Verification of the Temporal Properties of Dynamic Clean Processes *

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 this way 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 in this language by using the so called unique types. For example Clean programs perform I/O by accessing their unique environment. Furthermore, dynamic types of Clean make it possible to load some pieces of the program during run-time. We present a methodology for proving safety and liveness properties of concurrent, interleaved Clean Object I/O processes that also use dynamic types and show simple examples for verification of correctness of such 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 [14]. We also consider the *dynamic* possibilities of this language, viz. that certain components can be loaded into the system during run-time.

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 [12].

* This work is the continuation of the research introduced in [9]. The research was supported by the Hungarian State Eötvös Fellowship and by the Hungarian Ministry of Education, Grant Nr. FKFP 0206/97

2 Abstract specifications of reactive systems

Reactive systems may be formulated in Clean as part of a unique environment [16, 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].

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. Let $P, Q, R, U : A \longmapsto \mathcal{L}$ be logical functions.[1] 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 $R \in \mathrm{FP}$ (also denoted by $\mathrm{FP} \Rightarrow R$ for reasons of tradition [3]) 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. $Q \in TERM$ (or with the traditional notation, $Q \hookrightarrow \mathrm{FP}$) 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 $\mathrm{inv}P$.

We specify closed systems, that is we describe the joint behavior of all components. If dynamics are used then open specifications [3, 4] should be applied, i.e. we define statically the semantical properties of dynamically loaded components in forward. These semantical requirements should be satisfied run-time.

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 parallelism [10, 16] are not investigated. In [10] 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 [16]. 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 [14, 17]. I/O in Clean uses the *world as value* paradigm. In this paradigm, external resources such as the file system and event stream are

[1] Logical functions are characterized 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.

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.

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 initialization 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 13–14 (a *timer*), and Section 6.2, line 45–48 (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.

[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.

Dynamics in Clean [13, 15] are dynamically loadable, mobile components that consist of code of Clean expressions and certain type information. This latter, the so-called type code can be used to check and unify the type of the loaded Clean expression during run-time. So, Clean applications can be extended with data and code dynamically in a type safe manner with a functionality obtained e.g. from some other process or a location somewhere on the Internet.

Verification of Clean programs using *Dynamic*s may require the encapsulation of further semantic properties of the dynamically sent and for the dynamically received components, respectively. This semantical code of safety, and certain liveness properties could be given in an encoded form of some program properties described in section 2. This additional semantic information could then be used to check the appropriateness of a mobile component during runtime by comparing the requirement specification in the receiver and the behavior description in the sent component. Our research aims at the extension of the *Dynamic* construct of Clean in this direction.

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.

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 (see 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 [11]. By virtue of its definition the effect relation of a conditional assignment is total, i.e., its domain is equal to 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 sign $\|$ means that in the simultaneous assignment all v_i $(i \in [1,n])$ variables can get a new value, which value is chosen nondeterministically from the set of values that F_{j_i} deliveres, if the condition π_{j_i} holds. This F_{j_i} is a function of all v_k variables.

The program properties with respect to 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 the truth of which is 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 it 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 $\hookrightarrow_S \subseteq \mathcal{P}(A) \times \mathcal{P}(A)$ be 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 characterizes 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 $\{\lceil Q\rceil \mid (Q,\varphi_S) \in \hookrightarrow_S\}$.

The abstract parallel program $S \subseteq A \times A^{***}$ is a solution to 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 composition of programs [8] is also allowed in our model. We will use the *union* construct in this paper, which corresponds to the concurrent execution of programs. The initial assignment of the union program is the simultaneous execution of the initial assignments of the components (the union of programs can only be constructed if this initial assignment is valid), while the set of assignments of the union program is the union of the sets of assignments of the components. The union construct preserves an *invariant*, *unless* or *ensures* property if it holds in each of the components. The *union theorem* [8] shows how the behaviour relation of the union of programs can be computed using the behaviour relations of the components. This way we can prove the correctness of processes

using dynamics if we suppose that open specifications given for the dynamically loaded components are satisfied run-time.

