PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The version of the following full text has not yet been defined or was untraceable and may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/18758

Please be advised that this information was generated on 2019-02-17 and may be subject to change.
Translating IOA automata to PVS

M.C.A. Devillers

Computing Science Institute/

CSI-R9903 February 1999
Translating IOA automata to PVS
– Preliminary Report –

M.C.A. (Marco) Devillers

Computing Science Institute
University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
marcod@cs.kun.nl

February 19, 1999

ABSTRACT

IOA is a specification language for input/output automata based on the model of Lynch and Tuttle; IOA is developed at MIT by Garland and Lynch and is a part of the Larch family of specification languages. We present a compiler which translates IOA specifications to input for the Prototype Verification System (PVS) by means of examples.


1991 Mathematics Subject Classification: 03B35 Mechanization of proofs and logical operations,

Keywords and Phrases: Automata, Tools, Theorem Provers
1. Introduction

The IOA language is a specification language for distributed algorithms, developed at MIT by Garland and Lynch [GL98]. The IOA language is based on the well-known theory of input/output automata by Lynch and Tuttle [LT89] [Lyn96]. Tools supporting the IOA language are under development: they include a simulator, a compiler to the Larch theorem prover [GH93] and a compiler to the model checker SPIN [Hol91]. The tool described in this paper is intended to become part of this toolset: a compiler from IOA to the Prototype Verification System (PVS) [ORSH95], a theorem prover based on higher-order logic. The tool is under development, the current version can be tried at http://www.cs.kun.nl/~marcod/ioa.html.

A substantial amount of research efforts has gone into the verification of distributed systems described as input/output automata; both verifications involving pencil and paper proofs and verifications involving model checkers or theorem provers. In a number of cases, proof systems are successfully used to check, or construct, proofs of theorems which are not intellectually demanding but require a lot of bookkeeping. The compiler plays a significant rôle in the mechanization of the verification process of input/output automata. It supports the reuse of theories developed by others, be it supporting input/output automata theory or a specific theory such as, for instance, a theory which models communication over graph-like structures: it supports a standardized approach to input/output automata verifications in PVS.

The outline of the paper is as follows. In section two, we describe part of the IOA language briefly with an example automaton. In the section three, we describe a PVS data type for input/output automata and supporting automata theory specified in PVS; the compiler translates IOA automata specifications to members of this data type. In the fourth section, we show a number of examples how this translation is achieved, and touch on some future work. In section five, we describe implementation details of the compiler, and place the compiler in an IOA/PVS verification environment. As usual, we end with the conclusions in the last section.

2. The IOA Language

In this section we briefly present part of the IOA language by means of an example automaton specification, and an assertion over that automaton.

2.1 IOA automata

The example automaton is named CoffeeMachine, it is shown in figure 1 below. An IOA automaton expression denotes a labeled transition system where the action alphabet is partitioned into input, output and internal actions, and the transitions are input-enabled, i.e. each input action can be accepted in all states.

```plaintext
automaton CoffeeMachine

signature
    input Coin(val : nat) where val = 5 \ or \ val = 10 \ or \ val = 25
    internal TempHigh, TempLow
    output CupOfCoffee

states
    received : nat := 0,
    hotwater : bool := false

transitions
    input Coin(val : nat)
    eff received := received + val
    internal TempHigh
    pre received >= 50 \ or \ hotwater
```
The IOA Language

2. The IOA Language

In the signature of the automaton, the actions of the automaton are listed. The input action Coin specifies the events that the automaton accepts a coin. This action is parameterized with a natural number val denoting the value of the coin; in the where clause of the action, that value is restricted to be either five, ten, or twenty-five cents. The internal TempHigh action denote the events that the water in the machine has reached a high enough temperature to produce coffee; the TempLow action denotes the opposite. The output action CupOfCoffee expresses the event that a cup of coffee is produced.

In the states section, the state variables are listed. In this automaton two state variables are present: the received variable is a natural number used to hold an amount of money, and the hotwater boolean denotes whether the water in the machine is of a sufficient temperature to produce coffee. Initially the received variable is set to zero, and the hotwater variable false.

