Generating Counterexamples for Model Checking by Transformation

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
`hamilton@computing.dcu.ie`

VPT 2016

Outline

Introduction

- We consider the problem of verifying properties of reactive systems
    - continuously react to external events by changing their internal state and producing outputs
- Properties of such systems are usually expressed using a temporal logic
    - safety properties (nothing bad will ever happen)
    - liveness properties (something good will eventually happen)
- One well established technique for this verification is model checking
    - originally developed for finite state systems
    - reactive systems often have an infinite number of states
- A major advantage of model checking is that it can generate a counterexample explaining the reason why a property fails

Background

- Fold/unfold program transformation has been proposed as an automatic approach to model checking
  - folding corresponds to the application of a (co)-inductive hypothesis
  - generalization corresponds to abstraction
- Many techniques have been developed for logic programs
  - e.g. Leuschel & Massart, 1999; Roychoudhuri et al., 2000; Fioravanti et al., 2001; Pettorossi et al., 2009; Seki, 2011
- Less techniques have been developed for functional programs
  - notable exception: Lisitsa & Nemytykh, 2007 & 2008
- Unfortunately, none of these techniques generate counterexamples when the temporal property does not hold

Approach

1. Apply distillation to the program defining the reactive system
    - produces a simplified form of program which is easier to analyse
    - removes more intermediate data structures so less generalization is required
2. Define a number of verification rules on the resulting simplified form of program
    - less limitations than those associated with previous techniques
    - always terminates, but may not produce a meaningful result
3. Extend the verification rules to construct program traces
    - produces a witness when a property holds
    - produces a counterexample when a property fails

Language

Syntax

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

Language

Semantics

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

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

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

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

Specifying Reactive Systems

Reactive systems have to react to a series of external events by updating their state. We use a list datatype:

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

We will apply our techniques to example systems intended to model mutually exclusive access to a shared resource for two processes, so external events belong to the following datatype:

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

System states belong to the following datatype:

$$SysState ::= State\ ProcState\ ProcState$$

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

Transformation

We first of all transform the reactive systems definitions into simplified form $e^{\emptyset}$ using distillation where e^{ρ} is defined as follows:

Distilled Form

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

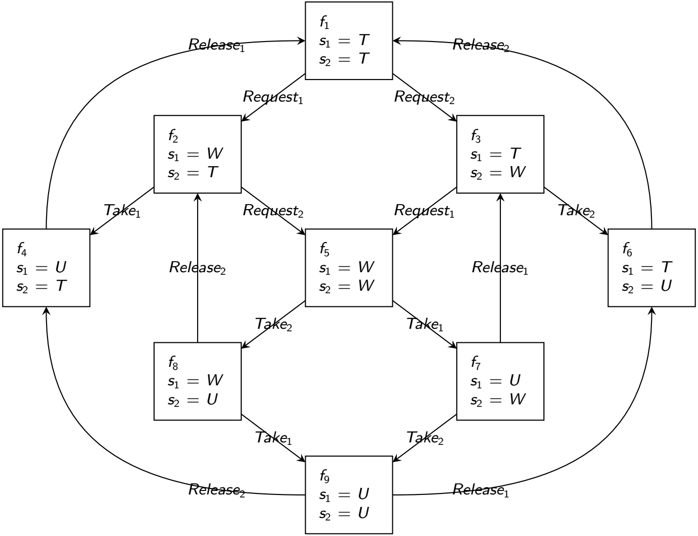
let variables are added to the set ρ, and will not be used as case selectors.

By abstracting over all **let** variables, we obtain a finite state approximation of the original system.

Example 1

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

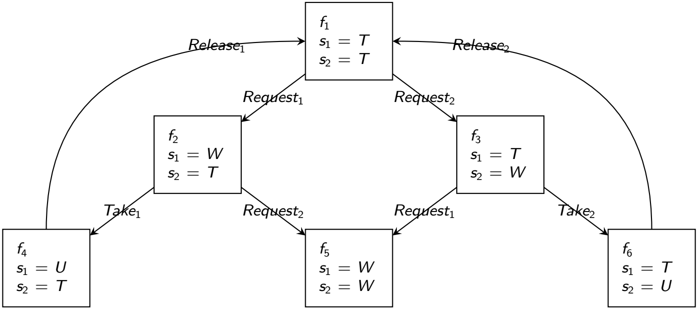
LTS Representation of Example 1



Example 2

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

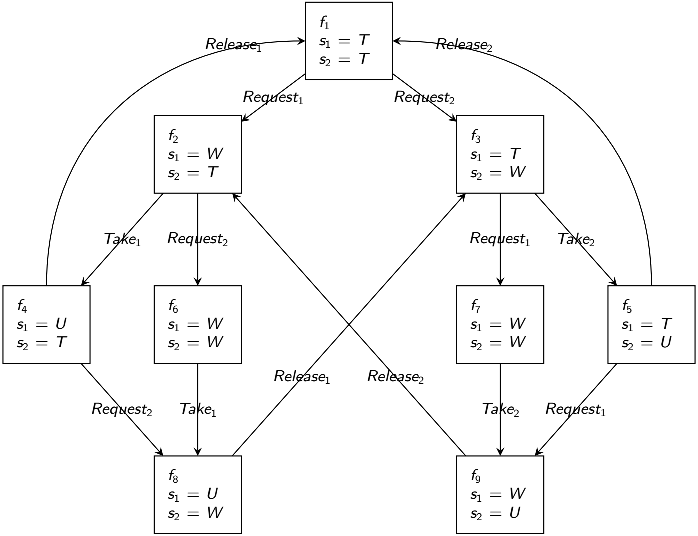
LTS Representation of Example 2



Example 3

$$
\begin{array}{lll}
\multicolumn{3}{l}{Cons\ (ObsState\ T\ T)\ (f_1\ es)} \\
\multicolumn{3}{l}{\mathbf{where}} \\
f_1 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Request_1 \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es) \\
 & & \qquad\qquad \mid Request_2 \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es) \\
f_2 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Take_1 \rightarrow Cons\ (ObsState\ U\ T)\ (f_4\ es) \\
 & & \qquad\qquad \mid Request_2 \rightarrow Cons\ (ObsState\ W\ W)\ (f_6\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es) \\
f_3 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Take_2 \rightarrow Cons\ (ObsState\ T\ U)\ (f_5\ es) \\
 & & \qquad\qquad \mid Request_1 \rightarrow Cons\ (ObsState\ W\ W)\ (f_7\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es) \\
f_4 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Release_1 \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es) \\
 & & \qquad\qquad \mid Request_2 \rightarrow Cons\ (ObsState\ U\ W)\ (f_8\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ U\ T)\ (f_4\ es) \\
f_5 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Release_2 \rightarrow Cons\ (ObsState\ T\ T)\ (f_1\ es) \\
 & & \qquad\qquad \mid Request_1 \rightarrow Cons\ (ObsState\ W\ U)\ (f_9\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ T\ U)\ (f_5\ es) \\
f_6 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Take_1 \rightarrow Cons\ (ObsState\ U\ W)\ (f_8\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ W\ W)\ (f_6\ es) \\
f_7 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Take_2 \rightarrow Cons\ (ObsState\ W\ U)\ (f_9\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ W\ W)\ (f_7\ es) \\
f_8 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Release_1 \rightarrow Cons\ (ObsState\ T\ W)\ (f_3\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ U\ W)\ (f_8\ es) \\
f_9 & = & \lambda es.\mathbf{case}\ es\ \mathbf{of} \\
 & & \quad Cons\ e\ es \rightarrow \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\qquad Release_2 \rightarrow Cons\ (ObsState\ W\ T)\ (f_2\ es) \\
 & & \qquad\qquad \mid \_ \rightarrow Cons\ (ObsState\ W\ U)\ (f_9\ es)
\end{array}
$$

LTS Representation of Transformed Example 3



Linear-time Temporal Logic

Syntax

$$\begin{array}{rcll}
\varphi, \psi & ::= & \top & \text{True} \\
 & \mid & \bot & \text{False} \\
 & \mid & p & \text{Propositional Variable} \\
 & \mid & \neg\varphi & \text{Negation} \\
 & \mid & \varphi \vee \psi & \text{Disjunction} \\
 & \mid & \varphi \wedge \psi & \text{Conjunction} \\
 & \mid & \varphi \Rightarrow \psi & \text{Implication} \\
 & \mid & \Box\varphi & \text{Always} \\
 & \mid & \Diamond\varphi & \text{Some Time} \\
 & \mid & \bigcirc\varphi & \text{Next Time}
\end{array}$$

Linear-time Temporal Logic

Semantics

Models π consist of an infinite number of states $\langle s_0, s_1, \ldots \rangle$ such that each state supplies an assignment to the atomic propositions. For a model π and position i:

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

Temporal Properties

We translate the atomic propositions of temporal formulae into our functional language, using the following datatype for truth values:

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

Property 1: Mutual Exclusion

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

Temporal Properties

Property 2: Non-Starvation (Process 1)

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

And similarly for Process 2.

