ASLAN User’s Manual

Brent Auernheimer
Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106

revised April 1992
1. Introduction

This document serves as a guide to the ASLAN specification language and use of the ASLAN language processor. Section 1 introduces the strategies underlying the ASLAN approach to system specification. A small, syntactically correct specification is presented to serve as motivation for the detailed exposition to follow.

The second and largest section of this document describes the syntax and semantics of the ASLAN language. Following sections describe the use of the ASLAN language processor and further explain the ASLAN approach towards correctness and consistency conjectures. A non-trivial, syntactically correct specification example and associated correctness conjectures are presented in the final section.

An appendix describes the current state of the ASLAN system, and known bugs.

For clarity, throughout this document keywords appear entirely in upper case.

1.1. The Finite State Machine Model

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 can be 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. In particular, given a state variable X, and an applicable transition T, ASLAN uses X' (pronounced X prime) 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. We can characterize this system with the single state variable "time". The only valid transition "tick" might assert that time increases by one unit:

\[ \text{time} = \text{time}' + 1 \]

This says that tick causes a transition to a new state in which the value of the variable time is one unit greater than its value in the immediately preceding state.

1.2. An Overview of Correctness Conjectures

How does ASLAN guarantee that a specification is "correct"? A reasonable goal would be 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.

To prove that a specification satisfies some invariant assertion, ASLAN generates proof obligations needed to construct an inductive proof of the correctness of the specification with respect to the invariant assertion. These proof obligations are known as correctness conjectures. It is the user’s responsibility to establish the validity of the correctness conjectures, possibly with the aid of a theorem prover.

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:

\[ \text{initial}\_\text{assertion} \rightarrow \text{invariant}\_\text{assertion} \]

where \(\rightarrow\) stands for logical implication.

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

\[ \text{invariant}\_\text{assertion}' \& \ T \rightarrow \text{invariant}\_\text{assertion} \]

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 for 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:

\[
\text{INITIAL } \text{inventory} \geq 0
\]

and the invariant assertion as:

\[
\text{INVARIANT } \text{inventory} \geq 0
\]

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 leads us to believe that 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. A reasonable ENTRY assertion for the consumer transition is:

\[
\text{ENTRY } \text{inventory} > 0
\]

The use of ENTRY assertions makes the inductive step:

\[
\text{invariant\_assertion}' \& \text{entry\_assertion}' \& \text{exit\_assertion} \\
\rightarrow \\
\text{invariant\_assertion}
\]

which for this example becomes:

\[
\text{inventory}' \geq 0 \& \text{inventory}' > 0 \& \text{inventory} = \text{inventory}' - 1 \\
\rightarrow \\
\text{inventory} \geq 0
\]

Some critical requirements cannot be expressed in terms of a state invariant alone. In particular, requirements relating the values of state variables before and after a transition to a new state serve as constraints governing state transitions. The following critical requirement might be added to the previous example: "The inventory may not be reduced by more than half at any one time". This is expressed in ASLAN as:

\[
\text{CONSTRAINT } \text{inventory} \geq \text{inventory}' / 2
\]
In general, the use of constraints makes the inductive step become:

\[ \text{invariant} \rightarrow \text{entry} \land \text{exit} \rightarrow \text{invariant} \land \text{constraint} \]

Thus, the correctness conjecture for this example is:

\[ \text{inventory} \geq 0 \land \text{inventory} > 0 \land \text{inventory} = \text{inventory} - 1 \rightarrow \text{inventory} \geq 0 \land \text{inventory} \geq \text{inventory} / 2 \]

This conjecture, however, is not true when inventory = 1!

1.3. A Simple Sample Specification

The following elaboration of the inventory example above is a syntactically correct ASLAN specification:

```
SPECIFICATION Producer_Consumer
LEVEL Top_Level

CONSTANT number_wanted : INTEGER

VARIABLE inventory : INTEGER

INITIAL inventory >= 0

INVARIANT inventory >= 0

CONSTRAINT inventory >= inventory / 2

TRANSITION produce
    EXIT inventory = inventory + 1

TRANSITION consume
    EXIT /* make sure constraint is not violated */
    IF (inventory / 2) < number_wanted
    THEN
        /* don’t consume! */
        inventory = inventory
    ELSE
        inventory = inventory - number_wanted
    FI

END Top_Level

END Producer_Consumer
```

2. The ASLAN Specification Language

The following description of the ASLAN language will make extensive use of Backus-Naur Form (BNF) to explain acceptable syntax. As a warm up, Figure 1 contains the syntax for letters, digits, and numbers.
2.1. Well-Formed Formulas

Well-formed formulas are the building blocks of ASLAN specifications. These expressions are the basic assertions that define critical system requirements or describe what happens when the functions of the system being specified are applied. As Figure 2 illustrates, wf_formulas are composed by applying unary and binary operators to terms (Section 2.1.2).

![Figure 1](Image)  
letters, digits, and numbers

2.1.1. Primitive Relations and Operations

ASLAN's rich set of operations and relations will be discussed in order of increasing precedence.

![Figure 3](Image)  
binary_operators, unary_operators

The logical operators take BOOLEAN arguments and return a BOOLEAN result. The negation operator is right-associative; all other logical operators associate to the left. Any binary logical operator may be immediately preceded by '¬'. In general, this yields the logical negation of the operation, that is, (A ¬¬ B) is equivalent to ¬(A -> B).
The relational operators are of the same precedence and do not associate. Like the logical operators, relational operators may be preceded by ‘Ä’. In the following tables T and ET stand for any consistently substituted type and enumerated type (including INTEGER).

<table>
<thead>
<tr>
<th>PREC</th>
<th>OP</th>
<th>LEFT</th>
<th>TYPE OF</th>
<th>RIGHT</th>
<th>RESULT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;-&gt;</td>
<td>BOOLEAN</td>
<td></td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>if and only if</td>
</tr>
<tr>
<td>2</td>
<td>-&gt;</td>
<td>BOOLEAN</td>
<td></td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>implies</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>BOOLEAN</td>
<td></td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>or</td>
</tr>
<tr>
<td>4</td>
<td>&amp;</td>
<td>BOOLEAN</td>
<td></td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>and</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>none</td>
<td></td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>logical negation</td>
</tr>
</tbody>
</table>

The lone membership relation ISIN may also be preceded by ‘Ä’:

<table>
<thead>
<tr>
<th>PREC</th>
<th>OP</th>
<th>LEFT</th>
<th>TYPE OF</th>
<th>RIGHT</th>
<th>RESULT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>ISIN</td>
<td>T</td>
<td>set of T</td>
<td>BOOLEAN</td>
<td></td>
<td>set membership</td>
</tr>
</tbody>
</table>

Like logical operators and relations, the set relations may be prefixed by ‘Ä’. All set relations have precedence 8, take "set of T" as arguments, return a BOOLEAN result and do not associate.

<table>
<thead>
<tr>
<th>PREC</th>
<th>OPERATOR</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>CONTAINED_IN</td>
<td>is subset of</td>
</tr>
<tr>
<td>8</td>
<td>SUBSET</td>
<td>is proper subset of</td>
</tr>
<tr>
<td>8</td>
<td>SUPERSET</td>
<td>is proper superset of</td>
</tr>
<tr>
<td>8</td>
<td>CONTAINS</td>
<td>is superset of</td>
</tr>
</tbody>
</table>

ASLAN provides the common numeric operations.

<table>
<thead>
<tr>
<th>PREC</th>
<th>OP</th>
<th>LEFT</th>
<th>TYPE OF</th>
<th>RIGHT</th>
<th>RESULT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>+</td>
<td>INTEGER</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>plus</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>INTEGER</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>minus</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>INTEGER</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>times</td>
</tr>
<tr>
<td>10</td>
<td>/</td>
<td>INTEGER</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>division</td>
</tr>
<tr>
<td>10</td>
<td>MOD</td>
<td>INTEGER</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>modulo</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>none</td>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>unary minus</td>
</tr>
</tbody>
</table>
In addition to the familiar union and intersection operators, ASLAN provides a set difference operator \( \text{SET DIFF} \) and a symmetric difference operator \( \text{SYM DIFF} \). The set difference of two sets \( A \) and \( B \) is a set of those elements of \( A \) that do not appear in \( B \). The symmetric difference of \( A \) and \( B \) contains all those elements in either \( A \) or \( B \), but not in both. A summary of the \textit{set operators} follows:

<table>
<thead>
<tr>
<th>PREC</th>
<th>OP</th>
<th>LEFT</th>
<th>TYPE OF</th>
<th>RIGHT</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>UNION</td>
<td>set of T</td>
<td>set of T</td>
<td>set of T</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>INTERSECT</td>
<td>set of T</td>
<td>set of T</td>
<td>set of T</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SET DIFF</td>
<td>set of T</td>
<td>set of T</td>
<td>set of T</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SYM DIFF</td>
<td>set of T</td>
<td>set of T</td>
<td>set of T</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{UNION}, \text{INTERSECT}, \text{and SYM DIFF} \) may also be used as unary operators applied to a set of sets of \( T \).