The transitions section describes the transition system of the automaton in precondition/effect style. In the example, the transitions associated with the Coin action are described first. Since it is an input action it has no precondition, i.e. the automaton always accepts a coin. The effect of the action increases the received variable with the value of the coin. An internal TempHigh transition is possibly taken if it is enabled; that is if the precondition of that transition holds. The precondition is enabled if a sufficient amount of money (fifty cents) is received; the effect sets the hotwater variable. Its counterpart, a TempLow transition is taken if the water temperature is too low to produce coffee, it sets the hotwater variable to false. The CupOfCoffee transition outputs a cup of coffee. The automaton can produce a cup of coffee if enough money is received and the water is heated; if a cup of coffee is produced the costs are subtracted from the received variable.

However, so far the automaton does not need to produce a cup of coffee when given money; it might just choose to do nothing. However, within the model of Lynch and Tuttle, such behavior can be excluded by restricting attention to the fair behavior of an automaton. Fair behavior of an automaton is usually defined as that behavior in which each partition –task– of a partitioning of the set of internal and output actions is treated fairly. In the coffee machine example we placed the internal TempHigh and TempLow actions in another task than the CupOfCoffee output action to express that, whatever happens, coffee is produced.

2.2 Invariants

After an automaton is defined, it is possible to specify invariants of that automaton: predicates which hold for all the reachable states of an automaton. In the following figure 2, an example invariant of the coffee machine automaton is listed.

![Figure 2: a coffee machine](image-url)

![Figure 2: an invariant of the coffee machine](image-url)
3. Input/output automata in PVS

The simple invariant above expresses that in every reachable state of the automaton, the value of the received variable is divisible by five.

3. Input/output automata in PVS

The aim of the compiler is to translate input/output automata as presented in the previous section to instantiations of a PVS datatype `IOAutomaton`. Roughly speaking, the `IOAutomaton` data type describes the class of all input/output automata according to the theory of Lynch and Tuttle. In this section, we list the PVS theories which define the `IOAutomaton` data type.

In the following figure 3, the PVS theory hierarchy is shown; the top theory named `IOA_theory` is the wrapper theory which imports all lemma's and definitions. As one might deduce from the figure, for the moment, only a theory of invariants of input/output automata is supported.

![PVS theory hierarchy](image)

In the theory `LTS`, shown in figure 4, labeled transition systems are defined as a tuple comprising a set of initial states, and a transition relation of labeled edges between states.

```pvs
LTS[actions: TYPE+, states: TYPE+] : THEORY
BEGIN
   LTS: TYPE = [# initial?: [states->bool],
            transitions?: [[states, actions, states]->bool] #]
END LTS
```

Signatures over an action alphabet `actions` are defined as a triples of disjoint subsets of the alphabet; the PVS theory is shown in figure 5 below. Also the sets of externally observable (input or output) and locally controllable (internal or output) actions are defined as the respective predicates `external?` and `local?`.

```pvs
Signature[actions:TYPE+] : THEORY
BEGIN
```

---

The simple invariant above expresses that in every reachable state of the automaton, the value of the received variable is divisible by five.

3. Input/output automata in PVS

The aim of the compiler is to translate input/output automata as presented in the previous section to instantiations of a PVS datatype `IOAutomaton`. Roughly speaking, the `IOAutomaton` data type describes the class of all input/output automata according to the theory of Lynch and Tuttle. In this section, we list the PVS theories which define the `IOAutomaton` data type.

In the following figure 3, the PVS theory hierarchy is shown; the top theory named `IOA_theory` is the wrapper theory which imports all lemma's and definitions. As one might deduce from the figure, for the moment, only a theory of invariants of input/output automata is supported.

![PVS theory hierarchy](image)

In the theory `LTS`, shown in figure 4, labeled transition systems are defined as a tuple comprising a set of initial states, and a transition relation of labeled edges between states.

```pvs
LTS[actions: TYPE+, states: TYPE+] : THEORY
BEGIN
   LTS: TYPE = [# initial?: [states->bool],
            transitions?: [[states, actions, states]->bool] #]
END LTS
```

Signatures over an action alphabet `actions` are defined as a triples of disjoint subsets of the alphabet; the PVS theory is shown in figure 5 below. Also the sets of externally observable (input or output) and locally controllable (internal or output) actions are defined as the respective predicates `external?` and `local?`.

```pvs
Signature[actions:TYPE+] : THEORY
BEGIN
```
3. Input/output automata in PVS

\[
\text{disjoint?}((A, B, C: \text{[actions]->bool])): \text{bool} = \\
\text{(disjoint?}(A, B) \text{ AND disjoint?} (B, C) \text{ AND disjoint?} (C, A))
\]

\[
\text{signature: TYPE} = \\
\{s: \text{[# input?, output?, internal?: [actions]->bool #]}
| \text{disjoint?}(\text{input?}(s), \text{output?}(s), \text{internal?}(s))\}
\]

\[
\text{external?}(s: \text{signature}): \text{[actions->bool]} = \\
\text{union}(\text{input?}(s), \text{output?}(s))
\]

\[
\text{local?}(s: \text{signature}): \text{[actions->bool]} = \\
\text{union}(\text{internal?}(s), \text{output?}(s))
\]

