Translation of a fragment of AsmL specification language to Higher Order Logic

Rostislav Yavorsky
yavorsky@microsoft.com

April 2003

Technical Report
MSR-TR-2003-22
The Abstract State Machine Language (AsmL) is an executable specification language developed at Microsoft Research. It is based on the notion of Abstract State Machine (ASM, formerly known as evolving algebra). According to the main ASM thesis, any algorithm at any level of abstraction can be faithfully simulated by an appropriate ASM. This specification method has been used for the description of the semantics of programming languages, communication protocols, distributed algorithms etc. [1, 2, 3].

AsmL is a powerful industrial strength language; it is a full member of the .NET family of languages. It is accompanied by tools for model based test generation and runtime verification [1]. On the other hand, tools for static analysis of the AsmL specifications by means of theorem provers have not been developed yet.

Our main goal is to define a fragment of AsmL that can be formalized in a theorem prover without much overhead, yet is rich enough to deal with reasonable properties of models. The algorithm for the AsmL to HOL translation described in this paper is implemented in a tool, which is intended to be part of integrated verification environment. This work is based on earlier research on verification of ASM like models in theorem provers (see [4, 5, 6]).

In section 1 we describe our technique using the example of a railroad crossing controller. In section 2, a general computation model is described that provides the theoretical background for the translation of a fragment of AsmL into the language of higher order logic. Then, in section 3, the corresponding fragment of AsmL and normal form of specification are outlined. In section 4, we describe the translation of this fragment of AsmL into the language of HOL theorem prover (HOL 4 Kananaskis [7]). Section 5 provides some useful HOL tactics for the verification of specifications.

1 Introductory example

Consider the following simple version of the railroad crossing controller studied in [8, 9]. An informal description of the problem is as follows.

We assume that the railroad crossing has only one track. The track has two sensors, one at some distance of the crossing in order to detect the incoming arrival of a train, and another one just after the crossing to detect that the train is leaving. An automatic controller receives the signals from the sensors and, on the basis of these signals, decides to send signal close or open to the gate. It takes at least time $D_{\text{min}}$ for a
train to reach the crossing after the sensor has detected it is incoming. And it takes at most $D_{open}$ for the gate to be really opened.

Below, $WaitTime$ denotes the difference $D_{min} - D_{open}$. The idea of the algorithm should be clear from the specification (Fig. 1): once $TrackStatus$ is changed to $Coming$, the algorithm computes the value of $Deadline$, the time when one should start closing the gate, and issues the $Close$ signal when the deadline is reached. After train has left the track, i.e. $TrackStatus$ becomes $Empty$, the $Open$ direction is issued.

We are going to verify the following safety properties of the control algorithm:

1. $GreenLight()$ invariant: once crossing is open and $TrackStatus$ is $Free$, it will remain open until $TrackStatus$ changes, i.e. gate closing could not be initiated by the controller.

2. A similar property of $RedLight()$: once $TrackStatus$ is $Coming$ and crossing is closed, it will remain closed while $TrackStatus$ does not change, i.e. gate opening could not be initiated by the controller.

The semantics of the AsmL model is as follows. Type declarations together with definitions of functions describe the background superuniverse of the system [11]. In this case it is a heterogeneous algebra with universes corresponding to types $Status$, $Signal$, $Infinite$, $Boolean$ and $Float$. We also add to the algebra one more universe $State$ corresponding to the tuple of the state variables $CurrentTime$, $TrackStatus$, $Deadline$ and $Direction$ correspondingly, that is:

$$State = Float \times Status \times (Float \text{ or } Infinite) \times Signal).$$

The signature has the following symbols: $=, +, and, or, \lt, less, SafeToOpen, RedLight, GreenLight$ and $is Infinite$ with the corresponding interpretation. Functions $SafeToOpen, RedLight$, and $GreenLight$ are state dependent, so they have one implicit parameter of type $State$.

The variables $CurrentTime$, $TrackStatus$, $Deadline$, and $Direction$ determine the state space of the system. It is convenient to think here that any state is characterized by the value of one variable of type $State$.

