Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is an author's version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/111088

Please be advised that this information was generated on 2019-09-21 and may be subject to change.

Proving the Temporal Properties of the Unique World *

Zoltán Horváth[1], Peter Achten[2], Tamás Kozsik[1], and Rinus Plasmeijer[2]

[1] Department of General Computer Science
University Eötvös Loránd,Budapest, Hungary
hz@ludens.elte.hu, tamas.kozsik@elte.hu
[2] Faculty of Mathematics and Computer Science
University Nijmegen, The Netherlands
peter88@cs.kun.nl, rinus@cs.kun.nl

Abstract. The behavior of concurrent and parallel programs can be specified in a functional style. We introduced a relational model for synthesizing abstract parallel imperative programs earlier. In this paper we investigate the applicability of the specification and verification tools of the model for proving temporal properties of concrete programs written in a pure functional language, in Concurrent Clean. Destructive updates preserving referential transparency are possible by using so called unique types. Clean programs perform I/O by accessing their unique environment. We present a methodology for proving safety and liveness properties of concurrent, interleaved Clean Object I/O processes and show examples for verification of simple Clean programs.

1 Introduction

The behavior of concurrent and parallel programs can be specified in a functional style. We introduced a relational model for synthesizing abstract parallel imperative programs earlier [7, 6]. We use the methodology and the abstract, programming language independent specifications presented in [3, 7, 8].

In this paper we investigate the applicability of the specification and verification tools of the model for proving temporal properties of reactive programs written in a pure functional language, in Concurrent Clean [12].

Verification of reactive ERLANG programs are investigated in [5, 4]. We use our UNITY like [3], temporal logic based reasoning instead of the first-order fixed-point calculus applied there. Automatic verification of Clean programs may be possible also by integrating temporal reasoning with CPS [11].

2 Abstract specifications of reactive systems

Reactive systems may be formulated in Clean as part of a unique environment [13, 1]. To specify the behavior of a reactive system we need temporal logic based notation [3, 6, 8] referring to program state and local state [1].

* Supported by the Hungarian State Eötvös Fellowship and by the Hungarian Ministry of Education, Grant Nr. FKFP 0206/97

A specification of a reactive system may be given as a set of properties of its unique environment (for full and detailed description of the model see [6]). Every property is a relation over the powerset of the state space[1]. Let $P, Q, R, U : A \longmapsto \mathcal{L}$ be logical functions. We define $\triangleright, \mapsto, \hookrightarrow \in \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A))$, and $\mathrm{FP}, \mathrm{INIT}, \mathrm{inv}, \mathrm{TERM} \subseteq \mathcal{P}(A)$.

A program satisfies the safety property $P \triangleright Q$, if and only if there is no direct state-transition from $P \wedge \neg Q$ to $\neg P \wedge \neg Q$ only through Q if any. A program satisfies the progress properties $P \mapsto Q$ or $P \hookrightarrow Q$ if the program starting from P inevitably reaches a state, in which Q holds. $P \mapsto Q$ defines further restriction for the direction of progress. The fixed point property $\mathrm{FP} \Rightarrow R$ defines necessary conditions for the case when the program is in one of its fixed point. The $Q \in \mathrm{INIT}$ property defines sufficient condition for the initial states of the program. $Q \hookrightarrow \mathrm{FP}$ expresses that the program starting from Q inevitably reaches one of its fixed points. P is said to be *stable* if and only if $P \triangleright \downarrow$, where $\downarrow$ denotes the constant function $False$. If P holds initially and P is stable, then P is an *invariant*, denoted by $\mathrm{inv}P$.

3 Clean constructs and libraries

The scope of the present paper is restricted to interactive programs written in standard Clean 1.3, using the Object I/O library 1.0.2 [1, 2]. Experimental Concurrent Clean systems supporting true concurrency and parallellism [9, 13] are not investigated. In [9] concurrent and parallel evaluation of Clean expressions are controlled by program *annotations*. Message passing needed for the communication of arguments and results is implicit. An explicit version is proposed in [13]. Here, functions are provided to create threads and channels, and also for message passing and receiving.

Clean is a strongly typed language based on term graph rewriting. Destructive updates preserving referential transparency are possible by using so called uniqueness types [12, 14]. I/O in Clean uses the *world as value* paradigm. In this paradigm, external resources such as the file system and event stream are passed explicitly as a value to functions. These values are also called *environments*. The external world is of type `World`. I/O programs are functions of type `:: *World -> *World`. The `*` in front of `World` means that the `World` argument is unique: when the function is evaluated it is guaranteed that this function has the only reference to the world. In addition, it must yield a world with one reference.