END Signature

— fig. 5: signature

In figure 6, the \text{IOAutomaton} data type is defined. This data type defines automata as a triple of a labeled transition system, a signature, and a tasks set. For this data type it is required that all input actions are enabled in all states, and tasks form a partitioning over the local states.

\text{IOAutomaton[actions: TYPE+, states: TYPE+]}: \text{THEORY}

BEGIN

IMPORTING \text{LTS[actions, states], Signature[actions]}

\[
\text{input_enabled?}(\text{Its}: \text{LTS}, s: \text{Signature}): \text{bool} = \\
\text{(FORALL (s0: states, a: actions):}
\text{input?}(s)(a) => \\
\text{EXISTS (s1:states): transitions?(\text{Its})(s0,a,s1))}
\]

\[
\text{tasks_partition?}(s: \text{Signature}, \text{tasks}:[[\text{actions->bool}]->\text{bool}]): \text{bool} = \\
\text{(FORALL (a:actions):}
\text{(EXISTS (t:[[\text{actions->bool}]->\text{bool}]): tasks(t) AND t(a)) => local?(s)(a))}
\text{AND}

\text{(FORALL (a:actions): local?(s)(a) => \\
\text{(EXISTS (t0:[[\text{actions->bool}]->\text{bool}]): tasks(t0) AND t0(a) AND}
\text{(FORALL (t1: [[\text{actions->bool}]]: tasks(t1) AND t1(a) IMPLIES t0=t1)))})}
\]

\text{IOAutomaton: TYPE} = \\
\{ M: \text{[# sig: Signature , lts : LTS, tasks? : [[actions->bool]->bool #]}
| \text{input_enabled?}(\text{lts}(M), \text{sig}(M)) AND \text{tasks_partition?}(\text{sig}(M),\text{tasks?}(M))\}
\]

END IOAutomaton

— fig. 6: input/output automata

The compiler produces an instantiation of the \text{IOAutomaton} data type for each translated automaton. As show in figure 3, supporting theory for invariants is developed around the \text{LTS} and \text{IOAutomaton} types. In theory \text{LTS_reachable}, a definition for the set of all reachable states of a labeled transition system is stated, together with interesting properties. The \text{LTS_stable} theory is a theory on stable predicates on states, predicates which are preserved by the transition relation. The \text{LTS_inductive} theory is a theory on properties of inductive predicates, predicates which hold on initial states and are stable; a similar result is derived for reachable inductive predicates, predicates which are inductive on reachable states, in the theory \text{LTS_reachable_inductive}. In the \text{LTS_invariant} theory, it is stated that predicates on states are invariant when they hold on all reachable states; it imports and combines the results of the preceding theories. The main result stated in that theory is, of course,
that inductive predicates on states are invariant. Below we list part of the LTS_invariant theory in figure 7.

\begin{verbatim}
LTS_invariant[actions: TYPE+, states: TYPE+]: THEORY
BEGIN

IMPORTING LTS[actions, states], LTS_reachable[actions, states],
    LTS_stable[actions, states], LTS_inductive[actions, states],
    LTS_reachable_inductive[actions, states]

invariant?(Its: LTS)(P: [states -> bool]): bool =
    (FORALL (s: states): reachable?(Its)(s) IMPLIES P(s))

inductive_invariant: LEMMA FORALL (Its: LTS): FORALL (P: [states -> bool]):
    inductive?(Its)(P) IMPLIES invariant?(Its)(P)

reachable_inductive_invariant: LEMMA FORALL (Its: LTS): FORALL (P: [states -> bool]):
    reachable_inductive?(Its)(P) IMPLIES invariant?(Its)(P)

invariant_subset:  LEMMA FORALL (Its: LTS): FORALL (P,Q: [states -> bool]):
    invariant?(Its)(P) AND subset?(P,Q) IMPLIES invariant?(Its)(Q)

invariant_intersection: LEMMA FORALL (Its: LTS):
    FORALL (P,Q: [states -> bool]):
    invariant?(Its)(P) AND invariant?(Its)(Q)
    IMPLIES invariant?(Its)(intersection(P,Q))
END LTS_invariant
\end{verbatim}

— fig. 7: the PVS invariant theory on LTS's

In the theory IOA_invariant mosts definitions and results are lifted from labeled transition systems to input/output automata.

