Model-based Conformance Testing for Android

Yiming Jing[1], Gail-Joon Ahn[1], and Hongxin Hu[2]

[1] Laboratory of Security Engineering for Future Computing (SEFCOM)
Arizona State University, Tempe, AZ85281, USA
{ymjing,gahn}@asu.edu
[2] Delaware State University, Dover, DE19901, USA
{hxhu@asu.edu}

Abstract. With the surging computing power and network connectivity of smartphones, more third-party applications and services are deployed on these platforms and enable users to customize their mobile devices. Due to the lack of rigorous security analysis, fast evolving smartphone platforms, however, have suffered from a large number of system vulnerabilities and security flaws. In this paper, we present a model-based conformance testing framework for mobile platforms, focused on Android platform. Our framework systematically generates test cases from the formal specification of the mobile platform and performs conformance testing with the generated test cases. We also demonstrate the feasibility and effectiveness of our framework through case studies on Android Inter-Component Communication module.

1 Introduction

According to a recent report from research firm [5], the worldwide smartphone market ballooned 65.4% year over year in the second quarter of 2011, indicating the total shipments of 100 million units. In addition, with the surging computing power and network connectivity of smartphones, more third-party applications and services are deployed on these platforms and enable users to customize their devices. Many legitimate applications tend to manipulate users' sensitive information such as contact list, locale information, and other credentials [14]. To protect such sensitive attributes, it is necessary to ensure that smartphones are properly configured and rigorously validated.

Fast evolving smartphone platforms, however, have raised considerable security concerns due to the lack of rigorous security analysis. At the same time, a large number of system vulnerabilities and security flaws on smartphone platforms have continuously been reported. For instance, an unprotected component was discovered in the phone application of Android version 1.1 [15]. This flaw allowed any malicious application to make phone calls without the permission it ought to have. Another recent work [10] indicated that the message passing system in Android can be a target for denial-of-service and hijacking if used incorrectly.

Software developers often utilize conformance testing as an indispensable step to check errors and flaws in both developing and maintaining software systems.

Conformance testing attempts to bridge the gap between system implementation and design requirements. It compares the expected behaviors described by the system requirements with the observed behaviors of an actual implementation. The observed results reflecting the conformance of implementation strongly depends on the adopted test cases [12]. In addition, test automation [17] has recently become quite common for reducing the cost of software testing procedures. A typical automated testing harness mainly offers automation in managing, executing and evaluating tests. However, such an approach cannot effectively support automated test generation. Manually creating test cases is tedious, error-prone, and often insufficient for proving the conformance of system implementation [19]. Such a problem exists in the widely used test harness for Android, Google's Android testing framework [3] [1]. Android testing framework only adopts hand-crafted test cases for conformance testing and fails to provide a comprehensive set of test cases.

Model-based testing involves developing a data model to generate tests. The model is developed based on the design requirements, and reflects the expected features of the System Under Test (SUT) [7]. Unlike hand-crafted tests, model-based approach helps reuse the generated test cases and improves the efficiency of testing procedures. If any requirement changes, a tester only needs to update the model and get a new suite of test cases, avoiding the tedious work of changing hand-crafted test cases.

In this paper, we present a model-based conformance testing framework for evaluating Android platforms. Our framework automatically generates and executes test cases. Moreover, we demonstrate the feasibility and practicality of our approach through case studies on Android Inter-Component Communication (ICC) module. We chose ICC for several reasons: (1) ICC is one of the core modules of Android as it supports collective interactions of applications; (2) the requirements of ICC are publicly available. To conduct conformance testing in our framework, we first derive the formal models and properties for Android ICC from design requirements. The formal specifications of models and properties are fed into an analysis module to automatically generate test cases, which systematically enable the rigorous conformance testing for the Android platform. MCTF checks whether the SUT's behaviors conform to functional and non-functional requirements. For example, the requirements specify a set of desired behaviors. Therefore, it is necessary to discover invalid and malformed inputs that may violate those requirements and should be caught and handled properly. Having comprehensive conformance testing would ensure the correctness and assurance of ICC in Android.

The remainder of this paper is organized as follows. Section 2 gives an overview of Android ICC. Section 3 discusses our framework and demonstrates how our framework can be applied to examine the conformance of Android ICC. Section 4 presents a tool chain designed with our framework followed by the discussion on performance analysis. Section 5 describes the related work. Section 6 concludes this paper and elaborates the future directions.

2 Overview of Android ICC

Smartphone applications inherently tend to communicate with each other. Android ICC is a sophisticated messaging system designed to support such interactions. In this section, we give a brief overview of Android ICC as described in Android documentation for SDK (SDKD) [2] and Android Compatibility Definition Document (CDD) [1].

2.1 Components

The basic unit in Android application communication is *component*. Each component is a logical building block that could support each other. Four types of components are defined with various requirements.

