Proc. of the International Workshop on Distributed Event-Based Systems (DEBS '04), pages 110-115, Edinburgh, Scotland, UK, May 2004.

A Transformational Framework for Testing and Model Checking Implicit Invocation Systems[1]

Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
{hellen, bradbury, cordy, dingel}@cs.queensu.ca

Abstract

With the growing size and complexity of software systems, software verification and validation (V&V) is becoming increasingly important. Model checking and testing are two of the main V&V methods. In this paper, we present a framework that allows for testing and formal modeling and analysis to be combined. More precisely, we describe a framework for model checking and testing implicit invocation software. The framework includes a new programming language – the Implicit Invocation Language (IIL), and a set of formal rule-based transformation tools that allow automatic generation of executable and formal verification artifacts. We evaluate the framework on several small examples. We hope that our approach will advance the state-of-the-art in V&V for event-based systems. Moreover, we plan on using it to explore the relationship between testing and model checking implicit invocation systems and gain insight into their combined benefits.

1. Introduction

With the growing size and complexity of software systems, software verification and validation (V&V) is becoming more and more important. Testing and model checking belong to the two categories of software V&V: testing/inspection and formal methods. While testing focuses on the actual behavior of the program, model checking focuses on the mathematical model. Testing and model checking are complementary: testing is lightweight but incomplete while model checking is more heavyweight but complete. A major problem with testing and model checking is that they usually require different software artifacts. In fact, there is often a large semantic gap between the software developer artifacts that are tested and the artifacts that are accepted by model checkers. The gap between artifacts typically has to be bridged by humans with little tool support. Thus, there is a possibility for spurious results when the finite-state model does not correspond exactly to the software system.

To alleviate this problem, we have developed a transformational framework for the testing and model checking of implicit invocation (II) or publish-subscribe systems. II systems are event-based and have two primitives. First, components can announce or publish events. Second, other components can listen or subscribe to events that are announced. A centralized message server or event dispatcher receives announced events and uses them to invoke the appropriate subscriber methods. We have chosen to focus on II systems for several reasons. In the context of testing, II systems feature a lot of non-determinism due to concurrent execution of components and the event dispatcher. In the context of model checking, this non-determinism often causes the model to be excessively large. Additionally, II has become increasingly popular as an event-based architecture.

Our framework includes a new programming language – the Implicit Invocation Language (IIL). IIL is a special-purpose language that is designed specifically for software systems that use the II architectural style. The primary advantage of IIL for programming II systems is that it leverages our knowledge about II and provides a notation with a level of abstraction that is convenient to read and write.

Rather than write a compiler, we have chosen to implement IIL for simulation and testing by source transformation to an existing executable language. We have chosen to translate II into Turing Plus, a concurrent programming language [11].

Model checking systems written in IIL involves the use of an existing II model checking system originally developed by Garlan and Khersonsky [6, 7] that we have extended in [2]. This system involves representing an II system in an XML intermediate representation that is translated into a finite state machine which is analyzed by a standard model checker. The model checker allows for the analysis

[1]This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Graduate Scholarship Program (OGS).

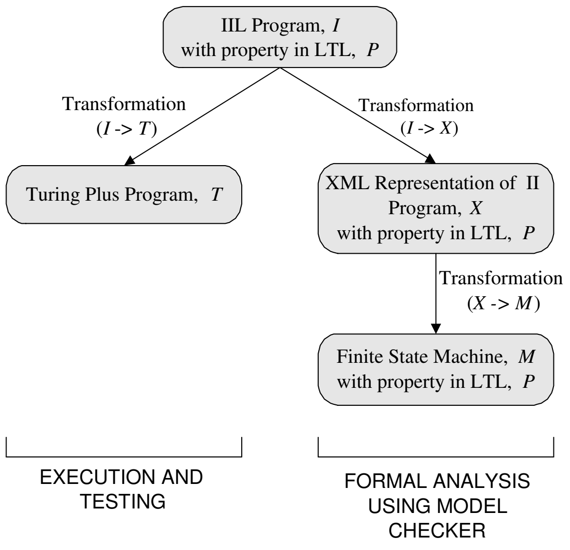
IIL Program, I with property in LTL, P
Transformation (I -> T)
Transformation (I -> X)
Turing Plus Program, T
XML Representation of II Program, X with property in LTL, P
Transformation (X -> M)
Finite State Machine, M with property in LTL, P
EXECUTION AND TESTING
FORMAL ANALYSIS USING MODEL CHECKER

Figure 1. Our transformational framework

of Linear Temporal Logic (LTL) properties. We integrate our previous model checking approach with IIL by providing an automated transformation from IIL into the XML intermediate representation.