<table>
<thead>
<tr>
<th>PREC</th>
<th>OPERATOR</th>
<th>TYPE OF</th>
<th>RESULT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>UNION</td>
<td>set of set of T</td>
<td>set of T</td>
<td>collected union</td>
</tr>
<tr>
<td>14</td>
<td>INTERSECT</td>
<td>set of set of T</td>
<td>set of T</td>
<td>collected intersect</td>
</tr>
<tr>
<td>14</td>
<td>SYM DIFF</td>
<td>set of set of T</td>
<td>set of T</td>
<td>collected difference</td>
</tr>
</tbody>
</table>

Finally, there are the \textit{list operators} \( \text{CONCAT} \) and \( \text{LIST LEN} \). \( \text{CONCAT} \) takes two lists and returns a list equal to the concatenation of the two lists. \( \text{LIST LEN} \) returns an \texttt{INTEGER} equal to the number of elements in its argument.

<table>
<thead>
<tr>
<th>PREC</th>
<th>OP</th>
<th>TYPE OF</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CONCAT</td>
<td>list of T</td>
<td>list of T</td>
</tr>
<tr>
<td>16</td>
<td>LIST LEN</td>
<td>none</td>
<td>list of T</td>
</tr>
</tbody>
</table>

\textbf{2.1.2. Terms}

Terms include identifiers (possibly followed by the prime operator and/or arguments), numbers, descriptions of lists or sets, \texttt{IF-THEN-ELSE-FI} and \texttt{NOCHANGE} statements, parenthesized \texttt{wF_formulas}, and built-in identifiers.

\textbf{2.1.2.1. Identifiers}

ASLAN identifiers must start with a letter and can be any combination of letters, digits and underscore \( (\_ ) \) thereafter. The case of letters within an identifier is not significant. Thus, the identifiers inventory and Inventory are considered identical.

The \texttt{BOOLEAN} identifiers \texttt{TRUE} and \texttt{FALSE} are predeclared in ASLAN, as are the \texttt{EMPTY} set, and the \texttt{NIL} list. Because constant and argument declarations appear frequently in the following sections, "individual declarations" are defined in Figure 6 for future use.

The id following the colon must be the name of some previously declared type.
2.1.2.2. Lists

The keyword LISTDEF precedes any list description. A parenthesized list of elements follows LISTDEF:

```
LISTDEF (1, 2, two, three, 4)
```

where two and three are constants of type INTEGER.

2.1.2.3. Component Specifiers

Component specifiers are an indexing method allowing one to access any element of a list.

Assuming that the identifier "queue" has been declared as a list of "persons", an assertion that the first element of queue is not the person "Bob" is:
Component specifiers applied to lists must have either a number, constant or variable of type INTEGER between the square brackets.

Component specifiers may also be applied to identifiers of some STRUCTURE type. When used in this way an identifier must appear between the brackets. The identifier must be one of the "fields" of the STRUCTURE type. Further discussion of this use of component specifiers is delayed until the section dealing with STRUCTUREs (2.2.2.1).

2.1.2.4. Sets

Sets may be described by listing their elements between brackets:

\{1, 2, two, three, 4\}

where "two and "three" are INTEGER CONSTANTS.

Alternatively, an expression like "the set of all x’s between 1 and 100" may be stated as follows:

\{ SETDEF x : INTEGER (x >= 1 & x <= 100) \}

2.1.2.5. Quantification

ASLAN provides the universal quantifier FORALL, and the existential quantifiers EXISTS and UNIQUE. Each occurrence of the above keywords must be followed by a declaration of local bound variables and a parenthesized expression.

\(<\text{quantifier}> ::= "\text{FORALL}" | "\text{EXISTS}" | "\text{UNIQUE}"\)

\(<\text{wf_quantification}> ::= <\text{quantifier}> <\text{individual_declarations}> "\text{wf_formula}\"

\(<\text{set_description}> ::= "\{" ( <\text{setdef}> | <\text{list_elements}> ) "\}"\)

For example, a statement that every integer has a superior is:

\(\text{FORALL } x: \text{INTEGER} ( \text{EXISTS } y: \text{INTEGER} (y > x))\)

As another example, consider the statement that for every pair of integers there exists a unique integer equal to the sum of the first two:

\(\text{FORALL } x, y: \text{INTEGER} (\text{UNIQUE } z : \text{INTEGER} (z = x + y))\)
2.1.2.6. The "Procedural" Operations

2.1.2.6.1. Implied NoChanges

It is not hard to imagine a system defined by several state variables and having transitions which do not affect every variable. Consider embedding the transition "tick" (which affects only the variable "time") in a system with another integer state variable x.

TRANSITION tick
   EXIT time = time' + 1

A possible invariant assertion is:

INVARIANT
   (time >= 0) & (x >= 0)

Assuming the CONSTRAINT and ENTRY assertions are the boolean constant TRUE, it might seem reasonable that ASLAN generates the following correctness conjecture for the transition tick:

   (time' >= 0) & (x' >= 0) & (time = time' + 1)
   ->
   (time >= 0) & (x >= 0)

This conjecture is not provable since no information about the new value of x is available. Since it would becomes extremely 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. Therefore, tick’s EXIT assertion would appear in conjectures as

   (time = time' + 1) & x = x'

The correctness conjecture may now be proved.

The logical operators discussed in previous sections are used to explicitly state relationships about state variables; nothing is known about variables which are not explicitly stated. For example, if user_ok(person) is a BOOLEAN constant which is used to determine which persons may or may not log in, the following 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 set login_allowed to true would satisfy the specification for authenticate_users.

TRANSITION authenticate_users(p: person)
   ENTRY
      /* assume the user cannot login */
      login_allowed = FALSE
   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 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 authenticate_users(p: person)
ENTRY
  /* assume the user cannot login */
  login_allowed = FALSE
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 doesn’t change! */
  & "user_ok(p) -> login_allowed = login_allowed"

Since computer scientists are not used to stating what happens when nothing is to happen, ASLAN provides four procedural operators which work the way computer scientists tend to 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 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

\[ i := j + 1; \]

states that only the value of \( i \) may change. The programmer can be assured that no other variable has changed.

Similarly, computer scientists assume that conditional statements which are not satisfied have no effect on the state variables. For example, the above transition may be written using the procedural conditional statement as:

TRANSITION authenticate_users(p: person)
ENTRY
  /* assume the user cannot login */
  login_allowed = FALSE
EXIT
  /* if the user is ok, let him log in */
  IF user_ok(p) THEN login_allowed FI

The procedural conditional statement excludes the possibility of an implementation always setting login_allowed to TRUE.

The following four sections discuss the ALTernative, IF-THEN-ELSE-FI, BECOMES, and NOCHANGE statements. A fifth section details the "nochanges" implied by the four statements and EXIT assertions in general.

2.1.2.6.2. ALTernative Statements

The ALT operator separates alternative actions. It has priority lower than any other operator, and associates to the left. ALT differs from logical disjunction (\( \lor \)) in that state variables mentioned on one side of ALT and not on the other are assumed to have remained unchanged. Suppose that in addition to the previous requirements, the screen_users transition must sound an alarm if an unauthorized user attempts to log on. The EXIT assertion

\[ \text{user_ok(p) \& login_allowed} \]
\[ \text{ALT} \]
\[ \neg \text{user_ok(p) \& sound_alarm} \]

would appear in conjectures as
(user_ok(p) & login_allowed & sound_alarm = sound_alarm"
| (~user_ok(p) & sound_alarm & login_allowed = login_allowed"

Notice that the first conjunction states that the value of sound_alarm has not changed, and the second conjunction states that login_allowed does not change.

2.1.2.6.3. Conditional Statements

ASLAN adopts the Algol68 convention of matching each IF keyword with a FI. The "ELSE wf_formula" portion of the conditional is optional.