- *Activities* are components that provide graphic user interface (GUI). The Android GUI is implemented as a stack of activities starting one after another, where each activity is presented as a window on the screen.
- *Services* are components that run in the background to perform long-running operations. Unlike activities, a service does not have any graphic interface. Instead, services provide Remote Procedure Call (RPC) interfaces.
- *Broadcast Receivers* are asynchronous components that receive and reply to system-wide broadcasts from other components.
- *Content Providers* are components that provide public data interfaces to other components. A content provider provides common database commands such as query, insert, update and delete, through which other components can retrieve and store data.

2.2 Intents and Intent Filters

Intents play a leading role in connecting the components of applications. An intent object is a data structure carrying information about its desired recipients and optional data. Applications communicate with each other by sending and receiving intents. All intents are processed and delivered by a centralized "post office", the intent resolver.

Like a post office processing parcels in the real world, the intent resolver finds qualifying recipients by checking the attributes of an intent object.

Primary intent attributes include *action* and *data*:

- *Action* is a string naming the general action to be performed. An intent can contain at most one action.
- *Data* is a tuple consisting of both the URI of the data to be acted on and its MIME media type. This attribute indicates the data to be processed by the action.

Secondary attributes include *component*, *category*, *extras* and *flags*.

- *Component Name* is a string naming the component that should handle the intent.

- *Category* is a string containing additional information about the kind of component that should handle the intent.
- *Extras* is a key-value pair of additional information to be delivered to the recipient component.
- *Flags* is a set of strings that instruct the Android system to launch an activity.

Each component can be bound to one or more *intent filters*, which declare capabilities of the components. An intent filter includes three attributes describing the intents it would accept, including *action*, *category* and *data*. Intents and components are correlated via intent filters. Android maintains a map between public components and intent filters. The intent resolver finds the matching intent filters for a given intent, then delivers the intent to the corresponding components based on the map.

3 Model-based Conformance Testing Framework (MCTF)

In this section, we present our conformance testing framework, called model-based conformance testing framework (MCTF), which is depicted in Figure 1. Our framework is designed for generating test cases and facilitating rigorous conformance testing with the generated test cases. We divide the framework into four steps as follows:

1. ***System Modeling: Android Modeling.***
   First, all parameters and properties of Android are derived from Android CDD and Android SDKD. Based on the identified parameters and properties, a model is defined. Parameters describe data objects and attributes of the system. Properties lay out rules regulating interactions of parameters. Android parameters and properties are then formally represented.
2. ***Test Case Generation.***
   The most significant recent development in testing is the application of formal reasoning techniques, such as model checking [11], theorem proving [24] and SAT solving [23], to generate test cases from the formal specification. In this step, the formal model is utilized to automatically derive abstract test cases, leveraging a formal reasoning technique.
3. ***Test Case Translation.***
   The generated test cases from the previous step are not suitable for direct execution, since they are generated in an abstraction level. Therefore, it is crucial to bridge the gap between abstract test cases and executable test cases. The translation is performed to extract necessary information from abstract test cases and construct executable test cases.
4. ***Test Case Execution.***
   In this step, executable test packages are generated by compiling executable test cases. With the executable test packages, an Android device or emulator is tested. For each test case, the results are monitored and recorded. Finally,

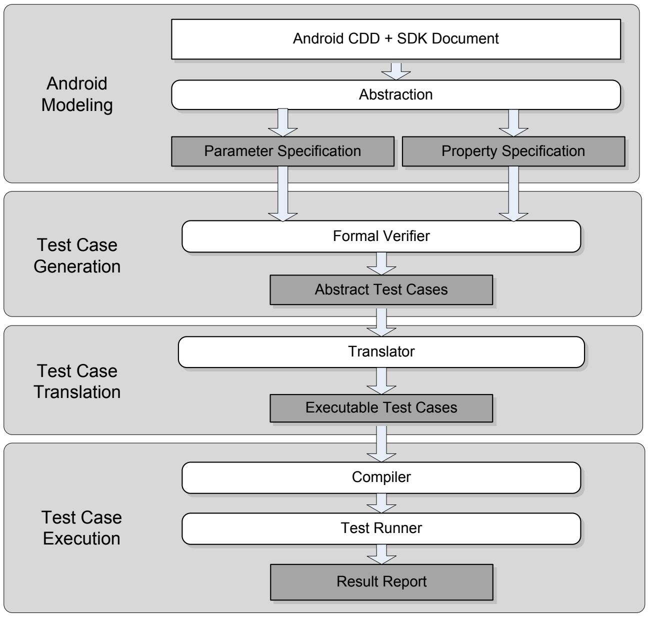


Fig. 1. Model-based Conformance Testing Framework

a human readable report is generated once all the tests are executed. The generated test report may contain supplemental information, such as screenshots, to further examine other functional and non-functional components.