Verification

We define verification rules $\mathcal{P}[\![e]\!]\ \varphi\ \phi\ \rho$

- e is an expression in distilled form
- φ is the temporal formula to be verified
- ϕ is a function variable environment
- ρ is the set of previously encountered function calls (used for the detection of loops)

Main aspects of verification rules:

- **let** variables are given the value *Undefined*.
- On encountering a loop:
  - If verifying a $\square$ property, return the value *True*; this corresponds to the standard greatest fixed point calculation.
  - If verifying a $\diamond$ property, return the value *False*; this corresponds to the standard least fixed point calculation.
  - Otherwise, return the value *Undefined*.

Verification

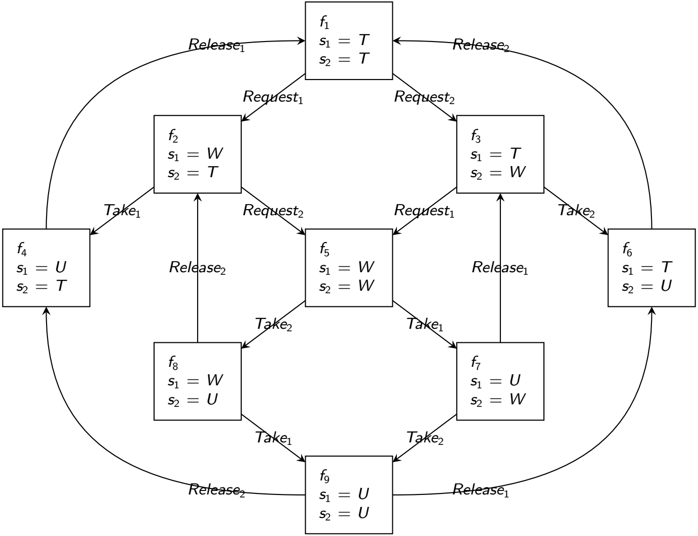
Theorem (Soundness)

$$\forall e \in \mathit{Prog}, \mathit{es} \in \mathit{List\ Event}, \pi \in \mathit{List\ State}, \varphi \in \text{WFF}:$$
$$(e\ \mathit{es} \overset{r*}{\rightsquigarrow} \pi) \Rightarrow (\mathcal{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = \mathit{True} \Rightarrow \pi, 0 \vDash \varphi)$$
$$\wedge\ (\mathcal{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset = \mathit{False} \Rightarrow \pi, 0 \nvDash \varphi)$$

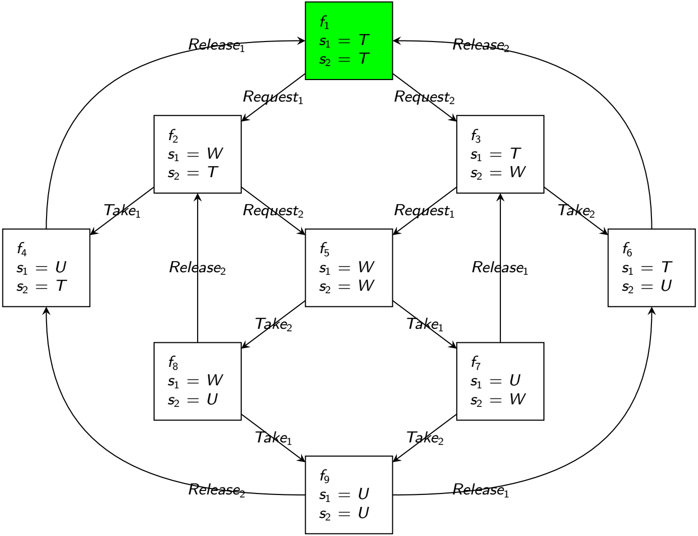
Theorem (Termination)

$\forall e \in$ Prog, $\varphi \in$ WFF: $\mathcal{P}[\![e]\!]\ \varphi\ \emptyset\ \emptyset$ always terminates.

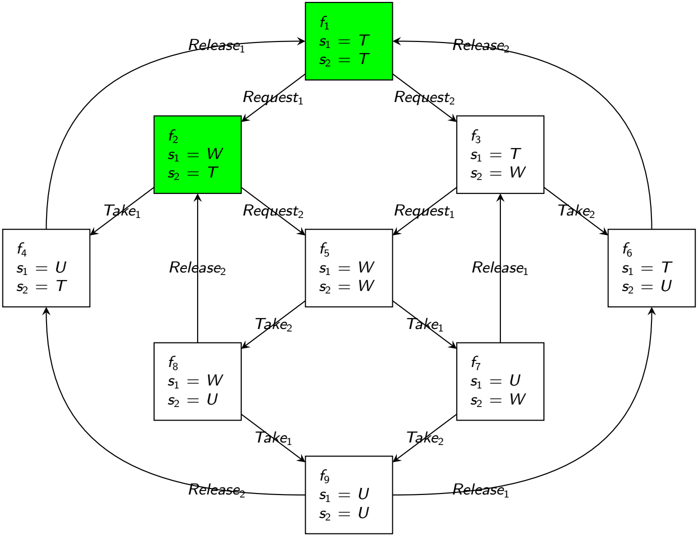
Verification of Property 1 for Example 1



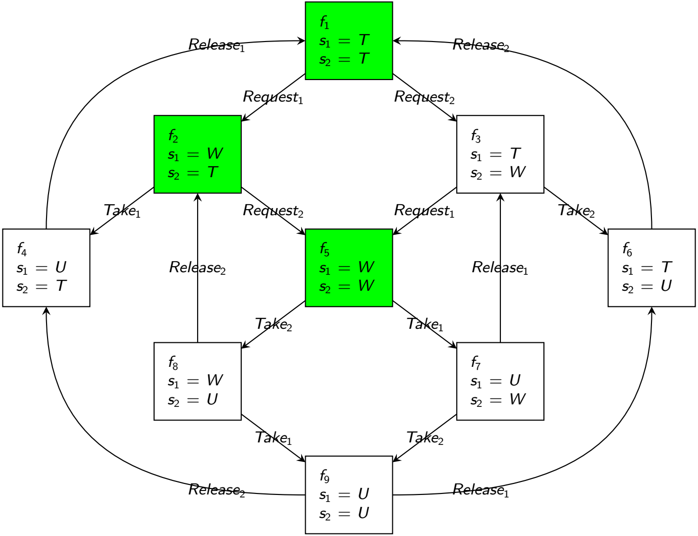
Verification of Property 1 for Example 1



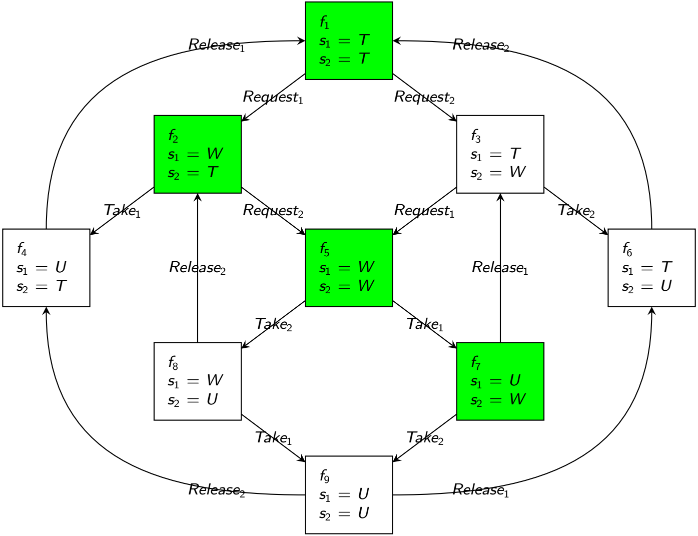
Verification of Property 1 for Example 1