Figure 1 shows the overall structure of our transformational framework. All of our transformation tools are implemented using TXL, a programming language and rapid prototyping system specifically designed to support rule-based source to source transformation [5]. Each tool is fully automated and is based on formal rewriting rules expressed in terms of the syntax of the source language and the target language. The framework allows for a powerful combination of two complementary V&V techniques. We hope that our automated approach will allow us to explore the relationship between testing and model checking and gain insight into the possible benefits of their combined use.

We will next provide an overview of the II architectural style in Section 2 before introducing IIL in Section 3. In Section 4 we will discuss the execution and testing of II systems using Turing Plus and in Section 5 we will discuss model checking II systems. Section 6 describes our evaluation of the framework and Section 7 discusses how our framework relates to existing research work. Finally, we outline our conclusions as well as possible future work in Section 8.

2. II systems

An II system is characterized by six parameters: components, events, event-method bindings, an event delivery policy, a shared state, and a concurrency model.

Events are the primary method of communication between components. The components in the system can announce events. 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 (Figure 2). The correspondence between events that are announced and the methods in a component instance that are 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 when and how to send them.

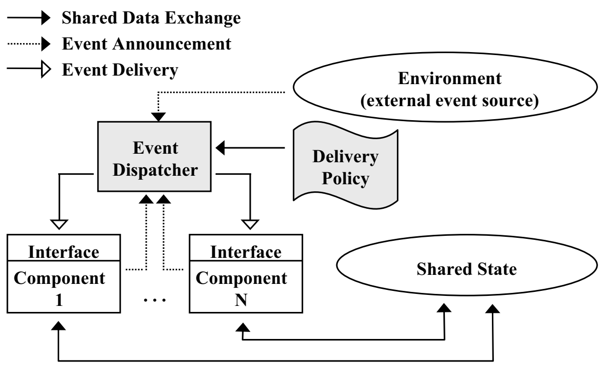

Figure 2. II system structure [6]

3. An II language (IIL)

IIL is a programming language designed especially for expressing II systems. IIL includes the following special features: component declarations, event declarations, announcement statements, dispatcher declaration, delivery statements, event-method bindings, and property declarations. We will now discuss these features in the context of a Set-Counter example [17]. The Set-Counter example in IIL is presented in Figure 3. Due to space limitations, this example has been elided and is provided primarily to show the main features and overall structure of an IIL program.

The Set-Counter system involves two examples of component declaration: a `Set` and a `Counter`. The `Set` component contains a set of objects and the `Counter` component keeps a count of the objects in the set. Figure 3 shows the IIL representation of the `Set` component. All components in IIL can have variables which describe component properties and methods which describe actions.