<conditional_term> ::= "IF" <wf_formula> "THEN" <wf_formula> {"ELSE" <wf_formula>} "FI"

Variables whose new values have been referred to in the THEN (ELSE) portion of the conditional but not in the ELSE (THEN) section are assumed to have not changed. The EXIT assertion for screen_users could be written as:

EXIT
  IF user_ok(p)
    THEN login_allowed
    ELSE sound_alarm
  FI

and would appear in conjectures as

IF user_ok(p)
  THEN login_allowed
    & sound_alarm = sound_alarm"
  ELSE sound_alarm
    & login_allowed = login_allowed"
  FI

As another example, suppose the variable "phone_number" associates "persons" with integers. If a person Bob’s phone number is to be changed to 9614321 an expression found in the body of a transition might be:

FORALL x : person {
  IF x = Bob
    THEN phone_number(x) = 9614321
    ELSE phone_number(x) = phone_number’(x)
  FI
}

or,
FORALL x : person ( 
    phone_number(x) = 
        IF x = Bob 
        THEN 9614321 
        ELSE phone_number'(x) 
        FI) 

It must be explicitly stated that if x is not Bob then x’s phone number is not changed. This is done by saying that the "new value" of phone_number(x) is equal to the "old value" of phone_number(x) for everyone except Bob.

2.1.2.6.4. The Becomes Operator

Using a universal quantification to specify the changing of a variable (such as phone_number) in exactly one case (phone_number(Bob)) can become tedious and is error prone. ASLAN provides the BECOMES statement as a shorthand for asserting that the value of a state variable is changed for some particular arguments, and remains unchanged for all others. BECOMES can be thought of as having priority higher than any operator or relation discussed in Section 2.1.1.

\[
\text{phone_number(Bob)} \ \text{BECOMES} \ 9614321 
\]

and will appear in correctness conjectures as:

\[
\text{FORALL _001 : person} \\
(\text{IF _001 = Bob} \\
\text{THEN phone_number(_001) = 9614321} \\
\text{ELSE phone_number(_001) = phone_number'( _001)} \\
\text{FI}) 
\]

Caution! Using the same variable, but different arguments, on the left side of a BECOMES statement will result in an inconsistent expression! For example,

\[
\text{phone_number(Bob)} \ \text{BECOMES} \ 9614321 \\
& \\
\text{phone_number(Bill) \ BECOMES} \ 5551212 
\]

results in
\[
\text{FORALL } _001 : \text{person} \\
\quad (\text{IF } _001 = \text{Bob} \\
\quad \quad \quad \text{THEN } \text{phone\_number}( _001) = 9614321 \\
\quad \quad \quad \text{ELSE } \text{phone\_number}( _001) = \text{phone\_number}\,'( _001) \\
\quad \quad \text{FI}) \\
\]
\[
\& \\
\text{FORALL } _001 : \text{person} \\
\quad (\text{IF } _001 = \text{Bill} \\
\quad \quad \quad \text{THEN } \text{phone\_number}( _001) = 5551212 \\
\quad \quad \quad \text{ELSE } \text{phone\_number}( _001) = \text{phone\_number}\,'( _001) \\
\quad \quad \text{FI}) \\
\]

which is a contradiction unless both \(\text{phone\_number\,'(Bob)} = 9614321\) and \(\text{phone\_number\,'(Bill)} = 5551212\)!

2.1.2.6.5. The NoChange Operation

As in previous examples, it is sometimes necessary to express the fact that certain state variables do not change value due to the application of a transition. ASLAN offers the NOCHANGE specification function as shorthand for stating the above. NOCHANGE may or may not take a list of variables as an argument.

\[
<\text{nochange}> ::= "\text{NOCHANGE}" \{ "(" <\text{id\_list}> ")" \}
\]

\textbf{Figure 13}

\textbf{nochange}

If an argument list is present ASLAN replaces the NOCHANGE statement with a conjunction of expressions asserting for every variable \(V\) in the argument list either:

1) \(V = V\,'\) if \(V\) has an empty domain (i.e., takes no arguments)

or,

2) a universal quantification stating that \(V(A_1,...,A_n) = V\,'(A_1,...,A_n)\) for all possible arguments \(A_1, ..., A_n\) of \(V\).

If NOCHANGE appears without a list of arguments, ASLAN assumes that \textit{no} state variables change value and repeats either step (1) or (2) for \textit{every} variable.

For these reasons you will never see NOCHANGE in any ASLAN generated conjecture. For example, the expression:

\[
\text{NOCHANGE}\left(\text{phone\_number}\right)
\]

is translated into:

\[
\text{FORALL } _001 : \text{person} \left( \text{phone\_number}( _001) = \text{phone\_number}\,'( _001) \right)
\]

where \(_001\) is an ASLAN generated identifier.

2.1.2.6.6. Implied NoChanges Revisited

There are three instances when ASLAN "automatically" generates NOCHANGE-like statements. First, variables whose new values were not referred to in an EXIT assertion are assumed to have not changed. Second, if the "new value" of a variable \(x\) is referenced in one half of an ALTernative statement and not in the other half, ASLAN essentially conjoins NOCHANGE\((x)\) to the half in which \(x\) is \textit{not} mentioned.
appears in correctness conjectures as

\[(\text{time} = 10 \& x = x) \mid (x = 1492 \& \text{time} = \text{time}')\]

Third, 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,

\[
\text{IF time'} = 10 \\
\quad \text{THEN time} = 11 \\
\quad \text{ELSE x} = 1958 \\
\text{FI}
\]

will turn up in conjectures as

\[
\text{IF time'} = 10 \\
\quad \text{THEN (time} = 11) \& (x = x') \\
\quad \text{ELSE (x} = 1958) \& (\text{time} = \text{time}') \\
\text{FI}
\]

When computing implied nochanges, ASLAN treats the appearance of a DEFINEd identifier in a \(\text{wf}_\text{formula}\) as a reference to each variable which appears unprimed in the body of the define.

Caution: it must be remembered that any implied NOCHANGEs ASLAN generates for variables that take arguments will be universal in nature. This is probably not what the specifier had in mind. It would have been wrong to write the previous transition to change Bob’s phone number as:

\[
\text{FORALL x : person (}
\quad \text{IF x} = \text{Bob}
\quad \quad \text{THEN phone_number}(x) = 9614321
\quad \text{FI)}
\]

since ASLAN will interpret this as:

\[
\text{FORALL x : person (}
\quad \text{IF x} = \text{Bob}
\quad \quad \text{THEN phone_number}(x) = 9614321
\quad \text{ELSE FORALL _001 : person (}
\quad \quad \quad \text{phone_number}(_001) = \text{phone_number}(_001)
\quad \quad \text{FI)}
\]

Unless \(\text{phone_number}(_001) = 9614321\), the above expression is equivalent to FALSE. Since the correctness conjecture corresponding to this transition will have this expression on the left of the implication, the conjecture will be vacuously true regardless of the invariant, constraint, or entry assertion. Such consistency issues are discussed further in Section 3.

The desired (consistent) effect can be obtained by:

\[
\text{phone_number}(\text{Bob}) \text{ BECOMES 9614321}
\]
2.2. ASLAN Specifications

2.2.1. Levels and Their Relationships

An ASLAN specification is a sequence of levels, bracketed by the keywords SPECIFICATION and END. SPECIFICATION must be followed with a (usually descriptive) identifier, and is matched with an END followed by the same (usually descriptive) identifier. The presence of the keyword INHIBIT immediately before the keyword LEVEL prevents ASLAN from generating correctness conjectures relating the level preceding the INHIBIT with the level following the INHIBIT. An INHIBITed top level prevents the generation of top level correctness conjectures. Correctness conjectures are discussed in detail in Section 3.

\[
\text{specification} ::= \text{"SPECIFICATION" } <\text{id}> \{\text{"INHIBIT"}\} <\text{tls}> <\text{lower\_levels}> \text{"END" } <\text{id}>
\]

\[
\text{tls} ::= \text{"LEVEL" } <\text{id}> <\text{top\_level\_elements}> \text{"END" } <\text{id}>
\]

\[
\text{lower\_levels} ::= <\text{lower\_level}> | <\text{lower\_levels}> <\text{lower\_level}>
\]

\[
\text{lower\_level} ::= \{\text{"INHIBIT"}\} \text{"LEVEL" } <\text{id}> \text{"REFINES" } <\text{id}> <\text{lower\_level\_elements}> \text{"END" } <\text{id}>
\]

\[
\text{lower\_level\_elements} ::= <\text{declarations}> <\text{requirements}> \{ <\text{transitions}> \}
\]

\[
\text{transitions} ::= <\text{transition}> | <\text{transitions}> <\text{transition}>
\]

Figure 14
specification, tls, lower\_level, lower\_level\_elements, and top\_level\_elements

The first level appearing in an ASLAN specification is the most abstract view of the system, and is colloquially called the "top level". Each level consists of a declaration, a requirements, and a transitions section. In addition, every level except the top level must REFINE an already existing level, and contain an implementation section. The IMPLEMENTATION section relates a lower level with the level it refines by showing the correspondence between types, variables, constants, and transitions at the higher level to types, variables, constants, and transitions at the lower level. ASLAN generates correctness conjectures to ensure that the lower level is a correct refinement of the upper level.