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 our first simple example is presented, 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 that the list stored in the local process state is a sorted permutation of the initial unsorted list.

$$\uparrow\!\hookrightarrow \mathrm{FP} \tag{1}$$

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

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).

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

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

5.2 Clean program

The program in Figure 1 starts an Object I/O process and initializes 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 initialized to be empty and subsequently modified by `initialize` to include a `Timer`. The `TimerFunction` is set to the `bubble` function, i.e. the timer handled by the I/O process calls `bubble` again and again after `0` seconds delay.

A function parameter is given to `bubble`, called `dynamicSweep`, which is loaded during run-time. The `typeCheck` function checks whether `dynamicSweep` has the appropriate type, while `semanticCheck` should check whether it satisfies

```
module dynamic_bubblesort
import StdEnv, StdIO
:: NoState = NoState
Start :: *World -> *World
Start world
  = startIO NDI unsortedlist NoState initialize [] world
  where
  unsortedlist  = [100,98..0]
  initialize :: (PSt [a] .p) -> PSt [a] .p   | < a
  initialize pst
      # (fname, pst) = getFileName pst
      # (dynamicSweep, pst) = readSweep fname pst
      = snd (openTimer undef (Timer 0 NilLS
            [TimerFunction (noLS1 (bubble dynamicSweep))]) pst)
  getFileName pst
     # (maybeFileName, pst) = selectInputFile pst
     | isNothing maybeFileName    = getFileName (appPIO beep pst)
     | otherwise                  = (fromJust maybeFileName, pst)
  readSweep :: String (PSt [a] .p) ->
     ( ([a] -> (Bool,Int,Int)), (PSt [a] .p) )    | < a
  readSweep fname pst
    # (dynamicFun, pst) = readDynamic fname pst
    # (checkedFun, pst) = typeCheck dynamicFun pst
    | semanticCheck specification checkedFun    = (checkedFun,pst)
    | otherwise                  = abort "Semantic check failed!"
    where
    typeCheck :: Dynamic (PSt [a] .p) ->
        ( ([a] -> (Bool,Int,Int)), (PSt [a] .p) )  | TC, < a
    typeCheck (f :: ([^a] -> (Bool,Int,Int)) | < a) pst = (f, pst)
    typeCheck pst = abort "Type check failed!"
    specification = Spec (
        \ list -> (isSorted, i, j)
          where
          isSorted = (forall k in 0..list.dom-2: list!!k <= list!!(k+1))
          not isSorted -> (i<j and list!!j < list!!i)
      )
  bubble :: ([a] -> (Bool,Int,Int)) NrOfIntervals (PSt [a] .p)
                           -> PSt [a] .p    | < a
  bubble sweep _ pst=:{ls=list}
    | sorted     = closeProcess pst
    | otherwise  = {pst & ls=updateAt i (list!!j)
                        (updateAt j (list!!i) list)}
    where (sorted,i,j) = sweep list
```

Fig. 1. The dynamic_bubble program

the given requirement specification. In the above program the algebraic data type value constructed by `Spec` contains this specification: `dynamicSweep` decides if a

list is sorted, and if not, it gives back two indeces, that are in inversion. Currently Clean does not support requirement specifications (`Spec` and `semanticCheck`), only checking of types is possible.

5.3 An abstract model of the program

The s_0 initialization assignment is a composition of the initialization 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. We suppose that the dynamically loaded function meets the requirement specification.

$$s_0 : pst := InitProc(Timer(bubble), (unsortedlist, NoState, empty))$$
$$S : \{\, pst := bubble(dynamicSweep, pst)\}$$

where

$$bubble(sweep, pst) = \begin{cases} (pst.ls, pst.ps, closed) & \text{if } sorted \vee pst.io = closed \\ (newlist(pst.ls), pst.ps, pst.io) & \text{otherwise} \end{cases}$$

where

$$newlist(list) = [list(0), \ldots list(i-1), list(j), list(i+1), \ldots list(j-1), list(i), list(j+1), \ldots list(list.length-1)]$$
$$(sorted, i, j) = sweep(pst.ls)$$

