Implementation and Verification of Implicit-Invocation Systems Using Source Transformation*

Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, Juergen Dingel
School of Computing, Queen's University, Kingston, Canada
E-mail: zhang@namzak.com, (bradbury,cordy,dingel)@cs.queensu.ca

Abstract

In this paper we present a source transformation-based framework to support uniform testing and model checking of implicit-invocation software systems. The framework includes a new domain-specific programming language, the Implicit-Invocation Language (IIL), explicitly designed for directly expressing implicit-invocation software systems, and a set of formal rule-based source transformation tools that allow automatic generation of both executable and formal verification artifacts. We provide details of these transformation tools, evaluate the framework in practice, and discuss the benefits of formal automatic transformation in this context. Our approach is designed not only to advance the state-of-the-art in validating implicit-invocation systems, but also to further explore the use of automated source transformation as a uniform vehicle to assist in the implementation, validation and verification of programming languages and software systems in general.

1. Introduction

With the growing size and complexity of software systems, software verification and validation techniques such as testing and model checking are increasingly important. While testing focuses on the actual behaviour of the program, model checking focuses on its mathematical model. Testing and model checking are complementary: testing is lightweight but incomplete while model checking is heavyweight but complete.

A major problem with testing and model checking is that they require different software artifacts. In fact, there is often a big semantic gap between the code artifacts that can be executed and tested and the modelling artifacts that can be verified using model checkers. This gap must typically be bridged by hand with little tool support, leading to a real possibility of errors and spurious results when the finite-state model does not correspond exactly to the implemented software system. Corbett, Dwyer, et al. note that hand-constructed models are *"expensive, prone to errors, and difficult to optimize"* [4]. The time required to convert artifacts by hand and the possibility of spurious results can be greatly reduced using automated transformations.

One kind of software system which is particularly difficult to validate is *implicit invocation* (II) or *publish-subscribe* systems, which are increasingly popular as an integration mechanism for loosely coupled components in software systems. II systems feature a lot of non-determinism due to concurrent execution of components. This high degree of non-determinism makes them particularly challenging to certify and hence a good proving ground for comparing and combining software verification and validation methods such as testing and model checking.

In previous work we proposed a framework for the uniform testing and model checking of II systems [23] based on an II model checking system originally developed by Garlan and Khersonsky [7, 8] and extended by Bradbury and Dingel [1]. Our framework leverages Garlan and Khersonsky's XML intermediate representation for II systems and its automated translation to finite state models checkable by the Cadence SMV model checker [14], a tool for exploring the state space of a program to check formal properties such as freedom from deadlock. Our previous short paper focussed on the testing and model-checking framework itself. In this paper we concentrate on the details of its implementation using source transformations.

At the core of our framework is the *Implicit-Invocation Language* (IIL), a new special purpose language specifically designed for expressing verifiable software systems that use the II architectural style. IIL is designed to address several problems: the lack of explicit features for II in existing programming languages, leading to code that does not well express its real semantics; the large gap between II code and its hand-created modelling representation, for example as Garlan and Khersonsky's XML representation; the lack of any convenient simulation and testing framework for II systems; the lack of the ability to both test and model check II systems in a uniform and consistent manner; and the lack of automated tools to assist in these processes.

We have chosen to implement IIL entirely using formal source transformations, both as an experiment in that technique and in order to allow for the future possibility of formal verification of the translations to execution and modelling artifacts themselves. One set of transformations provides the ability to execute and test IIL programs by translation to the existing general concurrent programming language Turing Plus [10], while another set provides the ability to verify and model check IIL programs by translation to the XML intermediate representation of Garlan and Khersonsky's II modelling method (Figure 1).

In the remainder of this paper, we provide a quick overview of the II architectural style in Section 2 and introduce the Implicit-Invocation Language (IIL) in Section 3. Section 4 discusses the programming, execution, and verification artifacts of our transformational framework. In section 5 we present the details of our automated source transforma-

*This work is supported by the Natural Sciences and Engineering Research Council of Canada and the Ontario Graduate Scholarship Program.

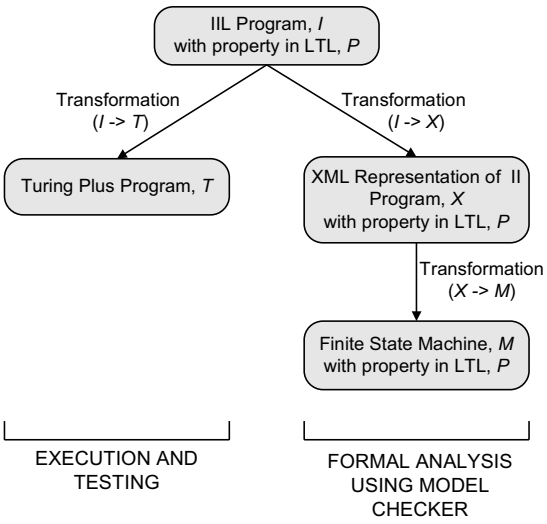

Figure 1. Our transformational framework

tions to both execution and modelling artifacts. We describe experience using our system and possible future directions for exploring the complementary relationship between testing and model checking in Section 6. Finally, we discuss related work and draw conclusions in Sections 7 and 8.

2. II Systems

II systems are characterized by six parameters: *components*, *events*, *event-method bindings*, an *event delivery policy*, a *shared state*, and a *concurrency model*. Components in the system can announce events, which are the primary method of communication between components. Upon receiving events from the components, the event dispatcher sends the events out to all subscriber components that have requested to receive that particular type of event.

The correspondence between announced events and the methods to be invoked in response to these announcements is defined in the event-method bindings. Event-method bindings instruct the dispatcher where to send events. The event delivery policy, a set of conditional delivery rules, instructs the dispatcher on when and how to send them.

