Model Checking IOA Programs with TLC

Summer 2001 Report

Stanislav Funiak

July 29, 2001

1 Introduction

This document describes the results from my work in June and July 2001 on connecting the IOA language to the TLC model checker. The reader is assumed to be familiar with the syntax of both IOA and TLA+. Appendix B assumes the knowledge of IOA's built-in data types.

The aim of this project is to design and implement a tool which performs validation of distributed systems written in the IOA language. The tool translates IOA programs to TLA+ specifications which can be then verified using TLC. The core of the research is to find ways to translate the IOA programs so that we can make best use of TLC's potential and maximize the spectrum of distributed systems which can be checked in reasonable time (preliminary results in [1] showed that time, more than memory, is the bottleneck for TLC's performance).

I started the work by hand-translating several examples of primitive IOA automata (see Appendix A for a complete list) and verifying their translations with TLC. Based on my hands-on experience and on the information on TLC given in [4], I came up with a scheme for translating primitive IOA automata to TLA+. This scheme was later augmented to support arbitrary imperative programs by translating the variables into a single record. The results are described in Section 2.

Beginning June 16, I started designing and implementing the prototype and later, the first usable version of the tool. Well into the implementation phase, I realized that the problem of constraining the model was more important (and more complex) than I originally thought; I virtually ignored this issue in the original design. Section 3 offers an analysis of possible contraints and a preliminary method for constraining IOA programs. Section 4 describes the requirements on the tool and its design.

Needless to say, I hope to continue working on this project and have several ideas about its future development. Section 5 suggests possible directions for the development of the tool.

2 Translating primitive automata to TLA+

2.1 Overview

In a sense, IOA and TLA+ are very similar[1]. Both IOA and TLA+ are used primarily for specifying and validating concurrent systems. Both IOA and TLA+ describe behaviors of the system, and both describe systems in terms of states and transitions. Both IOA and TLA+ are based on logic rather than relations, and both allow non-determinism in the specifications. There are some important differences between IOA and TLA+; for example, TLA+ is purely declarative, while IOA allows both imperative and declarative constructs, and composition is handled differently in the two languages. Nevertheless, the two languages have very much in common.

Seeing these similarities, I decided to translate the elements of IOA language to their most natural equivalents in TLA+: for instance, if automaton is the unit of specification in IOA+, then let automaton's translation be a module, a unit of specification in TLA+. Then, if TLC is good at model checking TLC specifications, it should also be good at model checking IOA specifications. My experience with hand-translating example IOA automata suggests that this approach is viable.

The TLA+ translations of elemens of an IOA program are summarized in the table below. The following subsections describe in detail each part of the translation scheme (in some cases, I chose a translation other than the most natural TLA+ equivalent; these decisions are justified in the respective subsections).

[1]It is dangerous for an undergraduate student to compare the work of two senior researchers. Inevitably, I will get something wrong. I would appreciate any comments and corrections on what I say in this overview.

Element of an IOA program	TLA+ translation
Automaton	module
Term	equivalent TLA+ term
Sort	a set (if possible)
Operator	equivalent operator (built-in or custom)
Formal type	a finite set
Formal parameter	a variable which is never modified
State variables	a single variable - record
Initial state	initial-state predicate
Action	a custom operator (predicate)
Transition	a custom operator (predicate) which acts on the pre-state, post-state, and formals
Invariant	a custom operator (predicate) which acts on the state and automaton's formals

2.2 Automaton

A primitive automaton translates into a TLA+ module. At present, the whole translation (e.g. custom operators implementing certain built-in operators, predicates for initial state, transitions, and invariants) is located in a single module. Later, if necessary, the translation can be more structured, e.g. the invariants can be placed into a separate module.

2.3 Terms

All terms are translated to their TLA+ equivalents: integer literals remain untouched, variable references are translated to references to appropriate TLA+ variables or expressions, application terms are translated by applying the operator's implementation (see next subsection) to the translation of the operands, and similarly, quantifier terms are translated by applying the $\exists$ and $\forall$ terms to the translation of the bound variable's sort and the translation of the predicate.

In some cases, translating the quantifier terms directly may result in specifications which are not checkable by TLC: since TLC cannot process quantifiers whose bound variable's set is infinite, programs with expressions of the form $\exists$ `e:E p(e)` or $\forall$ `e:E p(e)` where `E` is a sort with infinite number of elements will not be translated correctly. We can solve this problem in some cases (e.g. when `p(e)` is of the form `e` $\in$ `S` $\Rightarrow$ `q(e)` and `S` is a finite `Set[E]`). However, I do not know how to solve this problem in general.

2.4 Data types (sorts and operators)

Since TLC supports a large subset of TLA+'s rich set of built-in types and operators (defined either in TLA+'s syntax or in TLA+'s standard modules), it was not hard to come up with representations for the built-in IOA types. For most sorts and operators, TLA+ has a direct equivalent; in others, I chose what seemed to be the most natural representation. If a TLA+ operator or type is essential for the translation and is not yet supported by TLC, we may decide to submit a request to Yuan Yu (author of TLC).

Sorts are implemented as sets of TLA+ values which correspond to the elements of the given sort (for example, the sort `Int` is represented as *Int*, a TLA+ set of integers). Since at the time of writing, each sort's value is represented by precisely one TLA+ value, there is no ambiguity whether the set should contain *all* TLA+ values which correpond to elements of the given sort, or only one such value for each element of the sort[2].

The sort's implementation is used in quantifier terms as described above. In addition, the sort's implementation can be used to efficiently check (at runtime by TLC) if a given TLA+ expression is a member of the sort (thus, sort's implementation is also denoted as sort's *checkable set*). While this feature is not used in the tool now, it may become useful later (for example, to verify that the tool preserves type invariant or to validate the TLA+ constraints specified by user, see Section 3).

For some sorts, there is no expression which represents the set of its elements in a way which allows efficient verification by TLC. For example, the set of all `Mset[E]` could be expressed in TLA+ as $\text{UNION } \{[S \rightarrow Nat \setminus \{0\}] : S \in \text{SUBSET } E\}$ but this expression is useless since TLC cannot test whether a given value x belongs to this set. For these sorts, membership can be tested differently (for example, by checking that the domain of x is a subset of `E` and that each value of x is a positive integer).

Operators are implemented as equivalent TLA+ operators (for example, the operator
`__+__: Int, Int → Int`
is implemented by the operator `+` defined in the standard module *Integers*) or expressions. See Appendix B for a detailed description of built-in datatypes' implementations.