4. Translation to PVS

In this section, it is discussed how IOA automaton specifications are translated to PVS; i.e. how a specification of an automaton in IOA is translated to a member of the IOAutomaton data type presented in the previous section 3. The translation is described in a top-down fashion from the grammar: we start with type definitions and end with the translation of the transition relation.

4.1 LSL traits, LSL shorthands, and external theories

Within automata specifications, new data types may be defined, or externally defined data types may be included. For each automaton we generate a types theory which holds all the included or newly defined data types.

**LSL traits**  As stated, IOA is part of the Larch family of languages. It is possible to use axiomatic first-order theories, called traits, of the Larch Shared Language (LSL) within IOA. The translation of LSL traits to PVS theories is described in [Dev98]. An example of an LSL trait is shown in figure 8. The trait RingTopology defines a data type Node as the type which is generated by a finite number of application of start, and left, and that has an end node from which the start of the ring can be reached. Also, the nodes are uniquely numbered by an index operator, which has an inverse named select.
The index and select operators are added to the trait for technical reasons. We want to use this trait such that we can define an automaton for each node, see the automaton RingAutomaton defined later in this section in figure 10. A natural manner to specify such an automaton is by parameterizing it with a constant of the sort Node; unfortunately, at the moment it is not possible to use sorts, or constants of a sort, defined locally in traits as formal parameters of an automaton. In the example trait, this problem is solved by parameterizing the automaton with a natural number (the natural numbers are a predefined type) which refers to an element in the Node sort by means of the mentioned operators. Of course, we hope to implement a better solution in the near future.

Below, we list the translation of this trait, as described in [Dev98]. In that paper, it is also described how a user can prove that a trait is consistent by providing a model for the axiomatization in the higher-order logic of PVS.

The automaton below uses the RingTopology trait; it defines a protocol which elects the node with the maximum index number as the leader.
When a trait is used within an automaton, the theory associated with that trait is imported in the types theory of that automaton.

In the IOA language definition as given by Garland and Lynch [GL98], one is also allowed to assume traits, with as semantics that the axioms of a trait assumed must be satisfied by the formal parameters of an automaton. This construct is not supported in the current version of the compiler for reasons of time: it requires a non-trivial change to the compiler.

**LSL shorthands** A manner of defining new types is by using the LSL shorthands for enumerations, tuples, and unions [GH93]. Within the LSL language, shorthands are interpreted as axiomatic descriptions of often used LSL data types. However, since our compiler translates to PVS, we interpret shorthands as PVS types in order to be able to exploit PVS built-in proof strategies for these types. In figure 12 below, we give an example of an automaton in which an enumeration type is defined.

The types theory used by that automaton then includes a PVS data type `maybe` as shown below.

— fig. 10: a ring automaton

— fig. 11: the ring automaton types

— fig. 12: an enumeration
4. Translation to PVS

Chooser_types: THEORY
BEGIN
    maybe: DATATYPE
    BEGIN
        yes: yes?
        no: no?
        maybe: maybe?
    END maybe;
END Chooser_types

— fig. 13: translated enumeration type

The other LSL shorthands are translated in a similar fashion.

External theories Lastly, we allow users to re-use PVS theories with a small but useful full extension of the IOA language. For instance, in figure 14, the interface of an external PVS theory List is defined. The automaton Buffer listed in figure 15 includes that theory.

defines
List[T] : type;
nil : List[T];
cons : [[T, List[T]]->List[T]];
append : [[T, List[T]]->List[T]];
append : [[List[T], T] -> List[T]]

— fig. 14: external PVS theory

automaton Buffer(Msg: type)
uses List(Msg)
signature
input InMsg(m:Msg)
output OutMsg(m:Msg)
states
buffer : List[Msg] := nil,
working : bool := true
transitions
input InMsg(m:Msg)
eff buffer := append(buffer, m)
output OutMsg(m:Msg)
choose 1:List[Msg]
pre buffer = append(m, 1)
eff buffer := 1

— fig. 15: buffer automaton using lists

For this example, the types theory will include a reference to a user-defined PVS List theory as shown below.

