DiPSUnit: a JUnit Extension for the DiPS Framework

Sam Michiels, Dirk Walravens, Nico Janssens, Pierre Verbaeten
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium,
+32 16 327640
{Sam.Michiels, Dirk.Walravens}@cs.kuleuven.ac.be

ABSTRACT
Testing system software (such as protocol stacks or file systems) often is a tedious and error-prone process. The reason for this is that such software is very complex and often not designed to be tested. This paper describes DiPSUnit, a JUnit extension, which allows fine-grained as well as composed units to be tested consistently. Although non-trivial test support is provided, using DiPSUnit keeps testing simple and intuitive.

Keywords
Testing, framework, component software engineering

1 INTRODUCTION

Testing system software, such as a protocol stack or a file system, is a complex, tedious and error-prone task. The basic problem is that, for performance reasons, system software is often designed as a monolithic block of multi-threaded software. This prevents such software from being tested properly because of two reasons: first, it is very difficult to isolate the basic building blocks as stand-alone units that are independent from each other. Second, concurrency code, which is introduced in such multi-threaded system software, often crosscuts the code [3].

We present the DiPSUnit test framework, which is an extension of JUnit [2] specifically for DiPS (Distrinet Protocol Stack) [4] units. DiPSUnit offers a uniform way of testing because all DiPS units share the same interface. This keeps testing very intuitive and simple. However, the provided support for testing units in isolation in the presence of concurrent behavior and external control events is not trivial.

2 THE DIPS FRAMEWORK

DiPS is a Java component framework based on independent units that are connected as a pipe-and-filter architecture. The framework supports the development of system software such as protocol stacks or file systems. Communication between DiPS units is intercepted by the framework. This allows for units to communicate anonymously, since they have no explicit notion of other units in the system.

A DiPS unit is an object-oriented entity with a very specific (fine grained) responsibility. A distinction has been made between *purely functional units* and *concurrency units*. This separation allows the concurrency model to change, independent from the functionality in the system. DiPS units can be grouped together into more coarse grained composed units (such as a protocol layer in a protocol stack). All units process an explicit semantic entity (`Packet`). These packets can enter and leave a DiPS unit through one or more entry and exit points. A singular DiPS unit (such as a fragmenter or an encryption unit) with one entry and one exit (`PacketReceiver` and `Packet Forwarder`) is shown in detail in the left figure. Next to the data (`Packet`) flow there is a control flow that allows inter-unit control communication via *DiPS events*.

3 DIPS UNIT TESTING

The DiPSUnit framework provides a consistent way to test both singular and composed units. DiPSUnit has been developed specifically for DiPS but has some interesting generic test characteristics. Since the test framework is separated from a tested unit, a *test-first development* [2] approach is possible.

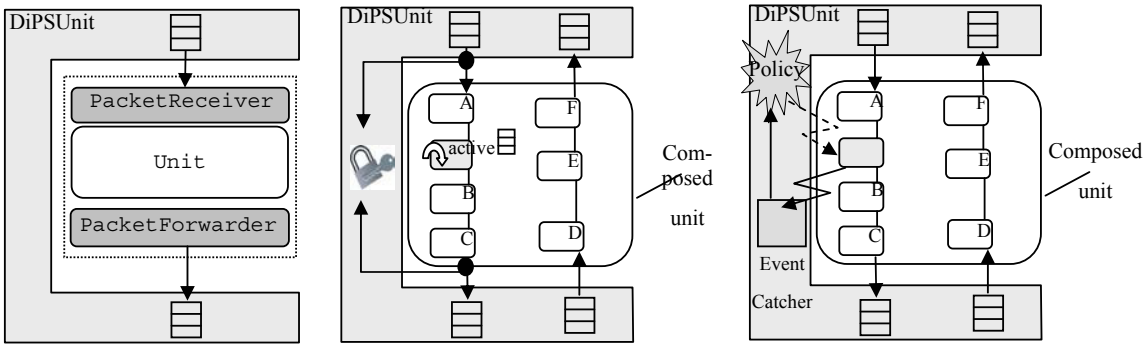
The DiPSUnit framework provides generic infrastructure support to plug in a unit (regardless of its number of entries or exits). First, a `PacketContainer` is provided to manage so-called packet buffers. An entry buffer is used to store specific test packets created for the test. An exit buffer collects result packets that are received from the tested unit during a test run. The `PacketContainer` is initialized by creating a packet buffer for each entry and exit of the unit. Second, the `Linker` links a unit with the DiPSUnit framework. This means that each entry point of the unit is linked with an entry buffer in the `Packet Container` and each exit point is linked with an exit buffer. Linking an exit buffer is done by introducing a specific `PacketReceiver` that puts incoming packets from the unit's exit point into the exit buffer. This is done completely transparent for a test developer.

