The ASLAN Formal Specification Language

Brent Auernheimer
Richard A. Kemmerer †

Department of Computer Science
University of California
Santa Barbara, California 93106

January 12, 1998

† This research has been supported in part by the National Science Foundation under Grant No. ECS81-06688.

The ASLAN Formal Specification Language

Brent Auernheimer
Richard A. Kemmerer †

Department of Computer Science
University of California
Santa Barbara, California 93106

1. Introduction

This paper discusses the ASLAN verification system and is divided into two main parts: an overview of the ASLAN system and a discussion of "procedural" and "nonprocedural" semantics and the approach adopted for the ASLAN verification system.

2. An Overview of ASLAN

The ASLAN specification language is built on first order predicate calculus with equality and employs the *state machine* approach to specification. The system being specified is thought of as being in various *states*, depending on the values of the *state variables.* Changes in state variables take place only via well defined *transitions.*

Predicate calculus is used to make assertions about desired properties that must hold at every state and between two consecutive states. Critical requirements that must be met in every state are state *invariants*. To prove that a specification satisfies some invariant assertion, ASLAN generates candidate lemmas needed to construct an *inductive proof* of the correctness of the specification with respect to the invariant assertion. These lemmas are known as *correctness conjectures.* ASLAN leaves the proof of these conjectures to the user or other theorem prover.

An ASLAN *specification* is a sequence of *levels*. Each level is an abstract data type view of the system being specified. The first ("top level") view is a very abstract model of what constitutes the system (types, constants, variables), what the system does (i.e., state transitions), and the critical requirements the system must meet. Lower levels are increasingly more detailed. The lowest level corresponds fairly closely to high level code. ASLAN generates correctness conjectures whose proof ensure that lower levels correctly refine upper levels.

The ASLAN language processor has been implemented to parse statements of levels and critical requirements, and produces both the conjectures needed to prove a specification is correct with respect to the critical requirements, and those necessary to show correct refinement of levels. The ASLAN language and use of the language processor are detailed in the ASLAN User's Manual [AK 85].

† This research has been supported in part by the National Science Foundation under Grant No. ECS81-06688.

2.1. The Finite State Machine Model

As previously noted, changes in state variables take place only by the application of transitions. In particular, given a state variable X, and an applicable transition T, ASLAN uses X´ to denote the value of X before the application of transition T, and X to denote the resulting value of X.

Consider a system consisting solely of a clock. The system may be characterized by the single state variable "Current_Time" and a transition "tick" that asserts "Current_Time" increases by one unit:

$$\text{Current\_Time} = \text{Current\_Time}' + 1$$

This specifies a transition to a new state in which the value of the variable Current_Time is one unit greater than its value in the immediately preceding state.

2.2. An Overview of Correctness Conjectures

A reasonable goal is to show that the system defined by the state variables and transitions always satisfies some critical requirements. These critical requirements must be met in every state that the system may reach. In ASLAN terminology these requirements are state *invariants.*

As noted above, to prove that a specification satisfies some invariant assertion, ASLAN generates lemmas needed to construct an *inductive* proof of the correctness of the specification with respect to the invariant assertion. It is left to the user, possibly with the aid of a theorem prover, to actually establish the validity of these *correctness conjectures.*

As the basis of the induction it must be shown that the system starts only in states that satisfy the state invariant. Assuming that some *initial* assertion defines possible beginning states, it must be proved that:

$$\text{initial\_assertion} \rightarrow \text{invariant\_assertion}$$

where "$\rightarrow$" stands for logical implication.

The inductive step involves showing for every transition T if the system was in a state satisfying the invariant assertion before the application of T, the resulting state also satisfies the invariant assertion:

$$\text{invariant\_assertion}' \ \& \ T \rightarrow \text{invariant\_assertion}$$

Where invariant_assertion´ means applying the "old value" operator ´ to every variable in the expression, "&" is logical conjunction, and T represents the effect of applying transition T.