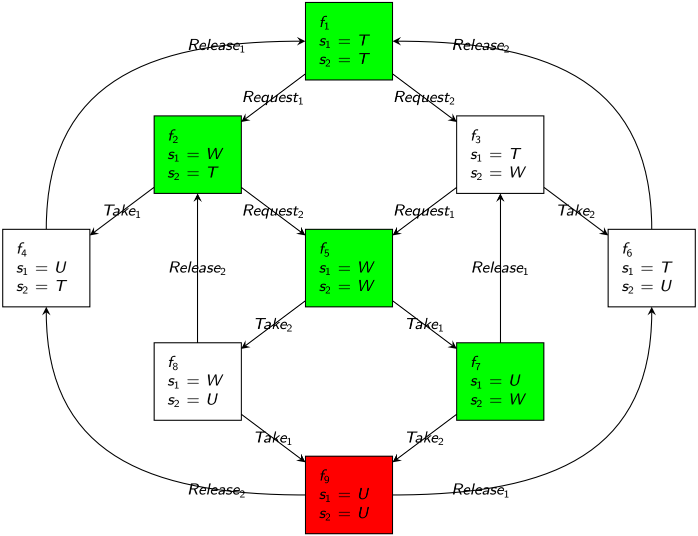
Verification of Property 1 for Example 1



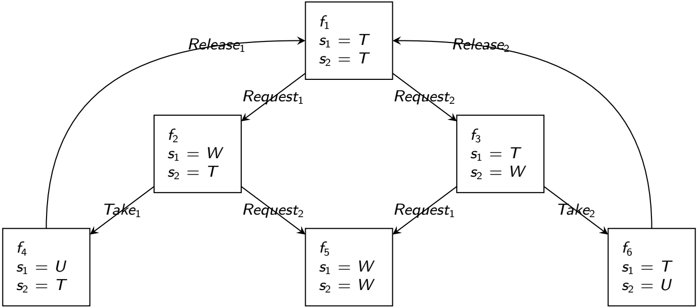
Verification of Property 1 for Example 1



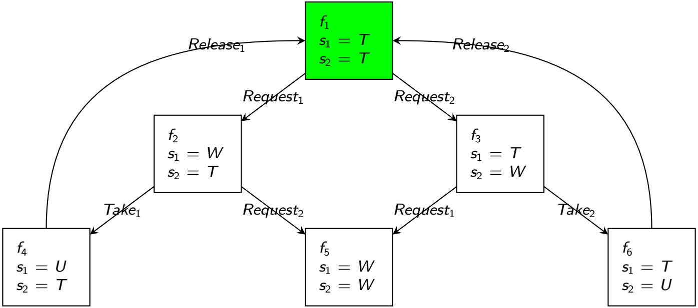
Verification of Property 1 for Example 1



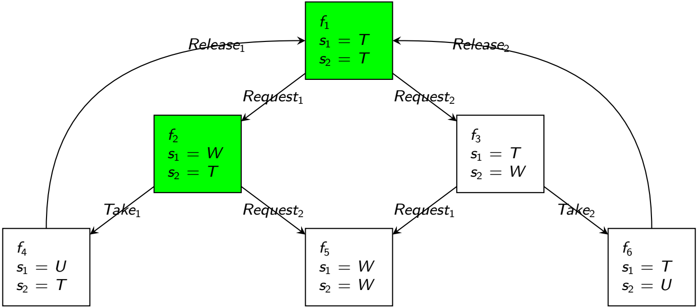
Verification of Property 1 for Example 2



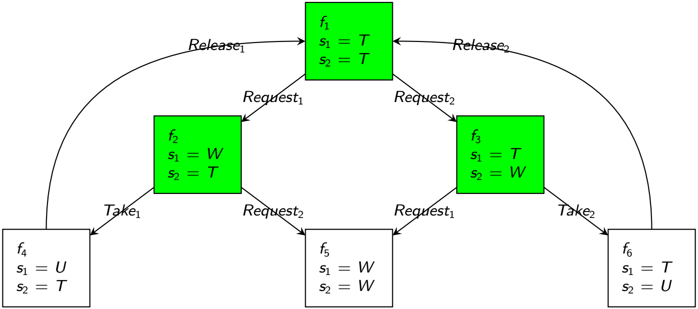
Verification of Property 1 for Example 2



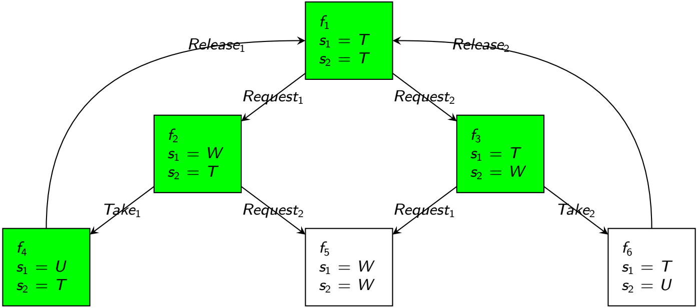
Verification of Property 1 for Example 2



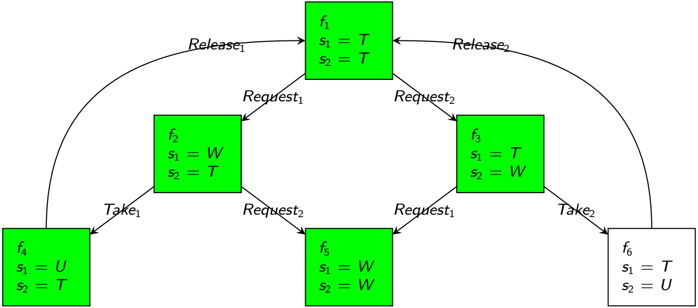
Verification of Property 1 for Example 2



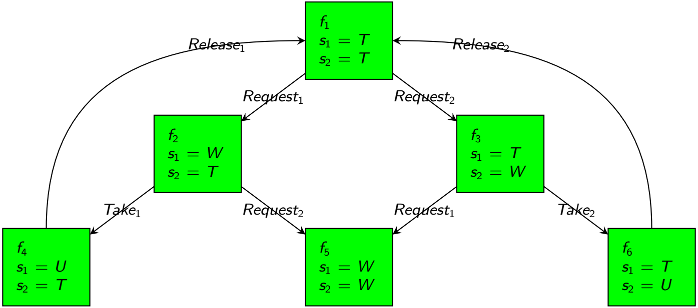
Verification of Property 1 for Example 2



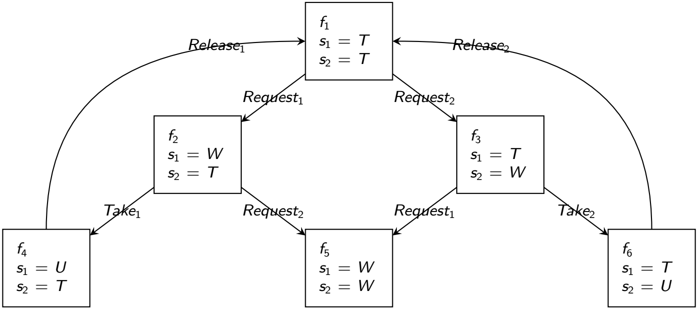
Verification of Property 1 for Example 2



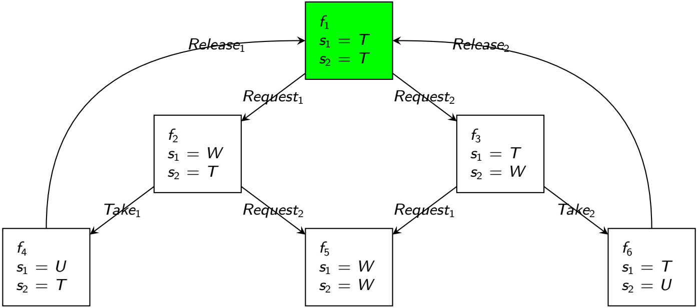
Verification of Property 1 for Example 2



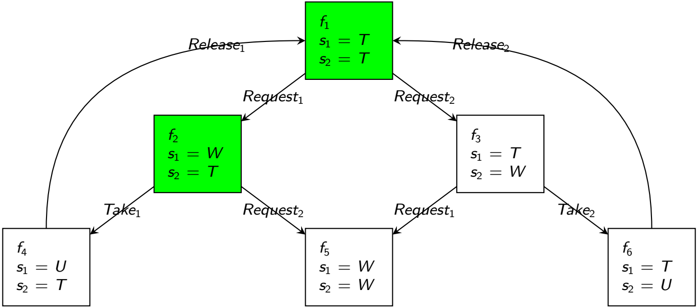
Verification of Property 2 for Example 2