2.5 States

State variables are translated into a single TLA+ variable, a record (denoted by s). This approach is different from my original proposal in [1] to translate each state variable to a separate TLA+ variable.

Translating state variables into a single record has an advantage, namely that all transitions (no matter whether they contain conditionals or nested loops or same lvalues in several statements) can be handled uniformly. In principle, both approaches are equally expressive; if necessary, we could always construct the record from the pre-state at the beginning of the transition, apply the effect clause, and express the post-state in terms of the result as shown below (s_i are state variables):

$$
\begin{array}{ll}
\text{LET} & state \triangleq [s_1 \mapsto s_1, s_2 \mapsto s_2, s_n \mapsto s_n] \\
 & result = applyEffect(state) \\
\text{IN} & \land\ s'_1 = result.s_1 \\
 & \land\ s'_2 = result.s_2 \\
 & \land\ \ldots \\
 & \land\ s'_n = result.s_n
\end{array}
$$

However, trivial transitions (e.g. transitions without conditionals or loops in which each variable appears in lvalue at most once) would then presumably be translated differently (to take advantage of translating state variables into separate TLA+ variables), which would complicate the tool's design and implementation unnecessarily.

To asses the impact of this decision, I compared the time it took to model check some of the hand-translated examples in both cases. I also explored the possibility of implementing states as a TLA+ tuple (in TLA+, *tuples* are sequences of fixed length). The results are summarized in the table shown below (measured with TLC v1.32):

[2] This also means that the $=$ and $\neq$ operators can be implemented simply by TLA+'s `=` and `#`.

Example	Model size	Relative time for variables translated separately	as a record	as a tuple
Adder	$\lvert IntR\rvert = 41$	1.00	1.28	1.45
Choice	$\lvert IntR\rvert = 41$	1.00	1.01	
Distribute	$\lvert IntR\rvert = 9$	1.00	1.21	1.28 [a]
ComposedLCR	$\lvert IntR\rvert = 4, n = 3$ [b]	1.00	1.02	
Multicast	(see below)[c]	1.00	1.03	
LossyBuffer	$\lvert M\rvert = 5, \lvert buff\rvert \leq 8$	1.00	1.10	
LossyBuffer	$\lvert M\rvert = 7, \lvert buff\rvert \leq 8$	1.00	1.09	

[a] Did not check the type invariant (otherwise, TLC thows an error).
[b] Only checked the type invariant.
[c] $Message = \{A, B\}$, $Node = \{v1, v2\}$, $MaxQueueLen = 2$, $BufSize = 1$; did not check the type invariant.

As seen in the table, expressing states in a single record leads to a 2-28% decrease in performance. This performance degradation is greatest for the simple examples of Adder, Distribute, and LossyBuffer; for the more complex examples (Multicast and ComposedLCR), it is at or below 3%, which is negligible (moreover, the performance degradation seems to be inversely proportional to the model size). 3% seems to be a small price to pay considering how much simpler the translation would be. Thus, decided to translate states as a single record (TLA+'s tuples are clearly not an option since TLC seems to handle tuples less efficiently than records and supports them only partially, e.g. products of three or more sets are not suppoted).

2.6 Formal types

Formal types are translated into sets (as other sorts). Since TLC explicitly computes the states of the specification, formal types need to instantiated before the specification can be checked (using the TLC terminology, this defines a *model* of the specification).

The set implementing a formal type can either be defined in the module itself, or it can be specified as a constant and defined in the configuration file. Both approaches seem equally viable, and it is up to the tool's design to pick a convenient one (the present design defines these sets in the module). The latter option allows one define the set in terms of *model values* (the advantage of model values is that they are guaranteed to be equal only to themselves). However, since IOA is a typed language, and variables of the given formal type will never be compared with variables of other sorts, a formal type can be translated to an arbitrary finite set (e.g. $1..n$).

2.7 Formal parameters

Formal parameters translate into TLA+ variables which remain unchanged over the course of the execution. In this way, user may model check automaton with various values substituted for its formal parameters. In principle, formal *types* could be treated similarly; however, this would probably complicate the tool unnecessarily. We can reexamine this issue when needed.

The values taken by a formal parameter are specified in the translation's initial-state predicate by conjuncts in the form $p_i \in P_i$ where p_i is the parameter's identifier (see Section 4.7 for the discussion of naming) and P_i is the set of values it can take. These conjuncts need to appear in the initial-state predicate *before* the translation of the initial state (see the next subsection) because formal parameters may appear in the variables' initial values and/or the **so that** predicate [3]

2.8 Initial state

Automaton's initial state translates into a predicate which defines the set of possible values for the record s. For each state variable v_i, we first express the set of its possible initial values V_i based either on the initial value of the variable (e.g. `constant(0)` or **choose** `x:Int x≥1`), or its sort if no initial value was specified. s is then a valid initial state as long as it is in the set S of all records with elements $v_1, \ldots, v_n$ in their respective sets $V_1, \ldots, V_n$ and satisfies the **so that** predicate:

$$
\begin{array}{l}
\mathit{Init} \ \triangleq \\
\quad \text{LET} \ \ V_1 \ \triangleq \ \{\ldots\} \\
\qquad\qquad V_2 \ \triangleq \ \{\ldots\} \\
\qquad\qquad \ldots \\
\qquad\qquad V_n \ \triangleq \ \{\ldots\} \\
\quad \text{IN} \ \ \ \wedge\, s \in [v_1 : V_1, v_2 : V_2, \ldots, v_n : V_n] \quad (= S) \\
\qquad\qquad \wedge\, \mathit{so\_that\_predicate}(s)
\end{array}
$$

When TLC evaluates the above expression, it first evaluates each of sets V_1, ..., V_n, then quickly constructs their product and for each element in the product, verifies if the **so that** predicate holds. Note that since TLC needs to bind each variable to a finite set when the variable is encountered in the evaluated expression for the first time[4], the set $[v_1 : V_1, v_2 : V_2, ..., v_n : V_n]$ needs to be finite; consequently, the sets V_i needs to be finite, too. The user may need to constrain the sets V_i as described in Section 3.

The translation proposed above might be sub-optimal in one case: when there are dependencies between state variables (as shown in Figure 1 below), only a few elements of the set S will satisfy the **so that** predicate. Not only will it be inefficient to filter out a few elements from S, TLC may not even be able to enumerate the set S if it is too large (S needs to fit into the memory). We will need to evaluate how frequent these cases are and find a way to deal with them (e.g. by letting the user construct the tuples of dependent variables). Maybe representing each state variable with a separate TLA+ variable would alleviate this problem[5]. I suspect, however, that we would run into the same problem once we use the composing tool. Presumably, the state of each subautomaton will be represented in the compoed automaton (again) as a single state variable or an element of an array.

```
states
  a: Array[I, E],
  b: Array[I, E]
  so that a = b
```

Figure 1: Dependent state variables.