The body of the $Control$ procedure defines the transition relation. Given a value of the parameter $WaitTime$ it computes the subsequent state of the system.

Before translation to HOL the procedure is transformed to a normal form with explicit definitions of updates for the state variables (Fig. 2).
enum Status
    Coming
    Empty

enum Signal
    Open
    Close

enum Infinite
    Infinity

function less(t1 as Float, t2 as Float or Infinite) as Boolean
    return (t2 is Infinite) or else (t1 < t2)

function SafeToOpen() as Boolean
    return (TrackStatus = Empty) or less(CurrentTime, Deadline)

function GreenLight() as Boolean //invariant to be verified
    return (TrackStatus = Empty) and (Direction = Open)

function RedLight() as Boolean //invariant to be verified
    return (TrackStatus = Coming) and (Direction = Close)

var CurrentTime as Float
var TrackStatus as Status
var Deadline as Float or Infinite
var Direction as Signal

procedure Control(WaitTime as Float)
    if Direction = Open and not SafeToOpen() then
        Direction := Close
    elseif Direction = Close and SafeToOpen() then
        Direction := Open
    if TrackStatus = Coming and Deadline is Infinite
        Deadline := CurrentTime + WaitTime
    if TrackStatus = Empty and not (Deadline is Infinite)
        Deadline := Infinity

Figure 1: AsmL specification of (one track) railroad crossing controller
procedure Control(WaitTime as Float)
  Direction := if Direction = Open and not SafeToOpen() then
    Close
  elseif Direction = Close and SafeToOpen() then
    Open
  else
    Direction
  Deadline := if TrackStatus = Coming and Deadline is Infinite then
    CurrentTime + WaitTime
  elseif TrackStatus = Empty and not (Deadline is Infinite)
    Infinity
  else
    Deadline

Figure 2: The control procedure in the normal form

Formalization of these semantics in HOL results in the corresponding RailroadTheory (fig. 3).
The safety properties are formalized by the following proof goals:

\begin{align*}
!S_0: \text{RailroadState}. \forall t. \text{GreenLight } S_0 & \Rightarrow \text{GreenLight} (\text{Control } t \ S_0)) \\
!S_0: \text{RailroadState}. \forall t. \text{RedLight } S_0 & \Rightarrow \text{RedLight} (\text{Control } t \ S_0))
\end{align*}

that correspond to the following formulas:

\begin{align*}
\forall S_0 & \in \text{RailroadState}\forall t(\text{GreenLight}(S_0)) \Rightarrow \text{GreenLight}(\text{Control}(t, S_0)) \\
\forall S_0 & \in \text{RailroadState}\forall t(\text{RedLight}(S_0)) \Rightarrow \text{RedLight}(\text{Control}(t, S_0))
\end{align*}

Application of the standard simplification technique (definition expansion, case splitting and simplification) resolves the first goal. In the second case HOL terminates with two subgoals that correspond to the case when Deadline is set to some finite value. Further analysis shows that RedLight defined as above is not an invariant of the procedure: even if gate is closed and TrackStatus is Coming the controller issues the Open signal if the Deadline is not reached.

2 The background model of computation

Each specification written in AsmL describes an initialized transition system, namely, the set of all admissible states, the subset of initial states and the transition relation. Currently, we aim at the verification of one step (safety) properties of the system, so the initial values of variables are ignored.
open HolKernel Parse boolLib bossLib realTheory;
val _ = new_theory "Railroad";

val _ = Hol_datatype ' Status = Coming | empty ';
val _ = Hol_datatype ' Signal = Open | Close ';
val _ = Hol_datatype ' Infinite = Infinity ';

val _ = Hol_datatype ' RailroadState = <|
  CurrentTime : real;
  TrackStatus : Status;
  Deadline : real + Infinite;
  Direction : Signal |>'

val _ = Define ' less = \t1: real. \t2: real + Infinite.
  if (ISR (t2)) then T else t1 < (OUTL t2) ';

val _ = Define ' SafeToOpen = \S0: RailroadState.
  (S0.TrackStatus = empty) \ (less S0.CurrentTime S0.Deadline) ';