Verification of Property 2 for Example 2



Verification of Property 2 for Example 2



f_1
$s_1 = T$
$s_2 = T$

f_2
$s_1 = W$
$s_2 = T$

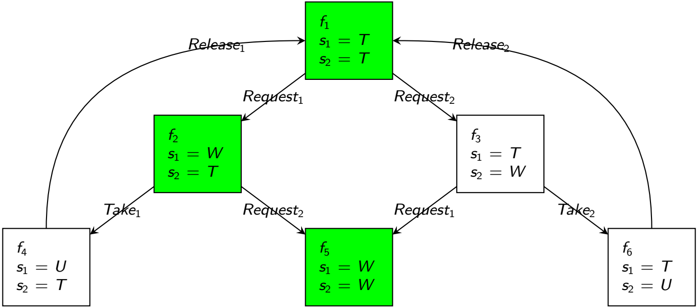
f_3
$s_1 = T$
$s_2 = W$

f_4
$s_1 = U$
$s_2 = T$

f_5
$s_1 = W$
$s_2 = W$

f_6
$s_1 = T$
$s_2 = U$

Verification of Property 2 for Example 2


f_1
$s_1 = T$
$s_2 = T$

f_2
$s_1 = W$
$s_2 = T$

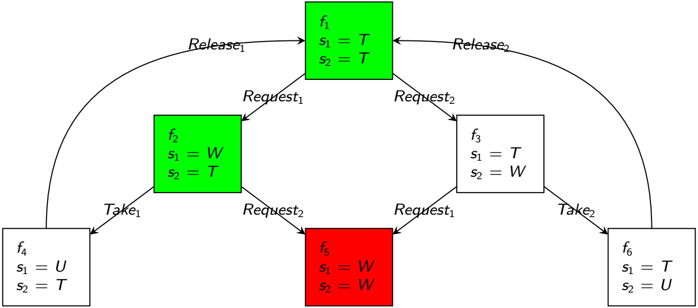
f_3
$s_1 = T$
$s_2 = W$

f_4
$s_1 = U$
$s_2 = T$

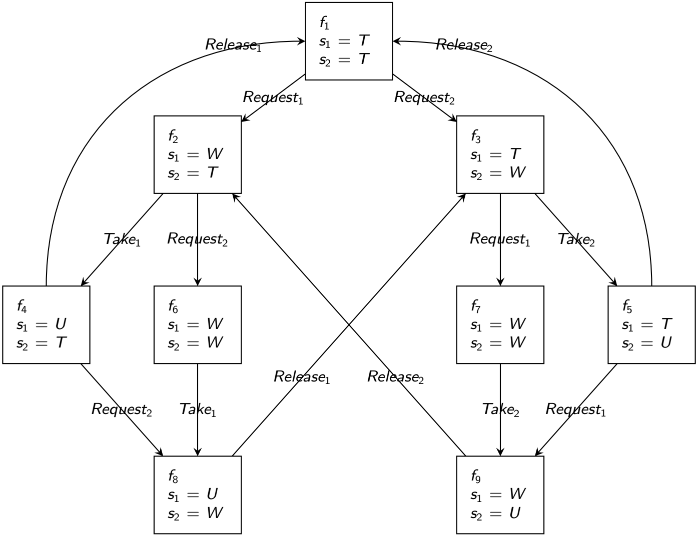
f_5
$s_1 = W$
$s_2 = W$

f_6
$s_1 = T$
$s_2 = U$

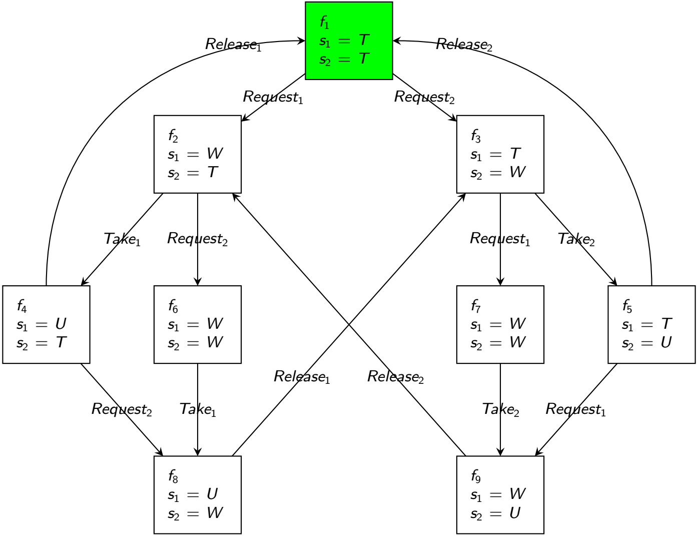
Verification of Property 2 for Example 2



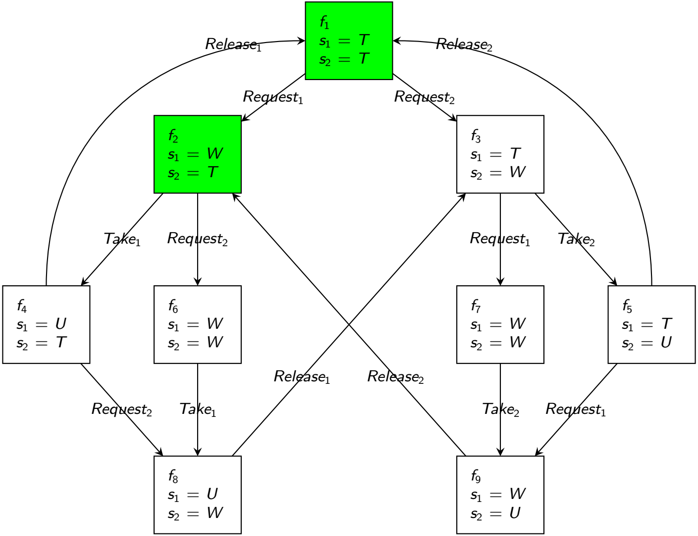
Verification of Property 1 for Example 3



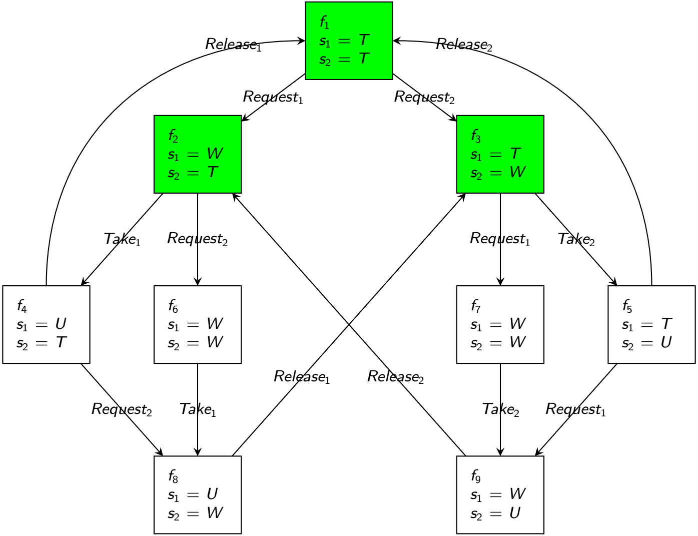
Verification of Property 1 for Example 3



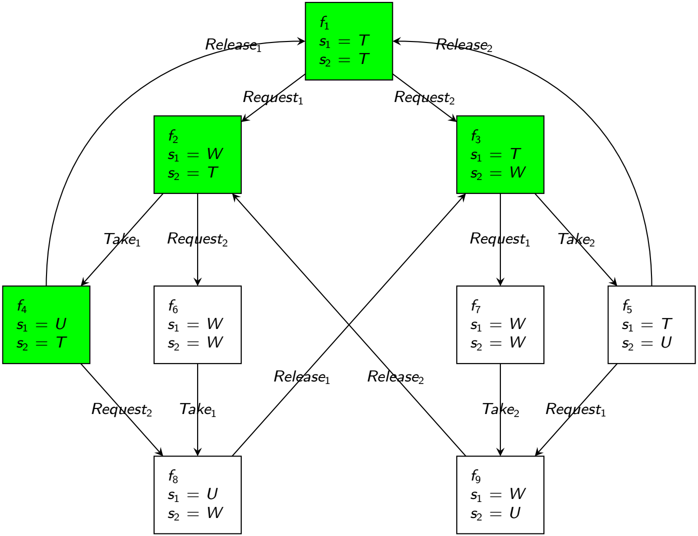
Verification of Property 1 for Example 3