[3] At the time of writing, the tool incorrectly places the conjuncts *after* the translation of automaton's initial state. I will fix this soon.

[4] We should ask Yuan Yu there are any exceptions to this requirement, or if they are planning any optimizations which would eliminate it

[5] I do not have sufficient experience with examples of this type to tell. Perhaps we could ask Yuan Yu for his opinion on this issue but I am not sure if he will be able to help us.

2.9 Actions

Action translate into predicates which relate pre-state to post-state and are true if and only if the post-state results from applying an action's transition to the current state. At the time of writing, only actions with formals of the form `var:sort` are supported (translating **const** formals should not be too hard; however, I will need to think carefully to ensure that they are model checked efficiently). If the action's formals are $a_1, \ldots, a_n$ and their respective sorts are $S_1, \ldots, S_n$, and the predicates for the action's transitions (see the next subsection) are named $T_1, \ldots, T_m$, the action is translated as follows (if the formals or the **where** predicate are missing, the expression is adjusted as necessary):

$$
\begin{array}{l}
\exists\, a_1 \in S_1, \ldots, a_n \in Sn \;:\; \textit{where\_predicate} \quad \land \\
\quad \lor\; T_1(a_1, \ldots, a_n) \\
\quad \lor\; \ldots \\
\quad \lor\; T_m(a_1, \ldots, a_n)
\end{array}
$$

When TLC evaluates the above expression, it first finds all the combinations of values a_1, ..., a_n which satisfy the given **where** predicate, and then generates the possible post-states using the transitions predicates T_1, ..., T_m. This translation is probably better than the expression

$$
\begin{array}{l}
\lor\; \exists\, a_1 \in S_1, \ldots, a_n \in Sn \;:\; \textit{where\_predicate} \land T_1(a_1, \ldots, a_n) \\
\lor\; \ldots \\
\lor\; \exists\, a_1 \in S_1, \ldots, a_n \in Sn \;:\; \textit{where\_predicate} \land T_m(a_1, \ldots, a_n),
\end{array}
$$

in which TLC would evaluate the set of values a_i several times (assuming that TLC does not attempt to optimize expressions and evaluates them left to right). Note that once we decide to support action actuals other than simple variable references, the translation proposed herein will no longer be optimal. For example, there is no reason to try several values of a formal if the actual is 0. I should definitely try to gain a deep understanding of how TLC works and have an idea of its future development before I implement **const** formals and non-trivial actuals; right now, my decisions are based merely on [4] and my intuition. *Note: Check if I don't say this elsewhere.*

2.10 Transitions

Transitions are translated as predicates which relate post-state to pre-state and are true if and only if the **where** predicate and the precondition are satisfied, and the post-state is the result of the **eff** clause on the pre-state. Only transitions with simple distinct variable actuals are supported at the time of writing (for the same reason that **const** action formals are not supported). Given the actuals $v_1, \ldots, v_n$, choose variables $c_1, \ldots, c_m$, and their respective sorts $C_1, \ldots, C_m$, a transition is translated as follows:

$$
\begin{array}{l}
T(v_1, v_2, ...) \;\triangleq \\
\quad \land\; \textit{where\_predicate} \\
\quad \land\; \exists\, c_1 \in C_1, \ldots, c_m \in C_m \;:\; \land\; \textit{precondition} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; \land\; s' = \textit{applyEffect}(s) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; \land\; \textit{so\_that\_predicate}(s')
\end{array}
$$

The effect clause (and any *program* as specified on page 41 of [3]) translates into a TLA+ expression equal to the the state record when the effect's statements are applied to the current state and any variables in the scope of the program. Statements also translate into a TLA+ expression equal to the state record when the statement (and any nested programs) is applied to the current, possibly intermediate state. If the program consists of a single statement, the translation is simply delegated to the statement. Otherwise, the program is translated using a sequence of n intermediate variables (where n is the number of statements in the program) as shown below:

$$
\begin{array}{lll}
\text{LET } s_1 & \triangleq & \mathit{statement}_1(s) \\
\phantom{\text{LET }} s_2 & \triangleq & \mathit{statement}_2(s_1) \\
\phantom{\text{LET }} \ldots & & \\
\phantom{\text{LET }} s_n & \triangleq & \mathit{statement}_n(s_{n-1}) \\
\text{IN } \; s_n & &
\end{array}
$$

2.11 Statements

2.11.1 Assignments

Assignments are translated by constructing a TLA+ record which is identical to the current state except the field in the lvalue, which is replaced with the value on the right-hand side of the assignment: $[s_i \text{ EXCEPT } s_i.\mathit{lvalue} = \mathit{value}]$. Value and lvalue are translated as described in Section 2.3. Since TLA+ supports shorthand notation for nested EXCEPT expressions, this translation also works for assigning values to elements in arrays and fields in tuples. However, as I realized while writing this document, the translation scheme then assumes a particular representation of arrays and tuples, and it is not general enough for custom data types (which may implement the `__[__]` and `__.*` operators arbitrarily). I will try to find a more general translation of assignments; this problem does not seem too hard.

Choose values are not supported yet, mainly because they introduce more complexity than I could handle in two months and because I was not sure how to represent them effectively. There are (at least) two ways to translate choices:

1. **Translate programs with choices as sets of states.** Instead of translating a program into state record which results from applying the program to the current state, translate it into a *set* of state records which can result from applying the program to the current state, taking all possible nondeterministic choices. This approach solves the problem in full generality (we could then support arbitrary programs in the `efect` clause); however, I am not sure wnat performance penalty we would have to pay.

2. **Convert choices to $\exists$ terms.** We could convert the choices to $\exists$ terms which would appear after the precondition. This approach cannot handle choices inside loops. Moreover, it is not clear that it would be more effectively checked by TLC than the first one.

2.11.2 Conditionals

If statements are translated using the TLA+ IF ... THEN ... ELSE ... construct[6]. If p_1 is the predicate after the **if** keyword, p_2 is the predicate after the first **elseif** keyword etc., the **if** statement is translated as follows:

$$
\begin{array}{l}
\text{IF } p_1 \text{ THEN } \mathit{program}_1 \\
\quad \text{ELSE IF } p_2 \text{ THEN } \mathit{program}_2 \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad \text{ELSE } \mathit{program}_n
\end{array}
$$

The predicates p_i and programs $\mathit{program}_i$ are translated using the current, possibly intermediate state. If the **else** keyword is omitted, $\mathit{program}_n$ is replaced with the current state (thus, if none of the predicate is satisfied, the **if** statement has no effect).

