{cases}$
$\qquad\qquad\qquad\qquad\qquad$ where $\phi(f) = \lambda x'_1 \ldots x'_n . e$

(6b) $\mathscr{C}[\![f\ x_1 \ldots x_n]\!] (\Diamond\varphi) \phi \rho \pi \quad = \begin{cases} (False, \pi), & \text{if } f \in \rho \\ \mathscr{C}[\![e[x_1/x'_1, \ldots, x_n/x'_n]]\!] (\Diamond\varphi) \phi (\rho \cup \{f\}) \pi, & \text{otherwise} \end{cases}$
$\qquad\qquad\qquad\qquad\qquad$ where $\phi(f) = \lambda x'_1 \ldots x'_n . e$

(6c) $\mathscr{C}[\![f\ x_1 \ldots x_n]\!] \varphi \phi \rho \pi \quad = \begin{cases} (Undefined, \pi), & \text{if } f \in \rho \\ \mathscr{C}[\![e[x_1/x'_1, \ldots, x_n/x'_n]]\!] \varphi \phi (\rho \cup \{f\}) \pi, & \text{otherwise} \end{cases}$
$\qquad\qquad\qquad\qquad\qquad$ where $\phi(f) = \lambda x'_1 \ldots x'_n . e$

(7a) $\mathscr{C}[\![\textbf{case}\ x\ \textbf{of}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n]\!] (\Diamond\varphi) \phi \rho \pi$

$$= (\bigvee_{p_i \in F} \mathscr{C}[\![e_i]\!] (\Diamond\varphi) \phi \rho \pi) \vee_v (\bigwedge_{i=1}^{n} \mathscr{C}[\![e_i]\!] (\Diamond\varphi) \phi \rho \pi)$$

(7b) $\mathscr{C}[\![\textbf{case}\ x\ \textbf{of}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n]\!] \varphi \phi \rho \pi$

$$= \bigwedge_{i=1}^{n} \mathscr{C}[\![e_i]\!] \varphi \phi \rho \pi$$

(8) $\mathscr{C}[\![x\ e_1 \ldots e_n]\!] \varphi \phi \rho \pi \quad = (Undefined, \pi)$

(9) $\mathscr{C}[\![\textbf{let}\ x = e_0\ \textbf{in}\ e_1]\!] \varphi \phi \rho \pi = \mathscr{C}[\![e_1]\!] \varphi \phi \rho \pi$

(10) $\mathscr{C}[\![e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!] \varphi \phi \rho \pi$

$$= \mathscr{C}[\![e_0]\!] \varphi (\phi \cup \{f_1 \mapsto e_1, \ldots, f_n \mapsto e_n\}) \rho \pi$$

Figure 11: Counterexample and Witness Construction Rules

These rules are very similar to the verification rules given in Figure 10, with the addition of the parameter π, which gives the value of the current trace thus far. As each observable state in the program trace is processed in rules (5a-d), it is appended to the end of π and when a final truth value is obtained it is returned along with the value of π. Counterexamples and witnesses can of course be infinite in the form of a lasso consisting of a finite prefix and a loop, while only a finite trace will be returned using these rules. However, loops can be detected in the generated trace as the repetition of observable states. To prove that the constructed counterexample or witness is valid, we need to prove that it satisfies the original temporal property which was verified.

Theorem 6.1 (Validity) $\forall e \in Prog, \varphi \in$ WFF:
$(\mathscr{C}[\![e]\!] \, \varphi \, \emptyset \, \emptyset \, [] = (True, \pi) \Rightarrow \pi, 0 \vDash \varphi) \wedge (\mathscr{C}[\![e]\!] \, \varphi \, \emptyset \, \emptyset \, [] = (False, \pi) \Rightarrow \pi, 0 \nvDash \varphi)$

Proof. The proof of this is by structural induction on the program e. $\qquad\qquad\square$

Using these rules, we try to construct counterexamples for the two properties (mutual exclusion and non-starvation) for the example programs given in Section 3.

Example 1 For the program shown in Figure 4, the application of these rules for Property 1 (mutual exclusion) is shown below where we represent Property 1 by $\square\varphi$ and the function environment by ϕ. We also use the shorthand notation (X,Y) to denote the state *ObsState X Y*.

