# A STRUCTURE FOR DEPENDABIL ITY CASES

ABZ 2010

Daniel Jackson & Eunsuk Kang
MIT

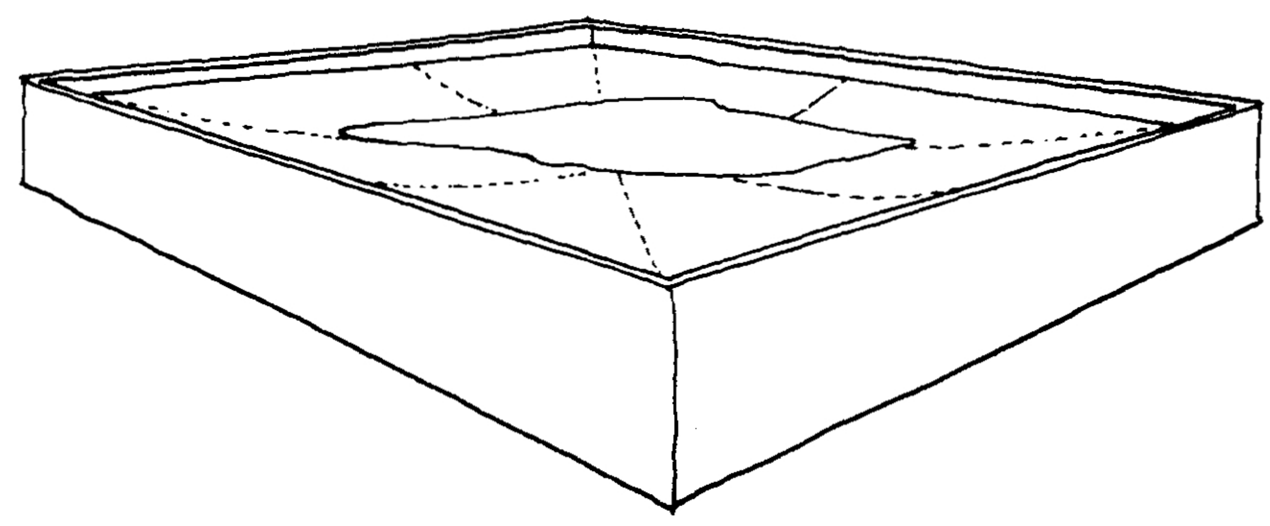
Funded by National Science Foundation and Northrop Grumman Cybersecurity Research Consortium

why does software fail?

# kemper arena, kansas city, 2007

# kemper arena, 1979

# what happened?

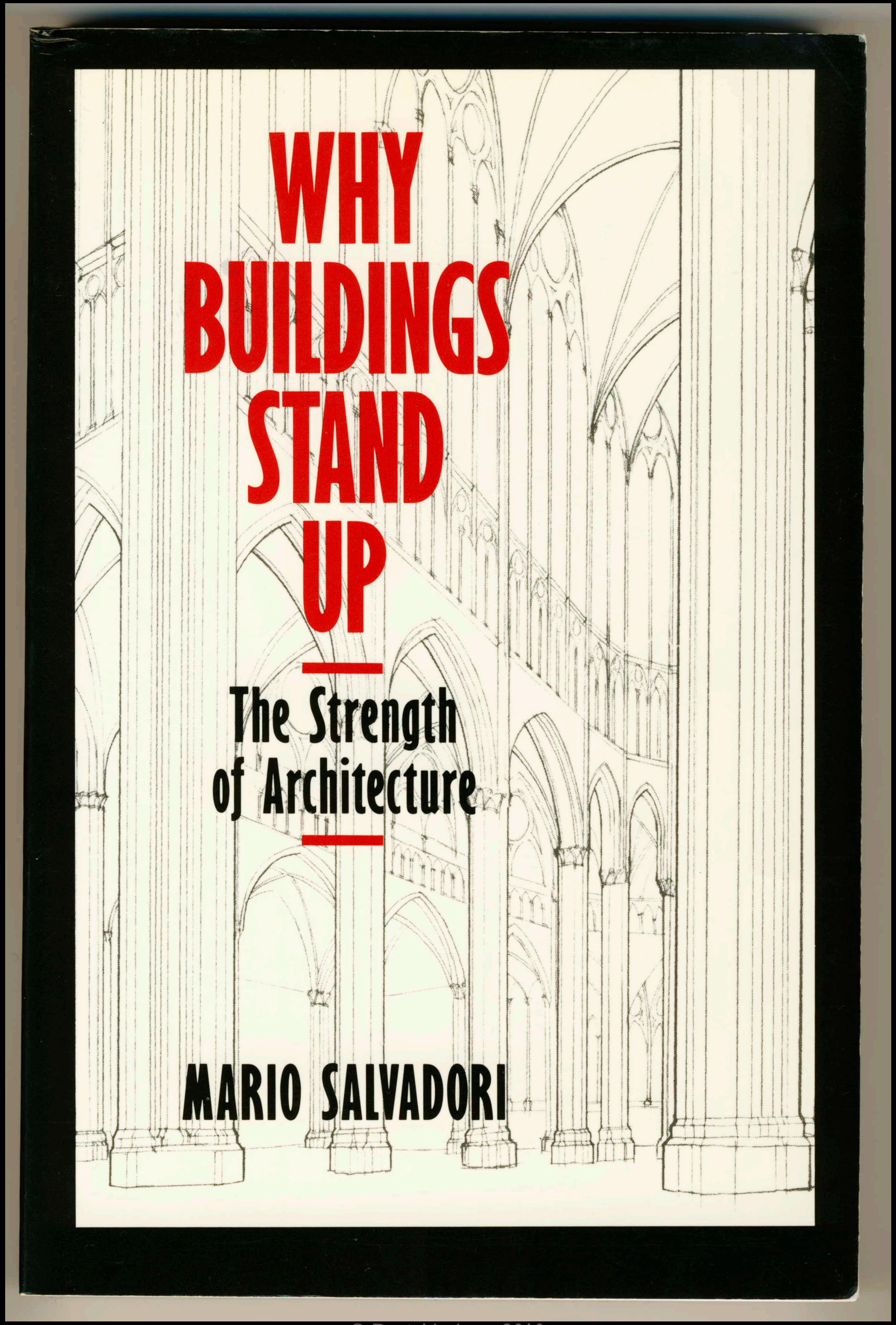


4.8 Ponding of a Flat Roof

For a common structure... ponding formulas have been derived and adopted in all structural codes... But when the ponding formulas were extended to a 4-degree system... including the long span portals... roof was unstable