[6]However, it may change to the TLA+'s CASE construct, which follows the IOA's syntax more closely.

2.11.3 Loops

Loops are translated using a recursive function with domain 2^S where S is the set (multiset) specified after the **in** keyword or the set implicitly defined by the predicate after the **so that** keyword. The function is defined separately for each loop in the transition as follows:

$$f(set) = \begin{cases} current\_state & \text{if } set = \{\} \\ applyBody(f(set \backslash \{x\})) \text{ when } loop\_var = x, \text{ for some } x \in set & \text{otherwise} \end{cases}$$

This definition naturally translates to

$$\begin{array}{l} F[set \in \text{SUBSET } S] \triangleq \\ \quad \text{IF } set = \{\} \text{ THEN } current\_state \\ \qquad\qquad \text{ELSE LET } loop\_var \triangleq \text{CHOOSE } x \in set : \text{TRUE} \\ \qquad\qquad\qquad\qquad state \triangleq F[set \backslash \{x\}] \\ \qquad\qquad\quad \text{IN} \quad loop\_body(state) \end{array}$$

When TLC evaluates this function on S, it picks an arbitrary (but deterministic) order, in which the loop variable should iterate over the given set. The order in which the loop variable iterates over the given set cannot yet be treated as an extra source of nondeterminism, since we do not yet support nondeterminism in effect programs (see Section 2.11.1 above). However, we could probably instruct TLC to check if all orderings lead to the same result (this feature should be optional since it may have severe impact on the speed of model checking).

2.12 Invariants

An invariant is translated by defining a custom TLA+ operator in the module of its automaton and specifying the invariant's TLA+ identifier in the module's configuration file:

Module:	Configuration file:
$InvId \triangleq predicate(s)$	INVARIANT *InvId*

Invariant's predicate is translated as described in Section 2.3.

3 Constraining the IOA programs

TLC checks the specifications by exploring the state space, generating the states and verifying the given invariants or step simulations. In many instances, it will be necessary (my viewpoint) or desirable (the user's viewpoint) to constrain the automata processed by the tool in order to allow TLC check the specification at all or to focus the TLC's attention on a particular part of the specification. No matter which viewpoint one takes, the problem of constraining the specifications seems important in model checking and should not be taken lightly.

3.1 What to constrain

The following elements of an IOA program can be constrained:

1. **Automaton formals and underspecified external traits.** Often, IOA program defines a *family* of automata rather than a particular automaton. This happens either because the automaton is parametrized by formal types and parameters or because its traits are underspecified (in the sense that multiple sorts/operators satisfy the axioms stated in the

trait). Since TLC works with "concrete" values, it will be necessary to specify which automata from this family should processed[7].

2. **State space.** If the automaton has infinite or large number states, the state space will need to be constrained so that TLC can explore it in reasonable time and disk space.

3. **Transitions.** The number of transitions enabled at a given time may be infinite or too large to check and thus, may need to be constrained.

4. **Choices, quantifiers, and some predicates.** No matter how sophisticated the tool is, there will be times when TLC will not be able to model check the translated automaton effectively. For example, the choice **choose** `x:Int` **where** `1≤x ∧ x≤10` would naturally translate into an existential quantifier $\exists\, x \in \mathit{Int} \,:\, 1 \leq x \wedge x \leq 10$. However, TLC cannot evaluate this expression since the set *Int* is not enumerable ("enumerable" in TLC terminology means that the set is finite and TLC knows how to generate its elements). Thus, at times, choices, quantifier terms, and even predicates will need to be constrained or replaced so that TLC can model check the rest.

The last point is different from the first three since it results from the inability of the tool and TLC to verify certain statements expressed in logic, and not an "infinite" or "large" nature of an IOA program. This is not surprising since IOA is expressive enough to describe solutions to unsolved mathematical problems.

Besides the obvious solution of replacing the declarative parts which fail to translate effectively with an imperative program or a TLA+ expression which computes the desired function, this problem could also be solved by giving TLC a *hint* in the form of a set. Often, choices and quantifier terms are over an infinite sort, while only a few variables define the result of the expression or satisfy the choice's predicate. If the predicate in a choice or a quantifier is easy to evaluate, TLC could find the value of the expression by iterating over a smaller set provided by the user. Alternatively, we can think of this set as hint which helps TLC focus on the interesting values. In the above example (**choose** `x:Int` **where** `1≤x ∧ x≤10`), giving the tool a hint that all the values of `x` which satisfy the predicate will be in the set $-20..20$ would suffice to obtain the desired result.

We need to work with more examples to find out if hints have any real value. However, they seem to have two advantages to the former approach:

1. They are easy to write. Expressing the set of all primes less than one hundred in TLA+ or in the language of simulator's scheduling blocks can be time-consuming. Writing the set 1..100 and letting TLC find which of its elements are primes is much easier.

2. They are safe. When the declarative expression is replaced with with an imperative program or a TLA+ expression, there is a danger that the user makes mistakes and misses precisely the boundary cases which he or she is trying to check.

3.2 Means of constraining IOA programs

I explored the available language tools for constraining IOA programs, and came up with the following options:

[7] Valid values for TLC are literals, model values, and values obtained by applying supported TLA+ constructs to valid values (subject to certain restrictions due to memory limitations and Java built-in type `int`). For example, $\textsc{subset}\ [1..10 - > \{\textsc{true}, 0\}]$ is a valid value, while $\textsc{choose}\ x : x \neq \textsc{false}$ is not. *Editorial note: check this in [4].*

1. **Predicate (expressed in IOA or TLA+).** This option seems most useful for constraining the state space globally (for example, to limit the queue length in a buffer) but could serve for other purposes as well. I strongly prefer predicates expressed in IOA for two reasons:

   - by expressing predicates in IOA, we make the tool less dependent on other languages and tools, and
   - to make predicates in TLA+ meaningful, we would need to define a scheme to name IOA variables in TLA+ expressions.

   In many cases, a direct translation of the IOA predicate (as described in Section 2.3) will be checkable by TLC. However, we would need to improve the translation of quantifier terms (e.g. by handling at least the common cases). On the other hand, any progress that we make here will likely be reused in other parts of the tool.

2. **Set (expressed in IOA or TLA+).** This option seems most useful for constraining the formals and the initial state, and to define hints as described in Section 3.1. For sets expressed in IOA, we will need to ensure that their TLA+ translation is enumerable. In case of sets expressed in TLA+, it will be up to the user to ensure that TLC can enumerate them.

