Two Translators for the IOA Toolkit

Holly Reimers

May 8, 2000

1 Summary

This paper presents two translator programs, ILUnparser and ILtoSource, that have been developed for the IOA Toolkit. IOA is a programming language based on the I/O automaton model for distributed systems. The IOA Toolkit is designed to support development of IOA programs from high level designs to system specific implementations. Translators are necessary to the IOA Toolkit to return intermediate language or source code listings to the designer for examination and editing. The translator ILUnparser returns the intermediate language code from the internal representation of IOA programs, and ILtoSource translates the internal representation to IOA source code.

2 Background

IOA is a language for specifying and designing distributed system applications and is formally based on the I/O automaton model for distributed systems. IOA has been designed to facilitate verification and simulation of distributed programs. The IOA Toolkit currently under development is designed to test the feasibility of generating, testing, and verifying distributed programs from such models. Members of this toolkit include a front end IOA parser and static semantic checker, a simulator for IOA programs, a composer for transforming a composite design to a primitive representation, and a code generator for generating source code for a target language on a target distributed system. The ultimate goal of the IOA Toolkit development project is to build an IOA-based programming environment for specifying, implementing, and testing distributed system

```
automaton <name>
      signature
            <input, output, internal> <actions>
      states
            <variables>: <type> := <initial value>
      transitions
            <input, output, internal> <actions>
                  pre <variables>
                  eff <value>
      tasks
            {<actions>, <variables>}
```

Figure 1: Structure of a Typical IOA Program

applications, supporting them from high level designs to functioning implementations [1].

The tools of the IOA Toolkit accept an intermediate language representation of IOA specified designs as input. IOA specified programs are first parsed into this intermediate language by the front end of the IOA Toolkit. This intermediate language representation is further parsed by the Toolkit's intermediate language parser into an internal (Java) representation. To aid designers in understanding changes that have been made to their programs by tools in the IOA Toolkit, translators are needed to return code listings for both source and intermediate languages from the internal representation of IOA programs in the IOA Toolkit [2]. Two translators have been developed: ILUnparser returns the intermediate language code listing from the internal representation, while ILtoSource returns the source code listing of these programs. Both translators contain methods supporting the current version of IOA and its internal representation in the IOA Toolkit.

3 Representations of IOA Programs

3.1 IOA Language

The structure of a primitive IOA program is shown in figure 1. The program sections are declared as follows [3]:

- Automaton declaration, including the name of the program and its formal parameters.

```
automaton Toggler
  signature
    internal action1, action2
  states
    alpha: Bool := false
  transitions
    internal action1
      pre alpha
      eff alpha := false
    internal action2
      pre ~alpha
      eff alpha := true
```

Figure 2: Code Listing for IOA Program Toggler

- Signature declaration, stating the actions performed in the program.
- States declaration, naming the state variables and their initial values.
- Transitions declaration, describing what is performed for each action.
- Tasks declaration, an optional section dividing the program's actions into sets of tasks to be performed together.

These sections correspond to the Automaton, ActionTable, StateTable, Action, and Task components of the internal representation, described in section 3.3 below.

3.2 Example: Toggler Program

Toggler, a simple IOA program, is presented in this paper to illustrate the functionality of both translators. Toggler contains one state variable, alpha, and two actions which toggle alpha between the Boolean values true and false. The IOA source code for this program is shown in figure 2.

3.3 Internal Representation

The internal representation of IOA programs consists of Java classes that represent components of an IOA design. Classes include Automaton, Action, State, Transition, and Task, among others, that correspond to components of typical IOA programs [3] (see figure 1 above.) An object tree for the internal representation of a typical IOA program is shown in figure 3. An Automaton object contains a String representing the name of the program, a StateTable representing the states for that program, and an ActionTable representing the actions for that program. ILUnparser and ILtoSource contain methods that traverse this tree of objects to access identifier information (names, etc.) from those objects to produce code listings describing the program. Figure 4 illustrates a snapshot of the internal representation of Toggler.

