Verifying Temporal Properties of Reactive Systems by Transformation

G.W. Hamilton
School of Computing and Lero
Dublin City University
Ireland
hamilton@computing.dcu.ie

We show how program transformation techniques can be used for the verification of both safety and liveness properties of reactive systems. In particular, we show how the program transformation technique *distillation* can be used to transform reactive systems specified in a functional language into a simplified form that can subsequently be analysed to verify temporal properties of the 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 the errors they contain being correctly identified.

1 Introduction

Formal verification of software components is gaining more and more prominence as a viable methodology for increasing the reliability and reducing the cost of software production. We consider here the problem of verifying properties of *reactive systems*, i.e., systems which continuously react to external events by changing their internal state and producing outputs. The properties of such systems are usually expressed using a *temporal logic* such as Computational Tree Logic (CTL) or Linear-time Temporal Logic (LTL). These logics are used to express *safety* properties which essentially state that nothing bad will happen, and *liveness* properties which essentially state that something good will eventually happen.

Model checking is a well established technique originally developed for the verification of temporal properties of finite state systems [4]. However, reactive systems usually have an infinite number of states. Model checking techniques therefore need to be extended to handle such systems, but the problem of verifying such systems is undecidable in general. Most proposed approaches to this problem are semi-automatic and involve either mathematical *(co-)induction* [3, 8] or *abstraction* to finite state models [11, 17]. Fold/unfold program transformation techniques have more recently been proposed as an automatic approach to this problem. Folding corresponds to the application of a (co-)inductive hypothesis and generalisation corresponds to abstraction. Many such techniques have been developed for logic programs (e.g. [13, 18, 5, 1, 9]). However, very few such techniques have been developed for functional programs (with the work of Lisitsa and Nemytykh [14, 2] using supercompilation [20] being a notable exception), and these deal only with safety properties.

In this paper, we show 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 a functional language. The program transformation technique which we use is our own *distillation* [6, 7] which builds on top of positive supercompilation [19], but is much more powerful. Distillation is used to transform programs defining reactive systems into a simplified form which makes them much easier to analyse. We then define a number of verification rules on this simplified form to verify temporal properties of the system. In these verification rules, intermediate structures are given an undefined value, thus abstracting the system to a finite number of states, but leading to a loss of information. We argue that, since distillation removes more intermediate structures than positive supercompilation, more accurate results are obtained. 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, revealing a number of errors.

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. Section 6 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.

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

Figure 1: Language Grammar

A program in the language 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. We assume that the patterns in a **case** expression are non-overlapping and 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** $p_1 \rightarrow e_1 \mid \cdots \mid p_k \rightarrow e_k$ 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 $\overset{r}{\leadsto}$ 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).

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

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

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

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

Figure 2: One-Step Reduction Relation

We use the notation $e \overset{r}{\leadsto}$ 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 $\overset{r}{\leadsto}^*$ denotes the reflexive transitive closure of $\overset{r}{\leadsto}$:

$$\begin{array}{ll} e \overset{r}{\leadsto}, \text{ iff } \exists e'.e \overset{r}{\leadsto} e' & e\Downarrow, \text{ iff } \exists v.e \Downarrow v \\ e \Downarrow v, \text{ iff } e \overset{r}{\leadsto}^* v \wedge \neg(v \overset{r}{\leadsto}) & e\Uparrow, \text{ iff } \forall e'.e \overset{r}{\leadsto}^* e' \Rightarrow e' \overset{r}{\leadsto} \end{array}$$

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*, our specifications can be trivially derived from these. Reactive systems have to react to a series of *external events* by updating their *state*. In order to facilitate this, we make use of a *stream* datatype, which is defined as follows:

$$\mathit{Stream}\ a ::= \mathit{Cons}\ a\ \mathit{Stream}$$

A stream is therefore an infinite list of elements of type a. Our programs will map an input stream of external events and an initial state to an output stream of *observable states*, which give 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:

$$\mathit{Event} ::= \mathit{Request}_1 \mid \mathit{Request}_2 \mid \mathit{Take}_1 \mid \mathit{Take}_2 \mid \mathit{Release}_1 \mid \mathit{Release}_2$$

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). In all of the following examples, the variable es represents the external event stream, and s_1 and s_2 represent the states of the two processes respectively.