II systems we have studied include [1]: a *Set-Counter* example in which one component stores elements in a set and another keeps count of the number of elements; the *Active Badge Location System* (ABLS), an electronic tagging alternative to pagers, in which different components issue requests, store information, and announce the location of users; and the *Unmanned Vehicle Control System* (UVCS), in which vehicle components announce information such as their movement plan, and other components monitor the movement to ensure vehicles reach their destinations without collision. All of these systems are specified and integrated using implicit invocation.

3. The Implicit-Invocation Language IIL

To help bridge the gaps between coding, testing and verifying implicit-invocation systems we have designed the special-purpose programming language IIL. IIL is explicitly designed to allow for direct expression of implicit-invocation semantics using custom syntax for II features and concepts on top of a Java-like core. In order to guarantee that all programs can be executed and tested, only features that can be directly implemented or transformed to simulated concurrent execution are included. In order to guarantee that all programs can be modelled, only language features that can be directly represented or transformed to Garlan and Kershonsky's XML intermediate modelling language are included. And to attach verification closely to code, properties to be verified are directly expressed as part of the program.

As an illustrative example, Figure 2 shows a standard implicit-invocation example, the Set-Counter system [20] expressed in IIL. In order to directly express verifiable II systems, IIL includes the following special features: component declarations, event declarations, announcement statements, a dispatcher declaration, delivery statements, event-method bindings, and property declarations.

The Set-Counter system declares two components: a `Set` and a `Counter`. The `Set` component contains a set of objects and the `Counter` component keeps count of the objects in the set. Figure 2 shows the IIL representation of both the `Set` and `Counter` components. All components in IIL can contain variables and methods.

The Set-Counter example declares four events. `EnvAdd` and `EnvRemove` are *external* or *environment* events, which represent external behaviour affecting the II system. Their declarations give the event name and its announcement properties. The other declared events `Insert` and `Delete` are *local* events which give the event name and optional data. Components in IIL use *announce* statements to send local events to the dispatcher. For example, an `Insert` event is announced in the `Add` method of the `Set` component.

As well as components and events, an *event dispatcher* is declared. The dispatcher is responsible for event delivery and defines the delivery policy. Environment events are delivered immediately, while local events are delivered according to the policy using *deliver* statements. In our Set-Counter example the delivery policy says that if there are more `Insert` events waiting to be delivered than `Delete` events, then an `Insert` event is delivered immediately and a `Delete` event is delivered randomly, otherwise the opposite occurs.

Event-method bindings are needed to register the methods to the events for event delivery. For example, in the Set-Counter example the `EnvAdd` event is bound to the `Add` method in the Set component `s`. That is, when an `EnvAdd` event is announced the `Add` method in `s` will be invoked.

IIL also allows for direct expression of the temporal logic *property declarations* to be verified for the program using the model checking process. For example the property `AlwaysCatchesUp` in the Set-Counter example says that global variable `setSize` will always eventually equal the `counter` variable in the Counter component `c`.

```
system SetAndCounter {
    external event EnvAdd {1..N},
        EnvRemove {1..N};
    event Insert(int {1..2} numElements);
    event Delete(int {1..2} numElements);

    dispatcher delivers Insert, Delete {
        if (Insert.count > Delete.count) {
            deliver Immediate Insert;
            deliver Random Delete;
        } else {
            deliver Random Insert;
            deliver Immediate Delete;
        }
    }

    int {0..3} setSize;

    SetAndCounter() {
        Set s = new Set();
        Counter c = new Counter();

        bind EnvAdd to s.Add();
        bind EnvRemove to s.Remove();
        bind Insert to
            c.CountIns(Insert.numElements);
        bind Delete to
            c.CountDel(Delete.numElements);

        property AlwaysCatchesUp =
            (G F (setSize = c.counter));
        property ...
    }

    component Set
        announces Insert, Delete
        accepts EnvAdd, EnvRemove {
        int {0..2} value;

        Add() {
            value = {1,2};  // nondeterministic
            if ((setSize + value) < 4) {
                setSize = setSize + value;
                announce Insert(value);
            }
        }

        Remove() {
            ...
        }
    }

    component Counter
        accepts Insert, Delete {
            int {0..3} counter = 0;

            CountIns(int {1..2} number) {
                counter = counter + number;
            }

            CountDel(int {1..2} number) {
                ...
            }
    }
}
```

Figure 2. The Set-Counter example in IIL (slightly elided due to space constraints)

4. II Framework Artifacts

Our transformational framework for running, testing and verifying IIL consists of three main types of artifacts:

- Programming/specification artifacts in IIL itself
- Execution/testing artifacts in the Turing Plus language
- Verification artifacts in the XML intermediate language and the SMV modelling language

Programs are expressed entirely in IIL, then automatically transformed to Turing Plus [10] for execution and testing and to the XML intermediate representation for modelling and SMV for verification. Because it is explicitly designed to express II systems, IIL programs are very concise – up to ten times smaller than both the corresponding Turing Plus implementations used for testing and the XML and SMV representations used for model checking.

4.1 Execution Artifacts in Turing Plus

Execution and testing artifacts are derived from IIL using a formal source transformation to Turing Plus [10], a general-purpose concurrent extension of the programming language Turing [11]. We decided to target Turing Plus for execution of II systems because of its simple, general concurrency model and randomized simulation scheduling framework, which allows for lightweight, realistic testing of concurrent programs.

A critical part of our transformation from IIL to Turing Plus is the design of a representation for implicit method invocation and component concurrency in Turing Plus that accurately reflects IIL semantics. In designing these, we used as a reference semantics for IIL the corresponding features of Garlan and Kershonsky's XML notation for II systems [7, 8].