Uniqueness is a function property. The type system derives the uniqueness properties of all functions. Many functions do not change the uniqueness of their arguments. The simplest example is the identity function `id` defined as: `id a = a`. Its derived type is: `id :: .x -> .x`. This type indicates that `id` can both be applied to a unique value and a non-unique (or *shared*) value.

[1] Logical functions are characterised by their truth set, so truth sets are used in the notations at place of logical functions and vice versa if it is not confusing.

Using uniqueness information the Clean compiler is able to generate better code by letting functions reuse unique argument components instead of rebuilding them. An example is the use of unique arrays.

Clean programs using the Object I/O library create their own unique state space and define initialisation and state-transition functions. The library supports *interactive processes*, which can be created and closed dynamically. Each interactive process may consist of an arbitrary number of *interactive objects*. These are: windows, dialogues, menus, timers, and receivers. Again, these objects can be created and closed dynamically. An interactive process is a *state transition system*. Its *state* is a value of type `PSt l p`, defined as:
`:: PSt l p = {ls::l, ps::p, io::*IOSt l p}`.
The `ls` and `ps` components constitute the 'logical' state of the interactive process which must be defined by the programmer. The `io` component is managed entirely by the Object I/O system. It contains, amongst others, the current state of all interactive objects of the interactive process. So the IO state represents the external environment of the interactive process and it is therefore uniquely attributed.

Interactive objects are created by passing an abstract description to the proper creation function. Such an abstract description is usually an algebraic data type value. One can find examples in the Clean programs in Section 5.2, line 10-11 (a *timer*), and Section 6.2, line 18-22 (a *dialogue*). The important point of such abstract descriptions is that they contain the *state transitions* of the interactive process. These state transitions are higher order function arguments of the algebraic data types. They are usually of type: `(PSt .l .p) -> (PSt .l .p)`. Event handlers of I/O processes on the same processor are interleaved. Atomic actions correspond to handling of one event.[2]

The Object I/O system keeps evaluating all interactive processes until each of them has *terminated*. An interactive process terminates by applying the library function `closeProcess` to its process state. This function will close all current interactive objects from the IO state component and turn it into the *empty* IO state, which is its final state.

4 A calculus of verification

We use a two phase model. In the first phase the abstract model of the Clean program is constructed. We give a formal specification and model the behaviour of the concrete program by an abstract program. The program text is analyzed and the state transition functions are extracted. The relatively well defined structure of the Object I/O processes makes this possible. A tool, which assists this first phase is needed in the future to ensure the correctness of the abstract model in respect of the main semantical properties of the concrete Clean program.

[2] The current implementation supports only processes on a single processor. However, using the Object I/O TCP/IP library (developed by Martin Wierich) allows one to create distributed communicating programs.

We prove the correctness of the abstract program in respect of the specification in the second phase. This phase has a well-developed mathematical basis [3, 6]. In the following we give a short description of the main concepts.

A problem is defined as a set of specification relations, more precisely a problem F is a relation over the parameter space B and ordered tuples of specification relations. The appropriate choice of the parameter space reduces the size of the problem and the number of verification steps. Every specification relation is defined over the powerset of the state space (section 2).

The abstract program is regarded as a relation generated by a set of nondeterministic (simultaneous) conditional assignments [7] similar to the concept of abstract program in UNITY [3] and to the concept of parallel program given by van Lamsweerde and Sintzoff [10]. By virtue of it's definition the effect relation of a conditional assignment is total, i.e., it's domain is equal with the whole state space. This means that a conditional assignment always terminates [6]. Some assignments are selected nondeterministically and executed in each step of the execution of the abstract program. Every statement is executed infinitely often, i.e., an unconditionally fair scheduling is postulated. If more than one processor selects statements for execution, then the executions of different processors are fairly interleaved. A fixed point is said to be reached in a state, if none of the statements changes that state [3]. We denote the conditional assignment $s_j \in S$ the following way: $(\underset{i\in[1,n]}{\|}(v_i :\in F_{j_i}(v_1,..,v_n), \text{ if } \pi_{j_i}))$.

The program properties with respect of an abstract parallel program are characterized as relations over the powerset of the state space. They are defined in terms of the weakest precondition (wp) of the element statements of the abstract program. We use the dual concept of strongest postcondition (sp) too.

We generalize the concept of weakest precondition for abstract parallel programs [7] $wp(S,R) ::= \forall s \in S : wp(s,R)$. Let us denote by $\text{inv}_S(Q)$ the set of logical functions of which truth are preserved by the elements of S if the program is started from a state satisfying Q. That is: $\text{inv}_S(Q) \subseteq \mathcal{P}(A)$.
$\text{inv}_S(Q) ::= \{\lceil P \rceil \mid sp(s_0, Q) \Rightarrow P \text{ and } P \Rightarrow wp(S,P)\}$.
Let us denote by $\triangleright_S$ the set of ordered pairs (P,Q) of logical functions for which holds that P is stable while $\neg Q$ during the execution of S. $\triangleright_S \subseteq \mathcal{P}(A) \times \mathcal{P}(A)$.
$\triangleright_S ::= \{(\lceil P \rceil, \lceil Q \rceil) \mid (P \wedge \neg Q \Rightarrow wp(S, (P \vee Q)))\}$.