3.4 Intermediate Language

The intermediate language for programs in the IOA Toolkit is an S-expression language. Tools in the IOA Toolkit parse the intermediate language representation into the internal representation and perform work on the internal representation. The intermediate language representation can be saved to a file during the design process to facilitate further work with the IOA Toolkit. The intermediate language representation of Toggler is shown in figure 5.

4 Design Requirements

The requirements of both translators call for code to be listed based on the correspondence between the IOA Toolkit's internal representation of an IOA program and the intermediate or source language. The following design requirements were considered during the implementation of the translators:

- Code listings must be semantically and syntactically accurate descriptions of the internal representation.
- Code listings must be complete and resolved of ambiguities.
- Translators must be reversible; that is they must return identical code listings when the output code of a translator is parsed into the IOA Toolkit and translated again with the same translator.
- Code listings must be formatted correctly for language and readability requirements.

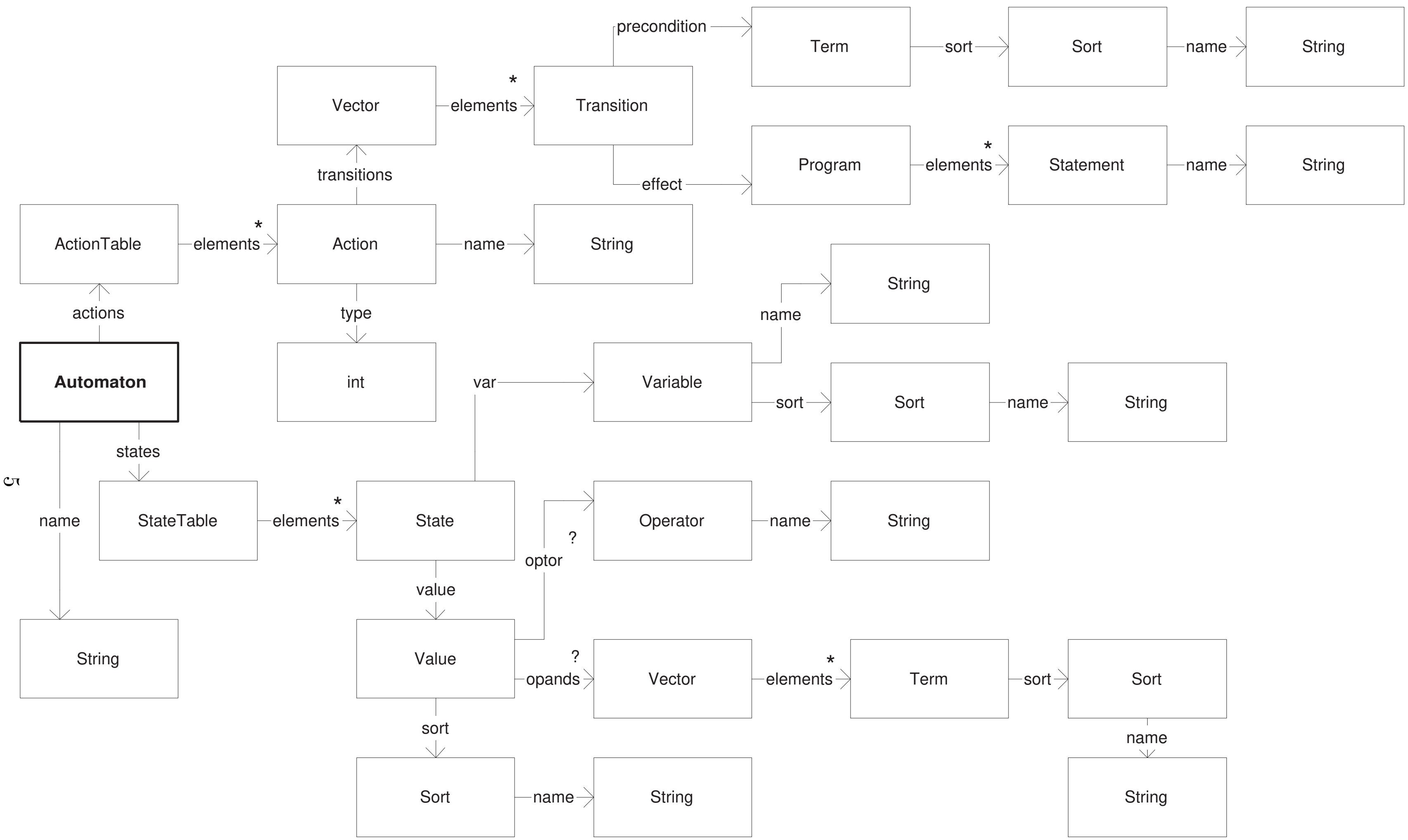

Figure 3: Object Tree for the Internal Representation of a Typical IOA Program. Names of actions, variables, and program statements can be obtained for use by translator programs by calling accessor methods (getName, etc.) of component objects in the internal representation.