As an example, suppose a critical requirement of some system is that "the number of items in the warehouse is never less than zero". Specifically, it must be shown that given that the system starts with a nonnegative inventory, it is not possible that the application of a transition results in a state in which the

inventory is less than zero. In ASLAN the initial conditions may be expressed as:

INITIAL

Inventory ≥ 0

and the invariant assertion as:

INVARIANT

Inventory ≥ 0

The correctness conjecture corresponding to the basis of the induction is then:

$$\text{Inventory} \geq 0 \rightarrow \text{Inventory} \geq 0$$

which is trivially true.

Suppose that one of the system transitions is a "consumer" transition that merely removes one item from the inventory:

$$\text{Inventory} = \text{Inventory}' - 1$$

This expression is called an EXIT assertion. EXIT assertions express what changes the application of a transition makes on system variables. For this example, the correctness conjecture corresponding to the inductive step is:

$$\text{Inventory}' \geq 0 \ \&\ (\text{Inventory} = \text{Inventory}' - 1) \rightarrow \text{Inventory} \geq 0$$

Notice that this conjecture is not always true, which means some part of the specification is incorrect with respect to the critical requirements. The problem arises because nothing prevents the application of the consumer transition when Inventory = 0. ENTRY assertions can be used to express the conditions necessary for a transition to be applied. An ENTRY assertion for the consumer transition is:

$$\text{Inventory} > 0$$

The use of ENTRY assertions makes the inductive step:

invariant_assertion′ & entry_assertion′ & exit_assertion

→

invariant_assertion

which for this example becomes:

$$\text{Inventory}' \geq 0 \ \&\ \text{Inventory}' > 0 \ \&\ \text{Inventory} = \text{Inventory}' - 1$$

→

Inventory ≥ 0

Thus, an ASLAN transition consists of an *entry assertion* and an *exit assertion*.

2.3. A Simple Sample Specification

The following specification models a portion of a security system. Each user has a password represented by the User_Password constant; for simplicity, users cannot change passwords. The Password_Ok constant is the abstraction of the actual system component that compares a user's real password with an attempted password. Since the Password type is unspecified, it could be that the underlying implementation stores only encrypted passwords (represented by User_Password), and that Password_Ok is implemented by encrypting the attempted password and comparing the result to the (encrypted) user's password.

A common security measure is to keep a log of all attempted logins. This is represented by the state variable Log of type Log_Set. An element of Log_Set is a set of Log_Entries, where each Log_Entry is a a 4-tuple, having Who, When, Tried, and Result fields. The idea is that every time a user attempts to login, their name, the current time, the password they attempted, and whether the system allowed them to log in, is recorded.

An attempt to login also causes the state variables Login_Allowed and Sound_Alarm to be set. The simple BOOLEAN variable Login_Allowed may represent setting up a process for the user in the actual implementation; at this abstract level, however, the mechanics of giving the user a process is immaterial. Similarly, Sound_Alarm could be implemented as anything from doing nothing, to sending a "beep" character to the terminal, to barring the windows and bolting the doors. All these options are left open at the top level.

The critical requirement the system must meet is that it is never the case that someone has logged on using a password that isn't theirs. This requirement is expressed as an INVARIANT stating that if a Log_Entry le is an element of the Log, then le[Result] is what it should be: true if and only if le[Tried] is compatible with le[Who]'s password.

```
SPECIFICATION Authentication LEVEL Top_Level
TYPE
   Time IS INTEGER,
   User,
   Password,
   Log_Entry IS STRUCTURE OF
     (Who: User, When: Time, Tried: Password, Result: BOOLEAN),
   Log_Set IS SET OF Log_Entry

CONSTANT
   User_Password(User): Password,
   Password_Ok(Password, Password): BOOLEAN

VARIABLE
   Log: Log_Set,
   Current_Time: Time,
   Login_Allowed, Sound_Alarm: BOOLEAN

INVARIANT
   FORALL le: Log_Entry
    (le ISIN Log → Password_Ok(User_Password(le[Who]), le[Tried]) = le[Result])

TRANSITION Login (u : User, p: Password) EXIT
   /* record the attempt... */
   Log = Log´ UNION {le: Log_Entry (le[Who] = u & le[When] = Current_Time
        & le[Tried] = p &le[Result] = Password_Ok(User_Password(u), p))}
   /* this took some time... */
 & Current_Time > Current_Time´
   /* the rest of login's duties... */
 & IF Password_Ok(User_Password(u), p)
    THEN Login_Allowed & ¬Sound_Alarm
    ELSE Sound_Alarm & ¬Login_Allowed
   FI

TRANSITION Tick EXIT
  Current_Time = Current_Time´ + 1

END Top_Level END Authentication
```