5.4 Formal proof

We prove invariant (3) first. Let us denote $(pst.ls \in perms(unsortedlist))$ by P. $sp(pst := InitProc(Timer(bubble), (unsortedlist, NoState, empty)), True) = (pst.ls = unsortedlist \wedge pst.ps = NoState \wedge pst.io = [Timer(bubble)]) \Rightarrow (pst.ls \in perms(unsortedlist)) = P$. Hence P holds initially. Furthermore, $wp(S, P) = (sorted \vee pst.io = closed \;\rightarrow\; pst.ls \in perms(unsortedlist)) \wedge (\neg sorted \wedge pst.io \neq closed \;\rightarrow\; newlist(pst.ls) \in perms(unsortedlist))$. Both implications follow from P, since $pst.ls \in perms(unsortedlist)$ and, if $\neg sorted$, $newlist(pst.ls) \in perms(pst.ls)$ by the definition of *newlist*. Thus P is stable, too. Being stable and initially true, P is indeed an invariant.

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

If we want to prove (4) we have to calculate the fixed point of the program. $\varphi_S = (\neg sorted \wedge (pst.io \neq closed) \rightarrow pst.ls = newlist(pst.ls) \wedge pst.ps = pst.ps \wedge pst.io = pst.io) \wedge (sorted \vee (pst.io = closed) \rightarrow pst.ls = pst.ls \wedge pst.ps = pst.ps \wedge pst.io = closed)$. If the condition of the first implication holds, then the consequence is false by the definition of *newlist* and the specified properties of *dynamicSweep*: two non-equal elements of $pst.ls$ are swapped in $newlist(pst.ls)$. After reduction we obtain $\varphi_S = ((sorted \vee pst.io = closed) \wedge (\neg sorted \vee (pst.io = closed))$, so $\varphi_S = (pst.io = closed)$. Notice, that $\varphi_S \wedge K \Rightarrow sorted$, so we can conclude, using the specification of *dynamicSweep*, that $\varphi_S \wedge K \Rightarrow sorted(pst.ls)$. Thus we have proved (4).

For proving (1) the theorem of the variant functions [6] can be applied. We introduce the variant function $v = inversion(pst.ls)$, that is the number of pairs of elements in $pst.ls$ that are in wrong order. It is obvious, that v is always a non-negative integer. We have to show that the variant function inevitably decreases until the program reaches its fixed point, i.e. $\neg\varphi_S \wedge v = t \wedge K \hookrightarrow_S (\varphi_S \vee v < t) \wedge K$. Using the facts that $\mapsto_S \subseteq \hookrightarrow_S$ and K is stable, it is sufficient to show, that: $\neg\varphi_S \wedge v = t \wedge K \mapsto_S \varphi_S \vee v < t$. The weakest precondition $wp(pst := bubble(pst), (\varphi_S \vee v < t)) = (\neg sorted \wedge (pst.io \neq closed) \rightarrow (pst.io = closed) \vee (inversion(newlist(pst.ls)) < k)) \wedge (sorted \vee (pst.io = closed) \rightarrow closed = closed \vee (inversion(pst.ls) < k)$. The definition of $newlist$ and the specification of $dynamicSweep$ guarantees, that in case of $\neg sorted$, the number of inversions in $newlist(pst.ls)$ is smaller than in $pst.ls$. Hence it is easy to show, that the weakest precondition calculated above follows from $(\neg\varphi_S \wedge v = t \wedge K)$, which equals to $((pst.io \neq closed) \wedge (inversion(pst.ls) = t) \wedge K)$. This proves the required $\mapsto_S$ property.