Let us denote by $\mapsto_S$ the set of ordered pairs (P,Q) of logical functions for which holds that P is stable while $\neg Q$ during the execution of S and there is a conditional assignment s_j which ensures the transition from P to Q.Let be $\hookrightarrow_S \subseteq \mathcal{P}(A) \times \mathcal{P}(A)$ the transitive disjunctive closure (denoted by ^{tdl}) of $\mapsto_S$. A fixed point is said to be reached in a state of the state space A, if none of the statements changes the state. φ_S characterises the set of fixed points, $\varphi_S ::= (\underset{j\in J, i\in[1..n]}{\wedge}(\neg\pi_{j_i} \vee (\pi_{j_{id}} \wedge v_i = F_{j_i}(v_1,\ldots,v_n))))$, where $\pi_{j_{id}}$ denotes the logical function, which characterize the set of states over which the relation F_{j_i} is deterministic. Let us denote by TERM_S the set $\{\lceil Q \rceil | (Q, \varphi_S) \in \hookrightarrow_S\}$.

The abstract parallel program $S \subseteq A \times A^{***}$ is a solution of the problem F, if $\forall b \in B : \exists h \in F(b)$, such that the program S satisfies all the specification properties given in the $\text{inv}_h, \triangleright_h, \mapsto_h, \hookrightarrow_h, \text{FP}_h, \text{TERM}_h$ components of the element of

the parameter space h assuming that the program starts from a state satisfying all the elements of INIT_h. The program S satisfies a specification property, if and only if there exists an invariant property K such that the program satisfies the specification property with respect to K. This means that a program is said to satisfy a specification property, even if the program fails to satisfy it over a subset of the unreachable states [3, 7].

The presented model has a temporal logic background but the verification steps are based on simple weakest precondition calculations.

5 Sorting a list stored in the unique process state

In this section we show our first simple example, in which we sort a list stored in the local process state component of an Object I/O process by bubble sort. We prove an invariant, a fixed point property and termination (progress).

5.1 Formal specification of the problem

The state space of the program is the PSt process state of the Object I/O process. $A = pst : PSt$. The process state consists of three components, the local state, the public state and the IO state. The list of elements to be sorted is stored in the local state. Type V is list of integers.
$PSt = (ls : V, ps : NoState, io : IOSt)$, $V = [Int]$.

We specify that the program inevitably reaches a fixed point and the list stored in the local process state is a sorted permutation of the initial *unsorted* list.

$$\uparrow\hookrightarrow \text{FP} \tag{1}$$

$$\text{FP} \Rightarrow (pst.ls \in perms(unsortedlist) \wedge sorted(pst.ls)) \tag{2}$$

The specification property (2) is allowed to be refined (substituted) by an invariant and a weaker fixed point property [6]. It is easy to show that from (3) and (4) follows (2).

$$\text{inv}(pst.ls \in perms(unsortedlist)) \tag{3}$$

$$\text{FP} \Rightarrow sorted(pst.ls) \tag{4}$$

5.2 Clean program

The Clean program below starts an Object I/O process and initialises the three components of the process state by calling the `startNDI` function. Local state is set to `unsortedlist`, public state is set to `NoState`, the IO state is initialised to be empty and subsequently modified by `initialise` to include a `Timer`. The `TimerFunction` is set to `bubble`, i.e. the timer handled by the I/O process calls function `bubble` repeatedly again and again after `0` seconds delays.

```
module bubblesort
import StdEnv, StdIO, StdDebug
:: NoState = NoState
Start :: *World -> *World
Start world
  = startNDI unsortedlist NoState initialise [] world
where
  unsortedlist = [100,99..0]
  initialise :: (PSt [a] .p) -> PSt [a] .p | Ord, toString a
  initialise pst = snd (openTimer undef (Timer 0 NilLS
                        [TimerFunction (noLS1 bubble)]) pst)
  bubble :: NrOfIntervals (PSt [a] .p) ->
                            PSt [a] .p | Ord, toString a
  bubble _ pst=:{ls=list}
     | not sorted = trace_n (show list) (closeProcess pst)
     | otherwise  = {pst & ls=updateAt i (list!!j)
                         (updateAt j (list!!i) list)}
  where (sorted,i,j) = sweep_is_sorted 0 list

// sweep_is_sorted :: Int [a] -> (Bool,Int,Int) | Ord a
// sweep_is_sorted answers the question whether the list is
// sorted and finds the indices (i,j) of two elements in the list
// l such that i<j and (l!!i)>(l!!j), if the list is not sorted.
```