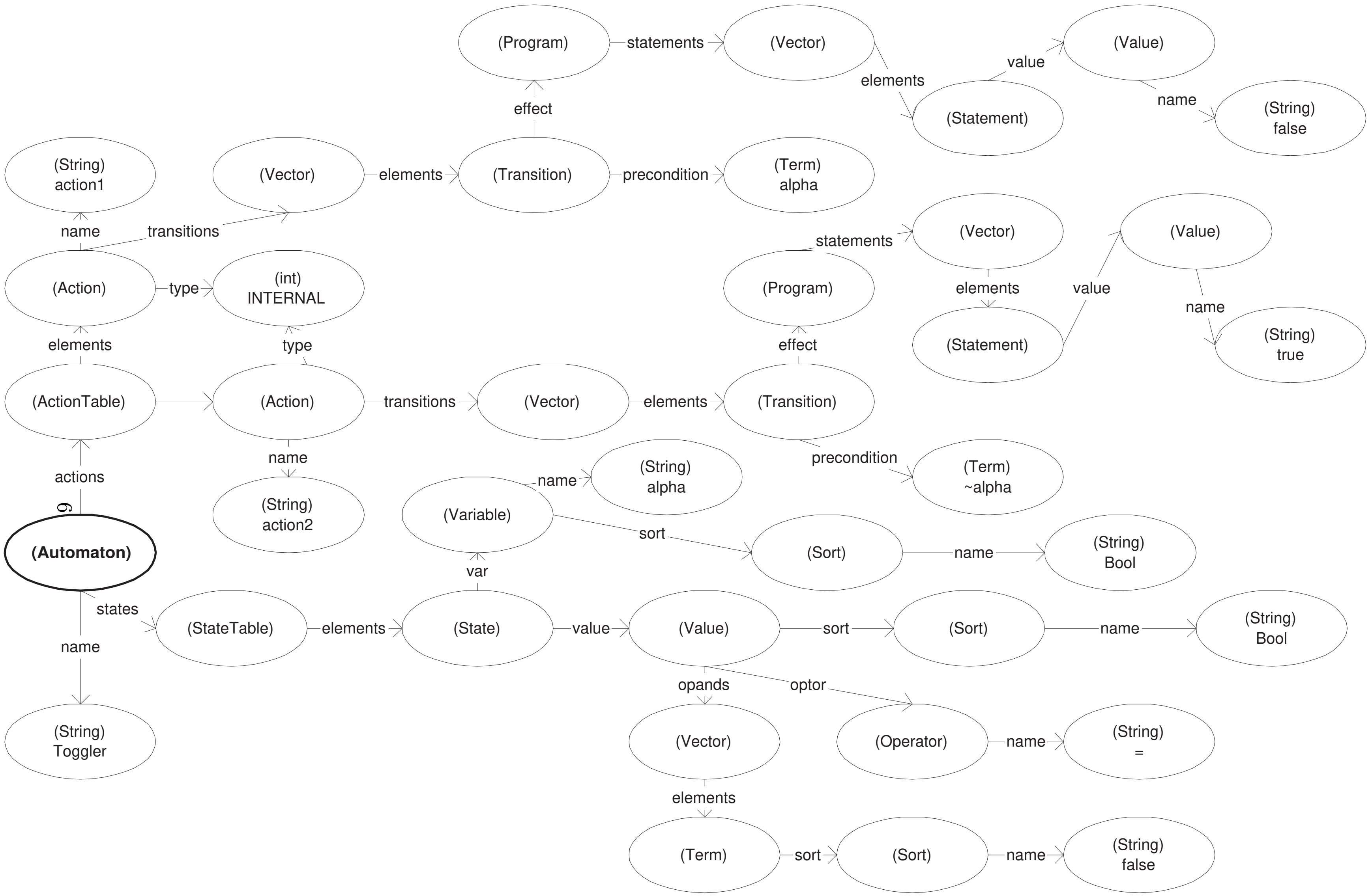

Figure 4: Object Snapshot of Internal Representation of Toggler Program. Specific identifier names are shown within their containing objects. The translators traverse this tree of objects to access the specific identifier names to produce code listings.

```
(ioa
 ((sorts (s1 "type" ()) (s0 "Bool" ()))
  (ops (op9 (id "true") (() s0) (scope 3))
   (op7 (prefix "~") ((s0) s0) (scope 3))
   (op4 (id "false") (() s0) (scope 3)))
  (vars (v0 "alpha" s0 (scope 37))))
 (automaton "Toggler"
  ((actions (a1 internal "action1" (subscope 38 37))
    (a2 internal "action2" (subscope 39 37)))
   (states (v0 op4))
   (transitions
    (a1 (pre v0) (eff ((assign v0 op4))) (subscope 40 37))
    (a2 (pre (apply op7 v0)) (eff ((assign v0 op9))) (subscope 41 37))))
  (subscope 37 3)))
```

Figure 5: Intermediate Language Representation of Toggler Program

The reversibility specification described above preserves the order of identifier declaration in the original program. While this requirement may not be necessary in most cases, misordered identifier declarations may lead to circular reference errors during processing by tools in the IOA Toolkit. In addition, preserving order of identifier declaration helps the designer understand changes that may have been made to the code by tools in the IOA Toolkit.

5 Implementation

Both translators were implemented using Java 1.1. Both contain a public static method that takes the internal representation of an IOA program and printer object as parameters. The printer is modified with the code listing created from string representations of program components. The public methods of both translators are supported by private methods to handle the translations of program components.

5.1 ILUnparser