The figure shows three (composed) units, which are linked in DiPSUnit. For clarity reasons, the figure shows the `PacketContainer` as 4 queues instead of 1 logical entity and unit details, such as entry and exit points, are removed in the composed units.

Dealing with Concurrency
When a concurrency (*active*) unit is present within a composed unit, a test must be suspended until all packets have arrived or until a timeout occurs (to avoid being suspended forever in case of an error). The active unit in the middle figure has an internal thread and a buffer to store incoming packets.

4 DIPSUNIT EXTENSIONS

In this section, we extend DiPSUnit to deal with more complex issues such as testing of multi-threaded (asynchronous) units and handling control flow stubs.



Consistent test approach in DiPSUnit: a singular DiPS unit, a composed unit with internal concurrency and a composed unit which sends (and receives) control events

DiPSUnit provides test support for such asynchronous units by offering a `Monitor`. A test should simply ask for such a `Monitor` to be present, the rest will be transparently handled by the framework. The framework provides the `Monitor` with a `PacketContainer` and a timeout value after which the test should continue. Because some DiPS units may introduce or remove packets, a test must specify the number of packets to expect at each of the exit points of a unit. Therefore we extend the `Packet Container` so that the number of packets to be expected at each exit buffer can be specified. DiPSUnit introduces a special `PacketReceiver` which not only stores incoming packets into the associated exit buffer, but also signals the `Monitor` for each packet it receives. This is again transparent for a test developer. The `Monitor` will send a wake-up signal to resume the test when all expected packets have arrived.

Control Flow Stubs

DiPS units can exchange control information by using DiPS events. DiPSUnit offers support to intercept control events in order to allow such units to be tested in isolation.

A test developer describes how to respond to a given event, by creating a `Policy` (which is semantically equivalent to a stub [1]). Such a `Policy` simulates the expected behavior or an error (such as a `Policy` that duplicates or omits the reply for a certain event, to simulate network errors). Separation of policies from test cases, allows reuse of generic policies. All `Policies` are registered in the DiPSUnit `EventManager`, together with the event type that triggers each of them. DiPSUnit offers an `Event Catcher`, which is responsible for catching specific types of events. The default `EventCatcher` simply delegates incoming events to the `EventManager` to look up the associated `Policy` (see right figure). More complex `EventCatchers` (e.g. to collect a number of (different) events before looking up the associated `Policy`) can easily be introduced. Note that DiPS units can be replaced (e.g. by stubs) without any code changes.

5 CONCLUSION

The combination of JUnit, DiPS and DiPSUnit is very promising. JUnit offers the basic infrastructure to develop test cases and test suites. DiPS facilitates unit testing because it forces to create modularized architectures and because it allows units to be replaced without changing any code, thanks to its anonymous communication model. Especially for complex system software, concurrency support in DiPS comes in very handy. Thanks to all this support, DiPSUnit can consistently test DiPS units, from fine grained to composed unit level. However, developing test cases is still intuitive and simple. Our experience is that DiPSUnit encourages a *test-first development* approach [2].

6 INFORMATION AND QUESTIONS

[5] provides a more detailed description of DiPSUnit. For more info, contact Sam.Michiels@cs.kuleuven.ac.be.

ACKNOWLEDGEMENTS

This research has been carried out in order of Alcatel Bell with financial support of IWT (project SCAN #010319).

REFERENCES

1. Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, 1999.
2. E. Gamma, K. Beck, *Test infected: Programmers love writing tests*, http://www.junit.org/, 1998.
3. G. Kiczales, e.a., *Aspect-Oriented Programming*, In proceedings of ECOOP'97, 1997.
4. F. Matthijs, *Component Framework Technology for Protocol Stacks*, Ph.D. thesis, K.U.Leuven, 1999. (Available at http://www.cs.kuleuven.ac.be/~samm/netwg/dips/)
5. S. Michiels, D. Walravens, e.a., *DiPSUnit: an Extension of the Junit Test Framework for DiPS*, Tech. Report CW-333, K.U.Leuven, Dept. Comp. Science, 2002.