The Set-Counter example contains four event declarations. Two of the declared events (`EnvAdd`, `EnvRemove`) are external events (also called environment events). The declarations contain an event name and event announcement properties. Environment events represent external be-

```
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 choice
         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 3. The Set-Counter example in IIL

havior that can affect the II system. The other two declared event are local events (`Insert`, `Delete`) which are declared with an event name and optional event data.

Components in an IIL program can contain announcement statements which define the announcement of locally declared events. For example, in the component `Set` the `Insert` event is announced in the `Add` method.

In addition to components and events, a dispatcher is declared. The dispatcher is responsible for event delivery and defines the delivery policy. Environment events will be delivered immediately by the dispatcher while local events will be delivered according to the delivery policy, which is composed of delivery statements. In our Set-Counter example we have included a delivery policy which states that if there are currently more `Insert` events waiting to be delivered than `Delete` events, then an `Insert` event is delivered immediately and a `Delete` event is delivered randomly (i.e., delivered sometime in the future). 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 we see that 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.

In addition to the special language features used to construct an II system, IIL also allows for LTL property declarations. The properties that are declared can be verified using our model checking process. For example the property `AlwaysCatchesUp` in the Set-Counter example states that the global variable `setSize` will always eventually be equivalent to the `counter` variable in the Counter component `c`.

4. Translating IIL programs into executable Turing Plus programs

As previously mentioned, IIL has no compiler and requires transformation to an executable language for testing and simulation. We have chosen to transform IIL into Turing Plus, an extension of the programming language Turing [10]. We decided to use Turing Plus for execution of II systems because Turing Plus, as a concurrent programming language [11], has a simple and general concurrency model.

Before implementing our automated transformation from IIL to Turing Plus we first had to develop a model of II in Turing Plus that captured the semantics of an IIL program. The two main design issues in developing this model were: implicit method invocation and the concurrency model.

4.1. Implicit method invocation

Turing Plus does not support II directly. The first problem we need to solve is to find a mechanism to carry out II in Turing Plus.

According to Garlan and Scott, "implicit invocation supplements, rather than supplants, explicit invocation" [8]. In our Turing Plus model, we take this approach and use three steps of explicit invocation to implement II (see Figure 4). That is, an explicit method call is used in event announcement, event delivery and bound method invocation. The three main elements of our implementation of II in Turing Plus are:

1. A system event warehouse, a set of queues, is built to receive all the announced events. When components announce an event or an environment event is generated, it will be sent to the system event warehouse.

2. The dispatcher removes the events in the system event warehouse and delivers them. Environment events will be delivered immediately. Local events will be delivered according to the delivery policy. The dispatcher delivers the events in the system event warehouse by calling the bound component to receive the event.

3. Each component has a component event warehouse to receive the events delivered by the dispatcher. The component will invoke the bound method after it receives the delivered events.

Our modeling of II in Turing Plus thus divides event-method bindings into two parts: the event-component binding information contained in the dispatcher and the event-method binding information contained in the components.

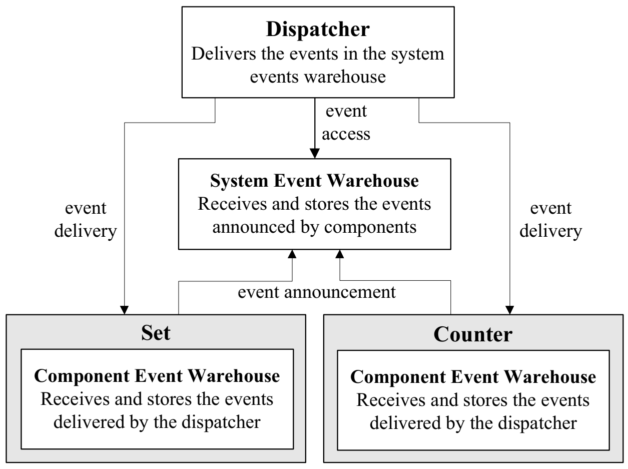
Dispatcher
Delivers the events in the system events warehouse
event access
System Event Warehouse
Receives and stores the events announced by components
event delivery
event delivery
event announcement
Set
Component Event Warehouse
Receives and stores the events delivered by the dispatcher
Counter
Component Event Warehouse
Receives and stores the events delivered by the dispatcher

Figure 4. Implicit method invocation for the Set-Counter example in Turing Plus

4.2. The concurrency model