The iprinter utility is used to format and print the S-expression code on which the IOA Toolkit's intermediate language is based. This code is created using accessor methods of the objects representing components of the program in the internal representation. To meet the specification for completeness and accuracy of code

```
protected static void unparseStateTable(StateTable st, iprinter ip) {
  ip.align();
  ip.put("states");
  Enumeration e = st.enumerate(); // get the states
  while (e.hasMoreElements()) { // translate each state
      unparseState((State)e.nextElement(), ip);
  }
  ip.end();
}


protected static void unparseState(State s, iprinter ip) {
  // <state> ::= (<varId> <value>?)
  ip.align();
  ip.put(s.getId()); // get varId
  if (s.getInitialValue() != null) { // translate value
      unparseValue(s.getInitialValue(), ip);
  }
  ip.end();
}
```

Figure 6: Private Methods in ILUnparser.java Supporting Translations of State from Internal Representation to S-expression Code

listings, each component of the internal representation is supported by a corresponding private method in ILUnparser. Each of these methods accesses the component's name and other identifying characteristics and sends the string representation of this data to the printer in accordance with grammar rules of the intermediate language. To meet the specification of reversibility, the order of identifier declarations is accessed where possible from the internal representation. The order information is used to format and print the identifier declarations in the same order. Figure 6 contains an example of two private methods supporting translations of state objects to S-expression code describing state declarations. Because the accessor of the StateTable returns an enumeration instead of an ordered set, the order of state declarations in the original program is not preserved. Enumerations in Java return elements in an arbitrary order.