3. **An imperative program.** This option would most likely be used to replace an expression translation of which cannot be evaluated by TLC. It is a possible approach; however, it seems very laborious for the user (we see this in simulator's scheduling blocks which are often longer than the program itself).

4. **Fixed set of parameters for each sort.** Each sort we would implement a fixed set of parameters (e.q. "maximum length" for a sequence) which would be set by the user, if necessary, to restrict a state variable or a formal. The tool would then generate the appropriate TLA+ code which takes the constraints into an account. This is a possible approach; however, it seems to require too much coding and is by far least expressive.

4 Design

4.1 Design goals

In my design, I focused on the following goals:

- **Modularity** Modularity promotes code reuse and helps in decreasing the complexity of the system by lowering the number of dependencies. By keeping the system modular, we will be able to divide the development of the tool amongst multiple people if needed.

- **Simplicity** Keeping the design simple ensures that it develops in the desired direction. Keeping the design simple is especially important in the early stages of research, since the system's specification keeps changing and is vastly incompete. I focused specifically on not overdesigning the tool to make sure that I would be able to implement it in time. Having a working version of the tool early will help us shape the ideas more clearly and get early feedback from the users.

- **Flexibility** The design should easily adapt to minor changes in IOA syntax and to moderate changes in the translation scheme.

- **Extendibility** It should be very easy to introduce new sorts and operators.
- **Code reuse** The design should attempt to reuse code developed earlier in the group, since reusing the code decreases the development time dramatically. If possible, I would rather improve the design of present frameworks than create new ones.
- **Clean design** I would like to come close to the ideals of the 6.170-style specification of ADTs and their methods. While this goal may not be fully attainable in real-world software development, I would like to achieve it in the long run for classes with stable design.

4.2 System specification

The scheme for translating automata and the related assertions was described in detail in Section 2. This section describes the overall functionality expected from the tool. Since we do not have any "official" customers, I was guided mainly by other tools which are being developed our group and by my own judgement of what would be useful for future users.

The tool translates intermediate language (IL) output by the front-end which parses the IOA program and ensures that it is syntactically correct (the front-end tool is written by S. Garland). The tool processes a single `.il` file which contains declarations of program's sorts, operators, primitive automata, and assertions. The tool should ignore any extensions to IOA which it does not recognize (e.g. the simulator's scheduling blocks). If instructed, the tool reads a model (configuration) file, in which the user constrains the specification and specifies the assertions to check.

For each automaton, the tool outputs one module file (with the translation of the automaton and its invariants) and the corresponding configuration file. In future, we may decide to produce a more structured output, in which, for example, the custom operators defined by program's traits are in a common file, and the automata and assertions are translated into two separate files).

The tool focuses merely on translating the specified .il file; it does not attempt to execute TLC or interpret its output (this task is left up to the user or a third-party program).

4.3 Design structure

The design can be split into three parts:

1. Classes which represent the elements of the source parse tree and generate the TLA+ source files. These include classes for the IOA program, IOA automata, terms, actions, states, and transitions.
2. Classes which implement operators and sorts in TLA+ and handle their registration in the implementation registry. Some of these classes stem from the registry (see [6]), some are designed from scratch.
3. Classes which represents the elements of the configuration file. Their role is to configure and appropriately restrict the IL elements in the source parse tree.

The absence of target tree makes the design noticably different from the design of the code generator. By generating the target tree first, the code generator can conveniently apply various transformations to the code before it is output to a Java source files. The approach taken in code generator is cleaner and more modular; however, it is not yet worth the development cost for my tool. The present design suffices to cleanly implement the features expected from the tool now and in the near future. Having a working version of the tool will be invaluable since we can now start working with bigger examples. If the tool begins to grow considerably or if we decide to implement feature which manipulate with the translation, we can move to the more modular design.

4.4 The source tree classes (package `ioa.tla.source`)

Classes in this package represent the nodes in the parse tree. The classes utilize the framework defined in the `ioa.il` package by extending the appropriate Basic... classes (e.g. `BasicAction`). The Basic... classes implement the corresponding interfaces (e.g. `Action`), which provide methods for accessing the children of the given node and its parameters, such as name and scope. To avoid namespace collision, the TLA+ classes start with "TLA," followed by the element type (e.g. `TLAAction`). The classes are instantiated by the `ILParser` class using the tool's factory.

Each class defines one or more methods which output a part of the TLA+ source file (see the JavaDoc documentation for a complete list of classes and their specifications). In general, nodes translate into the corresponding part of the TLA+ module (for example, state table translates to the initial-state predicate). The methods call accessors defined by the Basic... superclass to obtain the references to node's children. This poses a slight problem since the methods need to up-cast returned values each time they access their children, which is rather ugly and error-prone. I tried to avoid this problem by using the `toSourceForm` method built-in to the package or a *Visitor* design pattern. However, these options had their own disadvantages: `toSourceForm` method was insufficient the translation methods need to take arguments, and the visitor pattern would result in a less modular design.

The source tree package also defines four interfaces: `TLAAutomaton`, `TLAStatement`, `TLATerm`, and `TLAValue`. These interfaces declare translation functions common to all automata, statements, terms, and values. Thus, when a transition needs to translate its statements, it can treat them uniformly by casting each statement to `TLAStatement` and calling its approppropriate translate method. This results in greater extendibility and flexibility of the design[8].

4.5 Implementation classes (package `ioa.tla.impl`)

4.5.1 `TraitImpl`

The base class `TraitImpl` represents an implementation of a trait. At present, the main purpose of trait implementations is to insert the implementations of sorts and operators defined by the corresponding LSL traits into the registry (for this purpose, `TraitImpl` defines two abstract methods, `installSorts` and `installOps`). Thus, trait implementations act as *registration classes* for the registry (see [6]). However, the design of trait implementations is more structured and goes well beyound connecting the registry with the sort/operator implementations. For example, the notion of *installer* is missing completely; the methods which simplify insertion of new implementations or constructors into the registry are implemented directly in the base class `TraitImpl`.

Trait implementations have an option of inserting TLA+ code into the resulting source file. This is accomplished with an `init` method, which takes a single parameter, the output stream. This way, the trait implementations can define custom operators and reuse them later during the translation process. This approach can actually increase the size of the output file since the custom definitions appear in it even if the operators are never used. However, file size is insignificant, and these definitions improve the readability of the translation and the speed of model checking since even if the operands appear multiple times in the translation, they are evaluated only once.