Example 1 In the first example shown in Figure 3, each process can request access to the critical resource if neither process is using it, take the critical resource if it is waiting for it, and release the critical resource if it is using it.

```
f es T T
where
f = λes s1 s2.Cons (ObsState s1 s2) (case es of
                  Cons e es → case e of
                                Request1 → case s1 of
                                             U → f es s1 s2
                                           | _ → case s2 of
                                                   U → f es s1 s2
                                                 | _ → f es W s2
                              | Request2 → case s2 of
                                             U → f es s1 s2
                                           | _ → case s1 of
                                                   U → f es s1 s2
                                                 | _ → f es s1 W
                              | Take1    → case s1 of
                                             W → f es U s2
                                           | _ → f es s1 s2
                              | Take2    → case s2 of
                                             W → f es s1 U
                                           | _ → f es s1 s2
                              | Release1 → case s1 of
                                             U → f es T s2
                                           | _ → f es s1 s2
                              | Release2 → case s2 of
                                             U → f es s1 T
                                           | _ → f es s1 s2)
```

Figure 3: Example 1

Example 2 In the second example shown in Figure 4, each process can request access to the critical resource if it is thinking, 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.

```
f es T T
where
f = λes s1 s2.Cons (ObsState s1 s2) (case es of
                        Cons e es  →  case e of
                                 Request1 →  case s1 of
                                                T → f es W s2
                                              | _ → f es s1 s2
                               | Request2 →  case s2 of
                                                T → f es s1 W
                                              | _ → f es s1 s2
                               | Take1    →  case s1 of
                                                W → case s2 of
                                                      T → f es U s2
                                                    | _ → f es s1 s2
                                              | _  → f es s1 s2
                               | Take2    →  case s2 of
                                                W → case s1 of
                                                      T → f es s1 U
                                                    | _ → f es s1 s2
                                              | _  → f es s1 s2
                               | Release1 →  case s1 of
                                                U → f es T s2
                                              | _  → f es s1 s2
                               | Release2 →  case s2 of
                                                U → f es s1 T
                                              | _  → f es s1 s2)
```

Figure 4: Example 2

Example 3 In the final example in Figure 5, we implement Lamport's bakery algorithm [10] for two processes. In this example, to request access to the critical resource, each process must take a 'ticket' with a number, and the process with the lowest valued ticket is given precedence. A ticket value of zero indicates that a process has not requested to use the critical resource, so when a process releases the critical resource its ticket value is reset to zero. We therefore add two state variables t_1 and t_2 which give the current ticket number for each process, but are not part of the observable state. These are natural numbers belonging to the following datatype:

$$Nat ::= Zero \mid Succ\ Nat$$

Note that, since there is no limit to the number of a ticket (ticket numbers will keep increasing if each process alternately requests access to the critical resource before the other process has released it), this is an example of an infinite state system which can cause problems for some model checkers.

```
f es T T Zero Zero
where
f = λes $s_1$ $s_2$ $t_1$ $t_2$.Cons (ObsState $s_1$ $s_2$)
                    (case es of
                       Cons e es → case e of
                                      $Request_1$ → case $s_1$ of
                                                      T → f es W $s_2$ (Succ $t_2$) $t_2$
                                                    | _ → f es $s_1$ $s_2$ $t_1$ $t_2$
                                    | $Request_2$ → case $s_2$ of
                                                      T → f es $s_1$ W $t_1$ (Succ $t_1$)
                                                    | _ → f es $s_1$ $s_2$ $t_1$ $t_2$
                                    | $Take_1$    → case $s_1$ of
                                                      W → case $s_2$ of
                                                            T → f es U $s_2$ $t_1$ $t_2$
                                                          | _ → case ($t_1 < t_2$) of
                                                                  True  → f es U $s_2$ $t_1$ $t_2$
                                                                | False → f es $s_1$ $s_2$ $t_1$ $t_2$
                                                    | _  → f es $s_1$ $s_2$ $t_1$ $t_2$
                                    | $Take_2$    → case $s_2$ of
                                                      W → case $s_1$ of
                                                            T → f es $s_1$ U $t_1$ $t_2$
                                                          | _ → case ($t_2 < t_1$) of
                                                                  True  → f es $s_1$ U $t_1$ $t_2$
                                                                | False → f es $s_1$ $s_2$ $t_1$ $t_2$
                                                    | _  → f es $s_1$ $s_2$ $t_1$ $t_2$
                                    | $Release_1$ → case $s_1$ of
                                                      U → f es T $s_2$ Zero $t_2$
                                                    | _ → f es $s_1$ $s_2$ $t_1$ $t_2$
                                    | $Release_2$ → case $s_2$ of
                                                      U → f es $s_1$ T $t_1$ Zero
                                                    | _ → f es $s_1$ $s_2$ $t_1$ $t_2$)
```

Figure 5: Example 3

4 Specification of Temporal Properties

In this section, we describe how temporal properties of reactive systems defined in our functional language 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:

- $\neg\varphi$
- $\varphi \vee \psi$
- $\varphi \wedge \psi$
- $\varphi \Rightarrow \psi$

- $\Box\varphi$
- $\Diamond\varphi$
- $\bigcirc\varphi$

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.