In order to conduct model-based conformance testing, it is crucial to have a well-designed and general purpose language to represent the model. Alloy [20] is a structural modeling language based on first order logic, and has been widely used in the modeling community. The usage of Alloy for the representation of models is an attractive aim. Our framework adopts Alloy to formally represent an Android model. As we discussed earlier, the formal model is in turn utilized by formal reasoning tools such as Alloy Analyzer, to generate abstract test cases, which are then translated into executable test cases.

We now demonstrate how Android ICC can be rigorously tested through the four steps shown in Figure 1, identifying specific mechanisms for each MCTF task.

3.1 System Modeling: Android Modeling

A model for a specific software system is an abstract specification of the system's behaviors. Parameters and properties comprise a typical model for capturing such behaviors. The parameters are attributes or variables that appear in a piece of requirements. After parameters are identified, their types and valid

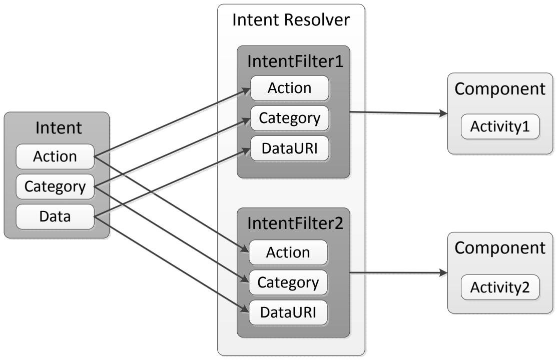


Fig. 2. Implicit Intent Resolution

value ranges should be identified as well. For example, if an input variable accepts integers in the range of 1 to 12, the identified parameters should use the same valid range. Properties are identified from the information about the relationships among parameters.

Android modeling procedure consists of three steps: model construction from requirements, specification of model parameters, and specification of model properties.

Model Construction from Android ICC Requirements For testing Android systems and applications, testers derive parameters and properties from Android SDKD and Android CDD. Android SDKD defines the requirements of Android system, including objects and logics of Android functions and packages. Android CDD complements Android SDKD by providing additional technical details of various versions of Android platform.

For example, a technical section in Android SDKD says that "there are three Intent characteristics that can be filtered on: actions, data and categories". From this, testers identify three parameters: *action*, *category* and *data*. The definition of these three attributes also shows the data type of each parameter. That is, *action* is any string, *category* is any string set and *data* is a pair (2-tuple) of strings.

Android SDKD and Android CDD describe Android ICC in two categories: *Explicit Intent Resolution* and *Implicit Intent Resolution*, depending on the target attributes for the resolution process. If the component name of an intent is a non-empty set, this intent is an *explicit* intent because the recipient component is given explicitly. The intent resolver delivers explicit intents to the recipients designated by the `ComponentName` attribute, regardless of other attributes in the intent. Such process is called *Explicit Intent Resolution*. Actually, no resolution process is occurred because the recipient is already specified by the sender.

Thus, *intent*, *component* and *intent resolver* are identified as parameters of explicit intent resolution. The attribute *ComponentName* is consulted. The property of explicit intent resolution is trivial, as abstracted below:

- ***Property 1***: The intent should be delivered to the recipient designated by the component name attribute of the intent.

Implicit intents do not specify any recipient component but wait for the intent resolver to determine which component they should be resolved to, based on the *action*, *data* and *category* attributes specified in the intent. This process is called *Implicit Intent Resolution*.

The parameters of implicit intent resolution include *intent*, *intent filter*, *component*, and *intent resolver*. *Action*, *category* and *data* are attributes that are consulted during the resolution process. Each attribute corresponds to a test, in which the attribute of the intent is matched against that of the intent filter. To be delivered to the component, an implicit intent must pass all the three tests on the intent filters bound with the component. Since a component can be bound with multiple intent filters, an intent that does not pass through one of a component's intent filters may pass another.

In the *action* test, the Android Intent Resolver tests both the action of the intent object and the action set of the intent filter. An intent names a single action while the intent filter specifies one or more actions. To pass the action test, the action specified in the intent object must match at least one of the actions specified in the intent filter. The action set of the intent filter object must not be empty. A special case is an intent without actions, which passes all action tests. The properties of *action* test can be summarized as follows:

- ***Property 2***: The action specified in the Intent object must match one of the actions listed in the filter.
- ***Property 3***: An Intent object that does not specify an action automatically passes the test as long as the filter contains at least one action.

The *category* fields in both the intent and intent filter are a set of category strings. To pass the category test, the category set of the intent should be the subset of the category set of the intent filter. The filter can list additional categories, but it cannot omit any in the intent. An intent without category passes all category tests by default. The properties of *category* test can be summarized as follows:

- ***Property 4***: Every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any in the intent.
- ***Property 5***: An Intent object with no category should always pass this test, regardless of the attributes in the filter.

Data contains URI and type. The URI specifies the location of the data in three sub-attributes: scheme, authority and path. The data type specifies the MIME type of the data. Android also allows wildcards when specifying data subtype in both the intent and intent filter.