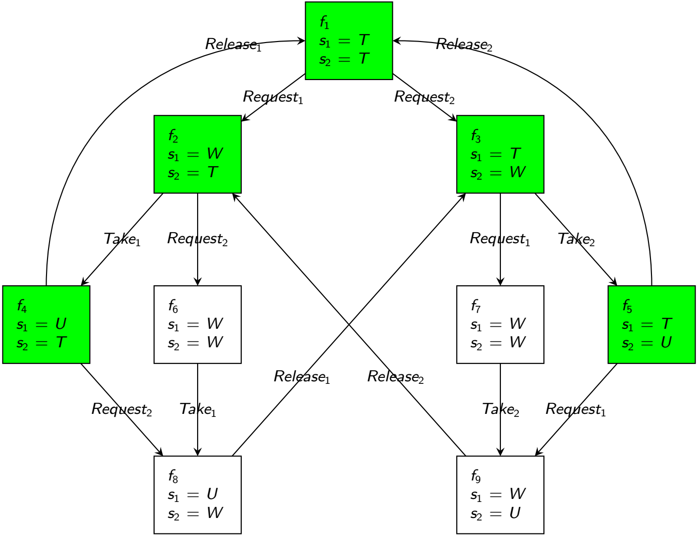
Verification of Property 1 for Example 3



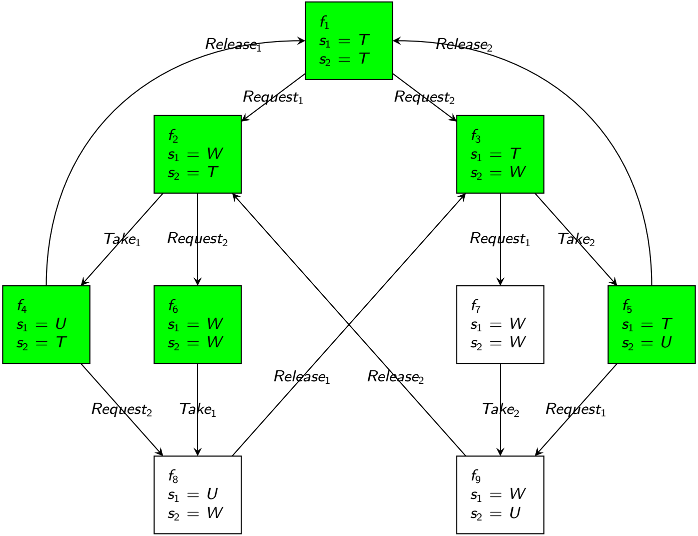
Verification of Property 1 for Example 3



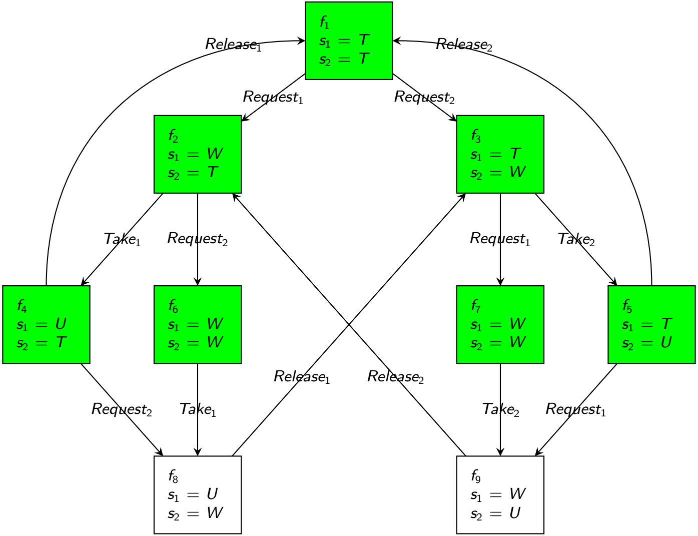
Verification of Property 1 for Example 3



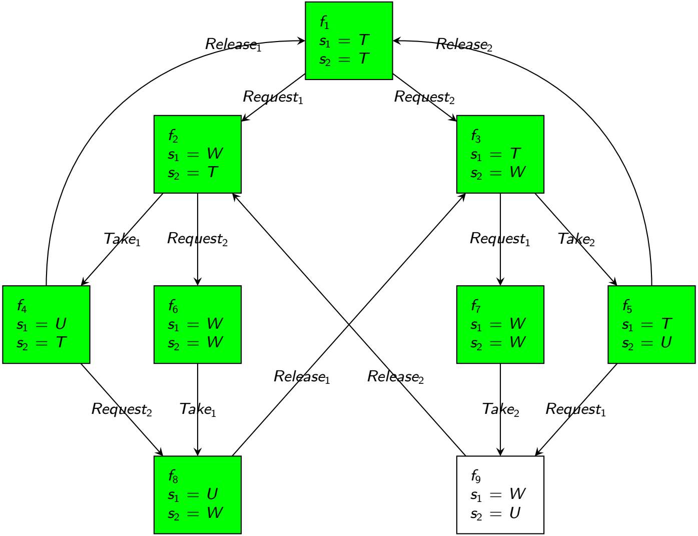
Verification of Property 1 for Example 3



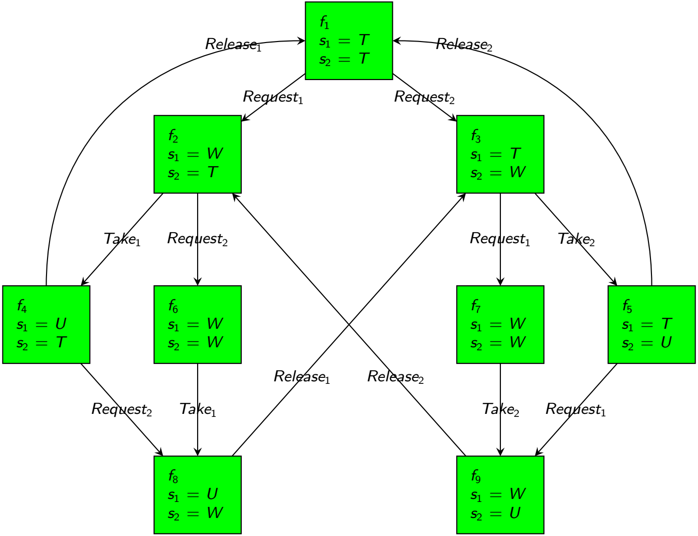
Verification of Property 1 for Example 3



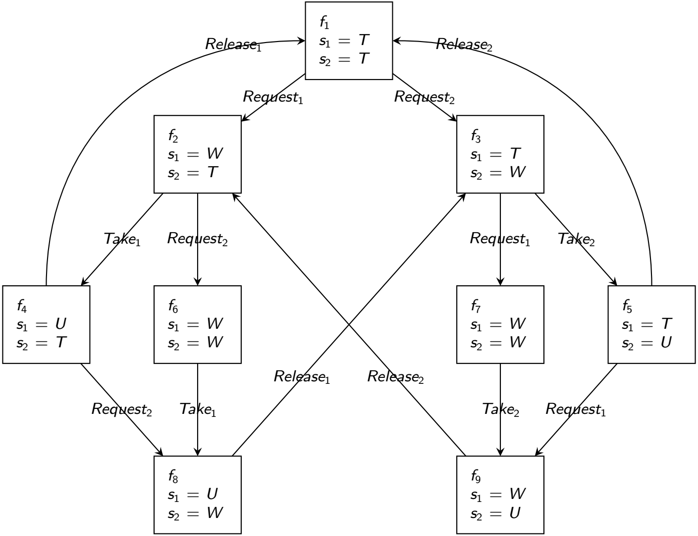
Verification of Property 1 for Example 3



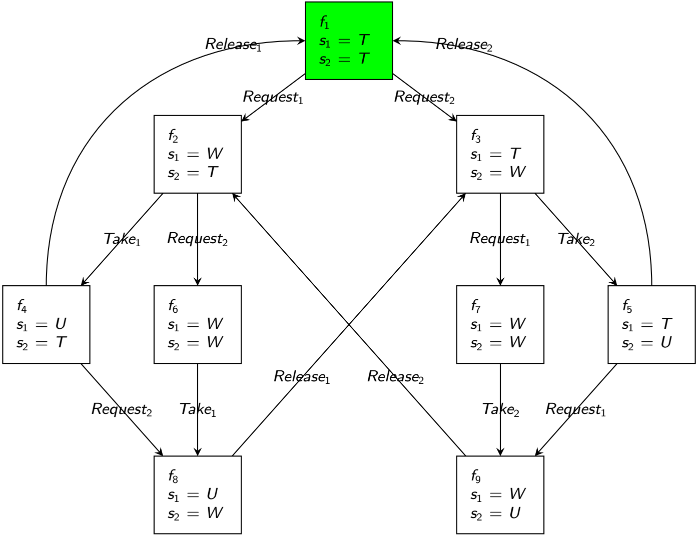
Verification of Property 1 for Example 3