2.2.2. The Declaration Section

ASLAN follows a "declared before use" policy. For example, if a constant is used in the definition of a type, the constant must have been previously declared. This causes no problems since type, constant, variable, and definition declarations may be freely mixed as long as the above mentioned policy is adhered to.

\[
<\text{declarations}> ::= <\text{declaration\_part}> | <\text{declarations}> <\text{declaration\_part}>
\]

\[
<\text{declaration\_part}> ::= \text{"TYPE" } <\text{type\_decl\_list}>
\]

\[
| \text{"CONSTANT" } <\text{constant\_decl\_list}>
\]

\[
| \text{"VARIABLE" } <\text{variable\_decl\_list}>
\]

\[
| \text{"DEFINE" } <\text{define\_decl\_list}>
\]

Figure 15
declarations

2.2.2.1. Types

ASLAN is a strongly typed language. Types themselves, however, can be very general and may be left unspecified. The simplest type declaration is:

\[
\text{TYPE person}
\]
Person is said to be an "unspecified" type. The only relations available on elements of unspecified types are \(=\) and \(\neq\). We might also wish to declare:

\[
\text{TYPE staff SUBTYPE person}
\]

Staff is then an "unspecified subtype" of person. Elements of staff may appear anywhere an element of person may appear.

\[
\text{Figure 16 type_decl_list}
\]

ASLAN also supports "specified" types. The simplest specified type is an alias, such as:

\[
\text{TYPE index IS INTEGER}
\]

Enumerated types are declared by following the keyword IS by a parenthesized list of elements:

\[
\text{TYPE small_symbols IS (a, b, c, d)}
\]

Enumerated types must have at least two elements. Elements of enumerated types are considered constants of that type. The order implied by the position of each element in the parenthesized list allows inequality relations to be applied to elements of enumerated types. We might also declare:

\[
\text{TYPE other_symbols IS (a, c)}
\]

Other_symbols is an "enumerated subtype" of small_symbols. Elements appearing in the parenthesized list of a subtype must be in the same relative order as they appear in the declaration of the supertype. For example, 'a' precedes 'c' in the declaration of small_symbols, and therefore 'a' must precede 'c' in the declaration of other_symbols.

Types representing sets or lists of previously defined types are declared simply as:

\[
\text{TYPE}
\begin{align*}
group & \text{ IS SET OF person,} \\
queue & \text{ IS LIST OF person}
\end{align*}
\]

ASLAN provides "structure types" that resemble PASCAL records. A type associating a "customer" with a "balance" could be:
TYPE debtor IS STRUCTURE OF
    (customer : person,
     balance : INTEGER)

As stated in Section 2.1.2.3 a component specifier may be used to pick fields out of variables and constants of STRUCTURE types. For example, given that the identifier borrower has been declared as a variable of type debtor, a statement that borrower is a person named "Bob" and owes 100 dollars is:

    borrower[customer] = Bob & borrower[balance] = 100

The bracketed identifier must be the name of one of the fields of the STRUCTURE type.

Finally, the keyword TYPEDEF allows types to be defined using an expression or quantification. For example,

    TYPE pos_int IS TYPEDEF x : INTEGER (x > 0)

states that pos_int consists of the positive integers.

The types INTEGER and BOOLEAN are built-in primitive types and cannot be redeclared.

### 2.2.2.2. Constants

Constants are unchanging mappings from some (possibly empty) domain to some range. Each identifier present in the list following the constant identifier, and the identifier following the colon must be a previously declared type.

\[
\begin{align*}
<\text{constant\_decl\_list}> & ::= <\text{parm\_id\_list}> "," <\text{id}> \\
& \quad | <\text{constant\_decl\_list}> "," <\text{parm\_id\_list}> "," <\text{id}>

<\text{variable\_decl\_list}> & ::= <\text{parm\_id\_list}> "," <\text{id}> \\
& \quad | <\text{variable\_decl\_list}> "," <\text{parm\_id\_list}> "," <\text{id}>

<\text{parm\_id\_list}> & ::= <\text{id}> \{ <\text{parms}> \} \\
& \quad | <\text{parm\_id\_list}> <\text{id}> \{ <\text{parms}> \}

<\text{parms}> & ::= "(" <\text{id\_list}> ")"
\end{align*}
\]

Figure 17
constant
\_decl\_list, variable
\_decl\_list

The constant declaration

    CONSTANT big_int : INTEGER

declares big_int to be an integer constant, or more formally, a mapping from the empty domain to the range consisting of integers. As another example, the constant mapping "ancestor" from pairs of persons to the boolean values may be declared as:

    CONSTANT ancestor(person, person) : BOOLEAN

Since constants cannot be changed, it is an error to apply the prime operator to any constant.
2.2.2.3. Variables

Like constants, variables are mappings from domains to ranges. Variables may be changed by the application of a transition, and therefore may have the operator applied to them. It is the value of variables that differentiates one state from another.

2.2.2.4. Definitions

ASLAN definitions can be thought of as parameterized macros. Definitions differ from macros in two ways. First, the formal parameters appearing in a define declaration are treated as constants local to the formula following the double equals. This implies that actual parameters may not contain any "new value" variables. Variables, however, may appear in the body (formula) of the definition. Second, instead of substituting the body of the define in place of an appearance of the defined identifier, ASLAN interprets the identifier as if a parenthesized copy of the formula (with appropriate substitution of actual for formal parameters) has replaced the identifier.

Every definition must be declared as being of some type.

\[
\text{Figure 18}
\]

\text{define\_decl\_list}

As an example, if it was frequently necessary to check whether two persons had a common ancestor we could make the definition:

\[
\text{DEFINE related}(x, y : \text{person}) : \text{BOOLEAN} == \\
\text{EXISTS } z : \text{person} (\text{ancestor}(z, x) & \text{ancestor}(z, y))
\]

Identifiers corresponding to DEFINEs which have no primed variables present in the formula portion of the declaration may appear in other formulas suffixed with . Such a primed DEFINEd identifier is taken to mean that is applied to every variable appearing in the formula portion of the DEFINE declaration. The "Available" DEFINE appearing in the sample specification of Section 4 is used in this manner in several of the specification's transitions.

It is an error to apply the prime operator to DEFINEd identifiers having at least one primed variable in the formula section of its declaration. For example, given the following definition

\[
\text{DEFINE inc}_x : \text{BOOLEAN} == x = x + 1
\]

it would be illegal to write

\[
\ldots \\
inc_x'
\]

\[
\ldots
\]

in the EXIT section of a transition.

2.2.3. The Requirements Section

The requirements section contains information necessary to generate correctness conjectures. Any of the AXIOM, INITIAL, INVARIANT, and CONSTRAINT portions may be omitted. Missing assertions are assumed to be TRUE. Unlike items making up the declarations section, the above expressions, when they do appear, must be present in the order shown in Figure 19.
2.2.3.1. Axioms

The AXIOM is an expression used to facilitate the proving of correctness conjectures, and is one of the more esoteric features of ASLAN. As an example consider the following AXIOM concerning the ancestor constant declared above:

\[
\text{AXIOM}\ \forall x, y, z : \text{person} \ (\text{ancestor}(x, y) \land \text{ancestor}(y, z) \rightarrow \text{ancestor}(x, z))
\]

This axiom expresses the transitivity of the ancestor relation.

2.2.3.2. Initial Conditions

The INITIAL section defines the set of possible starting states of the system being specified. Typically, this expression asserts something about the value of every state variable at startup time.