5.3 An abstract model of the program

The s_0 initialisation assignment is a composition of the initialisation of the components and the subsequent set of the IO state to include the timer. The program has an iterative structure, the state transforming step $pst := bubble(pst)$ is repeated until the fixed point is reached. A possible abstract model of the concrete Clean program is given below.

$$\begin{aligned} s_0 &: pst := InitProc(Timer(bubble), (unsortedlist, NoState, empty)) \\ S &: \{\, pst := bubble(pst)\} \end{aligned}$$

where

$$bubble(pst) = \begin{cases} (pst.ls, pst.ps, closed) \\ \qquad \text{if } sorted(pst.ls) \wedge (pst.io \neq closed) \\ (newlist(pst.ls), pst.ps, pst.io) \\ \qquad \text{if } \neg sorted(pst.ls) \wedge (pst.io \neq closed) \end{cases}$$

and $newlist(list) = [list(0), \ldots list(i-1), list(j), list(i+1), \ldots list(j-1),$
$list(i), list(j+1), \ldots list(list.length-1)],$
where $(i, j) = sweep'(list)$, the first i and j, for which $list(i) > list(j)$

5.4 Formal proof

We prove invariant (3) first. Let us denote $(pst.ls \in perms(unsortedlist))$ by P. $sp(pst := InitProc(Timer(bubble), (unsortedlist, NoState, empty), True) = (pst.ls = unsortedlist \wedge pst.ps = NoState \wedge pst.io = [Timer(bubble)]) \Rightarrow (pst.ls \in perms(unsortedlist)) = P$.
$wp(pst := bubble(pst), P) = (sorted(pst.ls) \wedge (pst.io \neq closed) \rightarrow pst.ls \in perms(unsortedlist)) \wedge (\neg sorted(pst.ls) \wedge (pst.io \neq closed) \rightarrow newslist(pst.ls) \in perms(unsortedlist))$.
Since $newlist(pst.ls) \in perms(pst.ls)$ by definition of $newlist$ if $\neg sorted(pst.ls)$ and $pst.ls \in perms(unsortedlist)$, so both the first implication and the second implication follows from P.

Let us denote $(\neg sorted(pst.ls) \rightarrow pst.io \neq closed)$ by K. Using the same kind of calculation it is easy to show that K is an invariant of the program.

If we want to prove (4) we have to calculate the fixed point of the program. $\varphi_S = (\neg sorted(pst.ls) \wedge (pst.io \neq closed)) \rightarrow pst.ls = newlist(pst.ls) \wedge pst.ps = pst.ps \wedge pst.io = pst.io) \wedge (sorted(pst.ls) \wedge (pst.io \neq closed) \rightarrow pst.ls = pst.ls \wedge pst.ps = pst.ps \wedge pst.io = closed)$. If the condition of the first implication holds, then the consequence is false by definition of $newlist$. We get $\varphi_S = ((sorted(pst.ls) \vee pst.io = closed) \wedge (\neg sorted(pst.ls) \vee (pst.io = closed))$, so $\varphi_S = (pst.io = closed)$ and $\varphi_S \wedge K \Rightarrow sorted(pst.ls)$. We proved (4).

We introduce the variant function $v = inversion(pst.ls) - \chi(pst.io = closed) + 2$, where $\chi(true) = 1$ and $\chi(false) = 0$. We apply the theorem of variant function [6] to prove (1). Since $\forall l : inversion(l) \geq 0$ and $\forall b : \chi(b) \leq 1$, $v \geq 0$. It is easy to see that for invariant K: $K \Rightarrow v > 0$, i.e. the variant function is positive in any reachable state. We have to show that the variant function inevitably decreases until the program reaches its fixed point, i.e. $\neg\varphi_S \wedge v = t' \wedge K \hookrightarrow_S (\varphi_S \vee v \leq t' - 1) \wedge K$. Using the fact $\mapsto_S \subseteq \hookrightarrow_S$ it is sufficient to show: $\neg\varphi_S \wedge v = t' \wedge K \mapsto_S (\varphi_S \vee v \leq t' - 1) \wedge K$. $wp(pst := bubble(pst), (\varphi_S \vee v \leq t' - 1) \wedge K) = (\neg sorted(pst.ls) \wedge (pst.io \neq closed) \rightarrow (sorted(newlist(pst.ls)) \vee (inversion(newlist(pst.ls) - \chi(pst.io = closed) + 2 \leq t' - 1 \wedge K)) \wedge (sorted(pst.ls) \wedge (pst.io \neq closed) \rightarrow ((sorted(pst.ls) \vee (inversion(pst.ls) - \chi(closed = closed) + 2) \leq t' - 1) \wedge K))$. It is easy to show, that from $(\neg\varphi_S \wedge v = t' \wedge K) = (pst.io \neq closed) \wedge inversion(pst.ls) - \chi(pst.io = closed) + 2 = t') \wedge K$ follows the calculated weakest precondition, hence we proved the $\mapsto_S$ relation.