Turing Plus does not support implicit method invocation directly, so in our Turing Plus model we used explicit invocation to implement implicit-invocation. Thus the Turing Plus

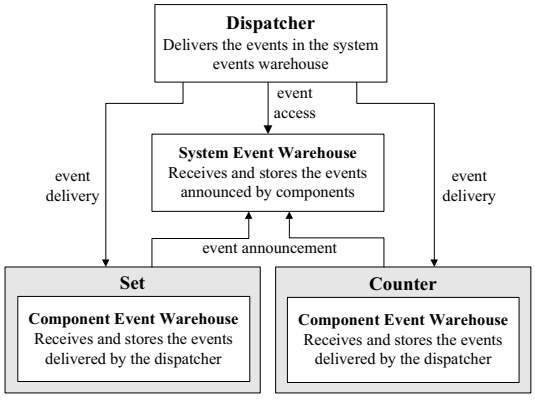

Figure 3. Implicit method invocation for the Set-Counter example in Turing Plus

implementation uses explicit method calls in event announcement, in event delivery (Figure 3), and in components to invoke bound methods when a delivered event is received.

The concurrency model determines how to assign and manage threads in the system. Based on the Garlan and Kershonsky modelling semantics, our implementation fixes the concurrency model to use a separate Turing Plus thread for each component, the event dispatcher, and the system itself. To ensure that the execution semantics of an IIL program in Turing Plus matches its model checking semantics in SMV, all of the threads in the Turing Plus implementation of an II system are synchronized using barrier synchronization.

Structurally, the Turing Plus implementation consists of a module and nested monitor for each component declaration and the dispatcher, and a main procedure that handles environment event generation. These vary with system and are derived from the IIL program by source transformation.

The Turing Plus implementation is based on a set of com-

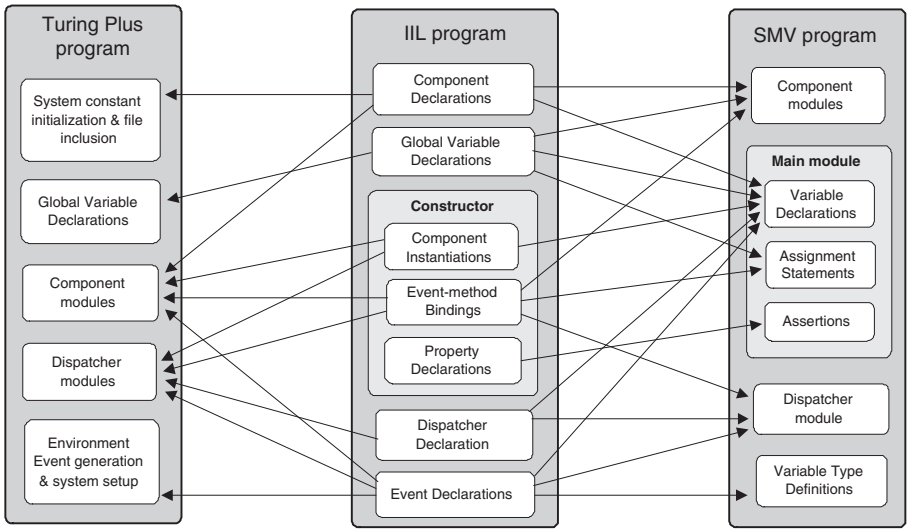

Figure 4. Diagram showing how the parts of an IIL program are used in generating the corresponding Turing Plus program and SMV model

mon definitions for the underlying mechanisms of II that are program independent, such as type definitions for events, event queues, and event warehouses (collections of event queues), as well as modules to manage the system event warehouse, component event warehouses, and thread synchronization. These modules are independent of the IIL program being transformed and are included from a library using generated *include* directives in each transformed result.

4.2 Verification Artifacts in SMV

To model check systems written in IIL, we use the approach previously presented in [1, 7, 8]. This approach focuses on the automatic analysis of II by representing an II system in an XML intermediate representation and using an existing Java tool to transform it into an SMV model accepted by the Cadence SMV model checker. The challenge therefore was to create a source transformation for IIL programs to the limited features of the XML modelling representation.

The SMV model for an IIL program represents each component and the dispatcher as an SMV module. There is also a main module which instantiates the other components. Modules in SMV have input and output parameters which are used for event announcement. For example, in the Set-Counter example an output parameter of the Set component is connected to an input parameter of the Dispatcher for the announcement of an Add event, and an output parameter of the Dispatcher is connected to an input parameter of the Counter component for delivery of the event. The model checking semantics of the SMV program is (by design) identical to the execution semantics of the Turing Plus program outlined above.

5. Transformations in the II Framework

Now that we have introduced the artifacts involved in our framework, we can discuss TXL and our automated transformation tools for artifact conversion. Figure 4 shows an overall summary of how the parts of an IIL program are used to automatically transform IIL programs into a Turing Plus program for execution and an SMV model for verification.

5.1 Source Transformation using TXL

TXL [5] is a programming language designed to support source transformation tasks. It combines features of both functional and rule-based programming, and supports unification, implied iteration and deep pattern match. A TXL program consists of two parts: a context-free, possibly ambiguous grammar describing the syntactic structure of the artifacts to be transformed, and a set of by-example formal transformation rules that use pattern-replacement pairs to describe the desired transformations. TXL has been used in a range of applications from software design recovery to artificial intelligence, in both academic and industrial contexts [6].