5.2 ILtoSource

The prettyprinter utility is used to format and print the IOA code from the internal representation of programs. Accessor methods of the objects representing components of the program were used to generate

```
private static void translateStateTable(StateTable st, prettyprinter pp) {
        pp.print(KI_STATES);
        pp.eol();
        Enumeration e = st.enumerate(); // get the states
        while (e.hasMoreElements()) { // translate each state
                pp.print(translateState((State)e.nextElement()));
                if (e.hasMoreElements()) {
                        pp.print(",");
                        pp.eol();
                }
        }
}

private static String translateState(State s) {
        StringBuffer sb = new StringBuffer();
        Variable v = s.getVariable(); // get state variable
        sb.append(v.getName()); // print state variable's name
        sb.append(":");
        sb.append(" ");
        sb.append(translateSort(v.getSort())); // get the type
        // get and print initial values
        if (s.getInitialValue() != null) {
                Value vl = s.getInitialValue();
                sb.append(" ");
                sb.append(":=");
                sb.append(" ");
                sb.append(translateValue(vl));
        }
        return sb.toString();
}
```

Figure 7: Private Methods in ILtoIOA.java Supporting Translations of State from Internal Representation to Source IOA Code

string representations of those components according to the syntactic and semantic rules of IOA. To meet the semantic and syntactic accuracy specification, the automaton, signature, states, transitions, and tasks sections of the standard IOA program format are each supported by private methods in the translator. Each of these methods accesses declared information from internal representation components. To meet the reversibility specification, identifier order information is accessed from the internal representation wherever possible.

Figure 7 contains an example of two private methods handling state object transitions. Because the accessor of the StateTable returns an enumeration instead of an ordered set, the order of state declarations in the original program is not preserved. Enumerations in Java return elements in an arbitrary order.

Because of differences in the grouping of information between IOA programs and internal representations, multiple private methods have been implemented for some internal representation components. In particular, the translations of State and Transition objects to IOA both access identifiers from Term objects; however, the selection and formatting of this information is different for the "states" and "transitions" sections in the standard IOA format. Methods with names such as translateTerm and translateTransTerm have been implemented to specialize these translation procedures.

6 Recommended Usage

The intended use for these translators is to allow components of the IOA Toolkit to return code listings based on the current representation of a program that is being processed with tools in the IOA Toolkit. This can be accomplished by calling the static public methods unparseSpec and translateSpec within ILUnparser and ILtoSource classes, respectively. In addition, toILForm, writeILForm, toSourceForm, and writeSourceForm methods can be called from any intermediate language element, and these methods will call one of the translators to obtain a code listing for that element. The code listing can be printed to the system output or redirected to a file for further examination. The source code for both translators may be found in the directory IOA_Toolkit/Code/ioa/il in the files ILUnparser.java, ILtoSource.java, and ILtoIOA.java. Test scripts are found in IOA_Toolkit/Test/il, and instructions for running tests can be found in IOA_Toolkit/Test/il/README.

7 Testing

Proper and thorough testing of the translators is necessary to ensure error-free operation. Translators that don't meet the design specifications outlined in this paper can cause problems for designers because errors in code listings returned to designers can be perpetuated in future versions of those designs. The following tests should be performed for each translator:

- Reversibility testing, in which the output of the translator is input to the intermediate language parser in the case of ILUnparser or the front end of the IOA Toolkit in the case of ILtoSource. The second output listing is then compared to the first for discrepancies.

- Completeness testing, in which a thorough selection of test programs are created to test the translation of program representations containing all varieties of program and data structures possible in the IOA Toolkit. The output listing of the translators should be compared to input listings as well as put through reversibility testing as described above.