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 state spaces of the three components. Process A is running over the subspace $pst_A : PSt \times ch_{ab} : Ch \times ch_{ba} : Ch \times button_A : Button$, where $PSt = (ls : Int, ps : NoState, io : IOSt), Ch = Channel(Int), Button = \{pressed, released\}$. Similarly process B is running over the subspace $pst_B : PSt \times ch_{ab} : Ch \times ch_{ba} : Ch \times button_B : Button$. The user is acting over the subspace $button_A : Button \times button_B : Button$. The overall state space of the system is:

$A = pst_A : PSt \times ch_{ab} : Ch \times ch_{ba} : Ch \times button_A : Button \times$
$\times\ button_B : Button \times pst_B : PSt.$

We use the concept of the 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 the channels are initially empty and the 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 `Process`-es are declared to belong to a `ProcessGroup` with shared state initialized to `NoState`, i.e. no useful shared public state exists. Communication channels are implemented implicitly. The local part of the process state PSt is initialized by the `Process` to 0. The `initialize` function initializes the IO state.

Two state transition functions are defined. Function `pressed` may cause a state transition by sending data and increasing the counter stored in the local state when the corresponding button is pressed. Function `received` is activated to receive data and to increase the counter stored in the local state when a message is arrived to the process. When the counter reaches the value `mc`, the IO state is closed.

The state transition used by `pressed` and `received` to increase the counter in the local state is loaded as a *Dynamic*. Similarly to the program on page 7 the `typeCheck` function is used to check whether this state transition has the proper type. Further semantic description should be given as a set of specification properties in the form presented in section 2. These properties should be represented using the data constructor `Spec` and be checked run-time by the `semanticCheck` function.

Synchronous sending causes 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 results in 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 could have been proved correct with respect to the posed problem. So the concrete program cannot be less synchronised but the abstract program can.

```
module dynamic_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 Process
process name (me,you)
  = ProcessGroup NoState (Process NDI 0 initialize [])
where
  mc = 10
  initialize :: (PSt Int .p) -> PSt Int .p
  initialize pst
    # (fname, pst) = getFileName pst
    # (dynamicCounter, pst) = readCounter pst fname
    = snd (openDialog undef (dialog dynamicCounter) pst)
  getFileName pst
     # (maybeFileName, pst) = selectInputFile pst
     | isNothing maybeFileName    = getFileName (appPIO beep pst)
     | otherwise                  = (fromJust maybeFileName, pst)
  readCounter :: (PSt Int .p) String ->
      ( (Int (PSt Int .p) -> (PSt Int .p)), (PSt Int .p) )
  readCounter pst fname
    # (dynCounter, pst) = readDynamic fname pst
    # (checkedCounter, pst) = typeCheck dynCounter pst
    | semanticCheck specification checkedCounter
                                 = (checkedCounter,pst)
    | otherwise                  = abort "Semantic check failed!"
    where
    typeCheck :: Dynamic (PSt Int .p) ->
          ( (Int (PSt Int .p) -> (PSt Int .p)), (PSt Int .p) )
    typeCheck (transition :: (Int (PSt Int .p) -> (PSt Int .p)))
              pst
      = (transition, pst)
    typeCheck _ pst = abort "Type check failed!"
    specification = Spec (
        \ mc pst :
          (pst.ls=ls and pst.io=io) in INIT
          (pst.ls=ls and pst.io=io) in TERM
          (pst.ls=ls+1) in FP
          (pst.io=(if (ls+1=mc) closed io)) in FP
      )
```

```
dialog counter
  = Dialog name ( ButtonControl "Press me" DefaultWidth
                      [ControlFunction (noLS (pressed counter))]
                  :+: Receiver me (noLS1 (received counter)) []
                  ) []
pressed :: (Int (PSt Int .p) -> (PSt Int .p)) (PSt Int .p) ->
           PSt Int .p
pressed counter pst
  # (_,pst)   = syncSend you undef pst
  = counter mc pst
received :: (Int (PSt Int .p) -> (PSt Int .p)) .i (PSt Int .p)
            -> PSt Int .p
received counter _ pst
  = counter mc pst
```

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 initialization assignments of the processes are a composition of the initialization of the components and the subsequent set of the IO state.

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

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

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

The dynamically loaded $pst := counter(pst)$ state transition appears in the assignments of S_A and S_B. Note, that it is safe to use it as an assignment, since its specification guarantees, that the computation defined by *counter* will always terminate. This state transition is atomic in our abstract program, as well as in the Clean implementation. In the formal proof below the fixed-point property specified for *counter* will also be used.