Note that at this level of specification the method of comparing passwords is not defined, but it is assumed that some method, which will be specified at a lower level, exists.

3. Procedural and Nonprocedural Semantics

In nontrivial systems, a particular transition rarely affects every state variable. That is, not all state variables are mentioned unprimed in the EXIT assertions of every transition. Consider the transition "tick", which affects only the variable "Current_Time". Since the ENTRY assertion is not present it is assumed TRUE. It is reasonable that ASLAN generates the following correctness conjecture for the transition tick:

FORALL le: Log_Entry
(le ISIN Log´ → Password_Ok(User_Password(le[Who]), le[Tried]) = le[Result])
& (Current_Time = Current_Time´ + 1)
→
FORALL le: Log_Entry
(le ISIN Log → Password_Ok(User_Password(le[Who]), le[Tried]) = le[Result])

This conjecture is not provable since no information about the "new value" of Log is available.

From the theorem prover's point of view, conjectures are first order predicate calculus statements built with relational operators such as > and =, logical operators, predicates (Password_Ok), constants, such as 1 and TRUE, and (new value) variables. The ASLAN language processor is the interface between the logical world and the more procedural world of the specifier. ASLAN bridges the gap by adding logical constructs to the specifier's assertions.

Since logical operators are used to explicitly state relationships about state variables, the specifier must be careful to use the *nonprocedural* semantics of first order logic when interpreting and writing ASLAN specifications. Imposing *procedural*, programming language semantics on logical statements is disastrous. In short, it is important to remember that nothing is known about variables which are not explicitly stated, or whose new value is stated for some cases, but not for other cases.

The most straightforward technique for adding constructs, and easiest to implement, is for the language processor to conjoin to each EXIT assertion "NOCHANGE(v)" statements stating that for each state variable v *not* mentioned in the EXIT assertion, the value of v does not change. That is, since it is tedious for the specifier to conjoin to every EXIT assertion an expression stating that each variable not otherwise mentioned does not change, ASLAN does this automatically during correctness conjecture generation. Simply stated, if a (unprimed) variable is *not* mentioned in an EXIT assertion of a transition, its value has not changed. The EXIT assertion for the tick transition would appear in conjectures as

(Current_Time = Current_Time´ + 1) & Log = Log´
& Login_Allowed = Login_Allowed´ & Sound_Alarm = Sound_Alarm´

The correctness conjecture may now be proved.

Now consider a slightly simplier system closely related to the above sample system. This system is even more abstract than the sample system; Passwords and Logs do not appear, only a BOOLEAN constant User_Ok(User). User_Ok determines which persons may or may not log in; it may be implemented at a lower level by the Passwords described above. The following Login transition says nothing about the value of the login_allowed variable if User_Ok(p) is false. That is, an implementation of the transition that *always* sets Login_Allowed to true would satisfy the specification for Login.

```
TRANSITION Login(p: User)
EXIT
 /* if the user is ok, let him log in */
 User_Ok(p) → Login_Allowed
```