- Uniqueness testing, in which the output file is compared to the input file to ensure it is identical and free of ambiguity, including noticing discrepancies in the order of language identifier declarations between the input files and the translated code listings.

7.1 Test Scripts

Existing test scripts can be found in the directory IOA_Toolkit/Test/il. They call the ILParser program on existing IL files in the directory IOA_Toolkit/Test/simulator/impltest, which has the side effect of printing the results of both translators to the system output. These results are redirected to text files that can be compared visually to the existing source and intermediate language files in IOA_Toolkit/Test/simulator/impltest. The results of these tests are described in the following sections.

7.2 ILUnparser Results

The structure and syntax of input intermediate language files were preserved in the code translations; however, some discrepancies were found between ILUnparser output and design specifications, as shown in figure 8. Language identifiers, such as sorts and operators, were declared in a different sequence from the input file order. This occurred because many accessor methods of internal representation components return enumerations that access members of a set in arbitrary order. The implementation of canonical ordering in the internal representation, as described in section 8.1 below, will facilitate the correction of this problem.

In addition, the intermediate language supports declarations of lexical scoping for components of a program; however, ILUnparser does not translate lexical scoping from the internal representation because no accessor methods exist for scope values. When such accessor methods are implemented for the internal

Input

```
(ioa
 ((sorts (s1 "type" ()) (s0 "Bool" ()))
  (ops (op9 (id "true") (() s0) (scope 3))
   (op7 (prefix "~") ((s0) s0) (scope 3))
   (op4 (id "false") (() s0) (scope 3)))
  (vars (v0 "alpha" s0 (scope 37))))
 (automaton "Toggler"
  ((actions (a1 internal "action1" (subscope 38 37))
    (a2 internal "action2" (subscope 39 37)))
   (states (v0 op4))
   (transitions
    (a1 (pre v0) (eff ((assign v0 op4))) (subscope 40 37))
    (a2 (pre (apply op7 v0)) (eff ((assign v0 op9))) (subscope 41 37))))
  (subscope 37 3)))
```

Output

```
(ioa
 ((sorts (s1 "type" ()) (s0 "Bool" ()))
  (ops
   (op9 (id "true") (() s0))
   (op7 (prefix "~") ((s0) s0))
   (op4 (id "false") (() s0)))
  (vars (v0 "alpha" s0)))
 (automaton  "Toggler"
  ((actions (a2 internal "action2") (a1 internal "action1"))
   (states (v0 (apply op4)))
   (transitions
    (a2 (pre (apply op7 v0)) (eff ((assign v0 (apply op9)))))
    (a1 (pre v0) (eff ((assign v0 (apply op4)))))))))
```

Figure 8: Pre-parsed Intermediate Language Representation of Toggler Program Followed by Translated Intermediate Language Code. Discrepancies exist between the identifier declaration order of actions and transitions in the two code listings. In addition, scope declaration is not yet handled by ILUnparser.

```
automaton Toggler
  signature
    internal action1, action2
  states
    alpha: Bool := false
  transitions
    internal action1
      pre alpha
      eff alpha := false
    internal action2
      pre ~alpha
      eff alpha := true
```

```
automaton Toggler
  signature
    internal action2, action1
  states
    alpha: Bool := false
  transitions
    internal action2
      pre ~alpha
      eff alpha := true
    internal action1
      pre alpha
      eff alpha := false
```

Figure 9: Source IOA Code Listing of Toggler Program (Left) and Translated IOA Code (Right). Discrepancies exist between the identifier declaration order in the two code listings.

representation, this discrepancy can be resolved.

7.3 ILtoSource Results

All syntax was maintained between the input and translated IOA listings; however, order of identifier declarations, such as action and transition declarations, was not maintained between input and output. An example of this discrepancy is shown in figure 9. This problem can be resolved once canonical ordering is implemented in the internal representation.