6 Properties of communicating I/O processes

In this section we prove the correctness of a program which consists of two simple communicating Object I/O processes. We present two processes, which send messages to each other, when the user presses a button. The processes are running in an interleaved way and are interactive. The overall system includes the user as a third component, who interacts with the processes by pressing buttons.

6.1 Formal specification of the problem

The overall state space of the program is the smallest common superspace of the subspaces of the three components. Process A is running over the subspace $pst_A : PSt_A \times ch_{ab} : Ch_1 \times ch_{ba} : Ch_2 \times button_A : Button_A$, where $PSt_A = (ls : Int, ps : NoState, io : IOSt), Ch_1 = Ch_2 = Channel(Int), Button_A = \{pressed, released\}$. Similarly process B is running over the subspace $pst_B : PSt_B \times ch_{ab} : Ch_1 \times ch_{ba} : Ch_2 \times button_B : Button_B$, where $Button_B = \{pressed, released\}$. The user is acting over the subspace $button_A : Button_A \times button_B : Button_B$. The overall state space of the system is:

$$A = pst_A : PSt_A \times ch_{ab} : Ch_1 \times ch_{ba} : Ch_2 \times button_A : Button_A \times \\ \times button_B : Button_B \times pst_B : PSt_B.$$

We use the concept of history of channels in our specification and verification. The history of a channel is the list of elements communicated on it ever. The history of channel c is denoted by $\overline{c}$.

We specify that the system reaches one of its fixed points (6) after sending mc messages (7), which messages are initiated by pressing the buttons (8),(9). We assume that channels are initially empty and buttons are released (5).

The parameter space is $B = mc : Int$.

$$ch_{ba} = ch_{ab} = \overline{ch_{ba}} = \overline{ch_{ab}} =<> \wedge button_A = button_B = released \in \mathrm{INIT}_{mc} \quad (5)$$

$$\uparrow\!\hookrightarrow_{mc} \mathrm{FP} \quad (6)$$

$$\mathrm{FP}_{mc} \Rightarrow |\overline{ch_{ba}}| + |\overline{ch_{ab}}| = mc \quad (7)$$

$$button_A = pressed \wedge |\overline{ch_{ab}}| = t' \triangleright_{mc} button_A = released \wedge |\overline{ch_{ab}}| = t' + 1 \quad (8)$$

$$button_B = pressed \wedge |\overline{ch_{ba}}| = t' \triangleright_{mc} button_B = released \wedge |\overline{ch_{ba}}| = t' + 1 \quad (9)$$

6.2 Clean program

The two processes behave in the same way. The function `process` is the functional abstraction of the common behaviour. The first parameter of the process is its name, the second parameter is an ordered pair of two identifiers, which are used to identify the process itself and the other process.

The `NDIProcess`-es are declared to belong to a `ProcessGroup` with shared state initialised to `NoState`, i.e. no useful shared public state exists. Communication channels are implemented implicitly. The local part of the process state PSt is initialised by the `NDIProcess` to 0. The `initialise` functions initialises the IO state.

There are two state transition functions defined. Function `pressed` may cause a state transition by sending data and incrementing the counter stored in the local state when the corresponding button is pressed. Function `received` is activated to receive data and to increment the counter stored in the local state when a message is arrived to the process. If the counter reaches the value `mc`, then the IO state is closed.

```
module processes
import StdEnv, StdIO
:: NoState = NoState
Start :: *World -> *World
Start world
   # (ridA,world) = openRId world
   # (ridB,world) = openRId world
   = startProcesses [process "A" (ridA,ridB),
                     process "B" (ridB,ridA)] world
process :: String (RId i,RId i) -> ProcessGroup NDIProcess
process name (me,you)
   = ProcessGroup NoState (NDIProcess 0 initialise [])
where
   mc = 10
   initialise :: (PSt Int .p) -> PSt Int .p
   initialise pst = snd (openDialog undef dialog pst)
   where
      dialog = Dialog name
                 ( ButtonControl "Press me"
                 [ControlFunction (noLS pressed)]
                 :+: Receiver me (noLS1 received) []
                 ) []
      pressed :: (PSt Int .p) -> PSt Int .p
      pressed pst=:{ls=i}
         # (_,pst)   = syncSend you undef pst
         | i‘<mc     = {pst & ls=i‘}
         | otherwise = closeProcess {pst & ls=i‘}
      where  i‘= i+1
      received :: .i (PSt Int .p) -> PSt Int .p
      received _ pst=:{ls=i}
         | i‘<mc     = {pst & ls=i‘}
         | otherwise = closeProcess {pst & ls=i‘}
      where i‘       = i+1
```