Although Login_Allowed was mentioned in the EXIT assertion, its value is not explicitly defined in all cases. This undesirable loophole could be closed by adding to the EXIT assertion a statement about the value of login_allowed when User_Ok is FALSE:

```
TRANSITION Login(p: User)
EXIT
 /* if the user is ok, let him log in */
 User_Ok(p) → Login_Allowed
 /* if the user ISN'T ok, make sure Login_Allowed is FALSE. */
 & ¬User_Ok(p) → ¬Login_Allowed
```

Since system designers are not accustomed to stating what happens when *nothing* is to happen, ASLAN provides *procedural* operators which work the way computer scientists often think logical operators should work. The operators are procedural in the sense that any state variables not explicitly mentioned are assumed *not to have changed.* This parallels programming language semantics in that (assuming no side effects) only variables explicitly mentioned (on the left side of an assignment statement) may change; unmentioned variables or those on the right side of an assignment do not. For example, the PASCAL assignment statement

```
i := j + 1;
```

where i and j are INTEGER variables, states that only the value of i changes. The programmer can be sure that no other variable has changed.

A design goal of ASLAN was to provide both 1) logical operators to allow purists to write specifications untainted by procedural semantics, and 2) procedural constructs for those wishing to use an approach closer to that used with programming languages. The purpose of procedural constructs is to ease the burden of explicitly specifying "nochanges".

There are three instances when ASLAN "automatically" generates nochange statements. First, variables whose new values were not referred to in an EXIT assertion are assumed to have not changed (as discussed above).

Second, if the new value of a variable x is mentioned in the THEN (ELSE) portion of a conditional statement, but *not* in the ELSE (THEN) portion of the same statement, it is assumed that the variable does not change in the ELSE (THEN) portion. For example,

IF User_Ok(p)
 THEN Login_Allowed
 ELSE Sound_Alarm & ¬Login_Allowed FI

will appear in ASLAN conjectures as

IF User_Ok(p)
 THEN Login_Allowed **& Sound_Alarm = Sound_Alarm´**
 ELSE Sound_Alarm & ¬Login_Allowed FI

The "implied nochanges" the ASLAN processor adds are in boldface.

Third, if the new value of a variable x is referenced in one half of an ALTernative (procedural disjunction) and not in the other half, ASLAN essentially conjoins NOCHANGE(x) to the half in which x is *not* mentioned.[1] For example,

 User_Ok(p) & Login_Allowed
ALT
 ¬User_Ok(p) & Sound_Alarm

appears in correctness conjectures as

 User_Ok(p) & Login_Allowed **& Sound_Alarm = Sound_Alarm´**
ALT
 ¬User_Ok(p) & Sound_Alarm **&Login_Allowed = Login_Allowed´**

The implementation of implied nochange statement generation for the ASLAN verification system is outlined in the next section and is described in detail in [Aue 85].

3.1. When Does A State Variable Change?

When trying to decide what NOCHANGE constructs to add problems arise with variables that take arguments. Specifically, how do you know when an *instance* of the parameterized variable x, such as x(4) does not change? There are two basic ways to look at this problem. The first is to view state variables as *functions* from a domain *D* to a range *R*. To change the value of an instance of the state variable requires the binding of a new function to the state variable; this means any instance whose value is not explicitly

[1] The only difference between the ASLAN ALT operator and logical disjunction (|) is that state variables mentioned on one side of an ALT and not on the other are assumed to have remained unchanged.

stated may take any value in *R*. This is a nonprocedural view, and is relatively easy to implement: since "one reference to an instance of a state variable is as good as another", it is easy to keep track of what state variable's function bindings have changed.