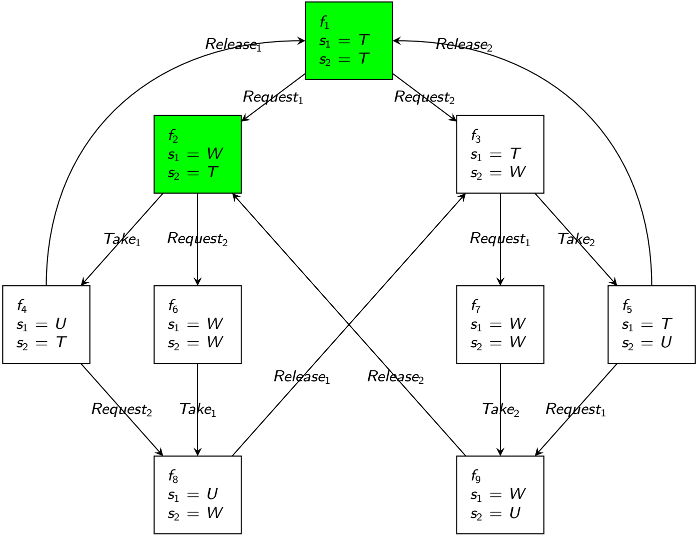
Verification of Property 2 for Example 3



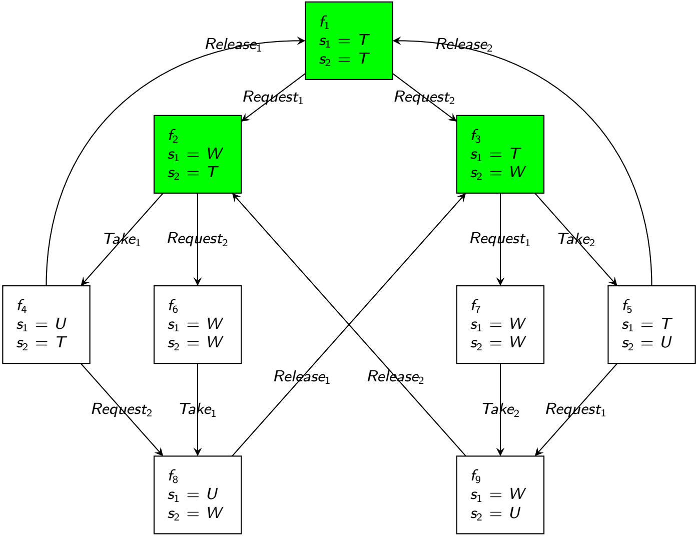
Verification of Property 2 for Example 3



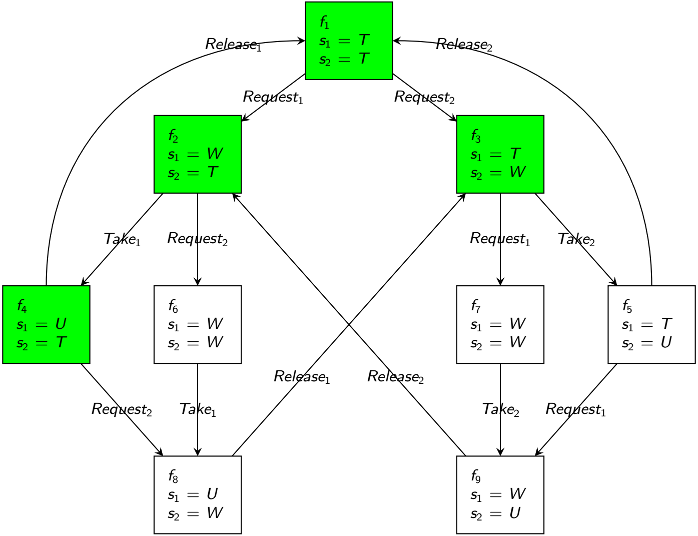
Verification of Property 2 for Example 3



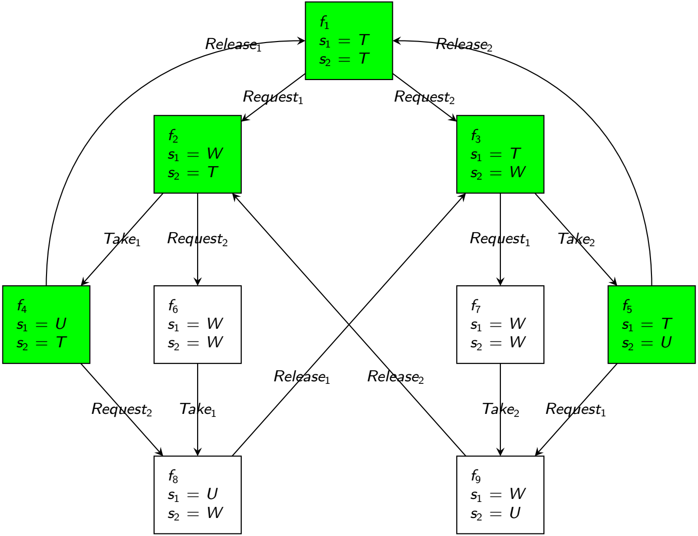
Verification of Property 2 for Example 3



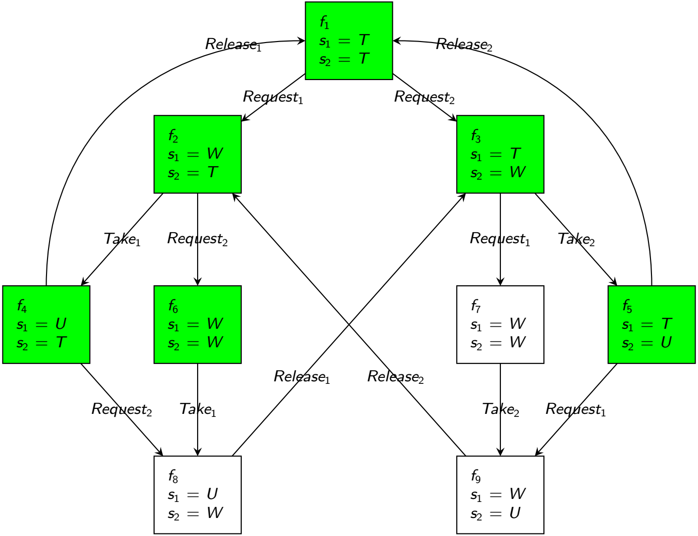
Verification of Property 2 for Example 3



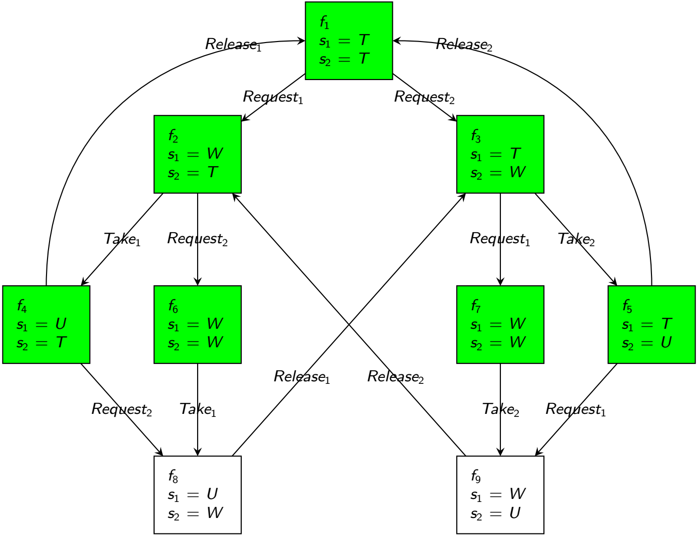
Verification of Property 2 for Example 3