Synchronous sending means an immediate call of the receiver of the partner process. The event handler processes a synchronous send and the corresponding receiving of the data in one single atomic action. This means a very strict ordering of events, which can be expressed by introducing strict explicit synchronization conditions at the abstract program level. The construction of the formal proof pointed out that a concrete program performing an asynchronous message passing instead of the synchronous one would not be correct. On the other hand an abstract program with weaker synchronization conditions was to be proved correct in respect to the posed problem, i.e. the concrete program can not be less synchronised but the abstract program corresponding to the concrete one is proved to be over-synchronized.

6.3 An abstract model of the program

We present the overall abstract program as the union of three components [3, 8]. The model behind the abstract program assumes an unconditionally fair scheduling which differs from the event driven semantics of the concrete program. Events are represented by conditions and assumed to be initiated by the user.

The s_0 initialisation assignments of the processes is a composition of the initialisation of the components and the subsequent set of the IO state.

$$
\begin{array}{ll}
s_{0A}: & pst_A := Init(Dialog(Button(pressed), Receiver(received)), \\
 & \qquad\qquad (0, NoState, empty)) \\
S_A: \{ & pst_A,\ button_A,\ ch_{ab} := pressed(pst_A),\ released,\ send(ch_{ab}, 0) \\
 & \quad \text{if } button_A = pressed \ \wedge\ |ch_{ab}| = |ch_{ba}| = 0 \ \wedge\ pst_A.io \neq closed \\
 & pst_A,\ ch_{ba} := received(pst_A),\ receive(ch_{ba}) \\
 & \quad \text{if } |ch_{ba}| \neq 0 \ \wedge\ |ch_{ab}| = 0 \ \wedge\ pst_A.io \neq closed \\
\}
\end{array}
$$

$$
\begin{array}{ll}
s_{0B}: & pst_B := Init(Dialog(Button(pressed), Receiver(received)), \\
 & \qquad\qquad (0, NoState, empty)) \\
S_B: \{ & pst_B,\ button_B,\ ch_{ba} := pressed(pst_B),\ released,\ send(ch_{ba}, 0) \\
 & \quad \text{if } button_B = pressed \ \wedge\ |ch_{ba}| = |ch_{ab}| = 0 \ \wedge\ pst_B.io \neq closed \\
 & pst_B,\ ch_{ab} := received(pst_B),\ receive(ch_{ab}) \\
 & \quad \text{if } |ch_{ab}| \neq 0 \ \wedge\ |ch_{ba}| = 0 \ \wedge\ pst_B.io \neq closed \\
\}
\end{array}
$$

$$
\begin{array}{ll}
s_{0U}: & SKIP \\
S_U: \{ & \\
 & button_A := pressed, \text{if } button_A = released \ \wedge\ |ch_{ab}| = 0 \ \wedge\ pst_A.io \neq closed \\
 & button_B := pressed, \text{if } button_B = released \ \wedge\ |ch_{ba}| = 0 \ \wedge\ pst_B.io \neq closed \\
\}
\end{array}
$$

where

$$
pressed(pst) \ = \ \begin{cases} (pst.ls + 1, pst.ps, pst.io) & \text{if } pst.ls + 1 < mc \\ (pst.ls + 1, pst.ps, closed) & \text{otherwise} \end{cases}
$$

$$
received(pst) \ = \ pressed(pst)
$$

6.4 Formal proof

Now we prove[3] that the former abstract program solves the specification properties (5-9) introduced in section 6.1. It is enough to show that the program

[3] For the sake of brevity (and also understandability) we omit the computation of weakest preconditions. We restrict ourselves to give a hint how these computations should be performed. Calculations of weakest preconditions are presented in section 5.4.

satisfies those properties for its reachable states. Thus we determine a couple of invariants first.