6.4 Formal proof

Now we prove[3] that the abstract program above solves the specification properties (5-9) introduced in section 6.1. It is enough to show that the program satisfies those properties for its reachable states. Thus we determine a couple of invariants first.

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

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

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

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

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

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

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 does not 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 specification of *counter*. Only this state transition can change the ls and io components of pst_A and pst_B. A $pst.ls$ is increased only by one, and it becomes mc at the very same time as the $pst.io$ becomes closed. After the $pst.io$ is closed, neither the send nor the receive statement can be executed, thus $pst.ls$ cannot 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 increases $|\overline{ch_{ab}}|$ or decreases $|ch_{ba}|$. The values in the left-hand side of the equation can also be changed by a send on ch_{ba}, but such

[3] For the sake of brevity and understandability we restrict ourselves to give a hint how weakest preconditions should be calculated. Examples are presented in section 5.4.

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 to 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 does not change its state when none of the conditions is true. After some steps of simplification the φ_S set of fixed points becomes the following:

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

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 first part of (13) 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 functionS. We choose the variant function $v = 2 * mc + 1 - pst_A.ls - pst_B.ls$. (The invariants guarantee that this 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.

As we saw earlier, φ_S is the conjunction of the negated conditions of the assignments: thus the program is not in fixed point when one of those conditions hold. Since "leads-to" is closed under disjunction, it is sufficient to prove separately for each condition, that the variant function inevitably decreases if the program is in a state in which the condition holds. Notice, that the statements in S_A and S_B decrease the variant function when their conditions permit state change. If C is a condition of one of the assignments in S_A or S_B, the property $C \wedge v = k \ \mapsto_{mc} \ v < k$ is satisfied. The statements of S_U keep the left-hand side stable, and the statements of S_A and S_B either keep the left-hand side stable (e.g. when not causing state change) or decrease the variant function. The assignment with C as condition will surely cause state change and decrease v.

What is left is the conditions of the assignments in S_U – as an example let us take the first one. The following property is easy to prove:

$$button_A = released \wedge |ch_{ab}| = 0 \wedge pst_A.io \neq closed$$
$$\mapsto_{mc} \; button_A = pressed \wedge |ch_{ab}| = 0 \wedge pst_A.io \neq closed$$

The right-hand side can be written as a disjunction of

a.) $button_A = pressed \wedge |ch_{ab}| = 0 \wedge pst_A.io \neq closed \wedge |ch_{ba}| = 0$ and
b.) $button_A = pressed \wedge |ch_{ab}| = 0 \wedge pst_A.io \neq closed \wedge |ch_{ba}| \neq 0$.

The first of these two formulas is the condition of the first assignment in S_A, while the second is a subset of that of the second. The rest of the proof follows from the basic properties of the "leads-to" relation.

7 Conclusions

We presented a two phase methodology for proving the correctness of Clean Object I/O processes with a well-defined set of state transition functions. Some of the state transition functions can be loaded dynamically if the temporal properties of the requirement specification are fulfilled. We propose to extend the structure of dynamics with additional semantical information. This can be used to perform further run-time checks in the next generation of the Clean compiler, implementing the extended form of dynamics, and a predefined library containing the `semanticCheck` function and the algebraic data type `Spec`.

The methodology can be applied for dynamically extended 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 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.

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] Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Proving the Temporal Properties of the Unique World. *Software Technology, Fenno-Ugric Symposium FUSST'99 Proceedings*, Technical Report CS 104/99, Tallin, 1999. pp 113-125.

[10] 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.

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

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

[13] Pil, M.: First Class File I/O. *In Proc. of Implementation of Functional Languages, 8th International Workshop, IFL '96, Selected Papers*, Bad Godesberg, Germany, Kluge Ed., Springer Verlag, LNCS 1268, pp. 233-246.

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

[15] Plasmeijer, R.: Mobile Expressions in the Functional Language Clean. *Software Technology, Fenno-Ugric Symposium FUSST'99 Proceedings*, Technical Report CS 104/99, Tallin, 1999. pp 51.

[16] 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.

[17] 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.