`TraitImpl` is subclassed by three groups of classes:

- **Implementations of built-in traits.** Each built-in trait has a corresponding trait implementation (for example, the trait `Sequence.lsl` has a corresponding trait implementation `SequenceTrait`).

[8] Tthe LP tool treats statements, terms, and values in the same way.

- **Implementations of trait shortcuts.** Trait shortcuts (enumeration, tuple, and union) have corresponding trait implementations, which are instantiated with proper fields by the class `TLASpec` based on the shortcuts defined in the IL tree. Thus, the design avoids the `DynamicRegistry` and the `DynamicRegistration` classes completely (in my opinion, they would add unnecessary complexity to the design).
- **Collections of traits.** The class `CompoundTrait` represents a collection of traits, and is used by `TLASpec` to conveniently store implementations of all traits in the program into a single object. The class `BuiltinTrait` is a special case of the `CompondTrait` which represents the collection of IOA's built-in traits.

It can be foreseen that additional trait implementations will be designed in future (for example, external traits implemented by the user or a class which can construct itself based on the specified LSL trait). The design will need to be extended before we proceed in this direction to ensure that we handle certain features of LSL traits correctly (mainly trace inclusion and parametrization).

Trait implementations are bound to the registry in which they install operator and sort implementations. This was probably a wrong decision, since trait implementations do not need to be a priori associated with a single registry. It is also questionable which registry should the `CompoundTraitImpl` be bound to. The best approach will probably be to let the `installOps` and `instalSorts` methods take the registry as an argument (as it is done in the code generator).

4.5.2 `TLAImpl`

TLAImpl is an abstract base class for sort and operator implementations. It declares methods common to both sorts and operators, including `clone` (to allow prototyping of implementations in constructors, see Section 4.5.5 below) and `init` (to allow the implementations write auxiliary definitions to the translation's module).

4.5.3 `TLASortImpl`

The class `TLASortImpl` represents an implementation of a sort. Each sort implementation is associated with a single `Sort` object in the parse tree (this sort object is used, for example, to lookup subsorts and their implementations). Each sort has a separate class which extends `TLASortImpl` and implements its abstract methods.

As discused in Section 2.4, sorts are implemented as sets. Since a given set may not be used in all expressions equally efficiently, the sort defines two sets: a *checkable set* and a *restricted set* (methods `getCheckableSet` and `getRestrictedSet`). Checkable set is one which can be appear on the right hand side of the $\in$ operator and lets TLC verify that the value belongs to a given sort. Restricted set is one which can appear on the right hand side of the $\in$ operator to assert that a given variable is in the given sort or its subset. Restricted sets are used in the present design to constrain the formals and the initial state; however, this part of the design will change once we fully design the scheme for constraining the primitive automata.

4.5.4 `TLAOpImpl`

The class `TLAOpImpl` represents an implementation of an operator. Similarly as sort implementations, each operator implementation is associated with precisely one `Operator` object. All operator implementations extend `TLAOpImpl` and need to override the `applyTo` method which translates the specified IL operands to TLA+ and applies the operator's implementation to them.

Virtually all operators are implemented with the `TemplateOpImpl` class. `TemplateOpImpl` takes a string template (which implements the operator) and substitutes any substrings in the form "$n" (where n denotes the operand) for the translation of the respective operand[9]. Since most built-in IOA operators have a direct TLA+ equivalent, this class provides a very concise way to implement operators for built-in datatypes (1 line per operator in most cases). We will see later if it suffices to implement custom traits, which may not have straightfoward translations.

4.5.5 Constructors

In order to use the implementation registry `ConstrImplRegistry` (see [6]), the tool defines two registry classes: `TLAOpConstructor` and `TLASortConstructors`. These classes are fairly simple: they implement the `construct` method which returns a clone of the prototype operator or prototype sort implementations specified when the constructor is created. The remaining work (e.g. matching the registry templates) is handled by the superclasses `OpConstructor` and `SortConstructor`.

4.6 Classes for constraining the model (package `ioa.tla.model`)

In order to complete the first version of the tool and gain some experience with constraining IOA programs, I designed and implemented a prototype scheme for constraining primitive automata. This scheme allows the user to constrain the automaton and action formals and the initial state by restricting them to a given set of values (see Section 3.2). The constraints are specified in a model (configuration) file with a scheme-like syntax and grouped by the target automaton. The implementation comprises of classes in the `ioa.tla.model` package which parse the given s-expression and apply the constraints to the given node in the source tree. Both the design and its implementation may change.

4.7 Naming

The main reason for developing a naming scheme is to ensure that identifiers in the output translation are "correct". Section 2 described how an IOA program is translated to TLA+; however, it often left aside the issue of assigning a name to the defined constants, variables, operators in the translation.

The tool's output can be used in various ways; thus, the requirements on the naming scheme vary. At the minimum, the names of newly defined variables and operators need to be unique in their scope and need to be valid TLA+ identifiers (as defined in [4]). If the tool's output is processed by some other tool (i.e. it needs to be merely machine-readable), this requirement suffices. If the tool's output is read by the user, the identifiers also need to be identical or at least similar to the identifiers in the original IOA program.

The current design attempts to satisfy the tighter requirements to make the output translation human-readable. This aim is accomplished with a centralized name registry implemented by the class `NameRegistry` and its subclass `TLANameRegistry`.

The class `NameRegistry` is an abstract base class, instances of which are able to return unique identifiers. The class defines an abstract method `newName` which takes as a parameter the suggested name and returns a name different from the previously returned names which is similar to the suggested name. In addition, the class has two static methods, `setInstance` and `getInstance` to set/get the global instance of the `NameRegistry`. The class `TLANameRegistry` implements `newName`

[9] Note that the "template" here is different from the template used by the registry.

such that it returns a valid TLA+ identifier (for simplicity, it filters out all non-letter characters from the suggested string and ensures that it does not return a TLA+ keyword).

Experience showed that while this design is a good short-term solution, it may be suboptimal for a serious user because often, variables which have the same identifier in the original program have different names assigned to them in the translation (because `newName` always returns a "fresh" identifier – even in cases when the identifiers' scopes are distinct). If necessary, we may alter the design to resolve this issue.