Buffer_types[Msg : TYPE+] : THEORY
BEGIN
    IMPORTING List[Msg]
END Buffer_types

— fig. 16: included PVS theory
4. Translation to PVS

4.2 Signature

The signature of a generator describes sets of input, output, and internal actions as subsets of an action (alphabet) type. Below, in figure 17, we list the translation of the signature of the coffee-machine automaton of figure 1.

CoffeeMachine_signature: THEORY
BEGIN
IMPORTING CoffeeMachine_types
actions: DATATYPE
BEGIN
Coin(n_0 : nat): Coin?
TempHigh: TempHigh?
TempLow: TempLow?
CupOfCoffee: CupOfCoffee?
END actions

input? : \([\text{actions}]\rightarrow \text{bool}\) = (LAMBDA (action : actions) :
CASES action OF
Coin(n_0): (LAMBDA (val : nat) : val = 5 OR val = 10 OR val = 25)(n_0),
TempHigh: false,
TempLow: false,
CupOfCoffee: false
ENDCASES)

output? : \([\text{actions}]\rightarrow \text{bool}\) = (LAMBDA (action : actions) :
CASES action OF
Coin(n_0): ((LAMBDA (val : nat) : false))(n_0),
TempHigh: false,
TempLow: false,
CupOfCoffee: true
ENDCASES)

internal? : \([\text{actions}]\rightarrow \text{bool}\) = (LAMBDA (action : actions) :
CASES action OF
Coin(n_0): ((LAMBDA (val : nat) : false))(n_0),
TempHigh: true,
TempLow: true,
CupOfCoffee: false
ENDCASES)

action? : \([\text{actions}]\rightarrow \text{bool}\) = (LAMBDA (action : actions) :
(input?(action) OR (output?(action) OR internal?(action))))
END CoffeeMachine_signature

— fig. 17: translated signature of the coffee automaton

The theory first imports the theory Coffee_types holding the declared or included data types. The signature is translated into the actions data type which comprises alphabet; the three predicates named input?, output?, internal? discriminate between the input, output, and internal actions respectively. In the theory, a predicate is used for the input action Coin to restrict the formals of that action to the legal values of five, ten, and twenty-five cents.

The last predicate, action?, is defined as the disjunct of the previous predicates and is used to restrict the actions alphabet type to the set of actions on the transitions of an automaton. In the
coffee-machine example, for instance, Coin(3) is a member of actions whereas it does not satisfy action?.

4.3 States
State variables are translated to observers and modifiers on a global state. For the coffee machine example of figure 1, this results in the following translation.

CoffeeMachine_states: THEORY
BEGIN
IMPORTING CoffeeMachine_signature
states: TYPE+ = [nat, bool]
received : [[states]->nat] = LAMBDA (st:states) : proj_1(st)
hotwater : [[states]->bool] = LAMBDA (st:states) : proj_2(st)
set_received : [[states, nat]->states] =
LAMBDA (st:states, arg:nat) : (arg, hotwater(st))
set_hotwater : [[states, bool]->states] =
LAMBDA (st:states, arg:bool) : (received(st), arg)
initial? : [[states]->bool] = (LAMBDA (st : states) :
((received(st) = 0) AND (hotwater(st) = false)))
END CoffeeMachine_states

The theory includes the Coffee_signature theory described before. First, a non-empty type states is defined; it is defined as the sum of the types of the state variables. For each state variable, an observer and a modifier on the state is constructed: in figure 18, the observer received returns the value of the received state variable given a state, and set_received updates a state given a value. The predicate initial? describes all the initial states of the automaton.

4.4 Transitions
Transitions described in precondition/effect style are translated to a ternary relation on pre-states, actions, and post-states: the precondition is translated to a predicate, the effect part is translated to a function, and an optional so that clause is translated to a relation. The transition relation of the automaton then is defined as the union of the transition relations associated with each action.

precondition/effect style transitions In the following figure 19, the transition relation of the coffee-machine automaton is listed.