5.2 Transformation to Execution Artifacts

Our automated tool for transforming IIL to Turing Plus consists of a set of formal transformation rules written in TXL. The structure and syntax of Turing Plus programs is very different from IIL – some of these differences have been discussed in Section 4.1. The transformation to Turing Plus is divided into four steps: component transformation, dispatcher transformation, system and environment setup generation, and restructuring of the resulting system body.

The fours steps form a tightly coupled transformation: each must be completely consistent with the other for the combined result to be correct. In order to facilitate this consistency, each of the steps is derived by formal source transformation from the same source artifact: the entire IIL source program itself. This demonstrates the advantages of the main design goal of IIL: to capture all aspects of the II system in one uniform source artifact. Each step takes as input the entire source program in IIL, using different parts of the source

```
 1 module Set
 2    export Fork, receiveEvent
 3    include "queueManager.i"
 4    queueManager.createEventQueue ("EnvAdd")
 5    queueManager.createEventQueue ("EnvRemove")
 6    var value : int
 7
 8    monitor SetMonitor
 9       export receiveEvent, getCount,
                Remove, Add
10       procedure Add
11          var e : event
12          queueManager.getEvent ("EnvAdd", e)
13          var Arr :
               array 1..2 of int := init (1, 2)
14          var Sel : int
15          randint (Sel, 1, 2)
16          value := Arr (Sel)
17          if ((setSize + value) < 4) then
18             setSize := setSize + value
19             var etba Insert : event
20             etba Insert.name := "Insert"
21             etba Insert.param (1).intPara :=
                  value
22             announce (etba Insert)
23          end if
24       end Add
25
26       procedure Remove
27          ...
28       end Remove
29       function getCount (ename : string) : int
30          result queueManager.getCount (ename)
31       end getCount
32
33       procedure receiveEvent (e : event)
34          queueManager.receiveEvent (e)
35       end receiveEvent
36    end SetMonitor
37
38    procedure receiveEvent (e : event)
39       SetMonitor.receiveEvent (e)
40    end receiveEvent
41
42    process run : 100000
43       for l : 1..999999999
44          Rendezvous.readySetGo
45          if SetMonitor.getCount ("EnvRemove")
                > 0 then
46             SetMonitor.Remove
47          elsif SetMonitor.getCount ("EnvAdd")
                > 0 then
48             SetMonitor.Add
49          end if
50       end for
51    end run
52
53    procedure Fork
54       fork run
55    end Fork
56 end Set
```

Figure 5. Generated Turing Plus module/monitor for the Set component of the Set-Counter example

as needed to transform or generate its result.

Step 1: Component transformation. Component transformation combines information from the event declarations, component declarations, and constructors in the IIL program. In Turing Plus components are represented as modules and the component transformation occurs in 5 parts. To clarify the component transformation we refer to the Turing Plus implementation of the Set component in Figure 5, which was automatically transformed from the Set-Counter IIL example in Figure 2. For each part of the transformation we make reference o the corresponding parts of Figure 5.

First, module and monitor names for components in the Turing Plus program are generated from the component names in IIL (*lines 1,8*). Second, an event warehouse (a collection of event queues) is created for each type of event that a component accepts (*lines 4,5*). Third, each method in a component is added to the export list for its monitor (*line 9*). This makes the methods public, so that they can be called from outside the monitor, for example in the run process (*line 48*).

Fourth, the method bodies for each component are generated. In addition to the syntactic transformation of the method bodies from IIL to Turing Plus, the invoking event must be retrieved (*lines 45-48*). A method requires the retrieval of the invoking event in order to use data contained in the event. Since the information about the invoking event is not included in the method body of the IIL program, we must extract this information from the remote component instantiations and the event-method bindings during transformation.

Fifth, the run process (*lines 42-51*) needs to check each event queue and invoke the appropriate bound method if the event queue is not empty. During transformation, the `accepts` statements in the IIL program are used to generate the conditional expression of the `if` statement in the run process, and event-method binding information is used to generate the method call.

As an example of the TXL transformation rules used in this step, Figure 6 shows the main rule used to generate the module and monitor structure from an IIL component. As is evident in this example, TXL's by-example concrete syntactic patterns and functional decomposition style help make it convenient to express and validate our source transformations.

Step 2: Dispatcher transformation. The dispatcher in Turing Plus is constructed using the event declarations, the dispatcher declaration, and the system constructor of the IIL program. All of this remote information must be combined using a global-to-local transformation to generate the result.

For each event in a system the dispatcher creates a queue in the system event warehouse. Event queues are not represented in the IIL program and are generated using the same method as described for component queues above. The event delivery policy is translated directly from the dispatcher body of the IIL program into code for the Turing Plus dispatcher module. In order to complete the event delivery transformation we also need to use information from the component instantiations and the event-method bindings.

Figure 7 shows the result of generating the Dispatcher for the Set-Counter example. Random delivery is simulated using the Turing Plus *randint* library routine to flip a coin. The main TXL rule to generate the Dispatcher module from the dispatcher section of the IIL program is similar in form to the rule for components shown in Figure 6.

Step 3: System and environment setup. In this step we generate declarations for global variables specified in the IIL program and initialize system constants of the Turing Plus implementation. We incorporate the parts common to all systems (discussed in Section 4.1) by generating file includes such as the *include "rendezvous.i"*, which adds the module that handles barrier synchronization. Finally, we generate statements at the end of the program to fork a concurrent process for each of the component and dispatcher modules.