This method has serious drawbacks. For instance, suppose the specifier has decided that User_Password really should be a variable[2] and that a Change_Password transition is needed. A first attempt at writing the transition results in

```
TRANSITION Change_Password
     (u: User, old_password, new_password: Password)
EXIT
 IF Password_Ok(User_Password´(u), old_password)
  THEN User_Password(u) = new_password
 FI
```

which would appear in conjectures as

```
TRANSITION Change_Password
     (u: User, old_password, new_password: Password)
EXIT
 IF Password_Ok(User_Password´(u), old_password)
  THEN User_Password(u) = new_password
  ELSE FORALL u1: User (User_Password(u1) = User_Password´(u1))
 FI
```

Unfortunately, if the condition is satisfied, nothing is known about other user's passwords. That is, the value of User_Password(u1) where u1≠u is unknown. The correct way to write the EXIT assertion is

[2] This means that the INVARIANT given in the sample specification is probably *not* what the specifier now wants.

```
TRANSITION Change_Password
    (u: User, old_password, new_password: Password)
EXIT
 IF Password_Ok(User_Password´(u), old_password)
  THEN FORALL u1: User
      (IF u1 = u THEN User_Password(u) = new_password
       ELSE User_Password(u1) = User_Password´(u1)
      FI)
 FI
```

This explicitly states that the passwords for all other users remain the same.[3]

The second point of view is to consider each instance of a state variable individually. This convention considers each instance of the domain to be bound to an element of the range. Changing the binding of an instance associated with an element in D has no effect on the other bindings. This is clearly a procedural approach, and corresponds to the way arrays are viewed in a high level programming language. Unfortunately, it is much harder for the specification processor to determine what changes and what doesn't in this scheme. Under the second assumption, the EXIT assertion for the Change_Password transition becomes

```
TRANSITION Change_Password(u: User, old_password, new_password: Password)
EXIT
 IF Password_Ok(User_Password´(u), old_password)
  THEN User_Password(u) = new_password
 FI
```

3.2. ASLAN Implementation of Nochanges

At this time ASLAN implements the first convention. In order to keep track of what "new value" variables are referenced the ASLAN processor needs to make two passes over each well-formed formula (wff). The first pass builds *reference tables*; the second pass uses the reference tables to compute the implied nochanges. For example, given the ALTernative statement

```
a ALT b
```

[3] ASLAN provides the BECOMES operator to change the value of an instance of a state variable. BECOMES is essentially a macro whose expansion results in a quantification that looks like the one above. Using BECOMES, the assertion would be:

```
IF Password_Ok(User_Password´(u, old_password)
 THEN User_Password(u) BECOMES new_password FI
```

where a and b are boolean variables, the ASLAN processor's first pass produces reference tables containing the number of times each new value variable appears on each side of the ALT.

ASLAN's second pass traverses the parse tree. Upon encountering the ALT node, the processor compares the tables associated with the right and left sides and augments each side with statements stating that unreferenced variables do not change.

Earlier versions of the ASLAN processor had problems with context. They failed because they built up variable reference tables and computed nochanges as the parse tree was being traversed during pass two; pass one had nothing to do with the building of reference tables or insertion of implied nochanges. For example, upon encountering an ALT node during the second pass, the processor would first traverse the left side and build a reference table, then similarly traverse and build the table for the right side. The tables were then compared, and nochanges were inserted based on this information. This approach did not take the context in which the ALT appeared into account.

Currently the ASLAN processor does more work during the first pass. In particular, reference tables are now formed during that pass. A reference table is built for both sides of an ALT statement, and for the IF, THEN, and ELSE parts of a conditional, as well as for the entire well-formed formula. This latter table supplies the context missing in the earlier versions of the processor.

This approach still has two major drawbacks:

1) Since "one reference to an instance of a state variable is as good as another", the implied nochanges generated by the language processor are "universal". This can cause consistency problems; for example

```
FORALL u1: User (IF u = u1 THEN User_Password(u1) = new_password FI)
```

appears in conjectures as

```
FORALL u1: User
  (IF u = u1
    THEN User_Password(u1) = new_password
    ELSE FORALL u2: User (User_Password(u2) = User_Password´(u2))
FI)
```

This is inconsistent since it states that User_Password(u1) changes when u1=u and that it does not change when u1≠u. Thus, the specification can not be satisfied and is equivalent to FALSE (Use of the BECOMES operator could avoid this problem.).

2) The use of a universal quantification to change an instance of a state variable is awkward.