Propositional models for linear-time temporal formulas consist of an infinite sequence of states $\pi = \langle s_0, s_1, \ldots \rangle$ such that each state s_i supplies an assignment to the atomic propositions. The satisfaction relation is extended to formulas in LTL for a model π and position i as follows.

$$\begin{array}{lll}
\pi,i \models p & \text{iff} & p \in s_i \\
\pi,i \models \neg\varphi & \text{iff} & \pi,i \not\models \varphi \\
\pi,i \models \varphi \vee \psi & \text{iff} & \pi,i \models \varphi \text{ or } \pi,i \models \psi \\
\pi,i \models \varphi \wedge \psi & \text{iff} & \pi,i \models \varphi \text{ and } \pi,i \models \psi \\
\pi,i \models \varphi \Rightarrow \psi & \text{iff} & \pi,i \not\models \varphi \text{ or } \pi,i \models \psi \\
\pi,i \models \Box\varphi & \text{iff} & \forall j \geq i.\pi, j \models \varphi \\
\pi,i \models \Diamond\varphi & \text{iff} & \exists j \geq i.\pi, j \models \varphi \\
\pi,i \models \bigcirc\varphi & \text{iff} & \pi,i+1 \models \varphi
\end{array}$$

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

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:

$$\mathit{TruthVal} ::= \mathit{True} \mid \mathit{False} \mid \mathit{Undefined}$$