$$|\overline{ch_{ab}}| \ + \ |\overline{ch_{ba}}| \ - \ |ch_{ba}| \ = \ pst_A.ls \quad \in \ inv_S \tag{10}$$

$$|\overline{ch_{ba}}| \ + \ |\overline{ch_{ab}}| \ - \ |ch_{ab}| \ = \ pst_B.ls \quad \in \ inv_S \tag{11}$$

$$pst_A.ls \leq mc \ \wedge \ (pst_A.ls = mc \ \leftrightarrow \ pst_A.io = closed) \quad \in \ inv_S \tag{12}$$

$$pst_B.ls \leq mc \ \wedge \ (pst_B.ls = mc \ \leftrightarrow \ pst_B.io = closed) \quad \in \ inv_S \tag{13}$$

$$|ch_{ab}| * |ch_{ba}| \ = \ 0 \quad \in \ inv_S \tag{14}$$

$$pst_A.ls + |ch_{ba}| \ = \ pst_B.ls + |ch_{ab}| \quad \in \ inv_S \tag{15}$$

Initially all of the above predicates hold, since the channels and their histories are empty, $pst_A.ls = pst_B.ls = 0 < mc$ and neither $pst_A.io$ nor $pst_B.io$ is closed. For proving the stability of the predicates — according to a well-known hint — it is sufficient to check those assignments that can change the value of one of the variables occurring in a predicate. Let's start with (14). It formulates that one of the channels should always be empty. Only the assignments containing the *send* function can make an empty channel non-empty. For example the first statement in S_A can extend channel ch_{ab}, but only when ch_{ba} is empty. Since this assignment doesn't change the value of ch_{ba}, there will be an empty channel after executing it, namely ch_{ba}. The sender assignment in S_B can be checked similarly.

Stability of the predicates in (12) and (13) is based on the definition of *pressed* and *received*. Only these two functions change the ls and io components of pst_A and pst_B. An ls is increased only by one, and it becomes mc at the very same time as the io becomes closed. After the io is closed, neither the send nor the receive operation can be called, thus ls can not grow above mc.

When calculating (10) we can observe that the change (growth) of $pst_A.ls$ is always accompanied by either a send or a receive operation on ch_{ab} and ch_{ba}, respectively, which either increase $|\overline{ch_{ab}}|$ or decrease $|ch_{ba}|$. The values in the left-hand side of the equation can also be changed by a send on ch_{ba}, but such an operation will increase both $|\overline{ch_{ba}}|$ and $|ch_{ba}|$, their difference will remain the same. (11) can be done similarly, and (15) is also based on the same phenomenon.

Now that the proof of the invariants is completed, we can compute the set of fixed points of the abstract program. All the assignments are such that the left-hand sides cannot be equal with the right-hand sides (either because channels are changed when a send or receive operation is performed on them, or because — in the case of S_U — the conditions do not allow it). Thus the set of fixed points is a conjunction of the negated conditions of the statements: the program doesn't change its state when none of the conditions is true. After some steps of simplification the φ_S set of fixed points is the following:

$$\varphi_S = (pst_A.io = closed \vee |ch_{ab}| \neq 0) \ \wedge \ (pst_B.io = closed \vee |ch_{ba}| \neq 0) \tag{16}$$

One of the channels must be empty (from (14)), so one of the IO states must be closed. Let them be for example ch_{ab} and $pst_A.io$. This means that $pst_A.ls = mc$

(from 12), thus (15) and the second part of (11) guarantees that $pst_B.ls = mc$ and $|ch_{ba}| = 0$, too. Using invariant (10) we can conclude proving the fixed point specification property (7).

The truth of property (8) is affected only by two assignments: the first from S_A and the first from S_U. The others cannot change the value of either $button_A$ or $\overline{ch_{ab}}$. Furthermore, the condition of the assignment from S_U guarantees that this assignment will never change the state of the program if the left-hand side of the $\triangleright$ property holds. Finally, the sender statement of S_A should be investigated. A simple weakest precondition calculation reveals, that when the condition of that assignment holds, thus the variables can change at all, both $button_A$ and ch_{ab} will take on values that satisfy the right-hand side of the $\triangleright$ relation. The proof is analogous for (9).

The correctness of the abstract program in respect of (6) can be proved by applying the theorem of variant function. We choose the variant function $v = 2{*}mc{+}1{-}pst_A.ls{-}pst_B.ls$. (The invariants guarantee that its value is a positive integer in every reachable state.) The calculation is similar to the calculations presented in section 5.4 but is more complicated from technical point of view. We can show that the variant function will inevitably fall off if the program is not in one of its fixed points. Let's suppose e.g. that $pst_A.io \neq closed \ \wedge \ |ch_{ab}| \neq 0$. It is possible to prove that the following progress properties hold:

$$pst_A.io \neq closed \ \wedge \ |ch_{ab}| \neq 0 \ \wedge \ |ch_{ba}| = 0 \ \wedge \ v = k \ \mapsto_{mc} \ v < k$$

$$pst_A.io \neq closed \wedge |ch_{ab}| \neq 0 \wedge |ch_{ba}| \neq 0 \wedge button_A = pressed \wedge v = k \mapsto_{mc} v < k$$

$$pst_A.io \neq closed \wedge |ch_{ab}| \neq 0 \wedge |ch_{ba}| \neq 0 \wedge button_A = released \wedge v = k \ \mapsto_{mc}$$
$$(pst_A.io \neq closed \ \wedge \ |ch_{ab}| \neq 0 \ \wedge \ |ch_{ba}| \neq 0 \ \wedge \ button_A = pressed \ \wedge \ v = k) \vee v < k$$