For these reasons, a strategy for implementing the second approach has been developed. In the same way the first method keeps a reference count of how many times a state variable is referenced, the new strategy involves keeping track of *arguments* to state variables. For example, the following is a fairly inscrutable pathological example:

```
VARIABLE a(BOOLEAN, INTEGER): BOOLEAN

TRANSITION t
  EXIT
    a(a(TRUE, 3), 42)
```

The new strategy builds the following table of arguments:

variable	arguments	
a	a(TRUE, 3), 42	TRUE, 3

The entries of the table are the arguments of "a" that are specified as possibly changing.

From the table a FORALL statement representing the implied nochanges would be constructed:

FORALL b: BOOLEAN, i: INTEGER
(¬(b = a(TRUE, 3) & i = 42) | (b & i = 3)) → a(b, i) = a´(b, i))

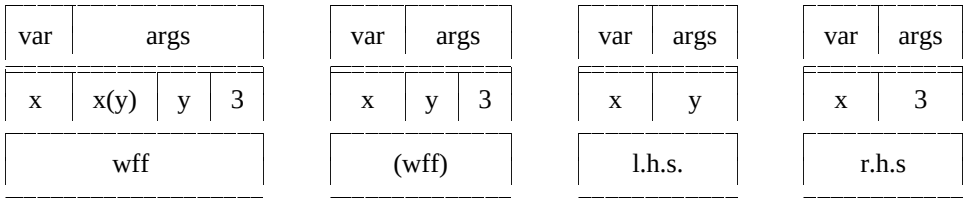
Thus the implied nochanges in this case are any instances of "a" that have not been explicitly referenced. The EXIT assertion would appear in conjectures as:

```
  a(a(TRUE, 3), 42)
& FORALL b: BOOLEAN, i: INTEGER
(¬(b = a(TRUE, 3) & i = 42) | (b & i = 3)) → a(b, i) = a´(b, i))
```

One of the hardest problems solved in the present implementation of the first convention is taking context into account. Fortunately the new argument tables can use the same basic strategy ASLAN presently uses. For example,

x(x(y)) = 4 & (x(y) = 2 ALT x(3) = 9)

would cause the language processor to produce the argument tables:



Using the same techniques for computing augmented tables and comparing them with reference tables from the corresponding other side results in:

x(x(y)) = 4 &
((x(y) = 2 & FORALL i:INTEGER (¬(i = x(y) | i = y) → x(i) = x´(i))
| (x(3) = 9 & FORALL i:INTEGER (¬(i = x(y) | i = 3) → x(i) = x´(i)))

The following is an example of a fairly complicated assertion using conditional statements:

IF predicate1
THEN x(1) = 1
ELSE x(2) = 2
FI
&
IF predicate2
THEN x(3) = 3
ELSE x(4) = 4
FI

The new method results in:

IF predicate1
THEN x(1) = 1 **& FORALL i: INTEGER (¬(i = 1 | i = 3 | i = 4) → x(i) = x´(i))**
ELSE x(2) = 2 **& FORALL i: INTEGER (¬(i = 2 | i = 3 | i = 4) → x(i) = x´(i))**
FI
&
IF predicate2
THEN x(3) = 3 **& FORALL i: INTEGER (¬(i = 1 | i = 2 | i = 3) → x(i) = x´(i))**
ELSE x(4) = 4 **& FORALL i: INTEGER (¬(i = 1 | i = 2 | i = 4) → x(i) = x´(i))**
FI

Notice that the implied nochange statments take the context of the other conditional statements into account.

Quantification is the stickiest part of implementing the new convention. To understand the problems with quantification and implied nochanges it is useful to review the problems and "solutions" encountered with the first convention and its implementation. In present ASLAN a universal quantification is commonly used for changing the value of a *single instance* of a state variable. The following statement is typical:

```
FORALL u: User (IF u = Bob THEN User_Password(u) = new_password FI)
```

which appears in conjectures as:

```
FORALL u: User
 (IF u = Bob THEN User_Password(u) = new_password
  ELSE FORALL u1: User (User_Password(u1) = User_Password´(u1))
  FI)
```

As already noted, the implied nochange is blatantly wrong.