Levy & Salvadori, *Why Buildings Fall Down*

# failure = flawed success story



# Therac 25

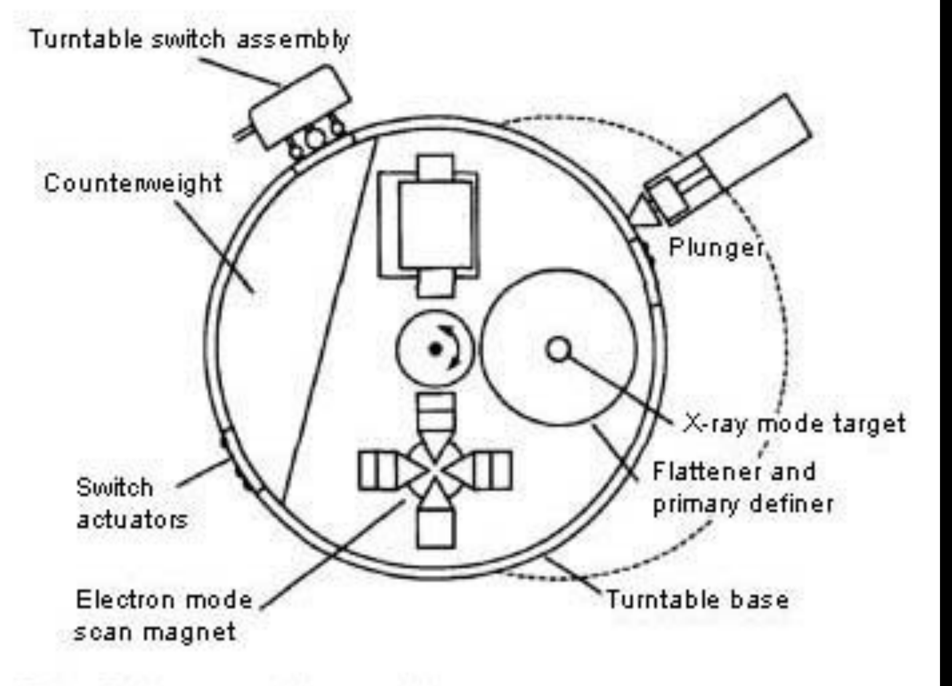


Figure B. Upper turntable assembly

AECL fault tree analysis (1983)

*did not include software*

*P(computer selects wrong energy) = $10^{-11}$*

Leveson & Turner (1993)