val _ = Define ' GreenLight = \S0: RailroadState.
  (S0.TrackStatus = empty) \ (S0.Direction = Open) ';

val _ = Define ' RedLight = \S0: RailroadState.
  (S0.TrackStatus = Coming) \ (S0.Direction = Close) ';

val _ = Define ' Control = \WaitTime: real. \S0: RailroadState.
  S0 with <| Direction :=
    if (S0.Direction = Open) \ ~(SafeToOpen S0) then 
      Close
    else if (S0.Direction = Close) \ (SafeToOpen S0) then 
      Open
    else
      S0.Direction;
  Deadline :=
    if (S0.TrackStatus = Coming) \ (ISR S0.Deadline) then
      INL (S0.CurrentTime + WaitTime)
    else if (S0.TrackStatus = empty) \ ~(ISR S0.Deadline) then
      INR Infinity
    else
      S0.Deadline |>'

val _ = export_theory();

Figure 3: HOL theory corresponding to the specification of railroad crossing controller
2.1 State background

The state background [11] of the system is a heterogeneous algebra. AsmL is strictly typed, so different data types correspond to disjoint universes $U_1, \ldots, U_k$. Union types correspond to a finite unions of the universes. The signature $\{f_1, \ldots, f_l\}$ of the algebra reflects operations on the data types. Each function has a fixed domain and range. The interpretations of both the signature and the universes are fixed during computation.

2.2 State space

Let $v_1, \ldots, v_m$ be all the state variables of the specification, with types $T_1, \ldots, T_m$ correspondingly. Then each potential state of the system is characterized by a vector of values of these variables. It is convenient here to assume that the algebra has a distinguished universe $U_{\text{state}}$ isomorphic to the cartesian product of these types. Concordantly, the signature should include all the projections and field update functions. Then any state corresponds to some value of this type, and transitions are just functions on this type.

To model initialized systems one should also specify here the subset of $U_{\text{state}}$ corresponding to initial states.

2.3 Transitions

Transition function of the system is represented by a term $t$ which is a combination of functions from the signature. It is convenient to consider the following kinds of transitions:

- pure transitions: type of $t$ is $U_{\text{state}} \rightarrow U_{\text{state}}$;
- parametrized transitions: type of $t$ is $(U_{\text{param}} \times U_{\text{state}}) \rightarrow U_{\text{state}}$, where $U_{\text{param}}$ is the parameter type;
- parametrized transitions with output: the corresponding type is $(U_{\text{param}} \times U_{\text{state}}) \rightarrow (U_{\text{state}} \times U_{\text{out}})$.

The last one is isomorphic to

$$(U_{\text{param}} \times U_{\text{state}} \rightarrow U_{\text{state}}) \times (U_{\text{param}} \times U_{\text{state}} \rightarrow U_{\text{out}}),$$

so one may consider output as a separate function $t_{\text{out}}$ associated with the transition.

In general, a system may have several transition functions (actions).
2.4 Computation

A computation is a sequence of values of the type $U_{\text{state}}$ such that each subsequent element is the result of application of a transition function (with some particular choice of parameters) to the previous one. The computation is initialized if the first element of the sequence belongs to the corresponding initial subset of $U_{\text{state}}$.

Computations that depend on input and nondeterministic computations are modeled via the corresponding choice of parameters of transitions.

2.5 One step properties

Given a precondition $\varphi$ and postcondition $\psi$, in order to verify that if $\varphi$ holds in a current state then $\psi$ must hold in a subsequent state under transition $t$, one has to prove that

$$\forall s \in U_{\text{state}}(\varphi(s) \Rightarrow \psi(t(s))).$$

In the most general case, to verify the following nondeterministic transition with output

\begin{verbatim}
choose p where $\chi(p)$
t(p)
return $t_{\text{out}}(p)$
\end{verbatim}

one proves that

$$\forall p \in U_{\text{param}}\forall s \in U_{\text{state}}(\chi(p) \land \varphi(s) \Rightarrow \psi(t(p, s)) \land \xi(t_{\text{out}}(p, s))),$$

where $\chi$ is a restriction on the choice of parameters, $\varphi$ and $\psi$ are the precondition and the postcondition correspondingly, $\xi$ is the condition on the output value.