To relieve the temptation of using the conditional-within-the-forall construct for changing a single instance of a variable, the BECOMES statement was introduced -- but it too can easily be misused. The conclusion is (not surprisingly) that universal implied nochanges are bad practice, and at the very least present ASLAN should not allow the specifier to use a (procedural) conditional statement within a quantification.

Things are somewhat better for the proposed implementation. Since changes in instances are merely *stated* in a straightforward way, specifications should contain fewer quantifications. Changing the passwords of both Bob and Elvira consists of only

```
 User_Password(Bob) = new_password_1
& User_Password(Elvira) = new_password_2
```

A specifier could, however, legitimately need to use a conditional statement within a quantification. Consider the problem of purging user accounts which have not been used. The following definition determines whether an account has been used:

```
DEFINE used(u: User): BOOLEAN ==
   EXISTS le: Log_Entry (le ISIN Log & le[Result] & le[Who] = u)
```

For example, a statement to set the password to a special "no access" constant for those users who have not

successfully logged on is:

```
CONSTANT
  No_Access: Password
 . . .
TRANSITION Purge
EXIT
  FORALL u: User
    (IF ¬used(u) THEN User_Password(u) = No_Access FI)
```

If the implied no changes were added directly to the conditional, as in the example above, the resulting assertion would be inconsistent:

```
TRANSITION Purge
EXIT
  FORALL u: User
    (IF ¬used(u)
      THEN User_Password(u) = No_Access FI
& FORALL u1: User (¬(u1 = u) → User_Password(u1) = User_Password´(u1))
```

Clearly, conditional statements within the scope of a quantification must be treated differently than "unbound" conditionals. There are two approaches to this problems. The first results from considering

FORALL i: INTEGER (IF predicate(i) THEN x(i) FI)

as a conjunction of an infinite number of individual conditional statements. That is,

```
   FORALL i: INTEGER (IF predicate(i) THEN x(i) FI)
≡
   IF predicate(1) THEN x(1) FI
 & IF predicate(2) THEN x(2) FI
 . . .
 & IF predicate(n) THEN x(3) FI
 . . .
```

Applying the implied nochange strategy on the individual conditional statements results in:

IF predicate(1) THEN x(1) **& FORALL i: INTEGER (¬(i = 1|i = 2|...) → x(i) = x´(i))**
ELSE FORALL i: INTEGER (¬(i = 2| i = 3 | ...) → x(i) = x´(i))
FI
& IF predicate(2) THEN x(2) **& FORALL i: INTEGER (¬(i = 1|i = 2|...) → x(i) = x´(i))**
ELSE FORALL i: INTEGER (¬(i = 1| i = 3 | ...) → x(i) = x´(i))
FI
. . .
& IF predicate(n) THEN x(3)**& FORALL i: INTEGER (¬(i = 1|i = 2|...) → x(i) = x´(i))**
ELSE FORALL i: INTEGER (¬(i = 1| ... |i = n - 1|i = n + 1| ...) → x(i) = x´(i))
FI
. . .

The antecedent of the implied no change expression conjoined to the THEN side of each conditional says that i is not equal to 1, 2, ..., which is false; the entire quantification is true, and may be dropped. Intuitively, the implied no change expression conjoined to the ELSE side of each conditional says that *only* x(n) does not change. Thus each conditional is equivalent to

IF predicate(n) THEN x(n) ELSE x(n) = x´(n) FI

and the original quantification is equivalent to

FORALL i: INTEGER (IF predicate(n) THEN x(i) ELSE x(i) = x´(i))

This leads to the implied nochange rule: "For each state variable instance, x(i), appearing in the THEN (ELSE) portion of a conditional statement occurring inside a quantification, conjoin to the ELSE (THEN) statement x(i) = x´(i)."