*race conditions, lack of interlocks, etc*

# real reasons for failure?

large attack surface

*bug anywhere can undermine entire system*

low quality throughout

*no defensive design*

*complex & brittle codebase*

no reason for success

*no articulation of critical properties*

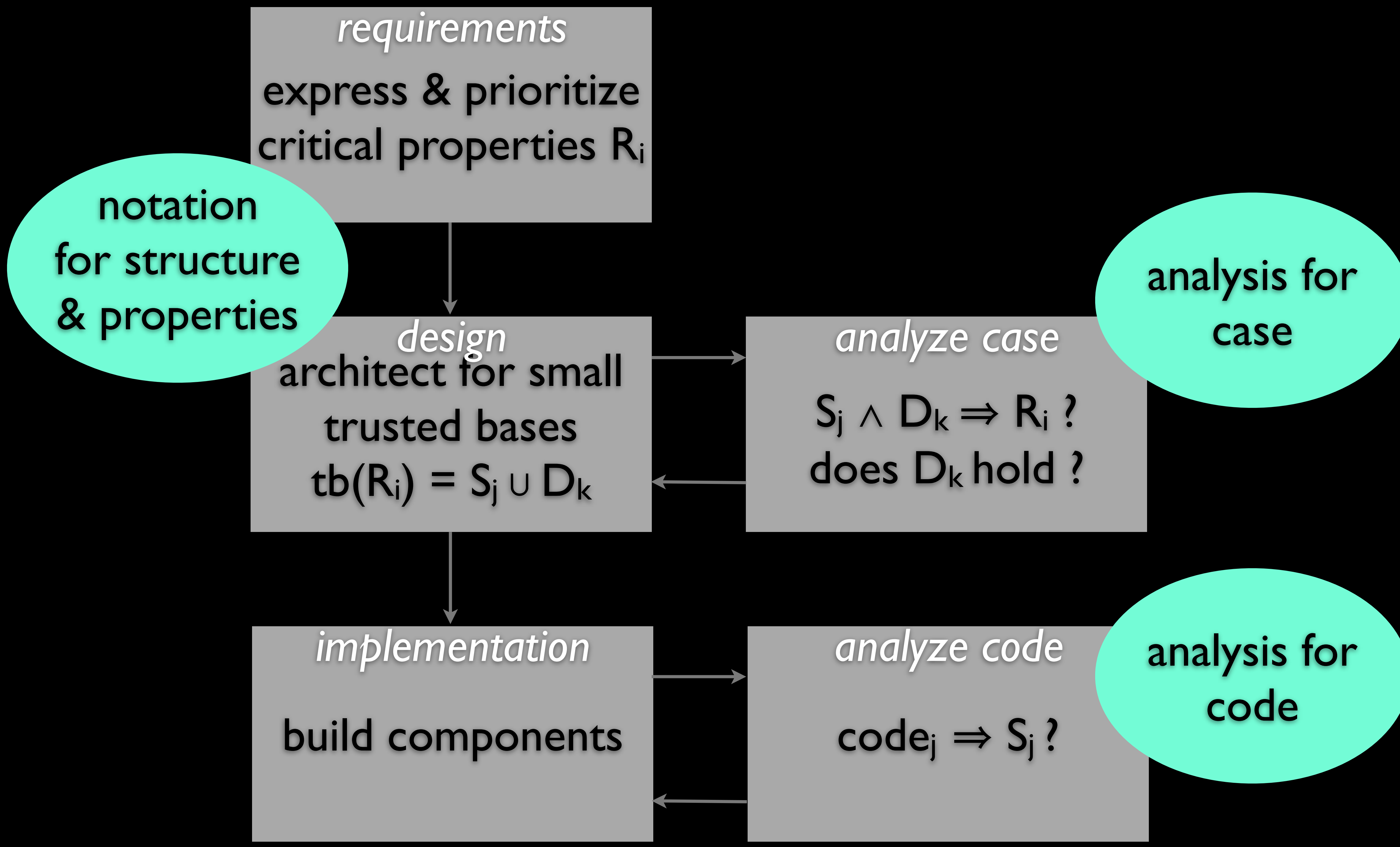
*no argument for why they hold*

# a case-based approach

# a paper about dependability cases


*A Direct Path to Dependable Software, CACM, March 2009*
*wordle thanks to Jonathan Feinberg, IBM Research, Cambridge*