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 be undecidable. 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:

```
□(case s of
      ObsState s₁ s₂ → case s₁ of
                          U → case s₂ of
                                 U → False
                               | _ → True
                        | _ → 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:

$$\begin{array}{l} \Box((\textbf{case } s \textbf{ of} \\ \quad \mathit{ObsState}\ s_1\ s_2 \ \rightarrow\ \textbf{case } s_1 \textbf{ of} \\ \qquad\qquad W \rightarrow \mathit{True} \\ \qquad\qquad \mid \_ \ \rightarrow \mathit{False}) \Rightarrow \Diamond(\textbf{case } s \textbf{ of} \\ \qquad\qquad\qquad\qquad \mathit{ObsState}\ s_1\ s_2 \ \rightarrow\ \textbf{case } s_1 \textbf{ of} \\ \qquad\qquad\qquad\qquad\qquad U \rightarrow \mathit{True} \\ \qquad\qquad\qquad\qquad\qquad \mid \_ \ \rightarrow \mathit{False})) \end{array}$$

The specification of this property for process 2 is similar.

5 Verification of Temporal Properties

In this section, we show how temporal properties of reactive systems defined in our functional language can be verified. To facilitate this, we first of all transform the reactive systems definitions into a simplified form using distillation [6, 7], a powerful program transformation technique which builds on top of the supercompilation transformation [20, 19]. Due to the nature of the reactive systems definitions, in which the input is an external event stream, and the output is a stream of observable states, the programs resulting from this transformation will take the form $e^{\emptyset}$, where e^{ρ} is defined as follows.

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

The **let** variables are added to the set ρ, and will not be used in the selectors of **case** expressions. These **let** variables are given an undefined value during verification, thus abstracting the system to a finite number of states.

We define our verification rules on this restricted form of program as shown in Figure 6. 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. The logical connectives $\wedge$, $\vee$, $\Rightarrow$ and $\neg$ are defined in the usual way for a Kleene three-valued logic in our language in rules (1-4). 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 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

$$
\begin{array}{lll}
(1) & \mathscr{P}\llbracket e\rrbracket\ (\varphi \wedge \psi)\ \phi\ \rho & = \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \varphi\ \phi\ \rho)\ \textbf{of} \\
& & \quad \begin{array}{ll} \mathit{True} & \rightarrow \mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho \\ \mid \mathit{False} & \rightarrow \mathit{False} \\ \mid \mathit{Undefined} & \rightarrow \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho)\ \textbf{of} \\ & \qquad \begin{array}{ll} \mathit{False} & \rightarrow \mathit{False} \\ \mid _ & \rightarrow \mathit{Undefined} \end{array} \end{array} \\
(2) & \mathscr{P}\llbracket e\rrbracket\ (\varphi \vee \psi)\ \phi\ \rho & = \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \varphi\ \phi\ \rho)\ \textbf{of} \\
& & \quad \begin{array}{ll} \mathit{True} & \rightarrow \mathit{True} \\ \mid \mathit{False} & \rightarrow \mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho \\ \mid \mathit{Undefined} & \rightarrow \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho)\ \textbf{of} \\ & \qquad \begin{array}{ll} \mathit{True} & \rightarrow \mathit{True} \\ \mid _ & \rightarrow \mathit{Undefined} \end{array} \end{array} \\
(3) & \mathscr{P}\llbracket e\rrbracket\ (\varphi \Rightarrow \psi)\ \phi\ \rho & = \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \varphi\ \phi\ \rho)\ \textbf{of} \\
& & \quad \begin{array}{ll} \mathit{True} & \rightarrow \mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho \\ \mid \mathit{False} & \rightarrow \mathit{True} \\ \mid \mathit{Undefined} & \rightarrow \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \psi\ \phi\ \rho)\ \textbf{of} \\ & \qquad \begin{array}{ll} \mathit{True} & \rightarrow \mathit{True} \\ \mid _ & \rightarrow \mathit{Undefined} \end{array} \end{array} \\
(4) & \mathscr{P}\llbracket e\rrbracket\ (\neg\varphi)\ \phi\ \rho & = \textbf{case}\ (\mathscr{P}\llbracket e\rrbracket\ \varphi\ \phi\ \rho)\ \textbf{of} \\
& & \quad \begin{array}{ll} \mathit{True} & \rightarrow \mathit{False} \\ \mid \mathit{False} & \rightarrow \mathit{True} \\ \mid \mathit{Undefined} & \rightarrow \mathit{Undefined} \end{array} \\
(5a) & \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ (\Box\varphi)\ \phi\ \rho & = \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ \varphi\ \phi\ \emptyset \wedge \mathscr{P}\llbracket e_1\rrbracket\ (\Box\varphi)\ \phi\ \rho \\