– ***Property 6***: An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.
– ***Property 7***: An Intent object that contains a URI but no data type passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type.
– ***Property 8***: An Intent object that contains a data type but no URI passes the test only if the filter lists the same data types and similarly does not specify a URI.
– ***Property 9***: An Intent object that contains both a URI and a data type passes the data type part of the test only if its type matches a type listed in the filter.

Figure 2 shows an example of implicit intent resolution. In this example, a public component is bound with two intent filters. An intent resolver attempts to resolve the intent shown on the left. If all of the tests pass for both intent filters, the intent is delivered to the two components on the right.

Specification of Model Parameters Based on Android SDKD and Android CDD, we formulate the identified parameters. We first define *Component* as follows:

Definition 1. *A component is represented with a* (τ)*, where* τ *is a unique name of the component;*

Intent can be defined as follows:

Definition 2. *An intent is represented with a 5-tuple* $(\tau, \alpha, \Gamma, \sigma)$*, where* τ *is the name of the recipient component;* α *is an action string that describes the action to be performed;* Γ *is a set of category strings that represent the type of components which should handle the intent; and* σ *is a 2-tuple* $(uri, type)$ *consisting of data URI and data type.*

Intents can be classified into two categories: *explicit intent* and *implicit intent*, as we discussed earlier. We formally define them as follows:

Definition 3. *Explicit intents designate the target component by its component name field. The set of explicit intents is denoted as EI.* $EI=\{i \mid i \in I \wedge i.\tau \neq null\}$

Definition 4. *Implicit intents do not specify a target. The set of implicit intents is denoted as II.* $II=\{i \mid i \in I \wedge i.\tau = null\}$

Then, the *intent filter* can be defined as:

Definition 5. *An intent filter is represented with a 3-tuple* $(\Lambda, \Gamma, \sigma)$*, where* Λ *is a set of action strings;* Γ *is a set of category strings; and* σ *is is a set of* $(uri, type)$ *tuples consisting of data URI and data type.*

We now formally define the *intent resolver* with sets and relations as:

- C is a set of components, $\{c_1, \cdots, c_p\}$;
- I is a set of intents, $\{i_1, \cdots, i_m\}$;
- F is a set of intent filters, $\{f_1, \cdots, f_q\}$;
- $FC \subseteq F \times C$, a many-to-many filter-to-component assignment relation;
- EIC, a one-to-one explicit intent-to-component assignment relation;
- IIF, a one-to-many implicit intent-to-filter assignment relation;

Based on the above-defined model, we now give the formal specification of identified parameters with Alloy as follows:

```
module android/ICC
abstract sig Str {}
sig actionStr extends Str{}
sig categoryStr extends Str{}
sig uriStr extends Str{}
sig typeStr extends Str{}
sig dataTuple {
    uri: lone uriStr,
    type: lone typeStr }

abstract sig Object {}
sig Component extends Object {
    componentName: lone componentStr }
```

```
sig Intent extends Object {
    componentName: lone componentStr,
    action: lone actionStr,
    category: set categoryStr,
    data: lone dataTuple }
sig Filter extends Object {
    action: set actionStr,
    category: set categoryStr,
    data: set dataTuple }
sig Resolver {
    IIF: Intent -> set Filter,
    IIF_A: Intent -> set Filter,
    IIF_C: Intent -> set Filter,
    IIF_D: Intent -> set Filter,
    FC: Filter -> set Component,
    EIC: Intent -> lone Component }
```

The first `sig` statement declares Str, which represents a string that can be assigned to other objects. Then, we define *component*, *intent* and *intent filter* which have all the necessary attributes for intent resolution. We then declare a resolver, which defines several relations which map intents to sets of intent filters. The value ranges of all the parameters are strings.

Specification of Model Properties Based on Android SDKD and Android CDD, we now formulate and specify properties of Android ICC. A `fact` statement in Alloy puts an explicit constraint on the model. In our cases, we need to represent the identified properties of intent resolution with facts. According to the properties identified from the requirements, we then give their formal specifications.

The formal specification of Property 1, which covers Explicit Intent Resolution, is shown below:

```
fact explicitIntentResolution {
  all r: Resolver, i: Intent, c:Component |
  i.componentName = c.componentName
  <=> i->f in r.EIC }
```

The following shows formal specifications of Property 2-9, which cover Implicit Intent Resolution:

```
fact implicitIntentResolutuion {
 all r: Resolver, i: Intent, f:Filter |
  i->f in r.IIF_A
  and i->f in r.IIF_C
  and i->f in r.IIF_D
<=> i->f in r.IIF }

fact actionTest {
all r:Resolver| all i:Intent |all f:Filter |
(f.action!=none and i.action!=none
and i.action in f.action)
or (f.action!=none and i.action = none)
<=> i->f in r.IIF_A }
```

```
fact categoryTest {
all r:Resolver| all i:Intent |all f:Filter |
(i.category!=none and i.category in f.category)
or (f.category!=none and i.category = none)
<=> i->f in r.IIF_C }

fact dataTest {
all r:Resolver| all i:Intent |all f:Filter |
(i.data.uri=none and i.data.type=none
and f.data.uri=none and f.data.type=none)
or (i.data.uri in f.data.uri
and i.data.type = none and f.data.type=none)
or (i.data.type in f.data.type
and i.data.uri = none and i.data.uri=none)
or (i.data.uri in f.data.uri
and i.data.type in f.data.type)
<=> i->f in r.IIF_D }
```

