



DITools: Application-level Support for Dynamic Extension and Flexible Composition

Albert Serra, Nacho Navarro, Toni Cortes

It would be nice to:

- Easily re-use/change programs (Paradyn/Dyninst is not easy!).....

It would be nice to:

- Easily re-use/change programs (Paradyn/Dyninst is not easy!)....
- ...without necessarily having the source code...

It would be nice to:

- Easily re-use/change programs (Paradyn/Dyninst is not easy!)....
- ...without necessarily having the source code...
- ...in user space.

Re-routing remote file reads to the network.

```
int fs_write(int fd, char* b, int s) {
    if (remote_fd(fd) ) {
        a = build_remote_request(b, s);
        r = send_request(server, WRITE, a);
    } else {
        r = base_fs_write(fd, b, s);
    }
    return r;
}
```

also,

```
(program, write) -> (mymodule, fs_write
```

Interesting example

- Replace

```
check_registration_code()
```

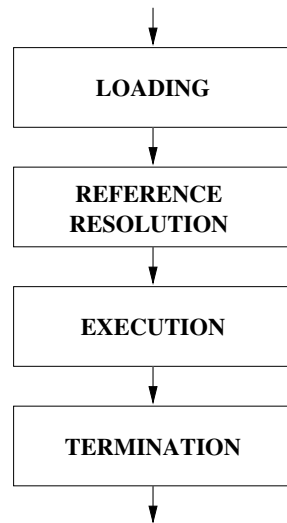
- with

```
{ return true; }
```

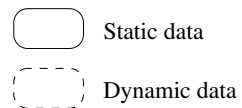
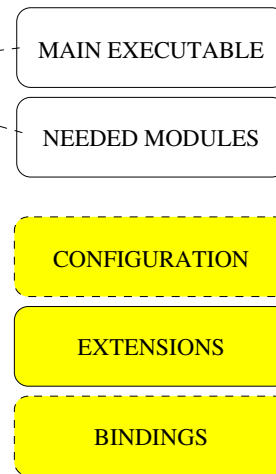
- !!!