(5b) & \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ (\Diamond\varphi)\ \phi\ \rho & = \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ \varphi\ \phi\ \emptyset \vee \mathscr{P}\llbracket e_1\rrbracket\ (\Diamond\varphi)\ \phi\ \rho \\
(5c) & \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ (\bigcirc\varphi)\ \phi\ \rho & = \mathscr{P}\llbracket e_1\rrbracket\ \varphi\ \phi\ \rho \\
(5d) & \mathscr{P}\llbracket \mathit{Cons}\ e_0\ e_1\rrbracket\ \varphi\ \phi\ \rho & = v, \text{ where } \varphi[e_0/s] \Downarrow v \\
(6a) & \mathscr{P}\llbracket f\ x_1 \ldots x_n\rrbracket\ (\Box\varphi)\ \phi\ \rho & = \begin{cases} \mathit{True}, & \text{if } f \in \rho \\ \mathscr{P}\llbracket e[x_1/x_1', \ldots, x_n/x_n']\rrbracket\ (\Box\varphi)\ \phi\ (\rho \cup \{f\}), & \text{otherwise} \end{cases} \\
& & \ \ \text{where } \phi(f) = \lambda x_1' \ldots x_n'.e \\
(6b) & \mathscr{P}\llbracket f\ x_1 \ldots x_n\rrbracket\ (\Diamond\varphi)\ \phi\ \rho & = \begin{cases} \mathit{False}, & \text{if } f \in \rho \\ \mathscr{P}\llbracket e[x_1/x_1', \ldots, x_n/x_n']\rrbracket\ (\Diamond\varphi)\ \phi\ (\rho \cup \{f\}), & \text{otherwise} \end{cases} \\
& & \ \ \text{where } \phi(f) = \lambda x_1' \ldots x_n'.e \\
(6c) & \mathscr{P}\llbracket f\ x_1 \ldots x_n\rrbracket\ \varphi\ \phi\ \rho & = \begin{cases} \mathit{Undefined}, & \text{if } f \in \rho \\ \mathscr{P}\llbracket e[x_1/x_1', \ldots, x_n/x_n']\rrbracket\ \varphi\ \phi\ (\rho \cup \{f\}), & \text{otherwise} \end{cases} \\
& & \ \ \text{where } \phi(f) = \lambda x_1' \ldots x_n'.e \\
(7a) & \multicolumn{2}{l}{\mathscr{P}\llbracket \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n\rrbracket\ (\Diamond\varphi)\ \phi\ \rho} \\
& & = (\bigvee\limits_{p_i \in F} \mathscr{P}\llbracket e_i\rrbracket\ (\Diamond\varphi)\ \phi\ \rho) \vee (\bigwedge\limits_{i=1}^{n} \mathscr{P}\llbracket e_i\rrbracket\ (\Diamond\varphi)\ \phi\ \rho) \\
(7b) & \multicolumn{2}{l}{\mathscr{P}\llbracket \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n\rrbracket\ \varphi\ \phi\ \rho} \\
& & = \bigwedge\limits_{i=1}^{n} \mathscr{P}\llbracket e_i\rrbracket\ \varphi\ \phi\ \rho \\
(8) & \mathscr{P}\llbracket x\ e_1 \ldots e_n\rrbracket\ \varphi\ \phi\ \rho & = \mathit{Undefined} \\
(9) & \mathscr{P}\llbracket \textbf{let}\ x = e_0\ \textbf{in}\ e_1\rrbracket\ \varphi\ \phi\ \rho & = \mathscr{P}\llbracket e_1\rrbracket\ \varphi\ \phi\ \rho \\
(10) & \multicolumn{2}{l}{\mathscr{P}\llbracket e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n\rrbracket\ \varphi\ \phi\ \rho} \\
& & = \mathscr{P}\llbracket e_0\rrbracket\ \varphi\ (\phi \cup \{f_1 \mapsto e_1, \ldots, f_n \mapsto e_n\})\ \rho
\end{array}
$$

Figure 6: Verification Rules

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) $\mathscr{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = True \Rightarrow \pi, 0 \models \varphi \wedge \mathscr{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = False \Rightarrow \pi, 0 \not\models \varphi$ where π is a model for e.

Proof
The proof of this is by recursion induction on the verification rules $\mathscr{P}$.

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 $\Box$ and $\Diamond$, and verification of each of these temporal operators will terminate when a function is re-encountered.

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 The result of distilling Example 1 is shown in Figure 7, and the LTS representation of this program is shown in Figure 8 (for ease of presentation of this and subsequent LTSs, transitions back into the same state have been omitted). Verification of Property 1 (mutual exclusion) fails for this transformed program; if the input event stream starts with $Request_1$, $Request_2$, $Take_1$, $Take_2$, $\ldots$, then the function calling sequence is f_1, f_2, f_5, f_7, f_9, $\ldots$ and we can see that we end up in the function f_9, where both processes are using the critical resource.

Example 2 The result of distilling Example 2 is shown in Figure 9, and the LTS representation of this program is shown in Figure 10. Verification of Property 1 (mutual exclusion) succeeds for this transformed program; we can easily see that there is no state in which both processes are using the critical resource. When trying to prove this property, as soon as we re-encounter any of the functions within the program, the value *True* is returned by verification rule (6a). However, verification of Property 2 (non-starvation) fails; if the input event stream starts with $Request_1$, $Request_2$, $\ldots$, then the function calling