2.2.3.3. Invariants and Constraints

As shown earlier, the INVARIANT expresses critical requirements of the system by making an assertion about relationships and values of state variables in any reachable state. INVARIANTs may not contain any primed variables.

The CONSTRAINT, on the other hand, makes an assertion about the values of state variables before and after the application of a transition. Thus, CONSTRAINTs must contain both primed and unprimed variables.

2.2.4. The Transitions Section

Transitions define the valid state changes that a system being specified can make.

\[
\text{transitions, entry_exits, except_exit_pairs}
\]

Transitions may take arguments and must have an EXIT statement. Formal parameters of a transition are considered local constants. The EXIT statement determines what changes the application of a transition has on the values of state variables. The optional ENTRY assertion expresses necessary conditions that must hold before a transition may be applied. An omitted ENTRY assertion is assumed to be TRUE. EXCEPT and EXIT pairs may be used to specify what is to happen under exceptional circumstances. For example, the "consumer" transition of Section 1.3 could be written as:
TRANSITION consume
   ENTRY inventory / 2 >= number_wanted
   EXIT inventory = inventory' - number_wanted

   EXCEPT inventory / 2 < number_wanted
   EXIT NOCHANGE(inventory)

2.2.5. The Implementation Section

The implementation section shows how types, constants, variables, and transitions appearing at an upper level are refined in an immediately lower level. A refinement statement relates a component of the upper level with an expression involving lower level components. In this section the subscript ‘u’ means the subscripted identifier is from the upper level while ‘l’ signifies that the identifier is of the lower (refining) level. Note that DEFINEs are not refined at the lower level.

\[
\langle \text{implementation_specs} \rangle ::= \langle \text{parmed_id} \rangle \text{ "==" } \langle \text{wf_formula} \rangle \nonumber \\
\quad \quad | \quad \langle \text{dotted_id} \rangle \text{ "==" } \langle \text{wf_formula} \rangle 
\]

\[
\langle \text{parmed_id} \rangle ::= \langle \text{id} \rangle \{ \langle \text{id_list} \rangle \}
\]

\[
\langle \text{dotted_id} \rangle ::= \langle \text{id} \rangle \text{ "." } \langle \text{number} \rangle
\]

\text{Figure 21}

implementation specs

Upper level types must be associated with lower level types. That is, a refinement statement about types must look like:

upper_type == lower_type

Such a statement implies the existence of an implementation function that maps upper level elements of upper_type to elements of lower_type. In functional notation:

\[
\text{Impl}_{\text{upper_type}}: \text{upper_type} \rightarrow \text{lower_type}
\]

Constants must be refined by a lower level wf_formula containing no references to variables. Variables may be refined by any lower level wf_formula. Upper level constants, variables and transitions which take arguments and appear in the left side of a refinement statement must be followed by a parenthesized list of dummy arguments. The type associated with each dummy argument is determined by its position in the upper level argument list. Dummy arguments may be referenced in the lower level expression following the double equals. When used in this way dummy arguments have types determined by the refinement of their upper level type. For example, if the following declaration appears in the upper level:

VARIABLE upper_var(upper_arg_type) : upper_type

and

VARIABLE lower_var(lower_arg_type) : lower_type

at the lower level, a few reasonable refinement statements are:

IMPLEMENTATION
   upper_type == lower_type,
   upper_arg_type == lower_arg_type,

   /* note use of dummy variable */
   upper_var(arg) == lower_var(arg)
The type of arg on the left side is upper_arg_type, while on the right side arg is of type lower_arg_type. The refinement statement should be interpreted as:

\[
\text{FORALL arg: upper_arg_type} \\
\quad (\text{Impl}_{\text{upper_type}}(\text{upper_var}(\text{arg})) = \text{lower_var}(\text{Impl}_{\text{upper_arg_type}}(\text{arg})))
\]

Consider the following example:

**SPECIFICATION Library**
**LEVEL top_level**

**TYPE**
- Book,
- Book_Set IS SET OF Book,
- Author,
- Title

**CONSTANT**
- Written_By(Book) : Author,
- Title_Of(Book) : Author

**VARIABLE**
- Library : Book_Set,
- Checked_Out(Book) : BOOLEAN

**END top_level**
**LEVEL second_level REFINES top_level**

**TYPE**
- Author IS (Shakespeare, Poe, Vonnegut),
- Title,
- Book IS STRUCTURE OF (written_by : Author, title_of: Title),
- Book_Set IS SET OF Book,
- User

**VARIABLE**
- Responsible_For(User) : Book_Set

**IMPLEMENTATION**
- Book == Book, /* A type refinement */
- Author == Author,
- Title == Title,
- Book_Set == Book_Set,
- Checked_Out == EXISTS u : User (b ISIN Responsible_For(u))
Transitions may be refined by any wf formula with the following restriction: the wf formula, if converted to disjunctive normal form, must have exactly one reference to a lower level transition in each conjunct. ASLAN provides a special notation for referring to ENTRY-EXIT and EXCEPT-EXIT pairs of a particular transition. Given a transition:

\[
\text{TRANSITION } T \{ \text{formal}_1: \text{type}_1, \ldots, \text{formal}_n: \text{type}_n \} \\
\text{ENTRY } \text{entry}\_\text{assertion} \\
\text{EXIT } \text{exit}\_\text{assertion} \\
\text{EXCEPT } \text{except}\_\text{assertion}_1 \\
\text{EXIT } \text{exit}\_\text{assertion}_1 \\
\ldots \\
\text{EXCEPT } \text{except}\_\text{assertion}_n \\
\text{EXIT } \text{exit}\_\text{assertion}_n
\]

'T' appearing in a refinement expression means the ENTRY-EXIT pair of transition T, while 'T.i' refers to the i\textsuperscript{th} EXCEPT-EXIT pair of T. For example, given the following upper level transition:

\[
\text{TRANSITION } T_u \\
\text{ENTRY } \ldots \\
\text{EXIT } \ldots \\
\text{EXCEPT } \ldots \\
\text{EXIT } \ldots
\]

and the two lower level transitions:

\[
\text{TRANSITION } T_{l1} \\
\ldots \\
\text{TRANSITION } T_{l2} \\
\ldots
\]

each having an ENTRY-EXIT pair and one EXCEPT-EXIT pair, the following are possible refinements:

\[
T_u = T_{l1}' \\
T_u.1 = T_{l1}.1
\]
or

\[
T_u = T_{l1}' \\
T_u.1 = T_{l2}
\]
or

\[
T_u = \text{IF boolean_expression THEN } T_{l1} \text{ ELSE } T_{l2} \text{ FI,} \\
T_u.1 = \text{IF boolean_expression THEN } T_{l1}.1 \text{ ELSE } T_{l2}.1 \text{ FI}
\]
2.3. Keywords and Comments

All ASLAN keywords are reserved. Keywords appearing in this document have been entirely in upper case letters, such as SPECIFICATION. ASLAN however does NOT require keywords to be upper case, and in fact SET, Set, set, sET, sEt, SeT are equivalent. The following are reserved words:

<table>
<thead>
<tr>
<th>ALT</th>
<th>AXIOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BECOMES</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>CONCAT</td>
<td>CONSTANT</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>CONTAINED_IN</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>DEFINE</td>
</tr>
<tr>
<td>ELSE</td>
<td>EMPTY</td>
</tr>
<tr>
<td>END</td>
<td>ENTRY</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>EXISTS</td>
</tr>
<tr>
<td>EXIT</td>
<td>FALSE</td>
</tr>
<tr>
<td>FI</td>
<td>FORALL</td>
</tr>
<tr>
<td>IF</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>INHIBIT</td>
<td>INITIAL</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTERSECT</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>IS</td>
</tr>
<tr>
<td>ISIN</td>
<td>LEVEL</td>
</tr>
<tr>
<td>LIST</td>
<td>LISTDEF</td>
</tr>
<tr>
<td>LIST_LEN</td>
<td>NIL</td>
</tr>
<tr>
<td>MOD</td>
<td>OF</td>
</tr>
<tr>
<td>NOCHANGE</td>
<td>SET</td>
</tr>
<tr>
<td>REFINES</td>
<td>SET_DIFF</td>
</tr>
<tr>
<td>SETDEF</td>
<td>STRUCTURE</td>
</tr>
<tr>
<td>SPECIFICATION</td>
<td>SUBTYPE</td>
</tr>
<tr>
<td>SUBSET</td>
<td>SYM_DIFF</td>
</tr>
<tr>
<td>SUPERSET</td>
<td>TRANSITION</td>
</tr>
<tr>
<td>THEN</td>
<td>TYPE</td>
</tr>
<tr>
<td>TRUE</td>
<td>UNION</td>
</tr>
<tr>
<td>TYPEDEF</td>
<td>VARIABLE</td>
</tr>
<tr>
<td>UNIQUE</td>
<td></td>
</tr>
</tbody>
</table>

ASLAN also uses these special symbols for operators, relations and punctuation:

- + = - \\
| | ( ) { } |
[ ] == < >
<= >= / & *

Comments are delimited by /* and */ and may appear anywhere a space may appear. Comments may not be embedded in identifiers, keywords, operators, or themselves.