$\mathscr{C}[\![Cons \, (T,T) \, (f_1 \, es)]\!] \, (\square\varphi) \, \emptyset \, \emptyset \, []$
$= \{5a\}$
$(\mathscr{C}[\![Cons \, (T,T) \, (f_1 \, es)]\!] \, \varphi \, \emptyset \, \emptyset \, []) \wedge_v (\mathscr{C}[\![f_1 \, es]\!] \, (\square\varphi) \, \emptyset \, \emptyset \, [(T,T)])$
$= \{5d\}$
$(\varphi[(T,T)/s]) \wedge_v (\mathscr{C}[\![f_1 \, es]\!] \, (\square\varphi) \, \emptyset \, \emptyset \, [(T,T)])$
$= \{$calculation, 6a, 7b, 5a, 5d$\}$
$(\mathscr{C}[\![f_1 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,T)]) \wedge_v (\mathscr{C}[\![f_2 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(W,T)])$
$\wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$
$= \{6a\}$
$(\mathscr{C}[\![f_2 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(W,T)]) \wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$
$= \{$calculation, 6a, 7b, 5a, 5d$\}$
$(\mathscr{C}[\![f_2 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(W,T)])$
$\wedge_v (\mathscr{C}[\![f_4 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(U,T)])$
$\wedge_v (\mathscr{C}[\![f_5 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(W,W)])$
$\wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$
$= \{6a\}$
$(\mathscr{C}[\![f_4 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(U,T)])$
$\wedge_v (\mathscr{C}[\![f_5 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(W,W)])$
$\wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$
$= \{$calculation, 6a, 7b, 5a, 5d$\}$
$(\mathscr{C}[\![f_1 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2,f_4\} \, [(T,T),(W,T),(U,T),(T,T)])$
$\wedge_v (\mathscr{C}[\![f_4 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2,f_4\} \, [(T,T),(W,T),(U,T),(U,T)])$
$\wedge_v (\mathscr{C}[\![f_5 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(W,W)])$
$\wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$
$= \{6a\}$
$(\mathscr{C}[\![f_5 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1,f_2\} \, [(T,T),(W,T),(W,W)])$
$\wedge_v (\mathscr{C}[\![f_3 \, es]\!] \, (\square\varphi) \, \phi \, \{f_1\} \, [(T,T),(T,W)])$

$= \{$calculation, 6a, 7b, 5a, 5d$\}$
   $(\mathscr{C}[\![f_5 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1, f_2, f_5\} \ [(T,T),(W,T),(W,W),(W,W)])$
   $\wedge_v (\mathscr{C}[\![f_7 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1, f_2, f_5\} \ [(T,T),(W,T),(W,W),(U,W)])$
   $\wedge_v (\mathscr{C}[\![f_8 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1, f_2, f_5\} \ [(T,T),(W,T),(W,W),(W,U)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$
$= \{$6a$\}$
   $(\mathscr{C}[\![f_7 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1, f_2, f_5\} \ [(T,T),(W,T),(W,W),(U,W)])$
   $\wedge_v (\mathscr{C}[\![f_8 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1, f_2, f_5\} \ [(T,T),(W,T),(W,W),(W,U)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box\varphi) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$
$= \{$calculation, 6a, 7b, 5a, 5d$\}$
   $(\textit{False}, [(T,T),(W,T),(W,W),(U,W),(U,U)])$

We can see that the rules that are applied closely mirror those applied in the verification of this property, and that the following counterexample is generated:

Cons (ObsState T T) (Cons (ObsState W T) (Cons (ObsState W W) (Cons (ObsState U W)
(Cons (ObsState U U) Nil))))

Example 2 For the program shown in Figure 6, the application of these rules for Property 2 (non-starvation) is shown below where we represent Property 2 by $\Box(\varphi \Rightarrow \Diamond\psi)$ and the function environment by ϕ. We again use the shorthand notation (X,Y) to denote the state *ObsState X Y*.