3.2 Test Case Generation

In conformance testing, testers need to generate positive and negative test cases to examine the implementation thoroughly. Positive test cases test whether the system behaves exactly as the specified properties when inputs are valid. Negative test cases test whether the system violates the properties when inputs are invalid. Formal reasoning tools can generate abstract test cases accordingly. They translate the model notations into boolean formulas. Then, the formulas are analyzed to find bindings of the parameters and their values that make the formulas true or false. Such true and false bindings are positive and negative test cases, respectively. To generate abstract test cases, we employ Alloy Analyzer to generate instances that satisfy both `facts` and `predicates`.

Positive test cases for a given property are derived from the formal model representation, in which the property specification serves as a predicate for generating instances that conform to the very property. Similarly, negative test cases are generated from the formal model representation, if we consider it as a predicate to identify counterexamples, which satisfy the negated property. As a model-based testing framework, MCTF can assist test activities at property and behavior levels [13].

Property Testing We take *Property 2* as an example to demonstrate the process of automated test generation for testing a given property from positive and negative aspects. To simplify the test case generation process, we remove the parameters and properties that are not related with action test. The following predicate is defined to derive the positive test cases for the corresponding facts in the formal property specification.

```
pred P2_pos(r: Resolver, i:Intent) {
  all r: Resolver, i: Intent, f:Filter |
  one i.action and i.action in f.action
      <=> i->f in r.IIF_A}
```

This predicate checks *Property 2* against the model representation of Android ICC, then instances are generated. The generated instances are used to construct positive test cases to ensure that the system should always permit a matched pair of intent object and intent filter object.

The corresponding negative test cases for *negated Property 2* are generated to ensure the system never denies a matching pair or accepts a mismatching pair. In order to derive negative test cases, we specify the negative property with Alloy as follows:

```
pred P2_negDeny(r:Resolver, i:Intent,
                f:Filter)
 {i->f not in r.IIF_A
 and i.action in f.action
 and i.action!=none
}
```

```
pred P2_negAccept(r:Resolver, i:Intent,
                  f:Filter
 ){i->f in r.IIF_A
 and i.action not in f.action
 and i.action!=none
}
```

Alloy Analyzer requires a bounded input domain, specified by the number of intents, intent filters, resolvers, action strings in our example, to generate instances and counterexamples. The size of input domain determines the total number of generated test cases. Then, we come up with the question of choosing an appropriate size for generating test cases that achieve reasonable coverage. Although testers can specify a large input domain and get millions of test cases for a trivial property with respect to the coverage, it is not always the case. The testers need to specify the input size based on practical test requirements [3].

For example, we specify the following input domain to test *Property 2*.

```
run P2_pos for
  exactly 1 Resolver, exactly 2 actionStr,
  exactly 2 Str, exactly 2 Intent,
  exactly 2 Filter
```

```
run P2_negDeny for
  exactly 1 Resolver, exactly 2 actionStr,
  exactly 2 Str, exactly 2 Intent,
  exactly 2 Filter
run P2_negAccept for
  exactly 1 Resolver, exactly 2 actionStr,
  exactly 2 Str, exactly 2 Intent,
  exactly 2 Filter
```

Figure 3 depicts a positive test case generated by Alloy Analyzer for *Property 2*. Both `Intent` and `Filter0` have the same action. Thus, `Resolver` allows the interaction between them. Figure 4 and Figure 5 depict two negative test cases. In Figure 4, `Resolver` unexpectedly denies `Intent` from accessing `Filter1` (marked by `(f)` and `(i)`). In Figure 5, `Resolver` unexpectedly accepts `Intent` and `Filter1` (marked by `(f)` and `(i)`), which have different actions.

Behavior Testing After each property has been tested independently, we can further check behaviors of the intent resolution module. Here, we give a more complex scenario to test all modeled intent filter properties. Based on the aforementioned properties, we instruct Alloy Analyzer to enumerate all assignments, simulating inter-component communications.

To test if a system always properly delivers the intent to correct recipients, we need positive test cases that are composed of matched pairs of intents and intent filters. In our model, it implies the set of `iif` relation should not be empty. Therefore, we have the following specification:

[3] The testers should balance the coverage and the input size, which are normally obtained from subject matter experts and prior testing results.