CoffeeMachine_transitions: THEORY
BEGIN
IMPORTING CoffeeMachine_states
transitions?: [[[states, actions, states]->bool] =
(LAMBDA (pre_st : states, action : actions, post_st : states) : (action?(action)
series of assignments and conditionals  The effect part of a transition relation described in precondition/effect style may hold a series of assignments, and conditional statements as was shown in the automaton RingAutomaton of figure 10. Below, in figure 20, we list the transition relation of that automaton.

choose parameters  As shown in the OutMsg action of the Buffer automaton of figure 15, it is possible to use choose parameters to relate the precondition to the effect. Below, we show the transition relation for that automaton.

Buffer_transitions[Msg : TYPE*] : THEORY
BEGIN
IMPORTING Buffer_states[Msg]

transitions? : ([states, actions, states]->bool) =
(LAMBDA (pre_st : states, action : actions, post_st : states) : (action?(action) AND
CASES action OF
  InMsg(m_0) : ((LAMBDA (m_0 : Msg) : (LAMBDA (m : Msg) : (post_st = set_buffer(pre_st,append(buffer(pre_st),m))))(m_0)))(m_0),
  OutMsg(m_0) : ((LAMBDA (m_0 : Msg) : (LAMBDA (m : Msg) : (send(pre_st) AND (NOT(leader(pre_st)) AND (n = max(pre_st)))) AND (post_st = set_send(pre_st,false))))(m_0)))(m_0)
ENDCASES))

— fig. 20: transitions of the ring automaton
4. Translation to PVS

\[
\text{OutMsg}(m_0) : \text{(LAMBDA } (m_0 : \text{Msg}) : \text{(LAMBDA } (m : \text{Msg}) : \\
\text{EXISTS } (1 : \text{List[Msg]}) : ((\text{buffer(pre_st) = append(m,1)) AND (post_st = set_buffer(pre_st,1))))}(m_0))(m_0)
\]

— fig. 21: transitions of the buffer automaton

**non-deterministic assignment** Also, non-deterministic assignments are allowed by means of the `choose` statement. The following theory of figure 22 lists the transition relation of the Chooser automaton of figure 12.

\[
\text{transitions? : [states, actions, states]->bool} = \\
\text{(LAMBDA } (\text{pre_st : states, action : actions, post_st : states}) : (\text{action?action}) AND \\
\text{CASES action OF} \\
\text{Out}(m_0) : \text{(LAMBDA } (m_0 : \text{maybe}) : \text{(LAMBDA } (m : \text{maybe}) : \\
\text{((send(pre_st) = m) AND EXISTS (choose_m0 : \text{maybe}) : (true AND (post_st = set_send(pre_st,choose_m0))))}}(m_0))(m_0)
\]

— fig. 22: transitions of the chooser automaton

We lift non-determinism of a single assignment to the transition-relation by generating variables for the non-deterministically ‘chosen’ values; in the example above, the variable `choose_m0` refers to such a value.

**so that** As an alternative to the precondition/effect style, it is also possible to define the transition relation associated with an action by means of the `so that` construct.

For instance, the `Out` transition of the Chooser automaton can also be written as:

\[
\text{output Out(m:maybe)} \\
\text{pre send = m} \\
\text{so that } \forall m:\text{maybe}. \text{ send' = m}
\]

— fig. 23: alternative Out transition

In that case, we get the following transition relation.

\[
\text{transitions? : [states, actions, states]->bool} = \\
\text{(LAMBDA } (\text{pre_st : states, action : actions, post_st : states}) : (\text{action?action}) AND \\
\text{CASES action OF} \\
\text{Out}(m_0) : \text{(LAMBDA } (m_0 : \text{maybe}) : \text{(LAMBDA } (m : \text{maybe}) : \\
\text{((send(pre_st) = m) AND EXISTS (m : \text{maybe}) : (send(post_st) = m))))}(m_0))(m_0)
\]

— fig. 24: translated so that predicate

Primed state variables are translated to observers on post-states, unprimed observers are translated to observers on pre-states.
4. Translation to PVS

4.5 Tasks

Tasks are translated in a straightforward manner to a set of sets of actions denoted as a predicate. Below, the translation of the tasks partition of the coffee machine automaton of figure 1 is listed.