To formalize this in HOL we need to translate the type declarations, definitions of functions, procedures, and the conditions.

3 Normal form of AsmL specification

We assume that a specification to be translated consists of the following five parts:

- type definitions;
• definitions of functions of the model;
• declarations of state variables;
• procedures (programs of one step transitions);
• constraints (invariants to be verified).

The purpose of these parts is the following.

Type definitions. Starting from the predefined AsmL basic types Null, Boolean, Integer, Float, String one can define new types utilizing the type constructors: Enum, Set of T, Seq of T, Map of T to S, structure, T or S, T?, where S and T run over types. Type definitions together with definitions of functions provide the description of the background universe of the model.

Functions correspond to specific operations of the modeled system. These could serve for both: abbreviations of frequently used terms and specifications of arbitrarily complex partially recursive functions. Jointly with type definitions this part describes the background of the system.

Declaration of state variables. Given the description of the background universe, any state of the computation is characterized by values of the state variables. A transition (action) of the system corresponds to an update of these variables. So, this part completes the definition of the state space of the model. All the variables are annotated with a type.

Procedures. Each procedure describes a one-step transition of the system. Procedures may depend on parameters. An important restriction is the following: all procedures should be deterministic (full AsmL allows non-determinism).

Constraints. From a theoretical point of view, constraints are just Boolean valued functions of the state. For the verification tool they could be utilized for formulating the proof goals.

The reduction of any AsmL specification to this normal form is a separate issue; many of the related problems have already been solved before, e.g. static checking of consistency and combining of parallel updates, partial updates, etc. (see [6, 12, 13]).

4 A more detailed description of the translation

This section describes the current implementation of the translation of the fragment of AsmL into the language of HOL.

8
4.1 Type definitions

The following built-in types are supported:

- **Boolean** — translated to `bool`;
- **Integer** — translated to `int` (`integerTheory` is loaded);
- **Float** — translated to `real` (`realTheory` is loaded);
- **String** — translated to `string` (`stringTheory` is loaded);
- **Null** — translated to its explicit definition as `Enum`.

And the following type families:

- **Set of T** — translated to `T set` (`pred_setTheory` is loaded);
- **Seq of T** — translated to `T list`;
- **Map of T to S** — translated to `T → S`;
- **Structure** — translated using `Hol_datatype`, e.g.
  
  \[
  \text{Structure F} \quad \text{corresponds to} \quad \text{Hol_datatype } 'F = < \mid x : T; \mid y : S \mid >';
  \]

- **Enum** — translated using `Hol_datatype`, e.g.
  
  \[
  \text{Enum Color} \quad \text{corresponds to} \quad \text{Hol_datatype } 'Color = \text{Red} \mid \text{Yellow} \mid \text{Green}';
  \]

- **T or S** — translated to `T+S` or `S+T` (see note below).

4.1.1 Note on translation of disjunctive types.

One can easily see that for all the types and type families listed above except the disjunctive one, AsmL and HOL have the same abstract syntax, so the translation is rather straightforward. For the disjunctive types including the option type `T?`, that is `(T or Null)`, it is not the case. In AsmL, type `(T or S)` corresponds to type union, so this operation on types is commutative,
associative and idempotent (i.e. \( T \lor T \) is the same as \( T \)). However, in HOL, the type operation \( T+S \) denotes disjoint sum, so it bears neither of these properties, in particular it is order dependent. Since the translator is implemented on the basis of the AsmL compiler, it is convenient to use the notion of core type. The following is taken from [10].

Core types are information computed by a type checker and annotated at the abstract syntax once the assembly stage is checked. Core type represents type information stripped off any semantic information like constraints, and resolved all type abbreviations. Equality on core type values is thus actual type equality.