Environment setup generates a procedure using a method call for each external event, similar to the component event announcement shown in Figure 3. An example TXL function from this step is shown in Figure 8. This function demonstrates the use of TXL's functional control paradigm to implement a source transformation that inherits global contex-

```
rule tr_component Bindings [repeat event_binding]
                   Events [list event_declarator]
  replace [component_declaration]
     'component CompName[id]
                EventAnn [opt event_announces]
                EventAcc [opt event_accepts]
     '{
         Body [repeat var_res_met_declaration]
     '}
     % Translate variable declarations
     construct VarDecls [repeat variable_declaration]
        _ [gather_var_decl Body]
          [tr_var_decl]
     % Translate method declarations
     construct MetDecls [repeat method_declaration]
        _ [gether_met_decl Body]
          [tr_met_decl Events Bindings CompName]
     % Method names to export
     construct ExportMets [list method_name]
        _ [get_list_method_name MetDecls]
     % Monitor name
     construct MonitorName [id]
        CompName [+ 'Monitor]
     % First method to run
     deconstruct ExportMets
        FirstMet [id], RestMets [list method_name]
     % Event name for first method
     deconstruct * [event_binding] Bindings
        'bind FirstEvent [id] 'to CompNameId .
              FirstMet '( _ [list expression]') ;
     construct QuotedFirstEvent [stringlit]
        _ [quote FirstEvent]
     % Rest of methods to run
     construct Elsifs [repeat elsif_inrun]
        _ [get_elsifs RestMets Bindings
                      CompName MonitorName]
  by
    'module CompName
       'export Fork, receiveEvent
       'include "queueManager.i"
        EventAcc [tr_event_accepts]
        NewVarDecls
       'monitor MonitorName
          'export receiveEvent, getCount, ExportMets
           NewMetDecls
          'function getCount(ename: string): int
             'result queueManager.getCount(ename)
          'end getCount
          'procedure receiveEvent(e: event)
             queueManager.receiveEvent(e)
          'end receiveEvent
       'end MonitorName
       'procedure receiveEvent(e: event)
          MonitorName.receiveEvent(e)
       'end 'receiveEvent
       'process 'run: 100000
          'for : 1 .. 999999999
              Rendezvous.readySetGo
             'if MonitorName.getCount
                    (QuotedFirstEvent) > 0 'then
                MonitorName '. FirstMet
              Elsifs
             'end 'if
          'end 'for
       'end run
       'procedure Fork
          'fork run
       'end Fork
    'end CompName
end rule
```

Figure 6. Main TXL Rule for generation of the Turing Plus module/monitor structure for an IIL component

```
module Dispatcher
  export Fork
  eventsManager.createEventQueue("EnvAdd")
  eventsManager.createEventQueue("EnvRemove")
  eventsManager.createEventQueue("Insert")
  eventsManager.createEventQueue("Delete")
  procedure deliverEvent
     % External events are always immediate
     if eventsManager.getCount("EnvAdd") > 0 then
        var e : event
        eventsManager.getEvent("EnvAdd", e)
        Set.receiveEvent(e)
     end if
     if eventsManager.getCount("EnvRemove") > 0 then
        var e : event
        eventsManager.getEvent("EnvRemove", e)
        Set.receiveEvent(e)
     end if
     % Delivery policy for internal events
     if eventsManager.getCount("Insert") >
           eventsManager.getCount("Delete") then
        var e: event
        eventsManager.getEvent("Insert", e)
        Counter.receiveEvent(e)
        var flip: int
        randint(flip, 0, 1)
        if flip = 1 then
           if eventsManager.getCount("Delete") > 0 then
              eventsManager.getEvent("Delete", e)
              Counter.receiveEvent(e)
           end if
        end if
     elsif eventsManager.getCount("Delete") > 0 then
        var e: event
        eventsManager.getEvent("Delete", e)
        Counter.receiveEvent(e)
        var flip: int
        randint(flip, 0, 1)
        if flip = 1 then
           if eventsManager.getCount("Insert") > 0 then
              eventsManager.getEvent("Insert", e)
              Counter.receiveEvent(e)
           end if
        end if
     end if
  end deliverEvent
  % The actual process of the Dispatcher
  process run : 100000
     for l : 1 .. 999999999
        Rendezvous.readySetGo
        deliverEvent
     end for
  end run
  % Procedure to start up Dispatcher when appropriate
  procedure Fork
     fork run
  end Fork
end Dispatcher
```

Figure 7. Generated Turing Plus Dispatcher module for the Set-Counter example

tual information to generate its result - in this case the list of events passed in from the main system setup generation rule.

Step 4: System body re-ordering. Unlike IIL, Turing Plus is a declaration-before-use language, and Turing Plus programs must follow a strict order and structure of declaration. In order to separate concerns and avoid overly constraining transformation rules, the previous three transformation steps ignore these constraints. This leaves the ordering problem to this last separate transformation, which involves reordering the program elements to match the order in Figure 4. In essence, this transformation is a topological sort of the program into declaration-before-use dependency order. A simple TXL rule used in this step is shown in Figure 9.

5.3 Transformation to Verification Artifacts

A major drawback to the model checking work we presented in [1] was that it was not completely automated, since user interaction was required to develop the XML modelling

```
function add_event_declaration
        Events [list event_declarator]

   % List of external events
   construct ExtEventNames [list reference]
      _ [get_ext_event_name_declarator Events]

   % List of external event limits
   construct EndOfEvents [list reference]
      _ [build_endOfEvents ExtEventNames]

   % List of external event counters
   construct EventsCount [list reference]
      _ [build_eventsCount ExtEventNames]

   % Make the "randonEvent(...)" statements
   construct RandomEvents [repeat randomEvent]
      _ [build_randomEvent ExtEventNames
                           EndOfEvents EventsCount]

   % Make the statement to print total events generated
   construct RandPuts [repeat randint_and_put]
      _ [build_randPut EndOfEvents ExtEventNames Events]

   replace * [repeat declaration_or_constructor]
      % Add to end of generated code
   by
      'procedure 'randomEvent (eventName : string,
                  maxEventCount : int,
                  var eventCount: int, frequency : int)
         'if eventCount < maxEventCount then
            'var flip: int
             randint (flip, 1, frequency)
            'if flip = 1 then
               'var e: event
                e.name := eventName
                announce(e)
                eventCount += 1
             'end' if
          'end 'if
      'end 'randomEvent

      'var EndOfEvents : int
      'var EventsCount := 0
      'var clockLimit: int

      'process run
         'for l : 1 .. clockLimit
             Rendezvous.readySetGo
            'put "main loop ", l
             RandomEvents
         'end 'for
      'end 'run

      'randomize

       RandPuts

      'put "Please input clockLimit: " ..
      'get clockLimit
   end function
```