   $\mathscr{C}[\![\textit{Cons} \ (T,T) \ (f_1 \ es)]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \emptyset \ \emptyset \ []$
$= \{$5a$\}$
   $(\mathscr{C}[\![\textit{Cons} \ (T,T) \ (f_1 \ es)]\!] \ (\varphi \Rightarrow \Diamond\psi) \ \emptyset \ \emptyset \ []) \wedge_v (\mathscr{C}[\![f_1 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \emptyset \ \emptyset \ [(T,T)])$
$= \{$5d$\}$
   $((\varphi \Rightarrow \Diamond\psi)[(T,T)/s]) \wedge_v (\mathscr{C}[\![f_1 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \emptyset \ \emptyset \ [(T,T)])$
$= \{$calculation, 3, 6a, 7b, 5a, 5d$\}$
   $(\mathscr{C}[\![f_1 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(T,T)]) \wedge_v (\mathscr{C}[\![f_2 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(W,T)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ \pi_4)$
$= \{$6a$\}$
   $(\mathscr{C}[\![f_2 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(W,T)]) \wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$
$= \{$calculation, 3, 6a, 7b, 5a, 5d$\}$
   $(\mathscr{C}[\![f_2 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(W,T)])$
   $\wedge_v (\mathscr{C}[\![f_4 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(U,T)])$
   $\wedge_v (\mathscr{C}[\![f_5 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(W,W)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$
$= \{$6a$\}$
   $(\mathscr{C}[\![f_4 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(U,T)])$
   $\wedge_v (\mathscr{C}[\![f_5 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(W,W)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$
$= \{$calculation, 3, 6a, 7b, 5a, 5d$\}$
   $(\mathscr{C}[\![f_1 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2, f_4\} \ [(T,T),(W,T),(U,T),(T,T)])$
   $\wedge_v (\mathscr{C}[\![f_4 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2, f_4\} \ [(T,T),(W,T),(U,T),(U,T)])$
   $\wedge_v (\mathscr{C}[\![f_5 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1, f_2\} \ [(T,T),(W,T),(W,W)])$
   $\wedge_v (\mathscr{C}[\![f_3 \ es]\!] \ (\Box(\varphi \Rightarrow \Diamond\psi)) \ \phi \ \{f_1\} \ [(T,T),(T,W)])$

= {6a}
$$(\mathscr{C}[\![f_5\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1, f_2\}\ [(T, T), (W, T), (W, W)])$$
$$\wedge_v(\mathscr{C}[\![f_3\ es]\!]\ (\Box(\varphi \Rightarrow \Diamond\psi))\ \phi\ \{f_1\}\ [(T, T), (T, W)])$$
= {calculation, 6a, 7b, 5a, 3, 5b, 6b}
$$(\textit{False}, [(T, T), (W, T), (W, W), (W, W)])$$

The following counterexample with a loop at the end is therefore generated:

Cons (*ObsState T T*) (*Cons* (*ObsState W T*) (*Cons* (*ObsState W W*) (*Cons* (*ObsState W W*) *Nil*)))

7 Conclusion and Related Work

In previous work [6], we have shown how a fold/unfold program transformation technique can be used to verify both safety and liveness properties of reactive systems which have been specified using a functional language. However, counterexamples and witnesses were not constructed using this approach. In this paper, we have therefore extended these previous techniques to address this shortcoming to construct a counterexample trace when a temporal property does not hold, and a witness when it does.

Fold/unfold transformation techniques have also been developed for verifying temporal properties for logic programs [11, 15, 4, 1, 8]). Some of these techniques have been developed only for safety properties, while others can be used to verify both safety and liveness properties. Due to the use of a different programming paradigm, it is difficult to compare the relative power of these techniques to our own. However, none of these techniques construct counterexamples when the temporal property does not hold.

Very few techniques have been developed for verifying temporal properties for functional programs other than the work of Lisitsa and Nemytykh [12, 2]. Their approach uses supercompilation [17, 16] as the fold/unfold transformation methodology, where our own approach uses distillation [5, 7]. Their approach can verify only safety properties, and does not construct counterexamples when the safety property does not hold.