```
f₁ es
where
f₁ = λes.Cons (ObsState T T) (case es of
                                 Cons e es → case e of
                                                Request₁ → f₂ es
                                              | Request₂ → f₃ es
                                              | _        → f₁ es)
f₂ = λes.Cons (ObsState W T) (case es of
                                 Cons e es → case e of
                                                Take₁    → f₄ es
                                              | Request₂ → f₅ es
                                              | _        → f₂ es)
f₃ = λes.Cons (ObsState T W) (case es of
                                 Cons e es → case e of
                                                Request₁ → f₅ es
                                              | Take₂    → f₆ es
                                              | _        → f₃ es)
f₄ = λes.Cons (ObsState U T) (case es of
                                 Cons e es → case e of
                                                Release₁ → f₁ es
                                              | _        → f₄ es)
f₅ = λes.Cons (ObsState W W) (case es of
                                 Cons e es → case e of
                                                Take₁ → f₇ es
                                              | Take₂ → f₈ es
                                              | _     → f₅ es)
f₆ = λes.Cons (ObsState T U) (case es of
                                 Cons e es → case e of
                                                Release₂ → f₁ es
                                              | _        → f₆ es)
f₇ = λes.Cons (ObsState U W) (case es of
                                 Cons e es → case e of
                                                Release₁ → f₃ es
                                              | Take₂    → f₉ es
                                              | _        → f₇ es)
f₈ = λes.Cons (ObsState W U) (case es of
                                 Cons e es → case e of
                                                Release₂ → f₂ es
                                              | Take₁    → f₉ es
                                              | _        → f₈ es)
f₉ = λes.Cons (ObsState U U) (case es of
                                 Cons e es → case e of
                                                Release₁ → f₆ es
                                              | Release₂ → f₄ es
                                              | _        → f₉ es)
```

Figure 7: Result of Distilling Example 1

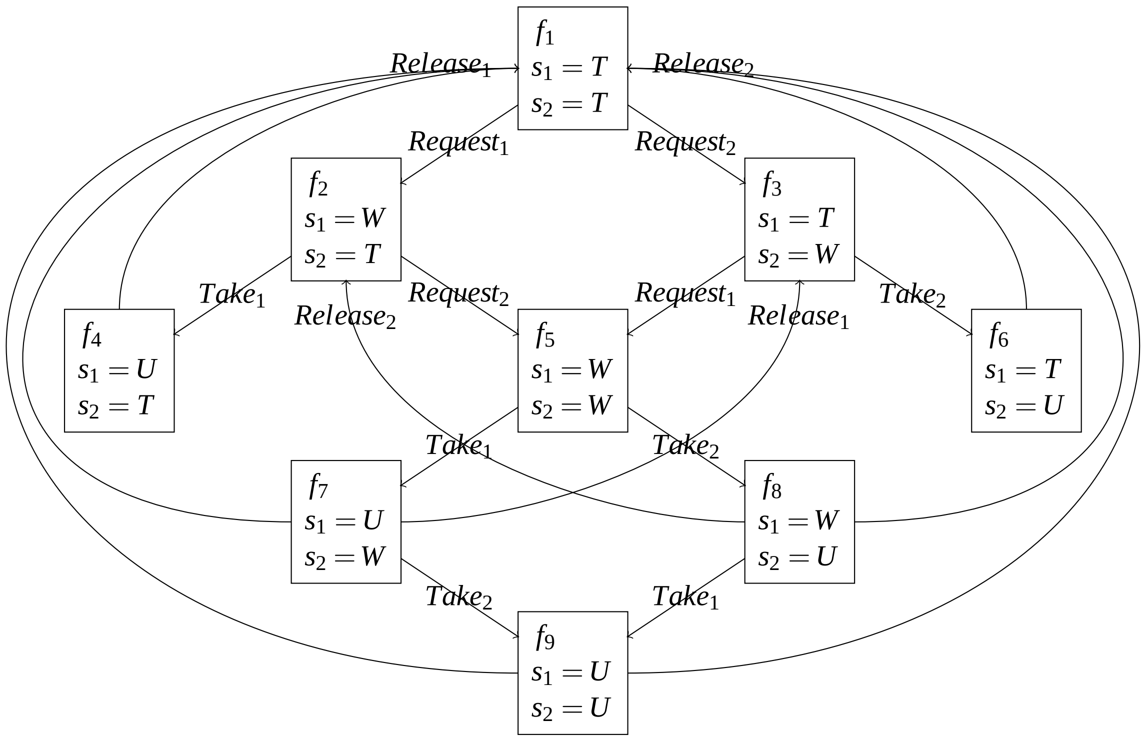
f_1 $s_1 = T$ $s_2 = T$
f_2 $s_1 = W$ $s_2 = T$
f_3 $s_1 = T$ $s_2 = W$
f_4 $s_1 = U$ $s_2 = T$
f_5 $s_1 = W$ $s_2 = W$
f_6 $s_1 = T$ $s_2 = U$
f_7 $s_1 = U$ $s_2 = W$
f_8 $s_1 = W$ $s_2 = U$
f_9 $s_1 = U$ $s_2 = U$
$Release_1$
$Release_2$
$Request_1$
$Request_2$
$Take_1$
$Request_2$
$Request_1$
$Take_2$
$Release_2$
$Release_1$
$Take_1$
$Take_2$
$Take_2$
$Take_1$

Figure 8: LTS Representation of Distilling Example 1

sequence is f_1, f_2, f_5, and we can see that we end up within the function f_5. At this point, both processes are waiting for the critical resource, so we need to prove that they will eventually get to use it. When trying to prove this eventuality property, we immediately re-encounter the function f_5, so the value *False* is returned by verification rule (6b).

Example 3 The result of distilling Example 3 is shown in Figure 11, and the LTS representation of this program is shown in Figure 12. We can see that the use of tickets is completely transformed away and that the resulting program has a finite number of states. This is where distillation provides an advantage over other transformation techniques such as positive supercompilation which are not able to remove as many intermediate data structures and thus to transform away the use of tickets. Verification of both Property 1 (mutual exclusion) and Property 2 (non-starvation) succeed for this transformed program. The proof of Property 1 is quite straightforward and similar to the proof of this property for Example 2. If we consider the proof of Property 2 for process 1, if the event $Request_1$ has just occurred, then we must be in one of the functions f_2, f_7 or f_9. There is a single exit from f_7 to f_9 by event $Take_2$, and a single exit from f_9 to path f_2 by event $Release_2$. Thus, we must eventually end up in function f_2 after a $Request_1$ event. From f_2, we must eventually end up in a state in which process 1 is using the critical resource, either directly by event $Take_1$, or indirectly with event $Request_2$ preceding $Take_1$. The proof of Property 2 for process 2 is similar.

6 Conclusion and Related Work

In this paper, 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. Many corresponding techniques have been developed for verifying temporal properties for logic