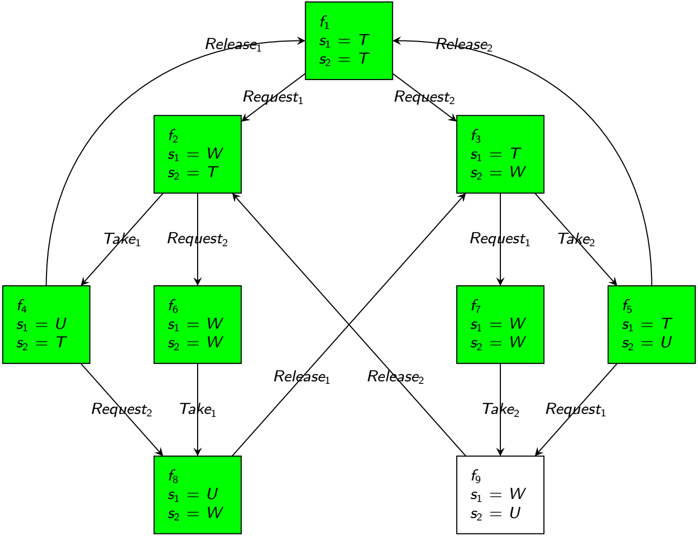
Verification of Property 2 for Example 3



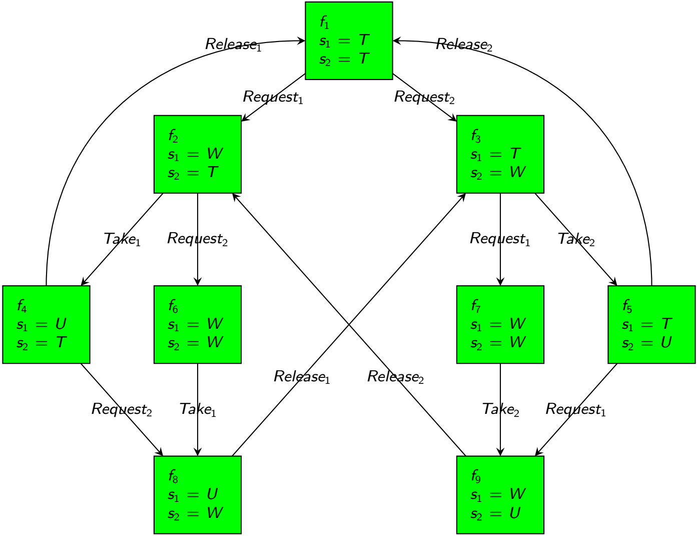
Verification of Property 2 for Example 3



Verification of Property 2 for Example 3



Verification of Property 2 for Example 3



Counterexample Construction

We extend the verification rules to produce counterexample construction rules $\mathcal{C}[\![e]\!]\ \varphi\ \phi\ \rho\ \pi$

- e is an expression in distilled form
- φ is the temporal formula to be verified
- ϕ is a function variable environment
- ρ is the set of previously encountered function calls
- π is the current program trace (a list of observable states)

The counterexample construction rules generate a verdict which consists of a program trace along with a truth value and belongs to the following datatype:

$$\mathit{Verdict} ::= \mathit{TruthVal} \times \mathit{List}\ \mathit{State}$$

The trace will give a counterexample if the associated truth value is *False*, and a witness if the corresponding truth value is *True*.

Counterexample Construction

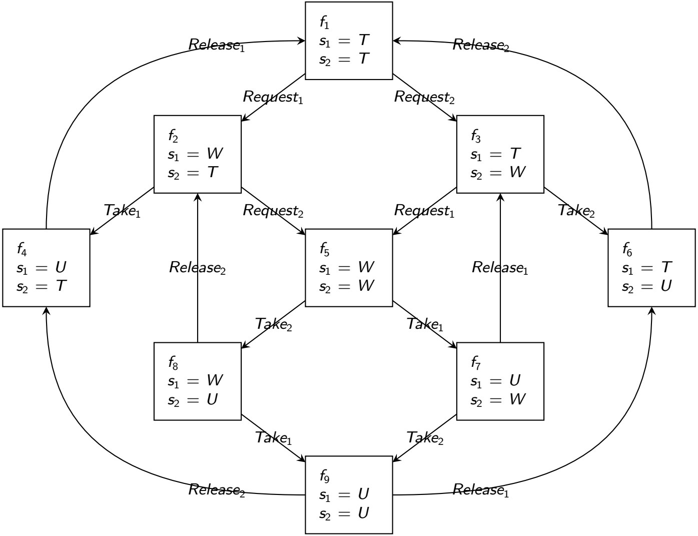
Main aspects of counterexample construction rules:

- If there is more than one counterexample or witness, the shortest one is always returned.
- As each observable state in the program is processed, it is appended to the end of the current program trace, and the final truth value is returned along with the value of this trace.
- Counterexamples and witnesses can be infinite, but the returned trace is finite; loops in the returned trace can be seen as the repetition of observable states.

Theorem (Validity)

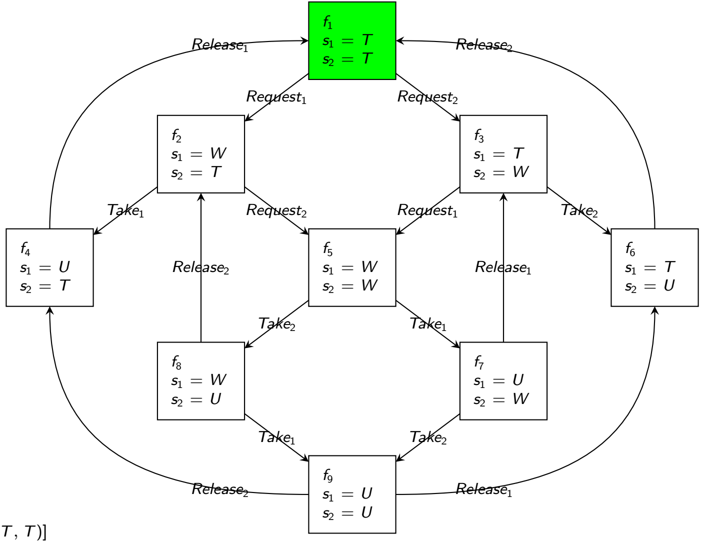
$\forall e \in \mathit{Prog}, \varphi \in \mathrm{WFF}$:
$(\mathcal{C}[\![e]\!]\ \varphi\ \emptyset\ \emptyset\ [] = (\mathit{True}, \pi) \Rightarrow \pi, 0 \models \varphi) \wedge$
$(\mathcal{C}[\![e]\!]\ \varphi\ \emptyset\ \emptyset\ [] = (\mathit{False}, \pi) \Rightarrow \pi, 0 \not\models \varphi)$

Counterexample Construction of Property 1 for Example 1



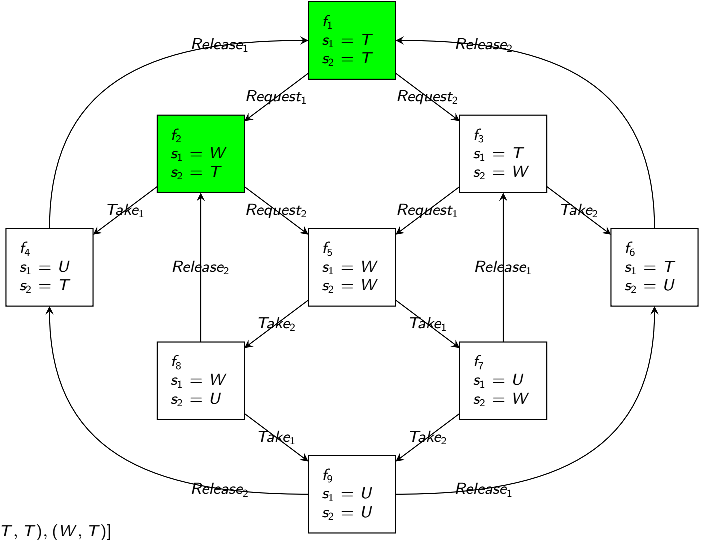
Trace: []