Figure 8. TXL function to make the Turing Plus external event generator for an IIL program

representation for the program. Our current approach overcomes this deficiency and bridges the gap between artifacts by completely automating the process of generating finite state models for software systems written in IIL.

The transformation from IIL to SMV finite state models involves three steps: program restructuring, conversion to XML, and finite state machine translation. The first two steps convert IIL into the XML modelling notation using cascaded TXL source transformations of the IIL program. The third step uses an existing Java tool to transform the XML representation to a set of finite state machine models in SMV that can then be verified using the Cadence SMV model checker.

Step 1: Program restructuring. The original goal of the IIL language was as a convenient replacement for the verbose XML representation that would be easier to read, write and understand. In the end, IIL has evolved into a full special-purpose language that includes many other notational conveniences, such as true global variables, local variables in methods, `for` loops and `switch` statements, none of which are available in the XML intermediate language. In this first step of our modeling transformation, these notational conveniences are resolved, in essence by compiling and reordering the IIL program using source transformation. The result is a simplified IIL program which is isomorphic to its XML modelling language equivalent, but not yet in XML notation.

Three main language features of IIL are not present in the XML representation and must be converted. First, global variable access is transformed to match the indirect global variable access of the XML representation. IIL components have direct access to globals, while the XML representation uses the SMV model, in which global variables must be accessed indirectly through special local input/output variables.

Second, IIL supports variable declaration at both the component and method level while the XML modelling representation allows variables at the component level only. This step involves moving all method level variables to the component level. To avoid potential name clashes, method variables are uniquely renamed using the method name as a prefix.

Third, IIL allows the convenience of `switch` statements in the dispatcher and both `switch` statements and `for` loops in component methods, while the XML modelling representation has only `if-then-else` statements in order to simplify its modelling task. The transformation therefore transforms `switch` statements into `if-then-else` and unrolls `for` loops into statement sequences, using classic transformations borrowed from the compiler community.

Recall that by design IIL is restricted to expressing programs that have a modelling language equivalent - thus because the XML modelling representation does not have loops, IIL `for` loops are constant bounded and can always be unrolled. Similarly, although the XML modelling representation has no `switch` statement, the transformation can convert them to their `if-then-else` equivalents. The TXL rule for converting `switch` statements used in this stage is shown in Figure 10.

Finally, the program is restructured into the strict order required by the XML modelling representation. In IIL there are no restrictions on ordering, but the XML representation must be strictly structured according to its schema. As in the transformation to Turing Plus, we simplify the previous steps by implementing the ordering constraints as a separate source transformation on the result.

Following this step the IIL program has been restructured into a statement-by-statement match to the target XML modelling representation, but has not yet been converted to XML. Again, rather than convert to XML tag notation while restructuring the IIL program, we have separated the conversion to XML tags into a separate cascaded source transformation in order to separate concerns. This cascaded transformation style is characteristic of complex TXL transformations and has served us well in this project as in others.

Step 2: Conversion to XML mark-up. The second step of the modeling transformation involves the syntactic mapping of the simplified and reordered IIL program to XML

notation. For example consider an event-method binding, defined in a `bind` statement, from Figure 2:

```
bind Insert to c.CountIns(Insert.numElements);
```

The `bind` statement causes an `Insert` event to invoke the `CountIns` method in the instance `c` of the `Counter` component. In the XML intermediate representation the bind statement is transformed into:

```
<event-binding event-name="Insert">
  <method-binding instance-name="c" method-name="CountIns"/>
</event-binding>
```

The TXL rule for the XML markup translation of `bind` statements is shown in Figure 11. The rule matches every IIL `bind` statement and captures its event name and list of method invocations. The event name is quoted so that it can be used in the XML tag, and a sequence of XML method-binding tags for the bound method invocations is generated by the subrule `construct_method_binding`. The rule then replaces the `bind` statement by an XML tag with the event name around the tagged sequence of method bindings.

Step 3: Generation of finite state machines. Following the transformation from IIL to the XML modelling notation, we use the Java tool developed by Garlan and Khersonsky [7] (modified in [1]) to transform the XML representation of the program into a set of finite state machines in SMV. These can

```
% In every scope, sort variable declarations
% before anything else
rule var_decl_first
    replace $ [repeat declaration_or_constructor]
        Anything [declaration_or_constructor]
        VarDecl [variable_declaration]
        Rest [repeat declaration_or_constructor]
    deconstruct not Anything
        _ [variable_declaration]
    by
        VarDecl
        Anything
        Rest
end rule
```

Figure 9. Example TXL rule used in reordering generated Turing Plus code

```
rule tr_switch_statement
   replace [statement]
      'switch '( Exp1 [expression] ')
      '{
         'case Exp2 [expression] ':
             CaseBlock [repeat declaration_or_statement]
          Rest [repeat switch_alternative]
      '}
   deconstruct Exp1
      Exp1RE [relational_expression]
   deconstruct Exp2
      Exp2RE [relational_expression]
   construct IfBlock [block]
      '{ CaseBlock [remove_break] '}
   construct ElseClause [opt else_clause]
      _ [tr_switch_alternative_1 Exp1RE Rest]
        [tr_switch_alternative_2 Rest]
   by
      'if '( Exp1RE == Exp2RE ')
         IfBlock
       ElseClause
end rule
```