Disjunctive types correspond to the following Core Type case [10].

\[
\text{structure OrCoreType extends CoreType}
\]
\[
\quad \text{opers as Set of CoreType}
\]

To translate disjunctive types to HOL we fix some order on core types (e.g. lexicographic order on the type names) and convert the corresponding set of disjuncts to a sequence with respect to this order. Thus each (unordered) disjunction of types in AsmL is mapped to sum of the types with a fixed order.

Accordingly, when translating expressions of a disjunctive type one should make type casting explicit, due to HOL rules. To define the corresponding type conversion from a simple type to a type union and vice versa, we use this fixed order of types to compute the place of a given type in the sequence, and then apply the appropriate combination of HOL functions \text{INL}, \text{INR} for the up casting, or \text{OUTL}, \text{OUTR} for the down casting. This could also be generalized to the type casting between two disjunctive types.

4.1.2 Type dependencies.

According to HOL rules, every value used in a definition should be defined previous to its use. So, type definitions in the AsmL specification should be placed in the corresponding order with respect to their dependencies.

4.2 Static functions

We use the term “static“ to emphasize the following restriction: a function should not update any state variable.

Again, the order of definitions should respect the dependency relation — each function used in the definition should be defined previous to its use.
Another general restriction: the translation currently does not support submachines, so the body of a function must be a pure expression: a combination (possibly self-referential) of static functions that computes a value to be returned.

Library functions are translated to the corresponding HOL definitions. This is quite simple due to expressive power of HOL and available HOL libraries. In the worst case one has to create manually the corresponding HOL definition. For example, function update \( f(a) := b \) is an abbreviation for \( f := \text{Update}(a, b, f) \) corresponds to the following HOL definition:

\[
\text{Define } \text{Update} a b f = \lambda x. (\text{if } x = a \text{ then } b \text{ else } (f x))\;
\]

that is

\[
\text{Update}(a, b, f)(x) = \begin{cases} 
  b, & \text{when } x = a; \\
  f(x), & \text{otherwise.}
\end{cases}
\]

4.3 State variables

Given all the state variables of the system, we define **ModelState** as a structure with fields corresponding to the variables. Initial values are ignored.

4.3.1 Note on HOL performance.

One can easily find out during an HOL interactive session that the definition of a structure with many fields takes a surprisingly amount of time. The reason is that during the definition process, HOL automatically creates and stores many basic yet useful properties of the structure and the field update functions.

4.4 Procedures: actions of the system

Contrary to static functions, procedures describe updates of the state variables (in other words — the system actions, or transitions). A procedure may depend on parameters yet it should not return values. The standard requirement holds: the order of definitions should respect the dependency relation.

According to the computation model given above, the body of a procedure should have only updates of the state variables. For convenience, our implementation of translation supports procedures with a little bit more complex, yet “simple” body. The definition of what is simple is given below:

- an update of a state variable \( v := \text{exp} \) is simple;
• a procedure call with particular instances of parameters is simple (the
procedure must be defined before);
• if Cond then Stm1 else Stm2 is simple, provided Stm1 and Stm2
are simple and Cond is a pure boolean expression (i.e. has no side
effects);
• once Stm is simple, Stm followed by a state variable update is again
simple, so one is allowed to have sequence of variable updates or a
procedure call followed by such a sequence.

A procedure with parameters (p1 as T1, p2 as T2, ...) is translated into HOL as function of the type

\[ T1 \rightarrow (T2 \rightarrow (\ldots \rightarrow (\text{ModelState} \rightarrow \text{ModelState})\ldots)) \]

that is given values of parameters and a state (i.e. values of the state
variables); it returns the subsequent state (i.e. next values of the state
variables).

4.5 Constraints — properties to check

These are just Boolean expressions on the state variables. They are trans-
lated to properties of a state:

\[ \text{PropertyN: ModelState} \rightarrow \text{bool}, \]

where N is the corresponding constraint number.