Counterexample Construction of Property 1 for Example 1



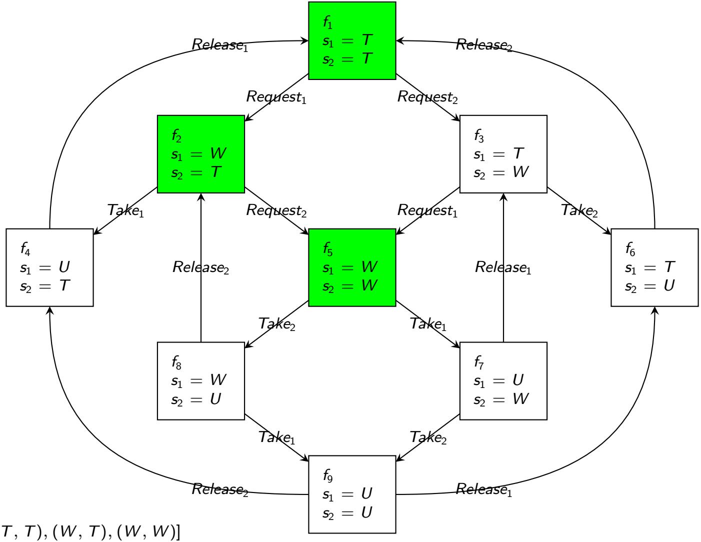
Trace: $[(T, T)]$

Counterexample Construction of Property 1 for Example 1



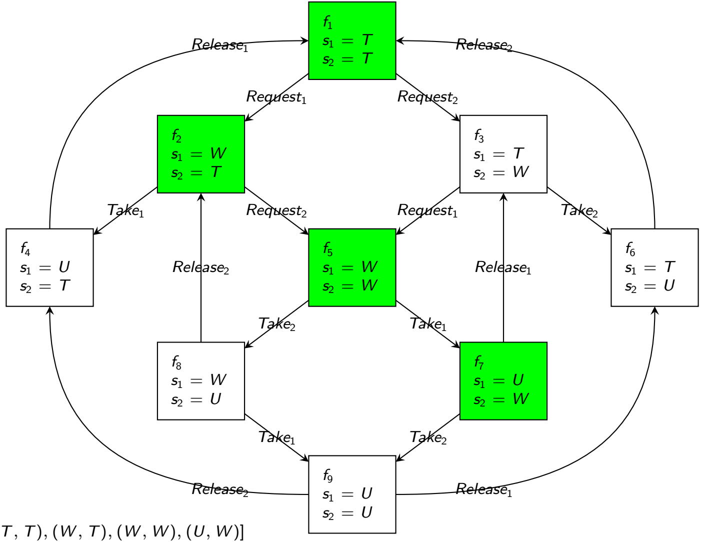
Trace: $[(T, T), (W, T)]$

Counterexample Construction of Property 1 for Example 1



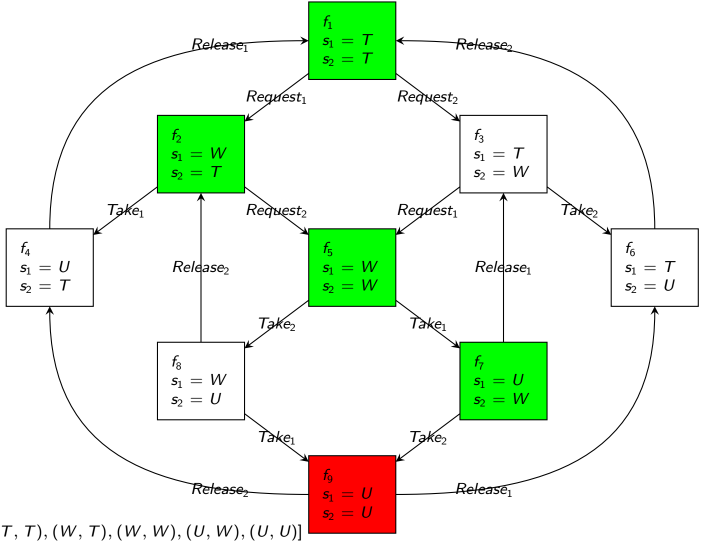
Trace: $[(T, T), (W, T), (W, W)]$

Counterexample Construction of Property 1 for Example 1



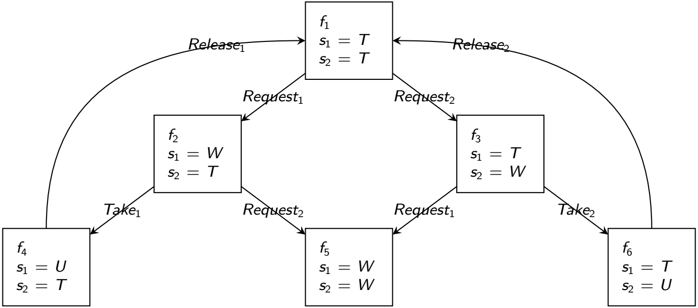
Trace: $[(T, T), (W, T), (W, W), (U, W)]$

Counterexample Construction of Property 1 for Example 1



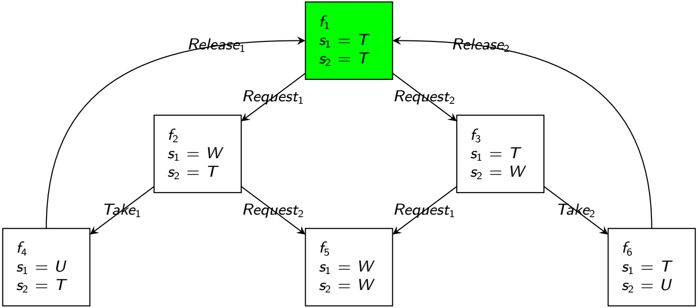
Trace: $[(T, T), (W, T), (W, W), (U, W), (U, U)]$

Counterexample Construction of Property 2 for Example 2



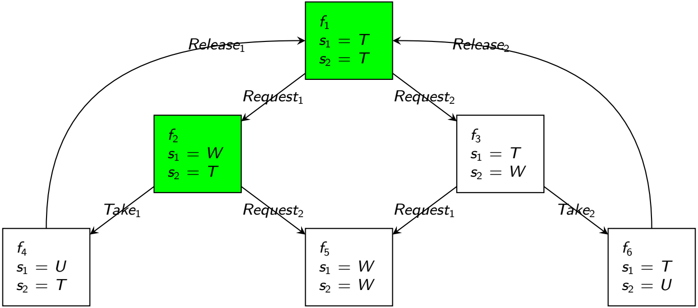
Trace: []

Counterexample Construction of Property 2 for Example 2



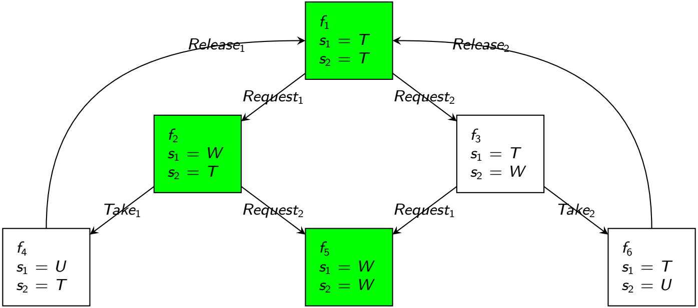
Trace: $[(T, T)]$

Counterexample Construction of Property 2 for Example 2



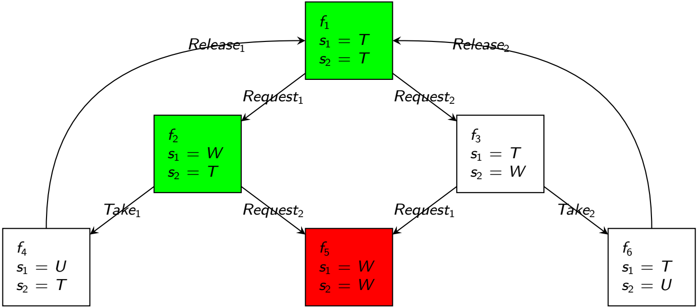
Trace: $[(T, T), (W, T)]$

Counterexample Construction of Property 2 for Example 2



Trace: $[(T, T), (W, T), (W, W)]$

Counterexample Construction of Property 2 for Example 2



Trace: $[(T,T),(W,T),(W,W),(W,W)]$

Conclusions

- We have previously shown how distillation can be used to verify both safety and liveness properties of reactive systems.
- Our technique gives a finite state approximation of the original system in which all intermediate data is given an undefined value.
- Standard finite state model checking techniques can then be applied.
- Counterexamples and witnesses were not constructed in previous work; this shortcoming has been addressed here.

Related Work

- Verification of temporal properties using logic programs:
  - Leuschel & Massart, 1999
  - Roychoudhuri et al., 2000
  - Fioravanti et al., 2001
  - Pettorossi et al., 2009
  - Seki, 2011
- Verification of temporal properties using functional programs:
  - Supercompilation: Lisitsa & Nemytykh, 2007 & 2008
  - Higher Order Recursion Schemes (HORS): Kobayashi, 2009; Lester et al., 2010
- None of this work constructs counterexamples when the temporal property does not hold.