Figure 10. TXL rule to convert ILL switch statements to if-then-else form

```
rule tr_event_binding
   replace [event_binding]
      'bind EventName [reference]
         'to ListMethods [list method_invocation];
   construct QuotedEventName [stringlit]
      _ [quote EventName]
   construct RepMethods [repeat method binding]
      _ [construct_method_binding ListMethods]
   by
      <event-binding event-name=QuotedEventName>
         RepMethods
      </event-binding>
end rule
```

Figure 11. TXL rule to convert bind statements to XML markup form

then be checked using the Cadence SMV model checker to verify the property constraints declared in the IIL program. Additional details of this step can be found in [1, 7, 8].

5.4 Evaluation of Transformations

To evaluate our framework we used the three examples introduced in Section 2: *Set-Counter*, the *Active Badge Location System* (ABLS) and the *Unmanned Vehicle Control System* (UVCS). For each example, our evaluation involved programming the system in the IIL language and verifying (by hand) that our transformation tools from IIL to Turing Plus and from IIL to the XML modeling representation performed correctly. We demonstrated that semantics was well preserved across all of the transformations by checking that the execution behaviour and the model checking behaviour matched the original semantics of the IIL programs. Finally, we verified that the specified properties of the IIL programs held, both empirically and formally, by testing and model checking the results of our transformations.

6. Testing and Model Checking using the II Framework

We have discussed the programming, execution and verification artifacts and the automated transformations used in our framework. We now detail how the framework can be used in both testing and model checking.

6.1 Testing

Our first transformation converts an IIL program into a semantically equivalent Turing Plus program which can then be compiled using the Turing Plus compiler and concurrency library to an executable program. The result of executing this Turing Plus program is the production of an execution trace. For purposes of validation, we used manual code instrumentation in the Turing Plus program to output run-time information into the traces. Here for example is a partial execution trace of the Set-Counter system:

```
...
clock tick 20
EnvAdd announced
clock tick 21
setSize = 1
Insert announced
clock tick 22
counter = 1
clock tick 23
...
```

In this example we have output the clock tick, the name of each announced event, and the new values of updated global and local variables. The above trace shows an announcement of the `EnvAdd` event by the environment, which causes the number of elements in the set (the variable `setSize`) to be increased. The `Set` component then announces an `Insert` event which is delivered to the `Counter` component causing the `counter` variable to be increased.

We have used execution traces to perform standard testing techniques on all three of our example II systems. Our Turing Plus programs are convenient for testing because the test harness for environment events is generated automatically as part of our transformation. Moreover, because Turing Plus uses a randomized simulation scheduler, multiple executions of the same program with the same inputs generally result in different execution traces, allowing for bulk testing of many different concurrent executions.

6.2 Model Checking

Our second automated transformation converts IIL programs to the XML modelling representation and then to SMV finite state models for formal verification. We have verified a variety of liveness and safety properties in the context of our IIL examples. IIL currently allows for expression of properties written in Linear Temporal Logic (LTL) but we also have the ability to check Computational Tree Logic (CTL). The LTL operators used in the expression of properties are: X ϕ (in the next state ϕ holds), G ϕ (ϕ holds globally), F ϕ (ϕ holds eventually), ϕ_1 U ϕ_2 (ϕ_1 holds at least until ϕ_2 does).

For example, consider the Unmanned Vehicle Control System (UVCS) example system. On one hand, we need to guarantee liveness properties such as each vehicle in the system eventually reaches its destination:

```
F ((Vehicle1.currRegion = Vehicle1.destRegion)
          & (Vehicle1.xpos = Vehicle1.destxpos)
          & (Vehicle1.ypos = Vehicle1.destypos))
```

On the other hand, we need to verify safety properties such as collision avoidance between two vehicles:

```
G (~(Vehicle1.currRegion = Vehicle2.currRegion)
          | ~(Vehicle1.xpos = Vehicle2.xpos)
          | ~(Vehicle1.ypos = Vehicle2.ypos))
```

In our experiments we verified that vehicles would in fact reach their destination location and that vehicles do not collide. Detailed results of our model checking experiments are presented in a previous paper [1].

6.3 Future Directions

Although the design purpose of our IIL language and transformational framework is the comparison and exploration of synergy between testing and verification, thus far we have only independently evaluated testing and model checking. Next we plan to use our framework to explore the relationship between testing and model checking. We believe that it provides a good testbed for studying the synergies between these two verification and validation methods, and in particular can allow us to investigate questions such as:

- *Can testing help increase confidence in model checking and the correctness of the model checker?*

One possibility would be representing execution traces as CTL properties and using model checking to verify that each trace is also possible in the model.

- *How can testing be used to simplify or optimize model checking?*

One of the primary optimizations in model checking is decomposition. Testing could be used to identify parts of the system that can be easily abstracted or removed for model checking. For example, if a component is not used in a test trace it may be safe to remove it from the verification.

- *Can model checking be used to evaluate the coverage offered by a test suite?*

Model checking could be used to guarantee output coverage in black box testing. For example, we could verify that a variable always is one of a set of values that covers the outputs, or model-check the converse to find a counter-example.

Model checking could also provide guarantees in white box testing. For example, if we wanted to provide statement coverage for an IIL program we could instrument the Turing Plus program to record a program counter in the execution trace. If we noticed that certain statements were not covered, we could use model checking to prove that the program counter can never have those values, or to generate a counter-example input to add to the test suite.