# elements of approach



# what I'll show you today

a diagram notation

*from KAOS: property tree*

*from Problem Frames: machines & domains*

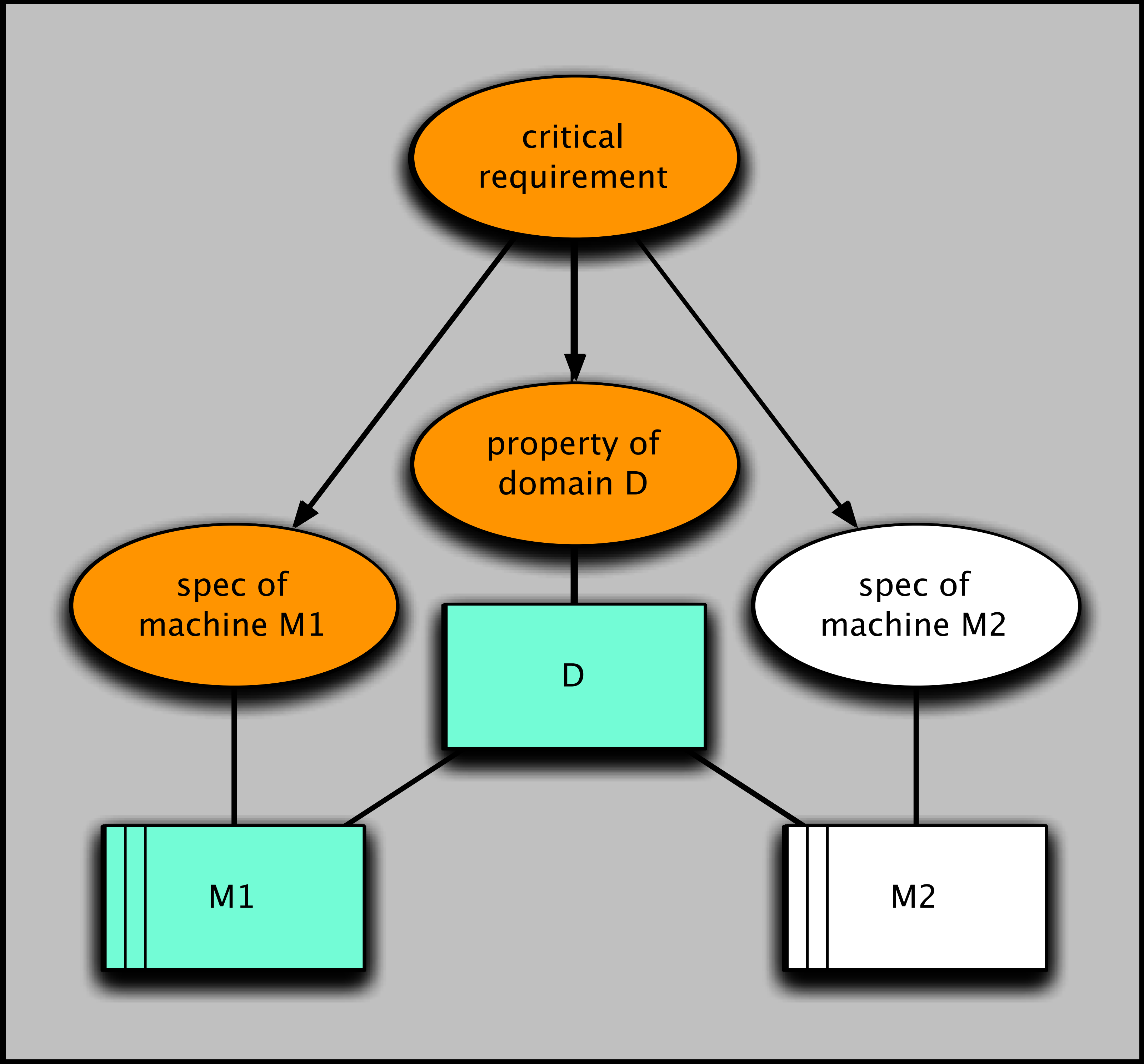
a specification idiom

*properties, machines, domains as objects*

*meta-structure becomes simple part of model*

*behaviour described statically*

# structure of a dependability case




elements

*requirement*

*machines*

*<u>domains</u>*

dependency

*requirements*

*on specs &*

*domain properties*

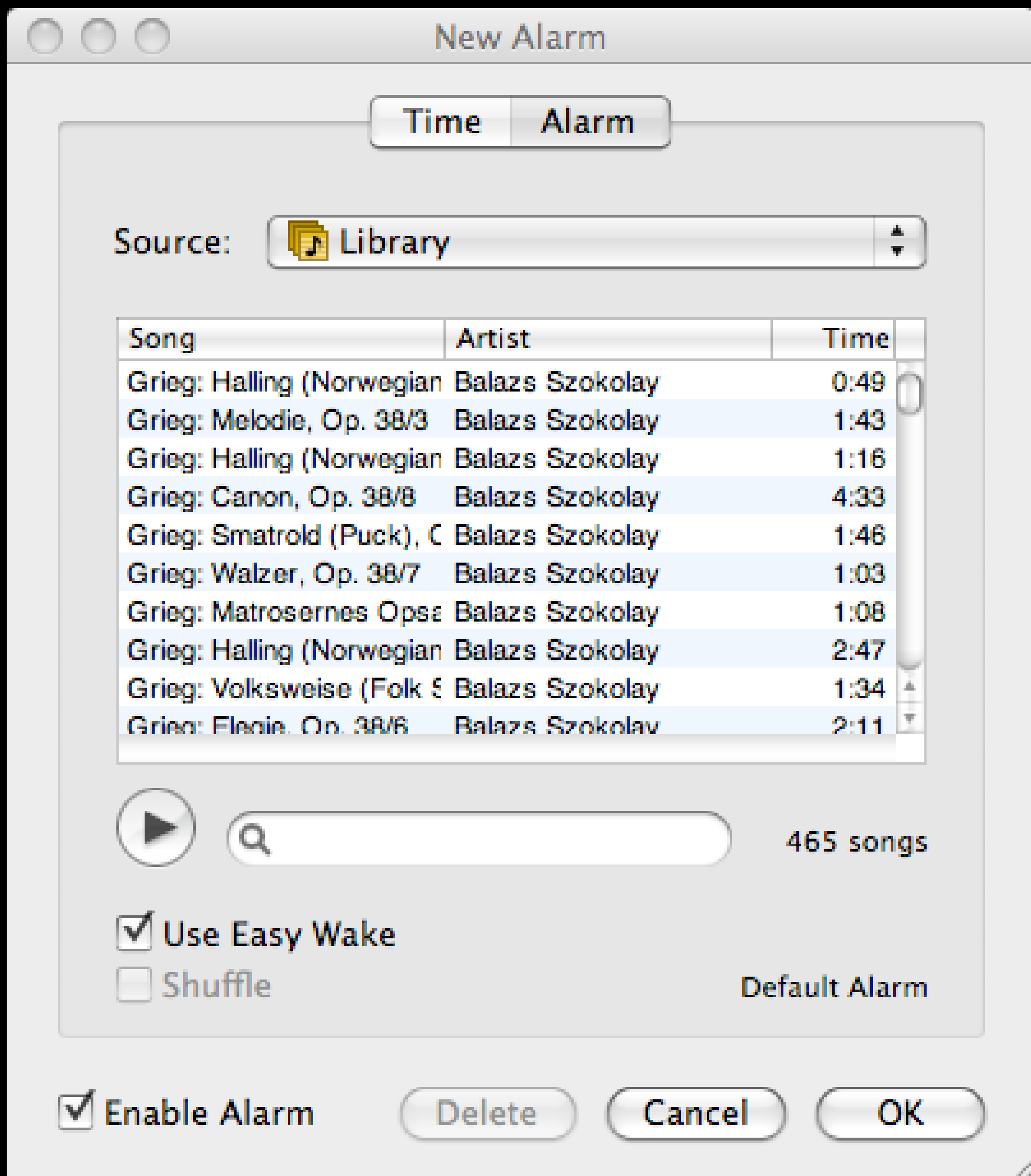
trusted base

*first find properties*

*then components*

# informal examples

# example 1: alarm clock



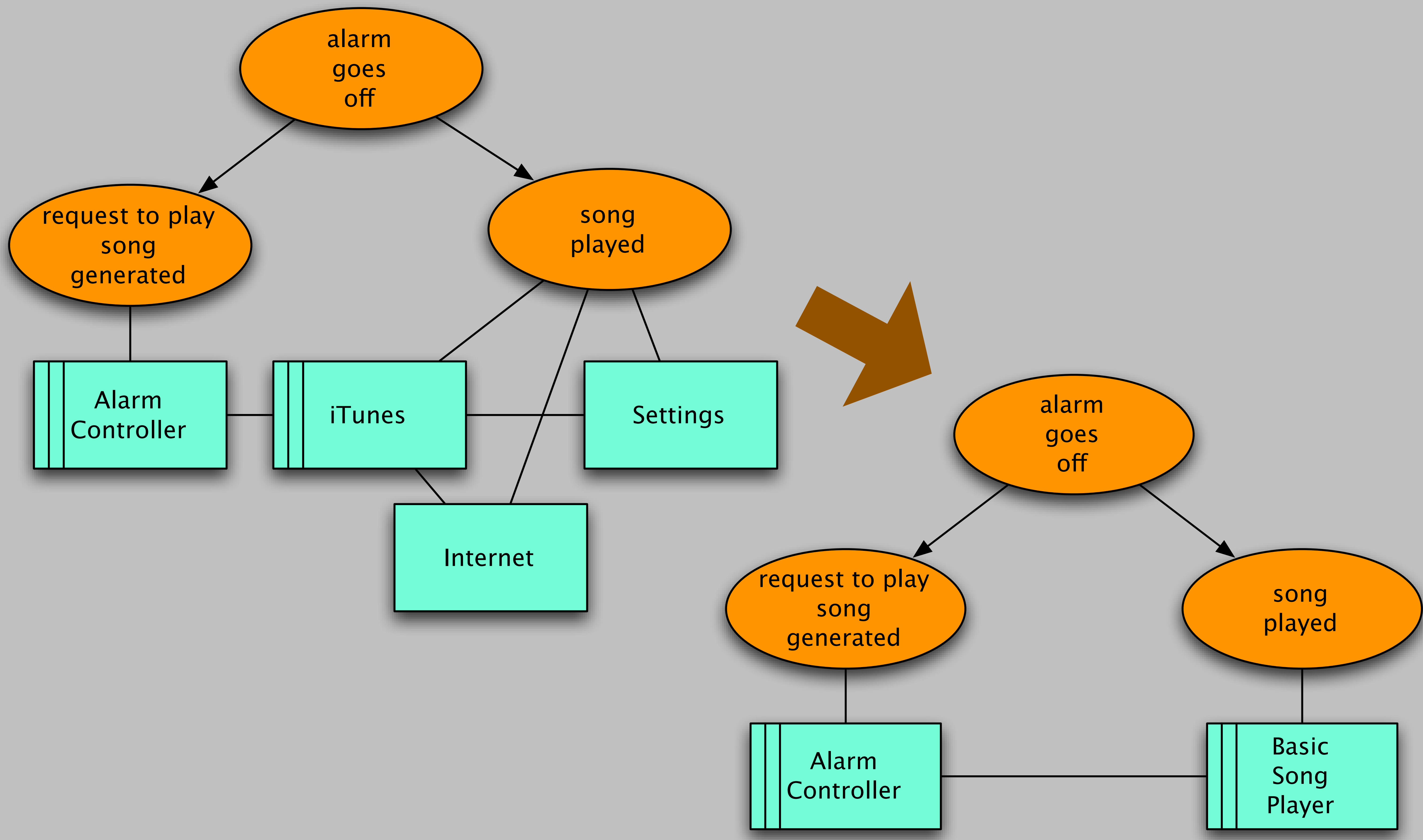

... It's only job is to wake you up in the morning, and I believe you'll find that it does it's job perfectly.

Most other alarm clock applications choose to play the alarms/music via iTunes (via AppleScript). I deliberately decided against this... Consider...

- The alarm is set to play a specific song, but the song was deleted.
- The alarm is set to play a specific playlist, but you renamed the playlist, or deleted it.
- The alarm is set to play a radio station, but the internet is down.
- iTunes was recently upgraded, and requires you to reagree to the license next time you launch it. The alarm application launches it for the alarm...
- You had iTunes set to play to your airTunes speakers, but you left your airport card turned off.
- You had the iTunes preference panel open. (Which prevents AppleScript from working)
- You had a "Get Info" panel open. (Which also prevents AppleScript from working)