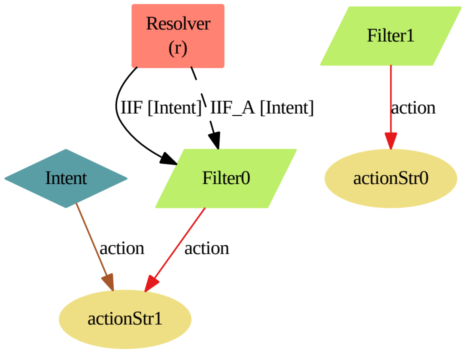


Fig. 3. Abstract Test Cases for Property Testing: Positive

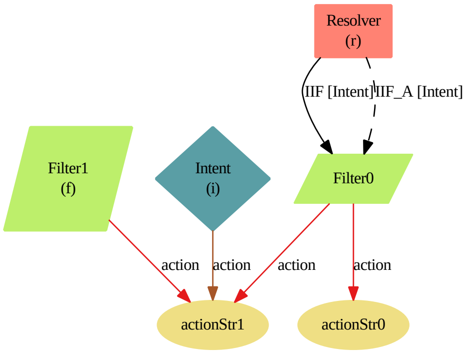


Fig. 4. Abstract Test Cases for Property Testing: Negative Deny

```
pred Positive(r: Resolver){
    #r.IIF>0 }
```

On the contrary, negative test cases are those without paired intents and intent filters. We simply set the size of `iif` to zero.

```
pred Negative(r: Resolver){
    #r.IIF=0 }
```

Figure 6 depicts a positive test case for behavioral testing. In this example, two successful intent deliveries can be identified from the arrows labeled with "`IIF[Intent]`": Intent0→Filter0, Intent1→Filter1.

In addition, the test case generation can be optimized to avoid generating *isomorphic* test cases by adopting the approach proposed in [8]. Finally, each abstract test case is exported to an independent file which contains the test conditions and variables for further processing. Because we are using Alloy Analyzer, one of the available choices is to export test cases to DOT files, which store test cases as hierarchical drawings of direct graphs. This is a perfect choice for visualizing abstract test cases. Another choice is to export test cases into

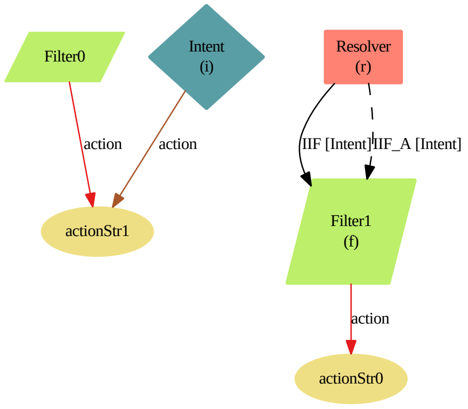


Fig. 5. Abstract Test Cases for Property Testing: Negative Accept

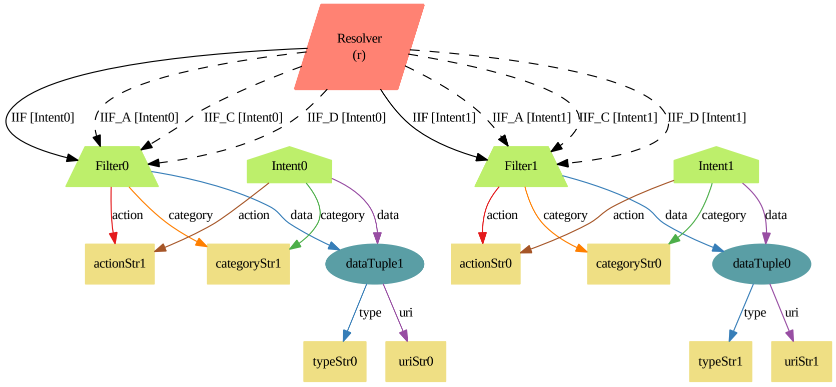


Fig. 6. A Positive Test Case for Behavior Testing

lightweight XML files, which are easy to parse with existing tools. We adopt the latter for generating executable test cases.

3.3 Test Case Translation

Except for requirements, Android SDKD also provides guidelines of Android testing framework and testing Android applications. Android CDD and Compatibility Test Suite (CTS) [1] provides additional guidelines for testing Android. Android test suites are based on JUnit [18] and Android's JUnit extensions. The extensions provide component-specific test classes and helper methods to help creating mock objects and controlling lifecycle of a component. In addition, CTS is shipped with an automated test harness. Testers can choose to use the test harness of Android CTS, use a third-party test harness, or write their own test runner based on the APIs provided by Android testing framework.

Abstract test cases generated by Alloy Analyzer in our approach cannot be directly integrated into test suites for execution as they are at different abstraction levels. Thus, an additional step is required to translate abstract test cases encoded in XML to executable test cases, involving information extraction and source code construction.

Extraction We employ a Python script to parse XML and regroup essential information fields with cElementTree [4]. cElementTree is a Python package for efficiently managing XML files.