2.4. Using the ASLAN Language Processor

The ASLAN language processor was constructed on UNIX using UNIX tools and is easy to use. Typically, one uses his/her favorite text editor to create a file containing the ASLAN specification. Typing:

```
aslan filename
```

UNIX is a trademark of Bell Laboratories.
in response to the UNIX prompt invokes the ASLAN processor causing input to be taken from the file
filename. After an appropriate amount of time either SUCCESS or FAILURE will appear on the terminal
and the UNIX prompt will return. ASLAN creates the file "filename.out" containing a dated source listing,
error messages, and correctness conjectures (unless INHIBITed).

3. Conjectures Revisited

3.1. Top Level Conjectures

3.1.1. Correctness Conjectures
In addition to an initial condition conjecture:

\[ \text{initial\_assertion} \rightarrow \text{invariant\_assertion} \]

for each top level transition of the form

\[
\text{TRANSITION } T \ (\text{arguments}) \\
\text{ENTRY } \text{entry\_assertion} \\
\text{EXIT } \text{exit\_assertion} \\
\text{EXCEPT } \text{except\_assertion}_1 \\
\text{EXIT } \text{exit\_assertion}_1 \\
\ldots \\
\text{EXCEPT } \text{except\_assertion}_n \\
\text{EXIT } \text{exit\_assertion}_n \\
\]

ASLAN produces the following correctness conjectures:

\[
\text{invariant\_assertion}' \ & \ \text{entry\_assertion}' \ & \ \text{exit\_assertion}' \\
\rightarrow \\
\text{invariant\_assertion} \ & \ \text{constraint\_assertion} \\
\]

and for \(1 \leq i \leq n:\)

\[
\text{invariant\_assertion}' \ & \ \text{except\_assertion}_i' \ & \ \text{exit\_assertion}_i \\
\rightarrow \\
\text{invariant\_assertion} \ & \ \text{constraint\_assertion} \\
\]

Where assertion' is the assertion with all variables \(V\) appearing in the expression being replaced by \(V'\).

3.1.2. Consistency Conjectures
At the present time ASLAN does not generate the conjectures necessary to build a proof of the consist-
sistency of the specification. A false invariant, entry, or exit assertion allows any invariant and constraint
to be deduced. It is therefore desirable that:

\[
\neg(\text{initial\_assertion} \rightarrow \text{FALSE}) \\
\]

and for every transition \(T\) of the above form,

\[
\neg(\text{invariant\_assertion}' \ & \ \text{entry\_assertion}' \ & \ \text{exit\_assertion} \rightarrow \text{FALSE}) \\
\]

and for \(1 \leq i \leq n:\)
¬(invariant\_assertion_i \& except\_assertion_j \& exit\_assertion_i \rightarrow FALSE)

Some other properties the specification writer should be aware of are:

- Determinancy. For a given transition T,

\[ \neg(entry\_assertion_i \& except\_assertion_i) \text{ for } 1 \leq i \leq n, \]

and

\[ \neg(except\_assertion_i \& except\_assertion_j) \text{ for } 1 \leq i, j \leq n, i \neq j. \]

- Universal applicability. For a given transition T,

\[ (entry\_assertion_i | except\_assertion_i | \ldots | except\_assertion_n) \]

3.2. Lower Level Conjectures

ASLAN generates conjectures to be used in an inductive proof that each lower level is a correct implementation of the upper level it REFINEs. As in Section 2.2.5, the subscript ‘u’ refers to the upper level and ‘l’ to the lower level. In addition, Impl(assertion) stands for the result of replacing each higher level constant or variable in an assertion with the lower level expression which refines that constant or variable.

3.2.1. Correctness Conjectures

As the basis of the induction an initial conditions conjecture is produced:

\[ initial_l \rightarrow Impl(initial_u) \& invariant_l \]

Two types of lemmas are generated for the inductive step. First, for ENTRY-EXIT pairs of lower level transitions that do not refine upper level transitions:

\[ Impl(invariant_u \& \neg entry \& exit) \rightarrow Impl(invariant_u \& invariant_l \& Impl(constraint_u) \& constraint_l) \]

and for each EXCEPT-EXIT pair:

\[ Impl(invariant_u \& \neg except \& exit) \rightarrow Impl(invariant_u \& invariant_l \& Impl(constraint_u) \& constraint_l) \]

The second type of correctness lemmas relate upper level transitions with lower level expressions that refine them. In general, a refinement statement

\[ T_u.\_k(a_1, \ldots, a_r) = \text{wf\_formula} \]

must satisfy the one transition reference per conjunction restriction of Section 2.2.5. That is, the disjunctive normal form of \(\text{wf\_formula}\) must look like:

\[ (A_1 \& T_1.1(t_{1,1}, \ldots, t_{1,n_1}) \mid \ldots | A_m \& T_m.\_m(t_{m,1}, \ldots, t_{m,n_m})) \]

where \(T_i\) are lower level transitions and the \(t_{i,j}\) are dummy variables from the left side, for \(1 \leq i \leq m, 1 \leq j \leq n_i\).

\[ ^5\text{In the spirit of Section 2.2.5, if the 'k' is omitted from the upper (lower) level transition of the refinement statement the conjectures are generated with except\_uk (except\_Ti.\_k) replaced by entry\_u (entry\_Ti) and exit\_uk (exit\_Ti.\_k) replaced by exit\_u.} \]
\[ \leq n_i. \text{ASLAN generates the following conjectures for each disjunct } A_i \land T_j(t_{i,1} \ldots t_{i,n_i}): \]

\[
\text{Impl(except}_{uk} & \text{Impl(invariant}_{u} & \text{invariant}_{l} \land A_i \rightarrow \text{except}_{T_i,j}
\]

and

\[
\text{Impl(except}_{uk} & \text{Impl(invariant}_{u} & \text{invariant}_{l} & A_i & \text{exit}_{T_i,j}
\rightarrow
\text{Impl(exit}_{uk} & \text{constraint}_{l} & \text{invariant}_{l}
\]

A proof of the former lemma guarantees correct application of the lower level transition. The validity of the latter lemma guarantees correct refinement. The following examples illustrate the above concepts. The simplest refinement statement is:

\[ T_{u,i} = T_{l,j} \]

ASLAN generates a conjecture asserting that the \( j \)th lower level exit expression may be applied whenever the \( i \)th upper level except expression, and the upper and lower invariants hold:

\[
\text{Impl(except}_{ui} & \text{Impl(invariant}_{u} & \text{invariant}_{l} \rightarrow \text{except}_{lj}
\]

A conjecture stating that the application of the lower level transition implies the implementation of the upper level exit assertion, the lower level invariant, and lower level constraint is also produced:

\[
\text{Impl(except}_{ui} & \text{Impl(invariant}_{u} & \text{invariant}_{l} & \text{expression} & \text{constraint}_{l} & \text{invariant}_{l}
\rightarrow
\]

A refinement statement of the form

\[ T_{u,i} = \text{IF expression THEN } T_{l,j} \text{ ELSE } T_{2,k} \text{ FI is equivalent to} \]

\[ T_{u,i} = \text{expression} & T_{l,j} | \text{expression} & T_{2,k} \]

in disjunctive normal form. Four conjectures are produced. As before, the first two conjectures concern the proper application of the lower level transitions:

\[
\text{Impl(except}_{ul} & \text{Impl(invariant}_{u} & \text{invariant}_{l} & \text{expression} \rightarrow \text{except}_{T1.j}
\]

\[
\text{Impl(except}_{ul} & \text{Impl(invariant}_{u} & \text{invariant}_{l} & \text{expression} \rightarrow \text{except}_{T2.k}
\]

where except\(_{Tc,d}\) is the \( d \)th except assertion of transition \( T_c \).