Further testing on both translators is necessary to ensure all language structures in the internal representation are properly handled. In addition, reversibility tests should be performed when changes are made to the IOA language or the internal representation, to ensure those changes can be processed correctly by the translators.

8 Recommendations for Future Work

Although the foundations for both translators are well in place, further work is needed on both translators before they can be used by programmers who require the IOA Toolkit to return accurate representations of their designs. Neither translator supports canonical ordering of language identifier declarations or program modification information, and ILUnparser does not yet handle lexical scoping information. In addition, testing of a wide variety of input files is required of both translators before they can be considered adequate for users of the IOA Toolkit. Finally, maintenance will need to be performed on the translators as changes are made to the IOA language and internal representation.

8.1 Support for Canonical Ordering

Both translators contain methods that print identifier declarations in a different order than their declaration in the input code. This discrepancy has the potential to cause circular reference errors in the IOA Toolkit, such as an operation on a variable that refers to a variable that has not yet been declared. The misordered identifiers are due to accessor methods of internal representation components that return Java enumerations of ordered sets of objects. Enumerations in Java return objects in an arbitrary, unspecified order and are not suitable for accessors that must acquire an ordered list. Alternative accessors that return ordered data structures, such as cloned vectors or linked lists, should be returned by the accessors of internal representation components, such as accessors of ActionTable objects, for which canonical ordering is a requirement.

8.2 Other Program Modification Information

Some tools in the IOA Toolkit make changes to the internal representation of the program. When these changes include modifying or deleting named components such as actions, states, or variables, the returned code listing may be difficult for the designer to understand. Enough information about the changes should be returned for the designer to understand what the changes are and the reasons for them; for example, it may be desirable for a composite automaton to contain information about the pre-composed automata and their components and return this data to the designer. It may be sufficient for this information to appear in comments, or it may be more useful for identifiers to include the name of the original automaton as part of the new identifier name. In any case, accessors for this information do not exist yet in the internal representation; therefore this functionality has not been implemented in the translators.

8.3 Testing and Maintenance

Reversibility, completeness, and uniqueness testing, as described in Section 7, should be performed on both translators. In addition, periodic maintenance of translators must be performed as changes are made to the IOA language and its internal representation in the IOA Toolkit. This will ensure that programs using newer versions of the language will not be incorrectly handled by the translators. Maintenance will require complete testing of changes to ensure modified language structures are correctly handled without ambiguity.

8.4 Additional Considerations

S-expression declarations of lexical scoping are not handled by ILUnparser because accessor methods in the internal representation do not exist for this data. Once these accessor methods are implemented, ILUnparser should be modified to include methods that include this information with the translation. In addition, parameter declarations for actions, states, transitions, and tasks in IOA are not yet handled by the ILtoSource translator. When test programs that use parameter declarations have been created, methods to support these declarations can be implemented and tested for this translator.

9 Conclusion

Translators for the IOA Toolkit have been developed to print intermediate and source language code from intermediate representations of IOA programs. Functionality exists to support tools in the IOA Toolkit to return source and intermediate language code listings to designers for inspection and additional editing. ILUnparser translates the internal representation to S-expression intermediate language, while ILtoSource translates the internal representation to source IOA code. Functionality should be added to both translators to support canonical ordering, lexical scoping, and program modification information when the appropriate accessors for this information have been implemented in the internal representation. Further testing of the translators is necessary, including testing for completeness, reversibility, and uniqueness. Finally, maintenance of the translators will be needed as changes are made to IOA and its internal representation.

References

[1] Stephen J. Garland and Nancy A. Lynch. *The IOA language and toolset: support for mathematics-based distributed programming.* Submitted for publication.

[2] Joshua Tauber. *IOA Code Generation.* Website http://theory.lcs.mit.edu/ josh/codegen.html.

[3] Stephen J. Garland, Nancy A. Lynch, Mandana Vaziri. *"IOA: a language for specifying, programming, and validating distributed systems."* MIT Laboratory for Computer Science, 1997.