From Alarm Clock, http://www.robbiehanson.com/alarmclock/faq.html

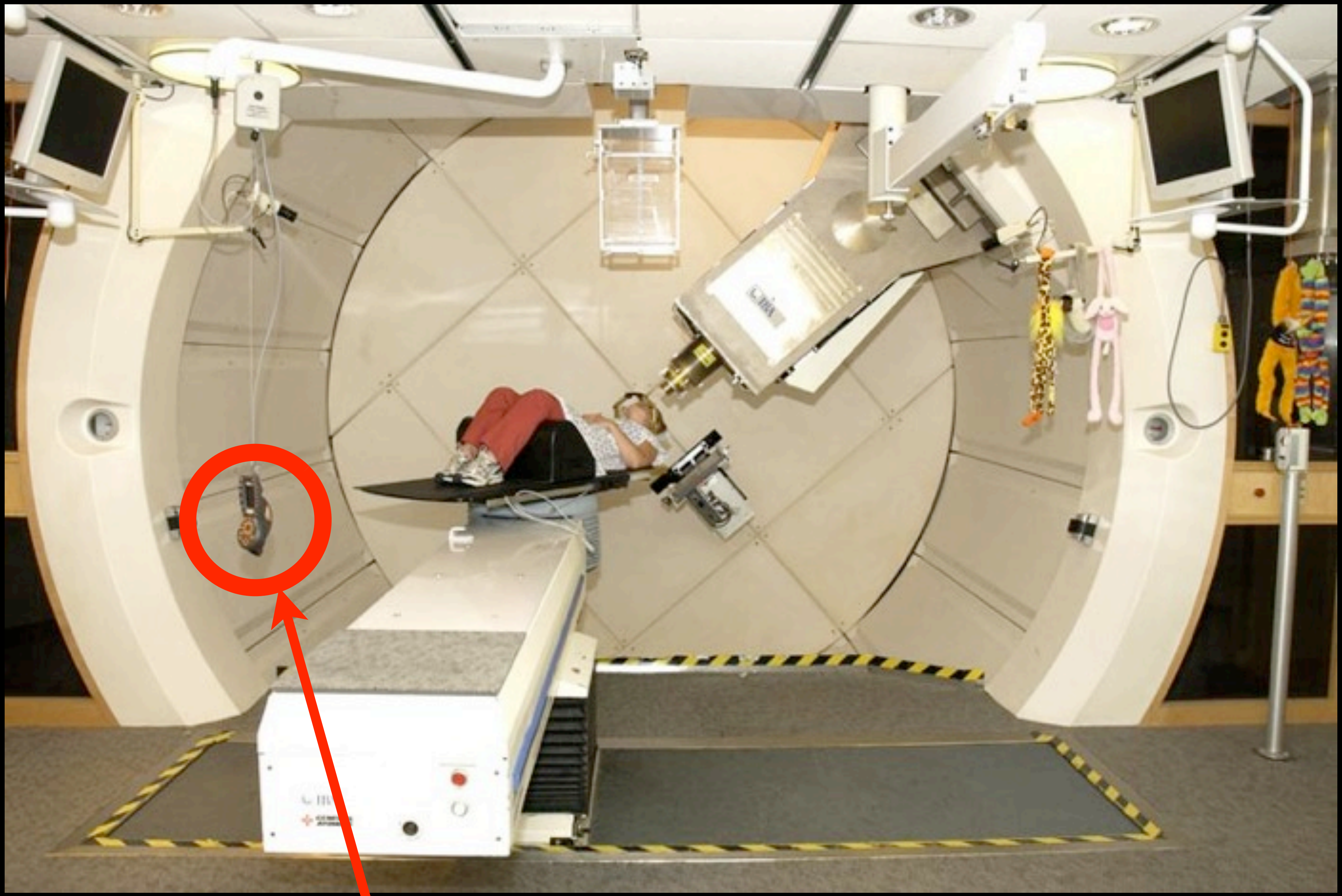
# example: alarm clock





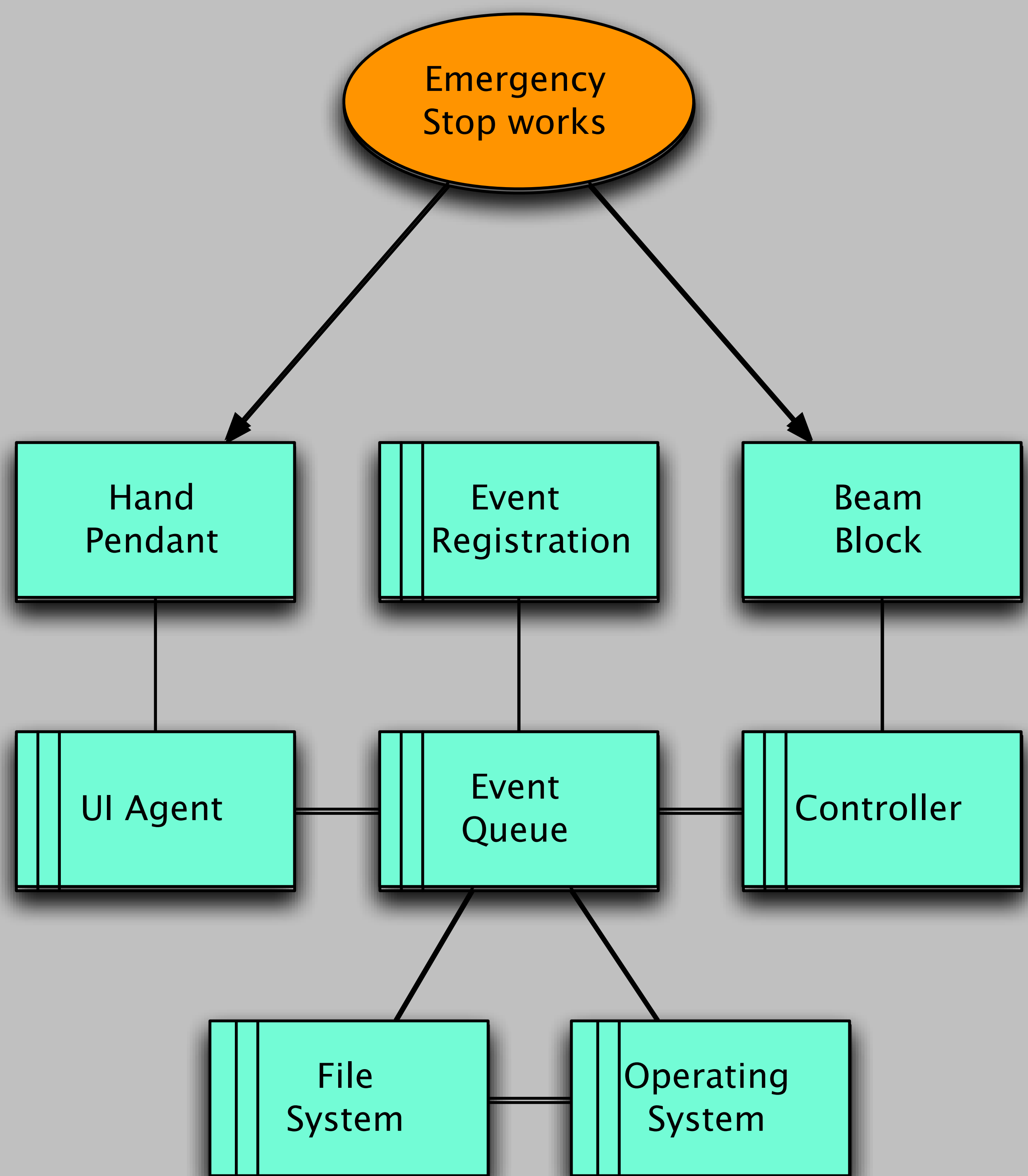

From Alarm Clock, http://www.robbiehanson.com/alarmclock/faq.html

# example: emergency stop

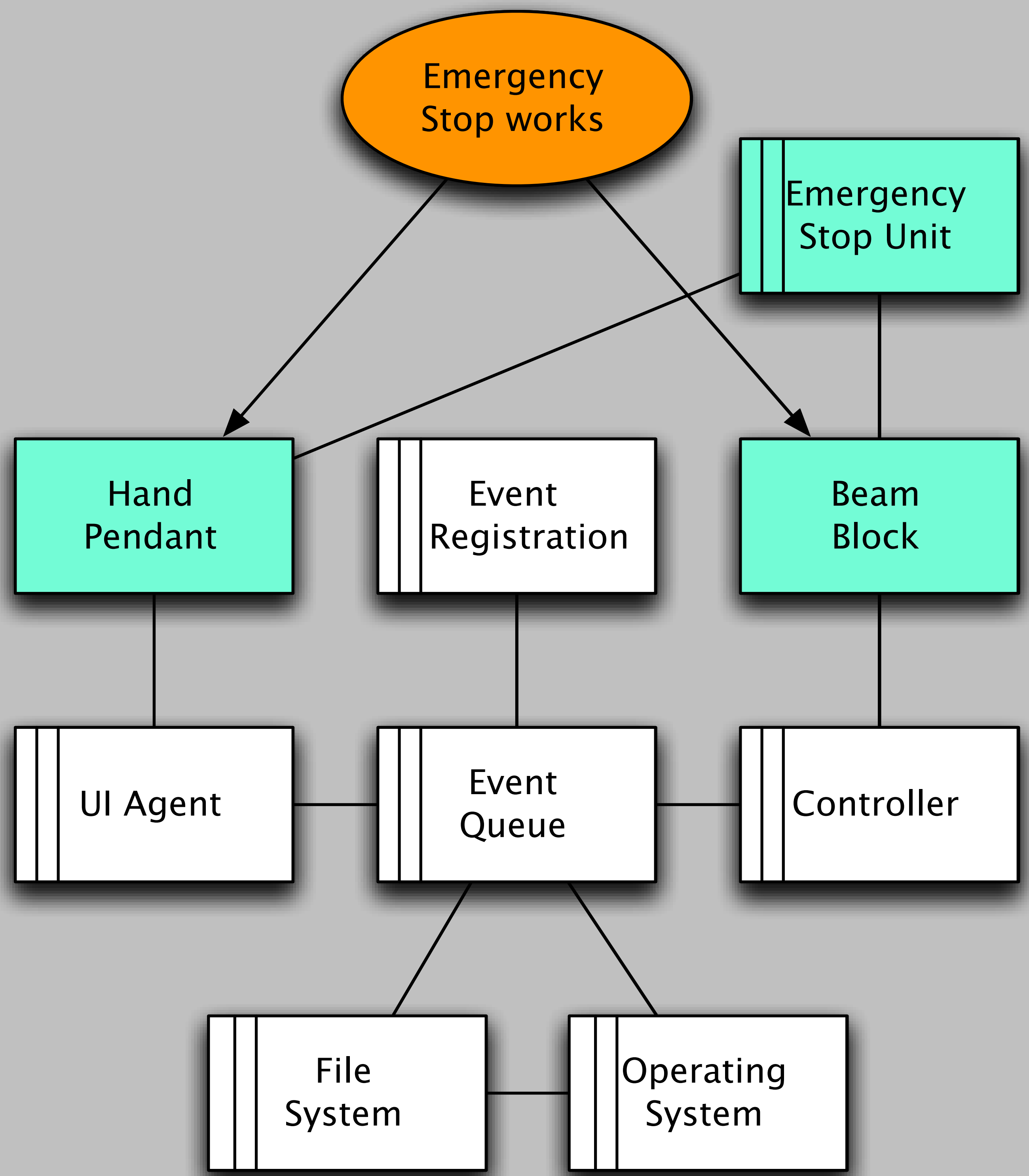