Execution Stages

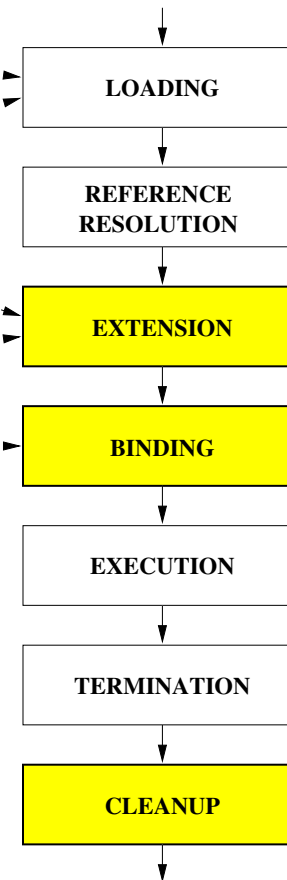
EXECUTION STAGES



INPUTS



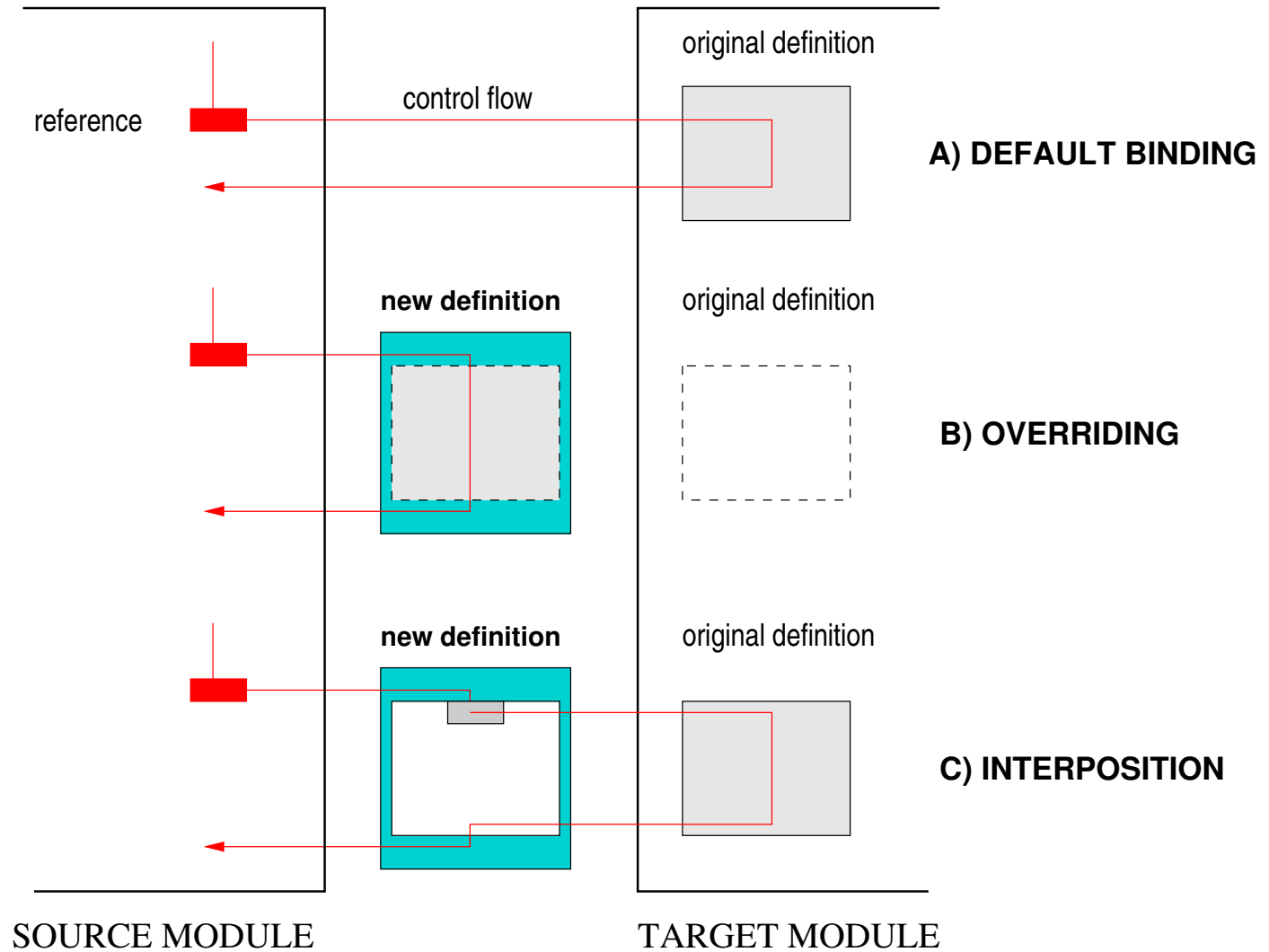
EXECUTION STAGES (EXTENDED)