In order to construct an executable test case for testing intent resolution, we need to know all the variables, attributes and their assigned values. In our case, the variables are intents and intent filters, and the attributes are component name, action, category, data, URI and type. An XML-encoded abstract test case is composed of several fields and tuples. Each field stands for an attribute. And each field consists of some tuples, which store a variable and the value of the attribute of that variable. Hence, information extraction can be achieved by enumerating tuples and fields and reorganizing them.

Suppose we have a fragment of an XML-encoded abstract test case as shown below:

```
<field label="action" ID="13" parentID="11">
<tuple> <atom label="Intent$2"/>
    <atom label="actionStr$0"/> </tuple>
</field>
<field label="category" ID="14" parentID="11">
<tuple> <atom label="Intent$2"/>
    <atom label="categoryStr$0"/> </tuple>
<tuple> <atom label="Intent$2"/>
    <atom label="categoryStr$1"/> </tuple>
</field>
```

From this fragment we can identify an Intent object `Intent2`. Its action is assigned to `actionStr0`, its category is assigned to {`categoryStr0, categoryStr1`}.

Code Construction The extracted information fields are utilized for a test case template and Java code fragments for Android Compatibility Test Suite (CTS). Our template is strictly complied with the format and syntax of test cases defined in Android CTS.

The sample code shipped with Android CTS offers practical examples of how to write executable test cases. We give a code template for testing Android ICC.

```
IntentFilter filter = new Match(
    String[] actions, String[] categories,
    String[] dataTypes, String[] uriSchemes,
    String[] uriAuthoroties, String[] uriPorts);
checkMatches(filter, new MatchCondition[] {
    new MatchCondition(
    int expectedResult,
    String action, String[] categories,
    String dataType, String dataURI); }
```

With the extracted information in the template, we get several Java code fragments at the end of this step.

3.4 Test Case Execution

After integrating the code fragments into existing test suites or a new test suite, executable test cases are derived by compiling fragments. Such test suites are run by a test runner that loads the test cases, runs and tears down each test. We use Android's Instrumentation Test Runner [3], which is a set of control methods and hooks in Android platform, to run our generated test cases. For each executable test case, the results are generated accordingly as we discussed in our framework. Finally, a report is presented in an HTML page including test results.

4 Implementation and Evaluation

In this section, we give a brief introduction of our tool set, which constitutes a tool chain for model-based conformance testing. As depicted in Figure 7, our tool chain consists of three tools: Alloy Analyzer, the Translator and the Android Instrumentation Test Runner. The formal representation of models and properties are fed into Alloy Analyzer for automatically generating test cases. Alloy Analyzer exports the generated abstract test cases to intermediate XML files. Then, our translator parses XML and constructs Java code fragments. The output of test case translation is an Android application package containing compiled JUnit test cases. Finally, Android Instrumentation Test Runner executes test suite and generates the test report.

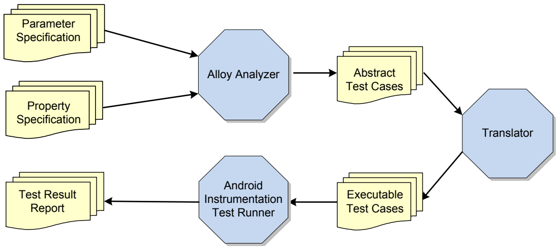
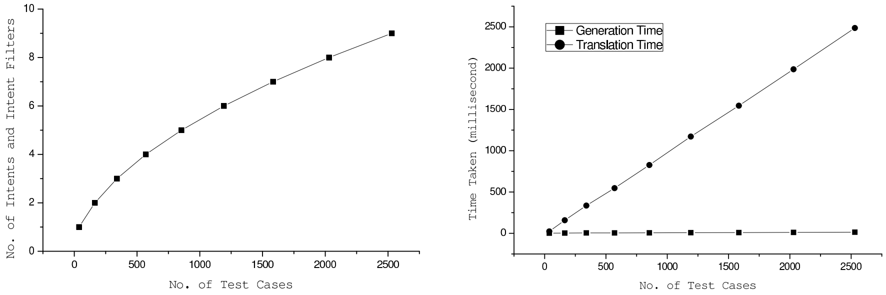


Fig. 7. A tool chain that supports MCTF

We provide a contrastive analysis between Android CTS and our generated test cases to demonstrate effectiveness of our framework in this section. For property testing, every property of the three tests need to be rigorously checked. We identified that Android CTS fails to check some properties from positive or negative aspects. Table 1 shows a comparison between Android CTS and the test cases generated by our approach. The table shows that Android CTS test suites are not offering sufficient test coverage. And our approach could achieve better coverage than that of Android CTS.

To evaluate the efficiency of our approach, we also examined two core processes, test case generation and test case translation, in our implementation.