The following two conjectures assert that the lower level transitions properly refine the upper level transition. They are basically the same as the second conjecture generated for the simpler refinement statement, with expression (expression) conjoined to the antecedent.

\[ (\text{exit}_{T_{p,l}}) \]
Impl(except $u_i$) & Impl(invariant $u_i$) & invariant $l_i$ & expression $l_i$ & exit $T_{1,j}$

$\rightarrow$

Impl(exit $u_i$) & constraint $l_i$ & invariant $l_i$

and

Impl(except $u_i$) & Impl(invariant $u_i$) & invariant $l_i$ & $\neg$expression $l_i$ & exit $T_{2,k}$

$\rightarrow$

Impl(exit $u_i$) & constraint $l_i$ & invariant $l_i$

where exit $T_{c,d}$ is the $d^{th}$ exit assertion of transition $T_c$.

3.2.2. Consistency Conjectures

ASLAN does not generate inter-level consistency conjectures. To prove that a refinement consistently relates two levels it is necessary to show that none of the antecedents of lower level correctness conjectures are FALSE. Lower level consistency conjectures are therefore analogous to the top level consistency conjectures of Section 3.1.2.

4. A Top Level Specification Example

The system to be specified consists of a university library data base. The transactions available include:

- Check out a book.
- Return a book.
- Add a copy of a book to the library.
- Remove a copy of a book from the library.
- Get a list of titles of books in the library by a particular author.
- Find out what books are currently checked out by a particular student.
- Find out what student last checked out a particular copy of a book.

The following restrictions apply to the use of these transactions: A book may be added to or removed from the library only by someone with library staff status. Library staff status is also required to find out which student last checked out a book. A student may find out only what books he or she has checked out. A person with library status may find out what books are checked out by any student.

In addition, the system must satisfy the following restrictions at all times: All books in the library must be either checked out or available for check out. No book may be both checked out and available for check out. A student may not have more than book limit books out at one time. A student may not check out more than one copy of the same book at one time.

These final pages contain a copy of the "library.out" file resulting from invoking the aslan processor on a top level specification of the university library data base. Because some of the correctness conjectures are long, only the initial conditions conjecture and the conjectures corresponding to the Check_Out and What_Checked_Out transitions are included.
1 SPECIFICATION Library
2 LEVEL Top_Level
3
4 TYPE
5 User,
6 Book,
7 Book_Title,
8 Book_Author,
9 Book_Collection IS SET OF Book,
10 Titles IS SET OF Book_Title,
11 Pos_Integer IS TYPEDEF i:INTEGER (i>0)
12
13 CONSTANT
14 Title(Book):Book_Title,
15 Author(Book):Book_Author,
16 Library_Staff(User):BOOLEAN,
17 Book_Limit:Pos_Integer
18
19 DEFINE
20 Copy_Of(B1,B2:Book) : BOOLEAN ==
21 Author(B1) = Author(B2)
22 & Title(B1) = Title(B2)
23
24 VARIABLE
25 Library:Book_Collection,
26 Checked_Out(Book):BOOLEAN,
27 Responsible(Book):User,
28 Number_Books(User):INTEGER,
29 Never_Out(Book):BOOLEAN,
30 User_Result:User,
31 Book_Result:Book_Collection,
32 Title_Result:Titles
33
34 DEFINE
35 Available(B:Book):BOOLEAN ==
36 B ISIN Library & ~Checked_Out(B),
37 Checked_Out_To(U:User,B:Book):BOOLEAN ==
38 Checked_Out(B)
39 & Responsible(B)=U
40
41 INITIAL
42 Library = EMPTY
43 & FORALL u:User (Number_Books(u) = 0)
44 & FORALL b:Book (~Checked_Out(b))
INVARIANT

FORALL b:Book (b ISIN Library ->
  Checked_Out(b) & ~Available(b)
  | ~Checked_Out(b) & Available(b))
& FORALL u:User (Number_Books(u) <= Book_Limit)
& FORALL u:User, b1, b2:Book(
  Checked_Out_To(u, b1)
  & Checked_Out_To(u, b2)
  & Copy_Of(b1, b2)
  -> b1 = b2)

TRANSITION Check_Out(U:User, B:Book)
EXIT
Available'(B)
& Number_Books'(U) < Book_Limit
& IF FORALL B1:Book (Checked_Out_To'(U, B1) -> ~Copy_Of(B, B1))
THEN
  Number_Books(U) BECOMES (Number_Books'(U) + 1)
  & (Checked_Out(B) BECOMES TRUE)
  & (Responsible(B) BECOMES U)
  & (Never_Out(B) BECOMES FALSE)
FI

TRANSITION Return(B:Book)
EXIT
( IF Checked_Out'(B)
  THEN Checked_Out(B) BECOMES FALSE
  & Number_Books(Responsible'(B))
  BECOMES (Number_Books(Responsible'(B)) - 1)
FI)

TRANSITION Add_A_Book(U:User, B:Book)
EXIT
( IF Library_Staff(U)
  & B ~ISIN Library'
  THEN Library = Library' UNION {B}
  & Checked_Out(B) BECOMES FALSE
  & Never_Out(B) BECOMES TRUE
FI)

TRANSITION Remove_A_Book(U:User, B:Book)
EXIT
( IF Library_Staff(U)
  & Available'(B)
  THEN Library = Library' SET_DIFF {B}
FI)

TRANSITION Last_Responsible(U:User, B:Book)
EXIT
( IF Library_Staff(U)
  & B ISIN Library'
& 'Never_Out'(B)
THEN User_Result = Responsible'(B)
FI)

TRANSITION What_Checked_Out(Requester,Whom:User)
ENTRY
Library_Staff(Requester) | Requester = Whom
EXIT
FORALL B1:Book (  
    Checked_Out_To'(Whom,B1) & B1 ISIN Book_Result
    ALT 'Checked_Out_To'(Whom,B1) & B1 ~ISIN Book_Result)
EXCEPT
Library_Staff(Requester) | Requester = Whom
EXIT
Nochange (Book_Result)

TRANSITION Titles_By_Author(By_Whom:Book_Author)
EXIT
Title_Result =
(SETDEF T1:Book_Title
    EXISTS B1:Book (Author(B1)=By_Whom
    & Title(B1)=T1
    & B1 ISIN Library'))

END Top_Level
END Library

UNREFINED IDENTIFIERS:
*****************************
None

-------------------------------------------------------------------------------
LEVEL: Top_Level
-------------------------------------------------------------------------------

Conjecture for Initial Conditions:

( Library = EMPTY
&
    FORALL u : User
    (Number_Books(u) = 0)
&
    FORALL b : Book
    (~'Checked_Out(b))
) -> {
FORALL b : Book
   (b ISIN Library) ->
   (Checked_Out(b) & ~Available(b) | ~Checked_Out(b) & Available(b))
)

& FORALL u : User
   (Number_Books(u) <= Book_Limit)
&
FORALL u : User, b1 : Book, b2 : Book
   (Checked_Out_To(u, b1) &
   Checked_Out_To(u, b2) &
   Copy_Of(b1, b2)
) ->
   (b1 = b2)
)

Conjectures for Transitions:

******* TRANSITION Check_Out (U : User, B : Book) *******

{ FORALL b : Book
   (b ISIN Library')
   ->
   (Checked_Out'(b) & ~Available'(b) | ~Checked_Out'(b)
   )
   

&
"Available'(b)
"
)

&
FORALL u : User
(Number_Books'(u) <= Book_Limit)

&
FORALL u : User, b1 : Book, b2 : Book
(
(\n  Checked_Out_To'(u, b1)
&
  Checked_Out_To'(u, b2)
&
  Copy_Of'(b1, b2)
)
->
(b1 = b2)
)
& (TRUE)
& (\n  Available'(b1)
&
  Number_Books'(u) < Book_Limit
&
IF
FORALL B1 : Book
(\n  Checked_Out_To'(B1, B1)
->
"Copy_Of(_001, B1)
)
THEN
(\n  FORALL _001 : User
  (\n    IF
    _001 = U
    THEN
      (Number_Books(_001) = Number_Books'(_001) + 1)
    ELSE
      (Number_Books(_001) = Number_Books'(_001))
    FI
  )
&
  FORALL _001 : Book
  (\n    IF
    _001 = B
    THEN
      (Checked_Out(_001) = TRUE)
    ELSE