\[ \text{CoffeeMachine\_tasks: THEORY} \]
\[ \text{BEGIN} \]
\[ \text{IMPORTING CoffeeMachine\_transitions} \]
\[ \begin{align*}
\text{tasks? : } & \{ \{ \text{actions} \} \mapsto \text{bool} \} \mapsto \text{bool} = \\
& (\text{LAMBDA} (\text{task} : \{ \text{actions} \} \mapsto \text{bool}) : \\
& \left( (\text{task} = (\text{LAMBDA} (\text{action} : \text{actions}) : ((\text{action} = \text{TempHigh}) \text{ OR } (\text{action} = \text{TempLow}))) \right) \text{ OR } (\text{task} = (\text{LAMBDA} (\text{action} : \text{actions}) : (\text{action} = \text{CupOfCoffee}))) \}
\end{align*} \]
\[ \text{END CoffeeMachine\_tasks} \]

fig. 25: translated coffee machine tasks

4.6 Automata

The somewhat cryptic translation below describes the final translation of the coffee machine automaton of figure 1; the coffee machine is defined as an instantiation of the IOAutomaton type, defined in the PVS library ioa.

\[ \text{CoffeeMachine\_automaton: THEORY} \]
\[ \text{BEGIN} \]
\[ \text{ioa} : \text{LIBRARY} = "~/ioa/pvs/" \]
\[ \text{IMPORTING CoffeeMachine\_tasks, ioa@IOA\_theory[actions,states]} \]
\[ \begin{align*}
\text{CoffeeMachine\_automaton: IOAutomaton} = \\
\text{its := } & (\# \text{ initial? := initial?, transitions? := transitions? #),} \\
\text{tasks? := tasks? #}) \\
\end{align*} \]
\[ \text{END CoffeeMachine\_automaton} \]

fig. 26: the coffee machine automaton in PVS

Because both the Signature and the IOAutomaton types must satisfy certain conditions, type checking the Coffee\_automaton theory will generate a number of type correctness conditions: the user will have to prove that the automaton is input enabled and that the tasks? sets denote a legal partitioning on local actions.

4.7 Invariants

In figure 27 below, we show the PVS translation of the invariant of figure 2: the received state variable is always dividable by five. This theory makes use of the PVS theories IOA\_invariant described in section 3.

\[ \text{CoffeeMachine\_invariants: THEORY} \]
\[ \text{BEGIN} \]
\[ \text{IMPORTING CoffeeMachine\_automaton} \]
\[ \text{CoffeeMachine\_inv\_0 : } \{ \text{states} \} \mapsto \text{bool} = \]
\[ (\text{LAMBDA} (\text{st} : \text{states}) : \text{EXISTS } (n : \text{nat}) : (\text{received(st)} = (5 \ast n))) \]
\[ \text{CoffeeMachine\_inv\_0 : LEMMA} \]
An invariant is translated to two expressions in PVS, as shown above: the invariant predicate is defined, in the example above `CoffeeMachine_inv_0`, and a lemma expresses that all reachable states satisfy the invariant. The proof of `CoffeeMachine_inv_0` is listed in appendix I: the lemma holds since the predicate is inductive.

4.8 Final remarks

At the moment, parts of the tool are still under development. For instance, the IOA language has constructs for (bounded) iteration in the language, allows one to compose multiple automata, and has several manners in which one can define assertions (such as invariants or simulation relations) over automata. Some of these constructs are only partly implemented (at the moment automata compositions and simulation relations are not supported), and for some of the IOA constructs, it is unclear how to translate them to PVS: we found a mistake in the semantics of the `forall` construct defined in the IOA language which allows iteration in the programs of the effect part of a transition, therefore a translation for this construct is not implemented; also, IOA (in some extend LSL) handles formal parameters of theories and theory inclusion different than PVS. IOA, based on LSL, assumes a linear syntactic inclusion mechanism, PVS assumes hierarchical inclusion mechanism based on operations on the semantics of a theory. Where the the syntactic operations on LSL traits are done by the LSL to PVS compiler; we felt these syntactic operations would result in an unreadable translation for IOA automata; for that reason, for instance, the `assumes` construct was not implemented.

5. The IOA to PVS compiler

In figure 28, a picture describing the structure of the compiler is shown. The compiler takes an IOA specification as its input and translates it in a number of steps to a PVS theory file. The boxes describe several components of the compiler, and the arrows the data structures passed between them. Also, it is shown which components can produce errors.

The compiler takes as input an automaton, or a trait, and outputs a PVS specification file. The scanner and parser are the usual components of a compiler. Our compiler is slightly non-standard in that we translate an IOA specification to a context, an internal representation of a higher-order logic theory. We use this internal representation both for the semantical analysis, and as the logical interpretation of an input file to be pretty printed to PVS.