```
f1 es
where
f1 = λes.Cons (ObsState T T) (case es of
                                   Cons e es → case e of
                                                    Request1 → f2 es
                                                  | Request2 → f3 es
                                                  | _        → f1 es)
f2 = λes.Cons (ObsState W T) (case es of
                                   Cons e es → case e of
                                                    Take1    → f4 es
                                                  | Request2 → f5 es
                                                  | _        → f2 es)
f3 = λes.Cons (ObsState T W) (case es of
                                   Cons e es → case e of
                                                    Request1 → f5 es
                                                  | Take2    → f6 es
                                                  | _        → f3 es)
f4 = λes.Cons (ObsState U T) (case es of
                                   Cons e es → case e of
                                                    Release1 → f1 es
                                                  | _        → f4 es)
f5 = λes.Cons (ObsState W W) (case es of
                                   Cons e es → case e of
                                                    _ → f5 es)
f6 = λes.Cons (ObsState T U) (case es of
                                   Cons e es → case e of
                                                    Release2 → f1 es
                                                  | _        → f6 es)
```

Figure 9: Result of Distilling Example 2

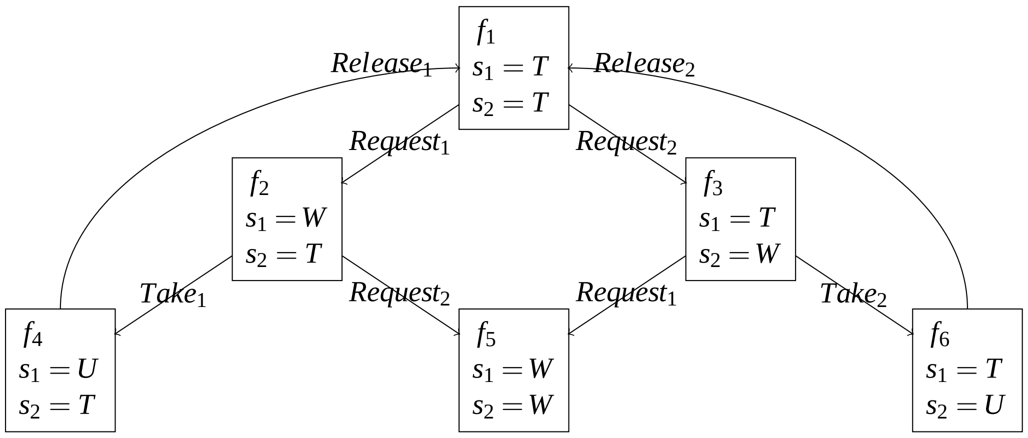
f_1 $s_1 = T$ $s_2 = T$
f_2 $s_1 = W$ $s_2 = T$
f_3 $s_1 = T$ $s_2 = W$
f_4 $s_1 = U$ $s_2 = T$
f_5 $s_1 = W$ $s_2 = W$
f_6 $s_1 = T$ $s_2 = U$
$Release_1$
$Release_2$
$Request_1$
$Request_2$
$Take_1$
$Request_2$
$Request_1$
$Take_2$

Figure 10: LTS Representation of Distilling Example 2