The concurrency model determines how to assign and manage threads in the system. In [7], Garlan and Khersonsky propose several models of concurrency including a single thread of control for all components and separate threads of control for components.

In our implementation, we fix the concurrency model to use a single thread for each component. Each component, the event dispatcher, and the system have a thread defined by a "run process". For example, in Figure 5 we see that the Set-Counter example has 4 threads. 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 an Turing Plus implementation of an II system are synchronized by the `Rendezvous` monitor.

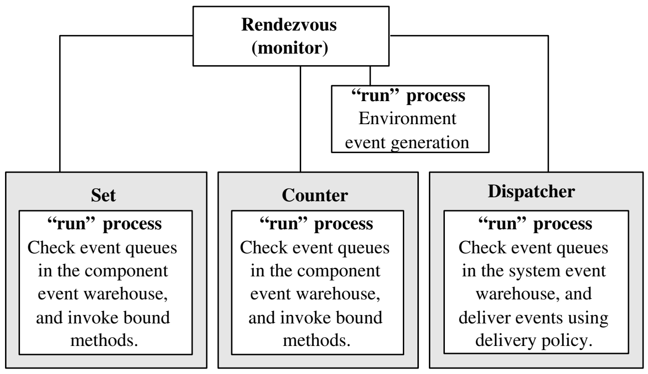
Rendezvous (monitor)
"run" process
Environment event generation
Set
"run" process
Check event queues in the component event warehouse, and invoke bound methods.
Counter
"run" process
Check event queues in the component event warehouse, and invoke bound methods.
Dispatcher
"run" process
Check event queues in the system event warehouse, and deliver events using delivery policy.

Figure 5. The II concurrency model for the Set-Counter example in Turing Plus

4.3. Transformation of IIL to Turing Plus

The structure and syntax of Turing Plus programs is very different from IIL programs. Figure 6 provides a summary of how information in an IIL program is used in generating each part of a Turing Plus program.

Our automated tool for transforming IIL to Turing Plus consists of a set of formal transformation rules written in TXL, a popular source-code transformation language that has been used in numerous industrial and academic projects over the past 10 years. Unfortunately, due to space restrictions we cannot include these rules in this paper. For examples of TXL rules see [18].

5. Translating IIL programs into SMV models

To model check systems written in IIL we use the existing approach we previously presented in [2]. This approach is an extension of an II model checking system originally developed by Garlan and Khersonsky in [6, 7]. This approach focuses on the automatic analysis of II by representing an II system in an XML parameterized representation.

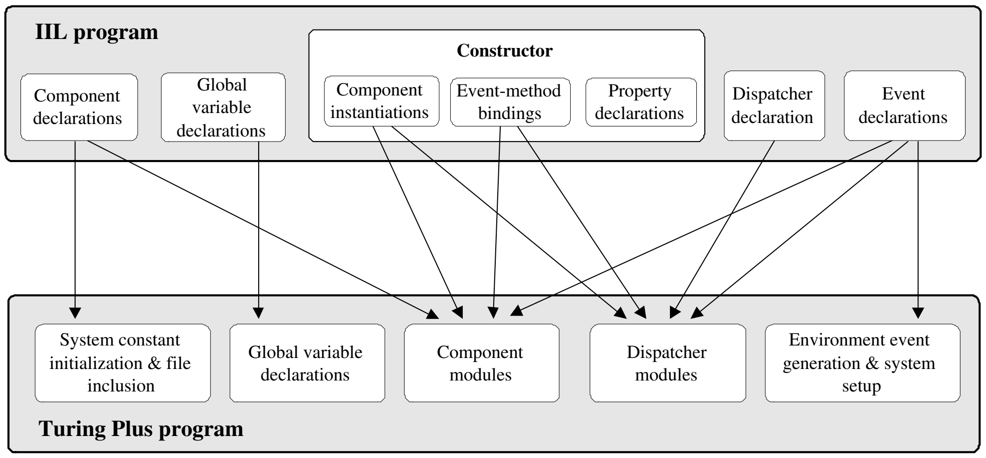

Figure 6. Information integration in IIL to Turing Plus transformation

Once the XML input has been created, a Java transformation tool converts the information into a set of finite state machines. This set of state machines can then be checked using the Cadence SMV model checker [14]. We currently check properties written in LTL.

A major drawback to our model checking work presented in [2] was that it was not completely automatic since user interaction was required in developing the XML representation. The approach presented in this paper overcomes this deficiency and completely bridges the gap between artifacts by automating the process of generating finite state models for software systems written in IIL.

5.1. Transformation of IIL to XML

Originally, due to the verbose nature of XML, IIL started out as a replacement to the XML intermediate language that would be easier to read and write. In addition to improving the syntax, IIL evolved into a special-purpose language that includes notational conveniences such as variable declarations in methods, the use of for loops and the use of switch statements.

As with the transformation to Turing Plus, our transformation from IIL to XML is done using an automatic transformation tool written in TXL. The transformation from IIL to XML involves two steps. In the first step the notational conveniences of IIL are removed. For example, for loops are unrolled. The program is also reorganized to follow the program order required by the XML representation. After the first step the IIL program is a statement-by-statement match to the XML representation. The second step of the transformation involves the syntax translation from IIL to XML.

6. Evaluation

To evaluate our transformational framework we use three examples: the Set-Counter example, the Active Badge Location System (ABLS), and the Unmanned Vehicle Control System (UVCS) [2]. Our evaluation of each example involved modeling the example in the IIL language and verifying that our transformation tools from IIL to Turing Plus and from IIL to XML worked correctly.

Our evaluation shows that IIL programs are substantially smaller in size than the corresponding Turing Plus implementation used for testing and both the XML and SMV representations used for model checking. Table 1 summarizes the results and illustrates the advantage of using a special-purpose language convincingly. Note that the ABLS and UVCS values in the table are the average of several example systems implemented in IIL. Our evaluation of our automatic transformation tools also demonstrated that the semantics was well preserved during all of the transformations.