4.8 Design limitations, possible improvements

I tried to come up with a good design and resolve all its limitations as they appeared. However, due to time constraints, the design may be suboptimal in the following parts:

- **Formals.** In the present design of the `ioa.il` package, formals are stored simply as a `List` of `Term` objects. While this may be an appropriate solution in the general framework of `ioa.il`, my tool may need to encapsulate formals in a class which defines convenient accessors.
- **Interface between `TLASpec` and `TLAAutomaton`.** The interface between `TLASpec` and `TLAAutomaton` is a little unclear in the present design. `TLASpec` constructs the trait implementations (including trait shortcuts and automaton formals) and passes them to `TLAAutomaton`, which initializes the traits and installs their operator and sort implementations into the registry. This behavior is not documented anywhere; furthermore, the work may not be split between the classes as dictated by the logic of the system.
- **One variable, multiple scopes.** I based my design on the assumption that each variable is declared only once in the IL input. I will need to think of what happens when variables are copied to other scopes and alter the design as (if) needed. The same problem appears with sorts and operators.

5 Future plans

I plan to make the following contributions to the project in the near future (the tasks are listed approximately in the order of their importance):

1. Ensure that the tool complies with the recent change to the IL (including multiple declarations of the same variable as described in the previous section). Implement invariants.
2. Develop the scheme for constraining primitive automata (together with Nancy Lynch and Stephen Garland).
3. Work on more advanced examples, including the transport protocol in [2] and distributed algorithms in [5].
4. Extend the design and implementation of the tool as needed to handle these examples.

In addition, I will contact Yuan Yu with questions mentioned in this document and bug reports and read the manual for the newest version of TLC to see if we could take advantage of some of its new features.

A Hand-translated examples

The table below summarizes the examples that I hand-translated at the beginning of the summer 2001. Most of them are taken from [3]. The translations are located in the CVS repository in the directory `IOA_Toolkit/Examples/tla/hand`.

Example	Explanation
Peasant.ioa	"Russian peasant multiplication" example by Andrej Bogdanov.
Adder.ioa	An automaton which adds integers.
Choice.ioa	An automaton which demonstrates nondeterministic choice in the initial state.
Shuffle.ioa	An automaton which shuffles cards.
LossyChannel.ioa	Message-passing channel which loses subsequence of messages in the buffer on crash.
Distribute.ioa	An automaton which distributes input integers into three sets.
Multicast.ioa	An automaton which multicasts packets to the specified sets of nodes.
LossyBuffer.ioa	A channel which loses one message for each message it delivers.
ComposedLCR.ioa	Leader election algorithm.

B Translations of built-in datatypes

This appendix describes implementations of built-in datatypes and their operators. The datatypes are listed in the order, in which they occur in [3]. Note that every datatype implicitly defines the infix operators =, ≠, and the conditional operator **if** `p` **then** `s1` **else** `s2`. These operators are translated to TLA+'s equivalent =, ≠, and IF p THEN s_1 ELSE s_2. If necessary, the TLC's operator *Assert* is used in the implementation of the operator to verify the operands (e.g. verify that the integer passed to the `nat` operator is greater or equal to zero).

B.1 Booleans

The type `Bool` is implemented using the TLA+ set BOOLEAN and the standard boolean operators.
Checkable set: BOOLEAN
Restricted set: BOOLEAN

B.2 Integers

The type `Int` is implemented by the TLA+ set *Int* and the operators defined in the standard module *Integers*. All operators either have direct TLA+ equivalents or short implementations with a conditional.
Checkable set: *Int*
Restricted set: unsupported

B.3 Natural numbers

The type `Nat` is implemented using the TLA+ set *Nat* and the operators defined in the standard module *Naturals*. All operators either have TLA+ euqivalents or short implementations with a conditional.
Checkable set: *Nat*
Restricted set: unsupported

B.4 Real numbers

The type `Real` is not supported yet since at the time of writing, TLC does not implement the standard module *Reals*.

B.5 Characters and strings

The type `Char` is not supported yet. The type `String` is supported only partially in a sense that a given element can be verified if it is a string; however, no comparison and sequence operators are supported. This is mainly due to the fact that TLC does not yet treat strings as sequences of characters (as defined by TLA+ syntax); thus, it is impossible implement string comparison and sequence operators. Theoretically, it would be possible to implement characters using string literals and strings as sequences of these literals; however, this implementation would be quite inefficient. We can consult this issue with Yuan Yu and see if TLC will support strings fully in near future.
Checkable set: STRING
Restricted set: unsupported

B.6 Arrays

Arrays are implemented as functions with the domain equal to the index set `I` (or in case of 2-dimensional array, a cross product of `I \X I`) and range equal to a subset of the element set `E`. This implementation is very efficient; all operators have direct translation into TLA+ (see table below). However, it is a rather restricted since the domain cannot be an infinite set (TLC cannot construct a function with an infinite domain). If this turns out to be very limiting, we can implement infinite arrays in the same way this is done in the code generator (by storing the values of only those elements which are different from the initial `constant` value).

Operators for `Array[I, E]`	Sample input	TLA+ translation
`constant`	`constant(e)`	$[i \in I \mapsto e]$
`__[__]`	`a[i]`	$a[i]$
`assign`	`assign(a, i, e)`	$[a \text{ EXCEPT } ![i] = e]$

Operators for `Array[I, I, E]`	Sample input	TLA+ translation
`constant`	`constant(e)`	$[\langle i, j \rangle \in (I \times I) \mapsto e]$
`__[__]`	`a[i, j]`	$a[i, j]$
`assign`	`assign(a, i, j, e)`	$[a \text{ EXCEPT } ![i, j] = e]$

Checkable set: $[I \rightarrow E]$, $[I \times I \rightarrow E]$ (assuming that both `I` and `E` are checkable)
Restricted set: $[I \rightarrow E]$, $[I \times I \rightarrow E]$ (assuming that both `I` and `E` have restricted sets)

B.7 Finite sets

Finite sets are implemented using the operator SUBSET ($\text{SUBSET } S \triangleq 2^S$) and the *Cardinality* operator from the standard module *FiniteSets*.

Operators for `Set[E]`	Sample input	TLA+ translation
`{}`	`{}`	$\{\}$
`{__}`	`{e}`	$\{e\}$
`insert`	`insert(e, s)`	$s \cup e$
`delete`	`delete(e, s)`	$s \setminus \{e\}$
$\in$	`e \in s`	$e \in s$
$\cup$, $\cap$	`s \U s', s \I s'`	$s \cup s'$, $s \cap s'$
`-`	`s - s'`	$s \setminus s'$
$\subset$	`s \subset s'`	$s \in \text{SUBSET } s' \land s \neq s'$
$\subseteq$	`s \subseteq s'`	$s \in \text{SUBSET } s'$
$\supset$	`s \supset s'`	$s' \in \text{SUBSET } s \land s \neq s'$
$\supseteq$	`s \supseteq s'`	$s' \in \text{SUBSET } s$
`size`	`size(s)`	$\mathit{Cardinality}(s)$

Checkable set: $\text{SUBSET } E$ (assuming that `E` is checkable)
Restricted set: $\text{SUBSET } E$ (assuming that `E` has restricted set)

B.8 Multisets

Multisets are implemented using the operators defined in the standard module *Bags*.