```
f_1 es
where
f_1 = λes.Cons (ObsState T T) (case es of
                                  Cons e es → case e of
                                                Request_1 → f_2 es
                                              | Request_2 → f_3 es
                                              | _         → f_1 es)
f_2 = λes.Cons (ObsState W T) (case es of
                                  Cons e es → case e of
                                                Take_1    → f_4 es
                                              | Request_2 → f_6 es
                                              | _         → f_2 es)
f_3 = λes.Cons (ObsState T W) (case es of
                                  Cons e es → case e of
                                                Take_2    → f_5 es
                                              | Request_1 → f_7 es
                                              | _         → f_3 es)
f_4 = λes.Cons (ObsState U T) (case es of
                                  Cons e es → case e of
                                                Release_1 → f_1 es
                                              | Request_2 → f_8 es
                                              | _         → f_4 es)
f_5 = λes.Cons (ObsState T U) (case es of
                                  Cons e es → case e of
                                                Release_2 → f_1 es
                                              | Request_1 → f_9 es
                                              | _         → f_5 es)
f_6 = λes.Cons (ObsState W W) (case es of
                                  Cons e es → case e of
                                                Take_1 → f_8 es
                                              | _      → f_6 es)
f_7 = λes.Cons (ObsState W W) (case es of
                                  Cons e es → case e of
                                                Take_2 → f_9 es
                                              | _      → f_7 es)
f_8 = λes.Cons (ObsState U W) (case es of
                                  Cons e es → case e of
                                                Release_1 → f_3 es
                                              | _         → f_8 es)
f_9 = λes.Cons (ObsState W U) (case es of
                                  Cons e es → case e of
                                                Release_2 → f_2 es
                                              | _         → f_9 es)
```

Figure 11: Result of Distilling Example 3

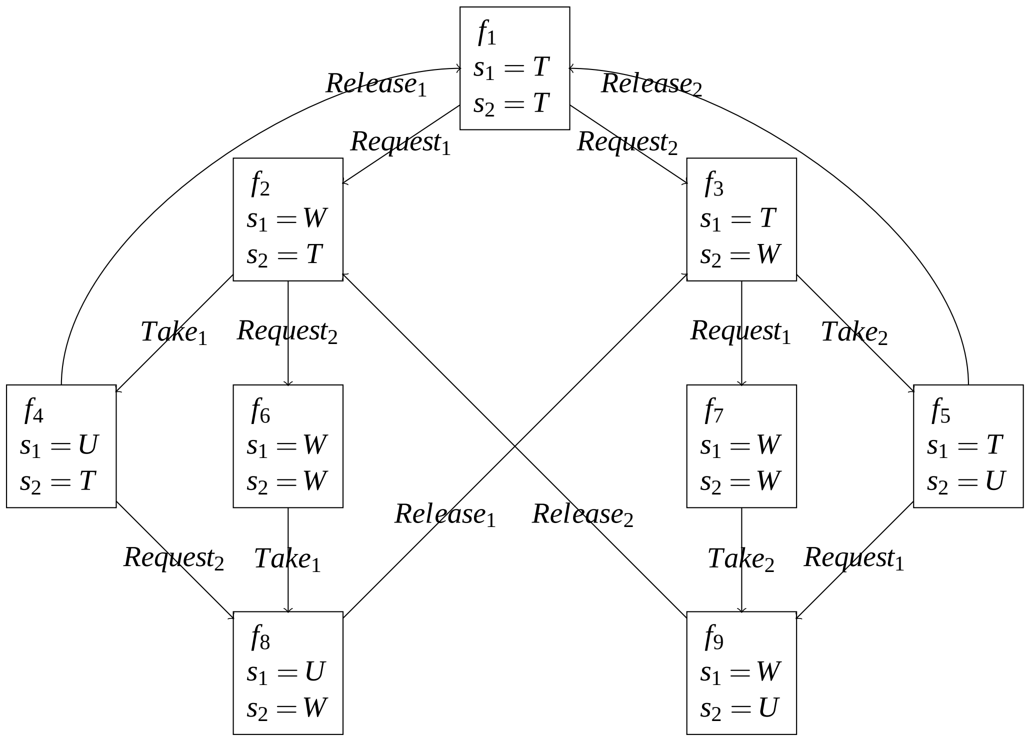
f_1 $s_1 = T$ $s_2 = T$
f_2 $s_1 = W$ $s_2 = T$
f_3 $s_1 = T$ $s_2 = W$
f_4 $s_1 = U$ $s_2 = T$
f_6 $s_1 = W$ $s_2 = W$
f_7 $s_1 = W$ $s_2 = W$
f_5 $s_1 = T$ $s_2 = U$
f_8 $s_1 = U$ $s_2 = W$
f_9 $s_1 = W$ $s_2 = U$
$Release_1$
$Release_2$
$Request_1$
$Request_2$
$Take_1$
$Request_2$
$Request_1$
$Take_2$
$Release_1$
$Release_2$
$Request_2$
$Take_1$
$Take_2$
$Request_1$

Figure 12: LTS Representation of Distilling Example 3

programs [13, 18, 5, 1, 9]). 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, we argue that the use of a more powerful program transformation algorithm will remove more intermediate data structures, and thus be capable of proving more properties directly within the same framework, without the need for making use of external solvers.

Very few techniques have been developed for verifying temporal properties for functional programs other than the work of Lisitsa and Nemytykh [14, 2]. Their approach uses supercompilation [20, 19] as the fold/unfold transformation methodology, where our own approach uses distillation [6, 7]. Since distillation has been shown to be more powerful than positive supercompilation, it follows that we should be able to verify more properties using our approach (such as the properties we verify for Lamport's bakery algorithm in Example 3). Also, the work of Lisitsa and Nemytykh can verify only safety properties, while our approach can be used to verify both safety and liveness properties.

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 [16]. Kobayashi [15] first showed how this approach can be used to verify safety properties of higher order functional programs. This approach was then extended to also verify liveness properties by Lester et al. [12]. 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

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

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] Amir Pnueli and Elad Shahar (1996): *A Platform for Combining Deductive with Algorithmic Verification*. In: *8th International Conference on Computer Aided Verification*, pp. 184–195 doi:10.1007/3-540-61474-5_68.

[4] 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.

[5] 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.

[6] 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.

[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] Henny Sipma and Tomás E. Uribe and Zohar Manna (1999): *Deductive Model Checking*. *Formal Methods in System Design* 15(1), pp. 49–74 doi:10.1023/A:1008791913551.

[9] 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.

[10] 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.

[11] Lenore D. Zuck and Amir Pnueli (2004): *Model Checking and Abstraction to the Aid of Parameterized Systems (A Survey)*. *Computer Languages, Systems & Structures* 30(3-4), pp. 139–169 doi:10.1016/j.cl.2004.02.006.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Parosh Aziz Abdulla and Giorgio Delzanno and Noomene Ben Henda and Ahmed Rezine (2009): *Monotonic Abstraction: on Efficient Verification of Parameterized Systems*. *International Journal of Foundations of Computer Science* 20(5), pp. 779–801 doi:10.1142/S0129054109006887.

[18] 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.

[19] 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.

[20] 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.