Example	IIL (KB)	TP (KB)	TP (% IIL)	XML (KB)	XML (% IIL)	SMV (KB)	SMV (% IIL)
Set-Counter	2	13	650%	9	450%	24	1200%
ABLS	3	13	444%	8	278%	23	767%
UVCS	8	16	200%	21	263%	38	469%
Overall	**5**	**14**	**315%**	**13**	**281%**	**28**	**622%**

Table 1. File size comparison

7. Related work

Rapide [13] and Eventua [15] are two existing special-purpose languages for event-based systems. Rapide is an executable architecture definition language. It is intended for modelling the architectures of concurrent and distributed systems. Eventua is an object-oriented language that includes native support for events by including classes, fields, a self keyword, and parameter passing for both methods and events. An Eventua program can be transformed to the $\varrho\varpi\varsigma$-calculus, the underlying formalism, for execution.

Bandera [4] and the Spin model checker [12] provide automatic translation from a general purpose programming language to a standard model checker. Our approach differs in that we limit ourselves to a special-purpose II language.

In the Cadena project at Kansas State University [9], the model checker Bogor was used to analyze BoldStroke – an event-based real-time middleware architecture developed by Boeing. BoldStroke was modeled using CORBA's Interface Definition Language. The model construction was only partially automatic.

As an alternative to our approach, it would be interesting to explore the use of Java to represent event-based systems (e.g. using the Message-Driven Thread API for Java [1], or publish/subscribe infrastructures like Elvin [16] or Siena [3]) and to use Bandera for automatic model extraction and analysis.

8. Conclusion

We have presented a framework for specifying, testing, and model checking II systems. It consists of a high-level language for specifying II systems and two fully automatic, formally specified translations: one into the Turing Plus language for execution and testing, and one into the input language of a standard model checker [18]. The framework demonstrates how automatic source code transformation can be used to combine the convenience of a special-purpose language with the benefits of two complementary V&V techniques: testing and model checking.

We believe that our work provides a useful test bed for studying the relationship and possible synergies between testing and model checking. In particular, it might allow us to investigate the following questions: To what extend can parallel testing be used to increase confidence in model checking results and in the correctness of the model checker? How can testing be used to simplify or optimize the model checking? Can model checking be used to evaluate the coverage offered by the test suite? Would it be useful to integrate temporal logic properties into the testing effort through, for instance, run-time safety analysis?

In addition to studying the relationship and possible synergies between testing and model checking, another future direction of research is the extension of our framework for use with more general forms of publish-subscribe systems. For example, II systems that support dynamic bindings and additional concurrency models.

References

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

[2] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation system. In *Proc. of ESEC/FSE 2003*, pages 78–87, Sept. 2003.

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

[4] J. Corbett, M. Dwyer, J. Hatcliff, et al. Bandera: Extracting finite-state models from Java source code. In *Proc. of the Int. Conf. on Software Engineering*, pages 439–448, June 2000.

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

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

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

[8] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proc. of the Int. Conf. on Software Engineering*, pages 447–455, 1993.

[9] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proc. of the Int. Conf. on Software Engineering*, pages 160–173, May 2003.

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

[11] R. Holt and D. Penny. The concurrent programming of operating systems using the Turing Plus language, University of Toronto, 1988.

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

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

[14] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Mar. 1999.

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

[16] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97*, Sept. 1997.

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

[18] H. Zhang. An implicit-invocation language and its implementation. Master's thesis, Queen's University, 2004.