Typically, the properties are expected to be invariants of the system
transitions. So, given a property and a transition the proof goal looks like:

\[ \forall S_0: \text{ModelState}.(\text{PropertyN } S_0 \Rightarrow \text{PropertyN(Transition } S_0)) \]

that is

\[ \forall S_0 \in \text{ModelState}(\text{PropertyN}(S_0) \Rightarrow \text{PropertyN(Transition}(S_0))). \]

5 Verification hints

In the following examples we assume for simplicity that the **ModelState**
structure has two fields x and y.
5.1 Expand definitions

This is rather trivial yet important step. Since the procedures and the state conditions are expected to be non-recursive, one could substitute the definitions for their names. For this reason, during the translation we collect the names of properties, actions and static functions in the HOL variables PropList, ActionList, and FuncList correspondingly. Then the definition expansion (with subsequent beta-reduction) is implemented by the following tactic:

\[
\begin{align*}
\text{PURE\_ONCE\_REWRITE\_TAC PropList} & \\
\text{THEN PURE\_REWRITE\_TAC ActionList} & \\
\text{THEN BETA\_TAC.}
\end{align*}
\]

Constraints in AsmL are not embedded, so it is enough to execute the rewriting once only. We use the PURE tactics to disable using the basic tautologies as rewrite theorems here just to speed up the performance.

5.2 Decompose the state structure

It is convenient from the theoretical point of view to have all the state variables integrated into one structure, yet for proving purposes one should make the goal as simple as possible. So, the first step of the simplification is to get rid of the state structure.

This turns out to be a rather easy task, since the ModelState structure is not included in the definition of any other data type. The other useful thing is that the goal always starts with the universal quantifier over the state variable S0:

\[
!S0: \text{ModelState. (\ldots S0 \ldots)}
\]

According to the translated fragment, the only way the variable S0 may occur in a proof goal after the definition expansion is either a field reference, e.g. S0.x, or an implicit field reference, e.g.

\[
(\text{if R then \ldots else S0}).x \quad \text{or} \quad (\text{S0 with y:=2}).x
\]

since the only operations on theModelState type are if then else, field update, and field reference.

To deal with it, we replace S0 with the precise constructor (ModelState x y) and then repeat the simplification rewritings:

\[
(\text{Cases\_on \textquoteleft S0\textquoteright})
\]

\[
\text{THEN REPEAT (CHANGED\_TAC}
\]

13
Here `ModelState_commute_with_if` denotes the commutativity theorem for all the field updates:

\[(\text{if Guard then } S0 \text{ else } S1).x = (\text{if Guard then } S0.x \text{ else } S1.x)\]

(it is created by the translator). The other three rewriting rules are created by the HOL system during the `ModelState` structure definition. They correspond to the following kind of simple properties:

\[
\begin{align*}
  & (\text{ModelState } x \ y).x = x & \text{accessors} \\
  & (S0 \text{ with } x:=a).y = S0.y & \text{accupds} \\
  & ((\text{ModelState } x \ y) \text{ with } x:=a) = (\text{ModelState } a \ y) & \text{updates}
\end{align*}
\]

The `trivial_if` denotes the equation `(if G then A else A) = A`.

### 5.3 Split cases, simplify and apply decision procedures

This part is surely the most important one, yet at this point almost nothing specific for the ASM approach is left. For automatic verification purposes one may use the `(RW_TAC list_ss FuncList)` tactic, where `FuncList` keeps all the names of (recursive) functions of model.

### 5.4 Analyze subgoals

HOL may fail to prove a property for two different reasons. First, the prover is not complete, so the property is true, yet hard to validate. On the other hand, rather often it means that the property does not hold. In both cases the remaining subgoals help to figure out the corresponding state of the modeled system.

For example, in the railroad crossing example, an attempt to prove the stability of `RedLight` terminates with the following subgoal:

\[(\text{ISR } s) \lor (r < \text{OUTL } s) \vdash (S' = \text{Coming}) \lor (S0' = \text{Close})\]

where the antecedent corresponds to the cases where `Deadline` is `Infinite` or `CurrentTime < Deadline`. Further analysis shows that these are exactly the cases when the property fails.
Acknowledgements

Author is thankful to Yuri Gurevich and Margus Veanes for the productive discussion on the subject. Great thanks to Mike Barnett and Vladimir N. Krupski for many helpful comments on the text.

References


7th International Joint Conference CAAP/FASE”, Springer LNCS 1214, 202-212