This method fails when two quantified expressions that mention the same state variables are conjoined. For example, suppose associated with each user is a security class:

TYPE
Security_Class IS (Confidential, Secret, Top_Secret)

CONSTANT
User_Security(User): Security_Class

A transition to purge all Confidential and Secret users may be needed. This transition is to set the passwords of all users not having Top_Secret clearance to No_Access. A specifier could write a transform which appears in conjectures as:

```
TRANSITION Top_Secret_Users_Only
EXIT
 FORALL u: Users (IF User_Security(u) = Confidential
            THEN User_Password(u) = No_Access
            ELSE User_Password(u) = User_Password´(u)
           FI)
& FORALL u: Users (IF User_Security(u) = Secret
            THEN User_Password(u) = No_Access
            ELSE User_Password(u) = User_Password´(u)
           FI)
```

The implied nochanges are **wrong**. There are several possible answers to this problem:

1) Force specifiers to write specifications in a "normal form" with all quantification pushed to the front. The language processor would then have flagged the above transition as being syntactically incorrect.

2) Write a smart language processor that converts all EXIT assertions to normal form before computing implied no changes. This has the disadvantages that 1) it's hard, and 2) any errors the language processor flags in the normalized EXIT assertion may be hard to track down to the corresponding original expression.

3) In addition to producing correctness conjectures, produce *consistency* conjectures as well. This has the disadvantage of at least doubling the number of conjectures to prove. An argument in favor of consistency conjecture generation is that it should be done anyway -- there is nothing stopping a specifier from writing FALSE as an EXIT assertion, and from the FALSE EXIT assertion being able to prove any INVARIANT. A possible consistency conjecture is

$$\neg(\text{exit\_assertion} \rightarrow \text{FALSE})$$

4) Whoever wrote the above transition obviously didn't know what they were doing. Training better specifiers leads to better specifications. Caveat emptor.

The second technique for computing implied nochanges for quantifications adds statements *outside* the scope of the original quantification. For example,

```
DEFINE Secret_User(u: User): BOOLEAN == User_Security(u) = Secret
DEFINE Confidential_User(u: User): BOOLEAN
    == User_Security(u) = Confidential

. . .
EXIT
 FORALL u: User (IF Secret_User(u) THEN User_Password(u) = No_Access FI)
& FORALL u: User (IF Confidential_User(u) THEN User_Password(u) = No_Access FI)
```

results in the table:

variable	arguments	
a	Secret_User(*)	Confidential_User(*)

Where the asterisk (*) stands for a universally quantified variable. The expression to be conjoined to the original is then:

FORALL u: User
(¬(Secret_User(u) | Confidential_User(u)) → User_Password(u) = User_Password(u)´)

Also note that since

EXISTS ... (wff)

is equivalent to

¬ FORALL ... (¬wff))

nochange processing for existential and universal quantification is similar. For example, an expression stating that at least one user's password becomes No_Access could be written as:

EXISTS u: User (User_Password(u) = No_Access)

Since the User_Password(u)'s that becomes No_Access could be *any* (or every) instance, the resulting argument table is:

variable	arguments
User_Password	*

It is possible that *all* instances of variable User_Password now have the value No_Access; the * as one of the argument table entries means that no implied nochanges should be generated.

4. Summary

An overview of the ASLAN specification language and the behavior of the language processor is described. The distinction between procedural semantics of programming languages and nonprocedural semantics of logic-based specification languages is discussed. Two solutions to the problem of "implied nochanges", the first of which ASLAN implements, are outlined and evaluated.

5. References

[AK 85] Auernheimer, Brent, and Richard A. Kemmerer, ASLAN User's Manual, TRCS84-10, Department of Computer Science, University of California, Santa Barbara, March 1985.

[Aue 85] Auernheimer, Brent, Implementation of Implied NoChanges in ASLAN, TRCS85-07, Department of Computer Science, University of California, Santa Barbara, March 1985