Advantages of this approach are that it results in more compact code for the compiler, and that it will be easier to produce code for other theorem provers based on higher-order logic, such as Isabelle/HOL.
6. Conclusions

A disadvantage is that at the parse tree level, less information is available than in most other compilers, which may influence future extensions of the compiler.

In the following picture, we describe the environment in which the compiler is to be used. As mentioned earlier, the compiler translates an IOA automaton to a member of the PVS IOAutomaton data type for which also supporting theory for input/output automata has been developed. PVS is normally used in emacs; we implemented an emacs mode for IOA which can be run in parallel with the PVS mode. The IOA mode provides syntax-colored highlighting, automatic compilation, and error reporting routines.

![fig. 29: the environment](image)

The IOA compiler has been developed in Java [AG96] in order to be able to easily link this compiler to other tools in the toolset. The following line describes how to start the tool:

```
java ioa [-/+tokens][-/+unparse][-/+check][-/+context][-/+pvs] fn.lsl
```

![fig. 30: compiler options](image)

The check switch may be used to semantically check an input file. The other switches produce the output of one of the components refered to. For instance, the context switch will produce a printed representation of the context, or the pvs switch will produce PVS output. The compiler is lazy in the sense that if only the unparse switch is set, for instance, it will only print the parse tree but will not perform semantical analysis.

6. Conclusions

We have presented a compiler which translates automaton specifications written in the IOA language [GL98] to equivalent PVS specifications[ORS95]; it does so by translating IOA automata to members of a predefined PVS input/output-automata data-type. Since IOA is part of the Larch family of specification languages, the compiler also supports the translation of LSL traits [GH93] to PVS as discussed in [Dev98]. A small extension of the IOA language allows the user to link to externally defined PVS theories. The tool is under development: we did not discuss constructs which are not (or partly) supported by the compiler. We found a small mistake in the semantics of an iteration construct of the IOA language; also, discrepancies between the inclusion mechanism of LSL/IOA and PVS are sometimes a source of problems. The tool is written in Java [AG96] to allow an easy link to other tools in the toolset; also, an IOA emacs mode to be used in parallel with PVS is provided.
References


I. An example proof of an invariant

In the following figure, the proof of CoffeeMachine_inv_0 defined in figure 27 is listed. Comments are provided on the significant steps of the proof.

```
(""
(USE "IGA_invariant.inductive_invariant") %we prove that the predicate is invariant
(PROP) %by showing it is inductive
(HIDE 2) %hide the old consequent
(EXPAND "inductive?") %we expand the definition of "inductive?" on IOA automata
(EXPAND "inductive?") %and on LTS's
(PROP)
("1"
(INDUCT "a")
("1" (INST 1 "n!1 + 1") (GRIND))
("2" (INST 1 "n!1 + 2") (GRIND))
("3" (INST 1 "n!1 + 5") (GRIND))))
("2" (GRIND))
("4" (GRIND))

(SKOSIMP*)
(EXPAND "CoffeeMachine_automaton")
(EXPAND "transitions?")
(PROP)
(REPLACE -3 :HIDE? T)
(SKOSIMP*)
(REPLACE -1 :HIDE? T)
(EXPAND "action?"
(EXPAND "internal?"
(EXPAND "output?"
(EXPAND "input?"

(PROP)
("1" (INST 1 "n!1 + 1") (GRIND)) %providing the right instantiations
("2" (INST 1 "n!1 + 2") (GRIND))
("3" (INST 1 "n!1 + 5") (GRIND)))

(SKOSIMP*)
(EXPAND "CoffeeMachine_automaton")
(EXPAND "transitions?")
(PROP)
(REPLACE -5 :HIDE? T) %we derive that received'=received-50
(SKOSIMP*)
(REPLACE -1 :HIDE? T)
(INST 1 "n!1 -10"
("1" (GRIND))) %therefore we instantiate n-10, which
("2" (GRIND))))))))

/*we derive the two cases
*/BS (predicate holds on initial states):
%trivially proven
*/IS (predicate is preserved):
%from the "action?" predicate we derive
%that n is 5, 10 or 25
*/we get three cases which we prove by
%providing the right instantiations
*/assume a = TempHigh, trivially proven
*/assume a = TempLow, trivially proven
*/assume a = CupOfCoffee
%we derive that received'=received-50
%therefore we instantiate n-10, which
%generates an extra TCC: n-10>=0, which
%is trivially proven.

■  fig. 31: proof of coffee machine automaton invariant
```