Operators for `Mset[E]`	Sample input	TLA+ translation
`{}`	`{}`	$EmptyBag$
`{__}`	`{e}`	$SetToBag(\{e\})$
`insert`	`insert(e, s)`	$s \oplus SetToBag(\{e\})$
`delete`	`delete(e, s)`	$s \ominus SetToBag(\{e\})$
$\in$	`e \in s`	$BagIn(e, s)$
$\cup$, $\cap$	`s \U s', s \I s'`	unsupported
`-`	`s - s'`	$s \ominus s'$
$\subset$	`s \subset s'`	$s \sqsubseteq s' \wedge s \neq s'$
$\subseteq$	`s \subseteq s'`	$s \sqsubseteq s'$
$\supset$	`s \supset s'`	$s' \sqsubseteq s \wedge s \neq s'$
$\supseteq$	`s \supseteq s'`	$s' \sqsubseteq s$
`size`	`size(s)`	$BagCardinality(s)$
`count`	`count(e, s)`	$CopiesIn(e, s)$

Checkable set: unsupported
Restricted set: unsupported

B.9 Sequences

Sequences are implemented using the native operators defined in the standard module *Sequences*.

Operators for `Seq[E]`	Sample input	TLA+ translation
`{}`	`{}`	$\langle\,\rangle$
$\vdash$	`s |- e`	$Append(s, e)$
$\dashv$	`e -| s`	$\langle\, e \,\rangle \circ s$
$\Vert$	`s || s'`	$s \circ s'$
$\in$	`e \in s`	$\exists\, i \in 1..Len(s) \;:\; s[i] = e$
`head`	`head(s)`	$Head(s)$
`last`	`last(s)`	$s[Len(s)]$
`init`	`init(s)`	$[i \in 1..(Len(s) - 1) \mapsto s[i]]$
`tail`	`tail(s)`	$Tail(s)$
`len`	`len(s)`	$Len(s)$
`__[__]`	`s[n]`	$s[n]$

Checkable set: $Seq(E)$ (assuming that `E` is checkable)
Restricted set: unsupported

B.10 Mappings

Finite partial mappings `Map[D, R]` are implemented as functions with domain being a subset of `D` and range a subset of 'R'. An element is defined if and only if it is in the domain of the function, and its image is the value of the function when the function is applied to the given element.

Operators for `Map[D, R]`	Sample input	TLA+ translation
`empty`	`empty`	$\{\}$
`__[__]`	`m[d]`	$m[d]$
`defined`	`defined(m, d)`	$d \in \text{DOMAIN } m$
`update`	`update(m, d, r)`	$[i \in (\text{DOMAIN } m) \cup \{d\} \mapsto$ $\quad \text{IF } i = d \text{ THEN } e \text{ ELSE } m[i]]$

Checkable set: unsupported

Restricted set: unsupported (will be supported in future for maps, in which `D` and `R` have restricted sets)

B.11 Enumerations

Enumerations are implemented as sets of string literals. Each element is converted to a valid TLA+ *identifier* (to simplify the implementation of the `succ` operator) such that all literals remain unique[10].

```
type enum = enumeration of e1, e2, ..., en
```

Operators for an **enumeration**	Sample input	TLA+ translation
`ei`	`e1`	$''e_1''$
`succ`	`succ(e)`	$[e_1 \mapsto'' e_2'', \ldots, e_{n-1} \mapsto'' e_n''][e]$

Checkable set: $\{''e_1'', \ldots,'' e_n''\}$

Restricted set: $\{''e_1'', \ldots,'' e_n''\}$

B.12 Tuples

Tupples are implemented using the built-in TLA+ type *record* (*tuples* in TLA+ terminology mean sequences of fixed length). The element identifiers are converted to valid TLA+ identifiers. *Note: tuples are not implemented yet.*

```
type atuple = tuple of e1: s1, e2: s2, ..., en: sn
```

Operators for a **tuple**	Sample input	TLA+ translation
`[__, ..., __]`	`[v1, ..., vn]`	$[e_1 \mapsto v_1, ..., e_n \mapsto v_n]$
`__.ei`	`t.e1`	$t.e1$
`set_ei`	`set_e1(t, v1)`	$[t \text{ EXCEPT } !.e_1 = v_1]$

Checkable set: $[e_1 : s_1, \ldots, e_n : s_n]$ (assuming that sorts s_i are checkable).

Restricted set: $[e_1 : s_1, \ldots, e_n : s_n]$ (assuming that sorts s_i have restricted sets).

B.13 Unions

Unions will be implemented as functions with domain consisting of a single element: the tag (or more precisely, a string with a valid TLA+ identifier of the tag). The range of the function is the current value in the union. This representation has nice properties, namely it is very simple, and TLC prints an error if the programm attempts to access a member variable with tag different from the one in the union. *Note: unions are not implemented yet.*

```
type aunion = union of e1: s1, e2: s2, ..., en: sn
```

Operators for a **union**	Sample input	TLA+ translation
`ei`	`e1(v1)`	$[e_1 \mapsto v_1]$
`__.ei`	`u.e1`	$u.e_1$
`tag`	`tag(u)`	$\text{CHOOSE } t \in (\text{DOMAIN } u) : \text{TRUE}$

Checkable set: $\text{UNION } \{[e_1 : s_1], \ldots, [e_n : s_n]\}$ (assuming that sorts s_i are checkable).
Note: this feature has not yet been tested

Restricted set: $\text{UNION } \{[e_1 : s_1], \ldots, [e_n : s_n]\}$ (assuming that sorts s_i have restricted sets).

[10] This is accomplished with a fresh instance of `TLANameRegistry` (see Section 4.7)

C Bibliography

References

[1] Bryan Ford, Stanislav Funiak, Jennifer Liang, Ken Takusagawa. *Object-Oriented Programming Group.* (6.897 Case Study)

[2] Stanislav Funiak. *Routing Protocol for the CyberBeanie Toy Network.* (6.033 Design Project) `http://web.mit.edu/stanf/6033dp1.ps`.

[3] Stephen J. Garland, Nancy A. Lynch, Mandana Vaziri. *IOA: A Language for Specifying, Programming, and Validating Distributed Systems.*

[4] Leslie Lamport. *Specifying Concurrent Systems with TLA+.*

[5] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[6] Michael Tsai. *Abstract Data Types (ADTs) for IOA Code Generation.*