32
\begin{verbatim}
(Checked_Out(_001) = Checked_Out'(_001))
FI )
&
FORALL _001 : Book
 {
 IF
  _001 = B
 THEN
   (Responsible(_001) = U)
 ELSE
   (Responsible(_001) = Responsible'(_001))
 FI )

&
FORALL _001 : Book
 {
 IF
  _001 = B
 THEN
   (Never_Out(_001) = FALSE)
 ELSE
   (Never_Out(_001) = Never_Out'(_001))
 FI )

 ELSE
 TRUE
 &
FORALL _001 : Book
 (Checked_Out(_001) = Checked_Out'(_001))
 &
FORALL _001 : Book
 (Responsible(_001) = Responsible'(_001))
 &
FORALL _001 : User
 (Number_Books(_001) = Number_Books'(_001))
 &
FORALL _001 : Book
 (Never_Out(_001) = Never_Out'(_001))
 FI
)
 &
 Library = Library'
 &
 User_Result = User_Result'
 &
 Book_Result = Book_Result'
 &
 Title_Result = Title_Result'
) -> {
 FORALL b : Book
 {
   (b ISIN Library)
   ->
   (Checked_Out(b)
\end{verbatim}
\[\forall u : \text{User} \ 
(\text{Number}_{-}\text{Books}(u) \leq \text{Book}_{-}\text{Limit})\]
\[\forall u : \text{User}, b1 : \text{Book}, b2 : \text{Book} \ 
(\ 
(\ 
\text{Checked}_{-}\text{Out}_{-}\text{To}(u, b1) \ 
\land \ 
\text{Checked}_{-}\text{Out}_{-}\text{To}(u, b2) \ 
\land \ 
\text{Copy}_{-}\text{Of}(b1, b2) \ 
\rightarrow \ 
(b1 = b2) \ 
) \) \land (\ 
\text{TRUE} \ 
) \)

******** TRANSITION What_{-}Checked_{-}Out (Requester : User, Whom : User) ********

\{(\ 
\forall b : \text{Book} \ 
(\ (b \text{ ISIN Library'})) \ 
\rightarrow \ 
(\ 
\text{Checked}_{-}\text{Out}'(b) \ 
\land \ 
\text{'Available'}(b) \ 
\lor \ 
\text{'Checked}_{-}\text{Out'}(b) \ 
\land \ 
\text{'Available'}(b) \ 
) \ 
) \)
\[\forall u : \text{User} \ 
(\text{Number}_{-}\text{Books}'(u) \leq \text{Book}_{-}\text{Limit})\]
\[\forall u : \text{User}, b1 : \text{Book}, b2 : \text{Book} \ 
(\ 
(\ 
\text{Checked}_{-}\text{Out}_{-}\text{To}'(u, b1) \ 
)}
&
  Checked_Out_To'(u, b2)
&
  Copy_Of'(b1, b2)
) ->
  (b1 = b2)
)
) & (Library_Staff(Requester)
| Requester = Whom
) & {
  FORALL B1 : Book
  {
    ((
      Checked_Out_To'(_001, B1)
    &
      B1 ISIN Book_Result
    )
    |
    (
      "Checked_Out_To'(_001, B1)
    &
      B1 "ISIN Book_Result
    )
    )
  }
  Library = Library'
  &
  FORALL _001 : Book
  (Checked_Out(_001) = Checked_Out'(_001))
  &
  FORALL _001 : Book
  (Responsible(_001) = Responsible'(_001))
  &
  FORALL _001 : User
  (Number_Books(_001) = Number_Books'(_001))
  &
  FORALL _001 : Book
  (Never_Out(_001) = Never_Out'(_001))
  &
  User_Result = User_Result'
  &
  Title_Result = Title_Result'
) -> {
  FORALL b : Book
  {
    (b ISIN Library)
    ->
    {
      Checked_Out(b)
      &
      "Available(b)
    }
  }
\[
\text{\texttt{FORALL u : User}} \\
\text{(Number_Books(u) <= Book_Limit)} \\
\text{\texttt{& \text{\texttt{FORALL u : User, b1 : Book, b2 : Book}}}} \\
\text{(} \\
\text{  \texttt{(Checked_Out_To(u, b1)}} \\
\text{  \&} \\
\text{  \texttt{Checked_Out_To(u, b2)}} \\
\text{  \&} \\
\text{  \texttt{Copy_Of(b1, b2)}} \\
\text{)} \\
\text{\texttt{-}->} \\
\text{\texttt{(b1 = b2)}} \\
\text{)} \\
\text{\texttt{& \text{\texttt{(TRUE}}}} \\
\text{\texttt{)}}
\]

******** TRANSITION What_Checked_Out.1 (Requester : User, Whom : User) ********

\[
\text{\texttt{FORALL b : Book}} \\
\text{(b ISIN Library')} \\
\text{\texttt{-}->} \\
\text{(} \\
\text{  \texttt{(Checked_Out'(b)}} \\
\text{  \&} \\
\text{  \texttt{~Available'(b)}} \\
\text{  |} \\
\text{  \texttt{~Checked_Out'(b)}} \\
\text{  \&} \\
\text{  \texttt{Available'(b)}} \\
\text{)} \\
\text{)} \\
\text{\texttt{& \text{\texttt{FORALL u : User}}}} \\
\text{(Number_Books'(u) <= Book_Limit)} \\
\text{\texttt{& \text{\texttt{FORALL u : User, b1 : Book, b2 : Book}}}} \\
\text{(} \\
\text{  \texttt{(Checked_Out_To'(u, b1)}} \\
\text{  \&} \\
\text{  \texttt{Checked_Out_To'(u, b2)}} \\
\text{)} \\
\text{)}
\]
\text{Checked\_Out\_To}(u, b2)
&
\text{Copy\_Of}(b1, b2)
)
->
(b1 = b2)
)
& ( TRUE
)}
5. Appendix - Current Status  
February, 18, 1992

Areas in which ASLAN is incomplete or in need of improvement are outlined below.

- Only the first error found is reported. That is, it is possible that an error makes the state of internal ASLAN system structures inconsistent.

- No warnings are issued for applying \( \dot{\cdot} \) to DEFINEd identifiers whose body contains no variables.

- No checks are made that "new value" variables do not appear in the body or as arguments of a BECOMES statement, or as arguments to a DEFINEd identifier.

- In section 2.2.5, the User’s Manual states that "Transitions may be refined by any \( wf \) formula with the following restriction: the \( wf \) formula, if converted to disjunctive normal form, must have exactly one reference to a lower level transition in each conjunct". The following restrictions apply to the refining \( wf \) formula:

  1) The \( wf \) formula must either be in disjunctive normal form,

  or

  2) the \( wf \) formula must be an IF-FI statement as illustrated at the end of section 2.2.5.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. The Finite State Machine Model</td>
<td>1</td>
</tr>
<tr>
<td>1.2. An Overview of Correctness Conjectures</td>
<td>1</td>
</tr>
<tr>
<td>1.3. A Simple Sample Specification</td>
<td>3</td>
</tr>
<tr>
<td>2. The ASLAN Specification Language</td>
<td>3</td>
</tr>
<tr>
<td>2.1. Well-Formed Formulas</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1. Primitive Relations and Operations</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2. Terms</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2.1. Identifiers</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2.2. Lists</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2.3. Component Specifiers</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2.4. Sets</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2.5. Quantification</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2.6. The &quot;Procedural&quot; Operations</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2.6.1. Implied NoChanges</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2.6.2. Alternate Statements</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2.6.3. Conditional Statements</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2.6.4. The Becomes Operator</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2.6.5. The NoChange Operation</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2.6.6. Implied NoChanges Revisited</td>
<td>13</td>
</tr>
<tr>
<td>2.2. ASLAN Specifications</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1. Levels and Their Relationships</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2. The Declaration Section</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2.1. Types</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2.2. Constants</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2.3. Variables</td>
<td>18</td>
</tr>
<tr>
<td>2.2.2.4. Definitions</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3. The Requirements Section</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3.1. Axioms</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3.2. Initial Conditions</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3.3. Invariants and Constraints</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4. The Transitions Section</td>
<td>19</td>
</tr>
<tr>
<td>2.2.5. The Implementation Section</td>
<td>20</td>
</tr>
<tr>
<td>2.3. Keywords and Comments</td>
<td>23</td>
</tr>
<tr>
<td>2.4. Using the ASLAN Language Processor</td>
<td>23</td>
</tr>
</tbody>
</table>