Figure 8(a) shows that the increase of the total number of generated test cases is proportional to the number of intents and intent filters. Figure 8(b) shows that the processing time taken for test case generation and translation increases linearly with the increase of the number of the test cases, indicating



(a) Amount of Generated Test Cases

(b) Processing Time

Fig. 8. Performance Evaluation

that our approach provides a feasible and promising solution to facilitate and enhance conformance testing for Android platform.

Table 1. Conformance testing achieved by Android CTS and our approach

Property	Positive/Negative	Android CTS		MCTF	
		Covered	#Test cases	Covered	#Test cases
Property 1	Positive	×	0	√	16
	Negative	×	0	√	18
Property 2	Positive	√	3	√	24
	Negative	√	2	√	14
Property 3	Positive	√	2	√	24
	Negative	×	0	√	10
Property 4	Positive	√	4	√	26
	Negative	√	4	√	10
Property 5	Positive	√	2	√	26
	Negative	×	0	√	12
Property 6	Positive	√	2	√	31
	Negative	√	2	√	18
Property 7	Positive	√	1	√	31
	Negative	√	3	√	20
Property 8	Positive	√	1	√	31
	Negative	×	0	√	20
Property 9	Positive	×	0	√	31
	Negative	√	2	√	26

5 Related Work

Most recent work related to software testing in Android addresses automated GUI testing for Android applications. Amalfitano et al. [6] proposed a crawling-based approach to generate GUI test cases. They designed a tool to simulate events on the user interfaces, generate event transition tree by capturing application responses, and predict future events at runtime. In contrast, our approach is the first attempt to explore rigorous conformance testing for Android. In particular, we adopt a model-based approach to automatically generate test cases.

Model-based approaches have been widely used for testing in various fields. Several researchers proposed automated frameworks for testing Java programs, such as Korat [8] and TestEra [21]. Korat constructs Java predicates and generates all non-isomorphic inputs for which the predicates return true, by searching and enumerating a given bounded input space. TestEra works in a similar way as Korat, but using a first-order relational language and existing SAT solvers. Both approaches use structural invariants on the input data to automatically generate test cases and then test the output against a set of predicates. However, the generated test cases are abstract and need to perform the translation task to generate the actual code. In our work, we attempt to extend model-based approaches to testing Android platforms. We also demonstrate how test cases can be integrated to perform conformance testing effectively.

Security for mobile devices and applications is a growing concern recently. TaintDroid [14] monitors and controls access to sensitive data by dynamic taint-based information flow tracking. Stowaway [16] identifies vulnerabilities in applications by static analysis on application packages, manifests and bytecodes. Chaudhuri [9] proposed a formal language to describe applications and reason about information flows and the consistency of security specifications.

6 Conclusion

While several automated testing frameworks have been proposed and developed for smartphone platforms, developers still need systematic approaches and corresponding tools to generate test cases for conformance testing efficiently and effectively. To address this issue, we have proposed a novel framework to enable rigorous conformance testing for the Android platform. Our framework adopted a model-based approach which utilizes formal verification techniques to automatically generate test cases. In addition, we have demonstrated the feasibility of our approach with Android ICC.

In our current framework, testers need to manually derive the model from requirements. As part of our future work, we would explore an approach for directly constructing model from the requirements, leveraging the capability of NLP techniques [22]. Moreover, we would apply our approach to other Android modules, such as Activity Manager and Package Manager.

References

1. Android compatibility. `http://source.android.com/compatibility/`.
2. Android sdk cocument. `http://developer.android.com/reference/android/package-summary.html/`.
3. Android testing fundamentals. `http://developer.android.com/guide/topics/testing/testing_android.html#Instrumentation`.
4. The elementtree xml api. `http://docs.python.org/library/xml.etree.elementtree.html`.

5. Apple rises to the top as worldwide smartphone market grows 65.4% in the second quarter of 2011, idc finds, August 2011. `http://www.idc.com/getdoc.jsp?containerId=prUS22974611`.
6. D. Amalfitano, A. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252–261. IEEE, 2011.
7. B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.
8. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM, 2002.
9. A. Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 1–7. ACM, 2009.
10. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, pages 239–252, 2011.
11. E. Clarke, O. Grumberg, and D. Peled. Model checking. 2000.
12. C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, 2007.
13. S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 285–294. IEEE, 1999.
14. W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI*, 2010.
15. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.
16. A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. Technical report, 2011.
17. M. Fewster and D. Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.
18. E. Gamma and K. Beck. Junit: A cooks tour. *Java Report*, 4(5):27–38, 1999.
19. H. Hu and G. Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, pages 195–204. ACM, 2008.
20. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
21. S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434, 2004.
22. D. Lewis and K. Jones. Natural language processing for information retrieval. *Communications of the ACM*, 39(1):92–101, 1996.
23. D. Mitchell. A sat solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85(112-133):12, 2005.
24. J. Robinson and A. Voronkov. *Handbook of automated reasoning*, volume 1. North Holland, 2001.