One other area of work related to our own is the work on using Higher Order Recursion Schemes (HORS) to verify temporal properties of functional programs. HORS are a kind of higher order tree grammar for generating a (potentially infinite) tree and are well-suited to the purpose of verification since they have a decidable mu-calculus model checking problem, as proved by Ong [14]. Kobayashi [13] first showed how this approach can be used to verify safety properties of higher order functional programs and for the construction of counterexamples when the safety property does not hold. This approach was then extended to also verify liveness properties by Lester et al. [10], but counterexamples are not constructed when the liveness property does not hold. These approaches have a very bad worst-case time complexity, but techniques have been developed to ameliorate this to a certain extent. It does however appear likely that this approach will be able to verify more properties than our own approach but much less efficiently.

Acknowledgements

References

[1] Alberto Pettorossi and Maurizio Proietti and Valerio Senni (2009): *Deciding Full Branching Time Logic by Program Transformation*. In: *19th International Symposium on Logic-Based Program Synthesis and Transformation*, pp. 5–21 doi:10.1007/978-3-642-12592-8_2.

[2] Alexei Lisitsa and Andrei P. Nemytykh (2008): *Reachability Analysis in Verification via Supercompilation*. International Journal of Foundations of Computer Science 19(4), pp. 953–969 doi:10.1142/S0129054108006066.

[3] E.M. Clarke, E.A. Emerson & A.P. Sistla (1986): *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems 8(2), pp. 244–263 doi:10.1145/5397.5399.

[4] Fabio Fioravanti and Alberto Pettorossi and Maurizio Proietti (2001): *Verification of Sets of Infinite State Processes Using Program Transformation*. In: *11th International Workshop on Logic Based Program Synthesis and Transformation*, pp. 111–128 doi:10.1007/3-540-45607-4_7.

[5] G.W. Hamilton (2007): *Distillation: Extracting the Essence of Programs*. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 61–70 doi:10.1145/1244381.1244391.

[6] G.W. Hamilton (2015): *Verifying Temporal Properties of Reactive Systems by Transformation*. Electronic Proceedings of Theoretical Computer Science 199, pp. 33–50 doi:10.4204/EPTCS.199.

[7] G.W. Hamilton & N.D. Jones (2012): *Distillation With Labelled Transition Systems*. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM, pp. 15–24 doi:10.1145/2103746.2103753.

[8] Hirohisa Seki (2011): *Proving Properties of Co-Logic Programs by Unfold/Fold Transformations*. In: *21st International Symposium on Logic-Based Program Synthesis and Transformation*, pp. 205–220 doi:10.1007/978-3-642-32211-2_14.

[9] L. Lamport (1974): *A New Solution of Dijkstra's Concurrent Programming Problem*. Communications of the ACM 17(8), pp. 453–455 doi:10.1145/361082.361093.

[10] Lester, M.M. and Neatherway, R.P. and Ong, C.-H. L. and Ramsay, S.J. (2010): *Model Checking Liveness Properties of Higher-Order Functional Programs*. Unpublished.

[11] M. Leuschel & T. Massart (1999): *Infinite State Model Checking by Abstract Interpretation and Program Specialisation*. In: *9th International Workshop on Logic Programming Synthesis and Transformation*, pp. 62–81 doi:10.1007/10720327_5.

[12] A. Lisitsa & A. Nemytykh (2007): *Verification as a Parameterized Testing (Experiments with the SCP4 Supercompiler)*. Programming and Computer Software 33(1), pp. 14–23 doi:10.1134/S0361768807010033.

[13] Naoki Kobayashi (2009): *Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs*. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pp. 416–428 doi:10.1145/1480881.1480933.

[14] C.-H. L. Ong (2006): *On Model-Checking Trees Generated by Higher-Order Recursion Schemes*. In: *Proceedings of Logic in Computer Science, LICS*, IEEE Computer Society Press, pp. 81–90 doi:10.1109/LICS.2006.38.

[15] Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan & Scott A. Smolka (2000): *Verification of Parameterized Systems Using Logic Program Transformations*. In: *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 172–187 doi:10.1007/3-540-46419-0_13.

[16] M.H. Sørensen, R. Glück & N.D. Jones (1996): *A Positive Supercompiler*. Journal of Functional Programming 6(6), pp. 811–838 doi:10.1017/S0956796800002008.

[17] V.F. Turchin (1986): *The Concept of a Supercompiler*. ACM Transactions on Programming Languages and Systems 8(3), pp. 90–121 doi:10.1145/5956.5957.