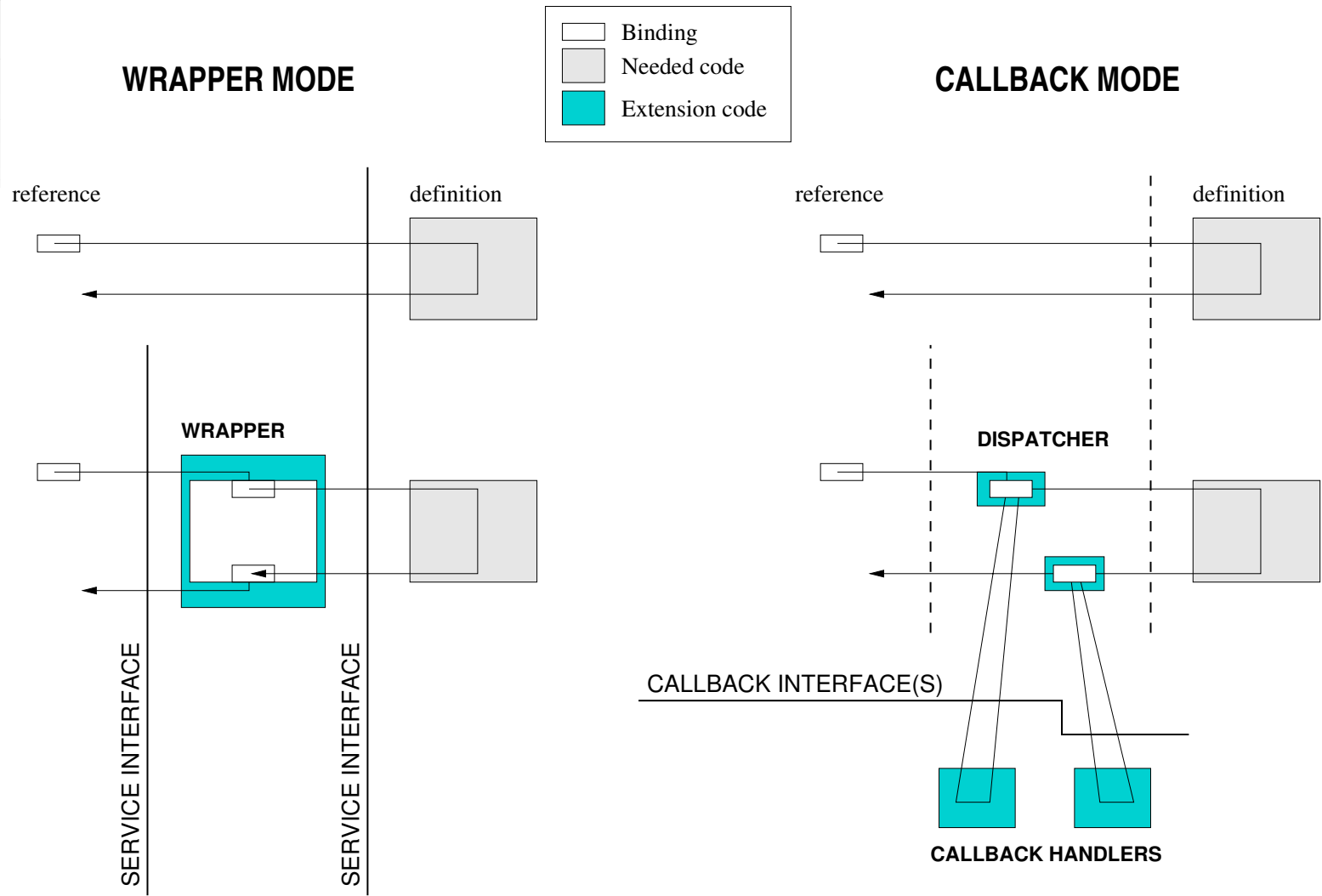
The DITOLS approach

- Object code format is mostly the same.
- Shared libraries/dynamic linking is common.
- Extend loader and linker.
- Relink everything at will - every possible combination.

Interposition



Extension Options



Implementation Characteristics

- Interposition can be done both statically and dynamically.
- Modules are allowed to initialize, and cleanup.
- Some checks to avoid errors.
- The DITools framework is “inherited” to forked processes.

Interposition examples

```
(program, read) -> (mymodule, myread)
```

Can also be done from within an (already loaded) module:

```
...  
di_rebind("program", "read",  
          "mymodule", "myread");  
...
```

Interpositions done statically, can be re-bound during run-time.

Limitations(!)

- Can handle only calls that are resolved during dynamic linking.
- Simple and easy but limited.
- Dynamic rebinding is not safe.

- Of course, it slows down everything.
- Efficiency is not the main goal.

Conclusions

- Dynamically change bindings between dynamically linked modules.
- No OS support needed.
- Good for monitoring, trace collection purposes.
- ... a good hack.



Type-Safe Delegation for Run-Time Component Adaptation

Gunter Kniesel

- OO programming eases code reuse with class inheritance.
- Component-oriented programming increases reuse even more.
- Usually, components cannot be reused “as-is”.
- Sometimes, adaptation is not possible at the source level.
- Dynamic adaptation might be necessary/wanted.

Adaptation techniques/problems

- The writer of the original component needs to have foreseen adaptation.
- Time of adaptation. Dynamical is difficult.
- Adapt the class or the objects?
- Code modification - what if there is no source code available?

Self-problem

- When replacing an object for a *wrapper* object, all calls to the original must go to the wrapper.
- Problem: even calls originating from the original.

Self-Problem - Example

```
public class DM {  
    // ... private data, constructor ...  
    int amount() { return ... }  
    void foo() { ... self.amount() ... }  
}
```

Example (cont)

```
public class DMtoEURO {
    // the wrapped component:
    DM parent;

    // constructor:
    DMtoEURO(DM p) { parent = p }

    //redefined method:
    int amount() { return parent.amount() }

    // forwarding method:
    void foo() { parent.foo() }
}
```

Example (cont)

```
public class DMtoEURO: public DM {
    // the wrapped component:
    //DM parent;

    // constructor:
    DMtoEURO(DM p) { parent = p }

    //redefined method:
    int amount() { return parent.amount() }

    // forwarding method:
    void foo() { parent.foo() }
}
```

- Orthogonal to class inheritance.
- Can be thought of as “object-inheritance.”
- Can be changed dynamically.
- Drawback: Safety.

The End

Thank you for your time