- *Might it be useful to integrate temporal logic properties into the testing effort through, for instance, run-time safety analysis?*

One possibility would be to use the safety specification of a program to automatically generate a run-time monitor that checks if a finite event trace satisfies the property [9].

7. Related Work

Rapide[13] and Eventua[17] are other special-purpose languages for event-based systems. Rapide is an executable language intended for modeling the architectures of concurrent and distributed systems. Eventua is an object-oriented language that includes native support for events. Eventua programs can be transformed to the $\varrho\varpi\varsigma$-calculus for execution. Our work differs from these approaches in that we focus on formal analysis in addition to execution traces.

Both Bandera [4] and Spin [12] provide automatic translation from a general purpose programming language to a standard model checker. Our approach differs in that we focus on a special-purpose II language that expresses all aspects of both program and properties in one uniform notation, and in that we achieve all our results using formal source transformation rules which at least in theory allow for formal verification of the translations themselves.

As an alternative to our approach, it would be interesting to explore using Java to represent event-based systems (e.g. using the Message-Driven Thread API for Java[15], or publish/subscribe infrastructures like Elvin[19] or Siena[2]) and to use Bandera for model generation and analysis. However,

because IIL expresses implicit invocation semantics and verification conditions in custom syntax rather than through library calls, and in a single uniform notation, it encodes the program, its execution and modelling much more directly.

Although as we have shown TXL expresses the source transformations in our framework very clearly, our method does not depend on any particular tool and other source transformation languages and systems such as Stratego [22], ASF+SDF [21], ANTLR [16] and others have their own advantages and could serve as well.

Ours is also not the only system that has proposed using formal source transformation to bridge gaps between verification and practice. In our own previous work we have used formal source transformation to extend the capabilities of the VeriSoft C++ model checker to handle Java RMI verification [3], and at Microsoft Research formal source transformation has been used to transform concurrent device drivers to sequential approximations that can be checked for some concurrency properties using sequential model checking [18].

8. Conclusion

We have presented a uniform source transformation-based framework for specifying, testing, and model checking implicit-invocation (II) systems. It consists of IIL, a special purpose high-level language for specifying II systems, and two fully automatic, formally specified source translations: one to the Turing Plus language for execution and testing, and one to the input language of a standard model checker for verification. The framework demonstrates how formal source transformation can be used to combine the convenience of a special-purpose language with the benefits of two complementary quality assurance techniques: testing and model checking. Furthermore, it shows how the significant gaps between artifacts can be bridged using transformation. Automatic source translation makes the analysis in our framework less error prone, less time consuming and more reliable.

The contribution of our work lies not only in the development of the transformation framework but also in the opportunities for future research. The framework provides an excellent testbed for exploring both automated transformation and the synergies between testing and model checking.

References

[1] J. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. ESEC/FSE 2003*, Sept. 2003.

[2] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[3] T. Cassidy, J. Cordy, T. Dean, and J. Dingel. Source transformation for concurrency analysis. In *Proc. 5th Int. Workshop on Language Descriptions, Tools and Applications*, pages 26–43, April 2005.

[4] J. Corbett, M. Dwyer, J. Hatcliff, et al. Bandera: Extracting finite-state models from Java source code. In *Proc. Int. Conf. on Software Engineering*, June 2000.

[5] J. Cordy. TXL – a language for programming language tools and applications. *Proc. 4th Int. Workshop on Language Descriptions, Tools and Applications, Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004.

[6] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *J. Inform. and Software Technology*, 44(13):827–837, 2002.

[7] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proc. Int. Work. on Software Spec. and Design*, Nov. 2000.

[8] D. Garlan, S. Khersonsky, and J. Kim. Model checking publish-subscribe systems. In *Proc. Int. SPIN Work. on Model Checking of Software*, May 2003.

[9] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[10] R. Holt and J. Cordy. The Turing Plus report. *CSRI, Univ. of Toronto*, 1987.

[11] R. Holt and J. Cordy. The Turing Programming Language. *Communications of the ACM*, 31(12):1410–1423, 1988.

[12] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, 2002.

[13] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. on Software Engineering*, 21(9):717–734, 1995.

[14] K. McMillan. Getting started with SMV. *Cadence Berkeley Laboratories*, 1998.

[15] mdthread.org. Message-driven thread API for the Java programming language. Web page: http://www.mdthread.org.

[16] T. Parr and R. Quong. ANTLR: A predicated- LL(k) parser generator. *Software-Practice and Experience*, 25(7):789–810, 1995.

[17] J. Patterson. An object-oriented event calculus. Technical Report TR02-08, Iowa State University, 2002.

[18] S. Qadeer and D. Wu. KISS: Keep it Simple and Sequential. In *Proc. PLDI 2004, ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 14–24, June 2004.

[19] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. AUUG'97*, Sept. 1997.

[20] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. In *Proc. SIGSOFT '90 Symp. on Software Development Environments*, Dec. 1990.

[21] M. van den Brand, P. Klint, et al. The ASF+SDF meta-environment: a component-based language development environment. *Proc. 1st Int. Workshop on Language Descriptions, Tools and Applications, Electronic Notes in Theoretical Computer Science*, 44(2), 2001.

[22] E. Visser. Stratego: A language for program transformation based on rewriting strategies. *Proc. Rewriting Techniques and Applications (RTA01), Lecture Notes in Comp. Science*, 2051:357–361, 2001.

[23] H. Zhang, J. Bradbury, J. Cordy, and J. Dingel. A transformational framework for testing and model checking implicit-invocation systems. In *Proc. Int. Work. on Distr. Event-Based Systems (DEBS '04)*, May 2004.