Based on the properties of the "leads-to" relation and using the theorems of refinements of specifications [6] one can easily derive that:
$pst_A.io \neq closed \ \wedge \ |ch_{ab}| \neq 0 \ \wedge \ v = k \ \hookrightarrow_{mc} \ v < k.$

7 Conclusions

We presented a two phase methodology for proving the correctness of Clean Object I/O processes with a static set of state transition functions. The methodology can be applied for dynamic set of state transition functions, if the condition of adding new and removing old state transition functions is well defined. In that case all the state transition functions should be included into the abstract model at the beginning extended with the appropriate conditions, which conditions allow change of state for the appropriate period only. The model can be used as well to prove the temporal properties of more general Clean programs using uniqueness types in principle[4] if unique objects are regarded as static entities at a different semantical view and abstraction level. Construction of an abstract model having the same semantics as the concrete program could be very difficult in practice.

[4] If dynamics are used then open specifications [3, 4] should be applied.

The methodology should be assisted by a two phase tool (not necessarily by a fully automatic one) in the future, which helps to formulate an abstract model of the state transitions defined in the Clean program text and ensures the correctness of the abstract program in respect of the main semantical properties of the concrete program. Also a proof tool, for example an appropriate extension of CPS which is working with propositional logic assertions would be necessary to support the verification of the abstract program in respect of the specification.

References

[1] Achten P., Plasmeijer R.: Interactive Objects in Clean. In: K. Hammond et al eds., *Proc. of Implementation of Functional Languages, 9th International Workshop, IFL'97*. St. Andrews. Scotland, UK, September 1997, LNCS 1467, pp. 304-321.

[2] Achten P.: A Tutorial to the Clean Object I/O Library - version 1.0.2., University of Nijmegen, 1999.

[3] Chandy, K.M., Misra, J.: *Parallel program design: a foundation*, Addison-Wesley, (1989).

[4] Dam, M., Fredlund, L., Gurov,D.: Toward Parametric Verification of Open Distributed Systems. To appear, in *Compositionality: The Significant Difference* (H. Langmaack, A. Pnueli, W.-P. De Roever (eds)), Springer-Verlag 1998.

[5] Arts, T., Dam, M., Fredlund, L., Gurov, D.: System Description: Verification of Distributed Erlang Programs. In Proc. of CADE'98, Springer-Verlag, vol 1421, pp. 38-41, 1998.

[6] Horváth Z.: *A relational programming model of parallel programs.* PhD thesis, in Hungarian. PhD program in Informatics, Department Group Informatics, University Eötvös Loránd Budapest, Hungary. 1996.

[7] Horváth Z.: The Formal Specification of a Problem Solved by a Parallel Program – a Relational Model. In: *Proc. of the Fourth Symp. on Programming Languages and Software Tools*, Hungary, 1995, 165-179. to appear in: *Annales Uni. Sci. Bp. de R. Eötvös Nom. Sectio Computatorica* (1997).

[8] Horváth, Z., Kozsik, T., Venczel, T.: On Composing Problems and Parallel Programs. Paakki, J., ed., *Proceedings of the Fifth Symposium on Programming Languages and Software Tools*, Jyväskylä, Finland, June 7-8, 1997 (1997) 1-12.

[9] Kesseler, M.: Lazy Copying and Uniqueness. In Proc. of 6th International Workshop on the Implementation of Functional Languages, UEA, Norwich,UK, Glauert Ed., pp. 4.1-4.11.

[10] van Lamsweerde, A., Sintzoff, M.: Formal Derivation of Strongly Correct Concurrent Programs. *Acta Informatica*, Vol. 12, No. 1 (1979) 1-31.

[11] de Mol M.: Clean Prover System. Master Thesis no. 442, University of Nijmegen, 1998.

[12] Plasmeijer, R., van Eekelen, M.: *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[13] Serrarens, P.R.: Explicit Message Passing for Concurrent Clean. In K. Hammond et al.: *Proc. of Implementation of Functional Languages, IFL'98*, 298-308, Sept. 1998, London, to appear in LNCS.

[14] Smetsers S., Barendsen E., van Eekelen M., Plasmeijer R.: Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In *Proc. of Graph Transformations in Computers Science*, International Workshop, Dagstuhl Castle, Germany, Schneider and Ehrig Eds., Springer-Verlag, LNCS 776, pp. 358-379.