*hand pendant with stop button*

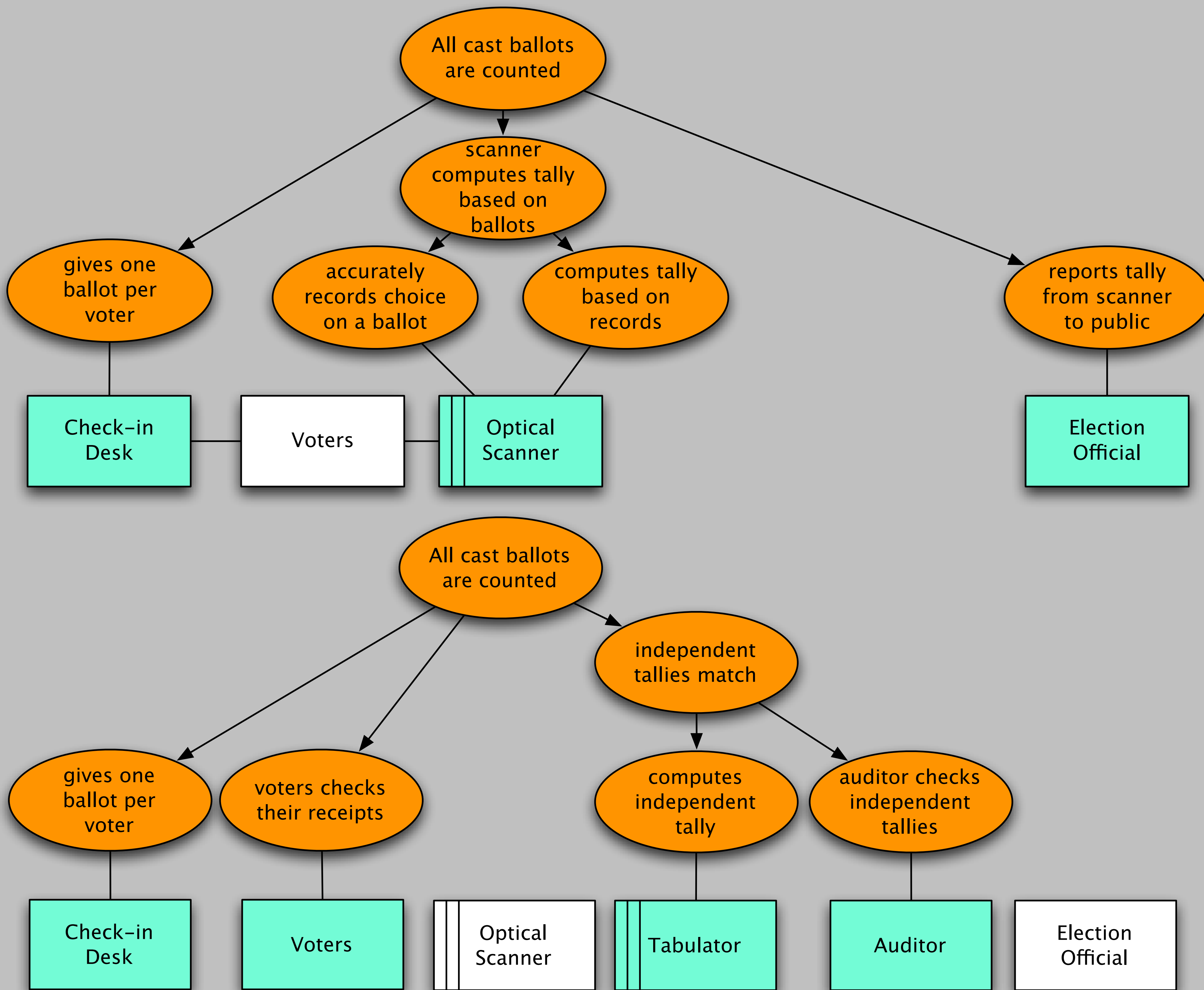
# emergency stop design



# emergency stop (re)design



# example: voting

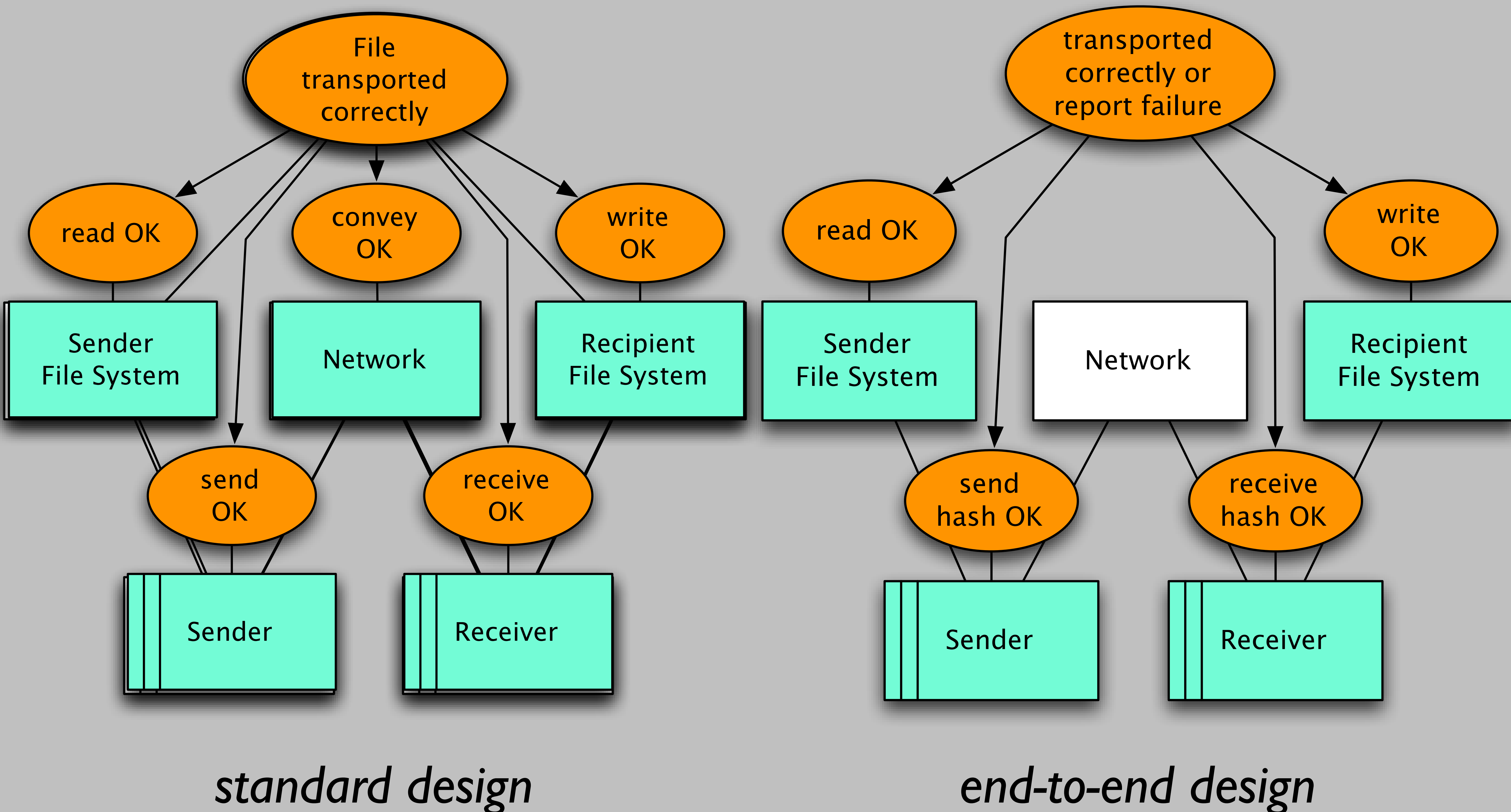



*standard design, relying on scanner*

*Scantegrity design, relying on voters and 3rd party tabulators*

# an example, formally

# file transfer




From: Jerome H. Saltzer, David P. Reed and David D. Clark. End-To-End Arguments In System Design (1984).

# aim

make this precise

*syntax & semantics for diagrams*

*textual form to elaborate in full*

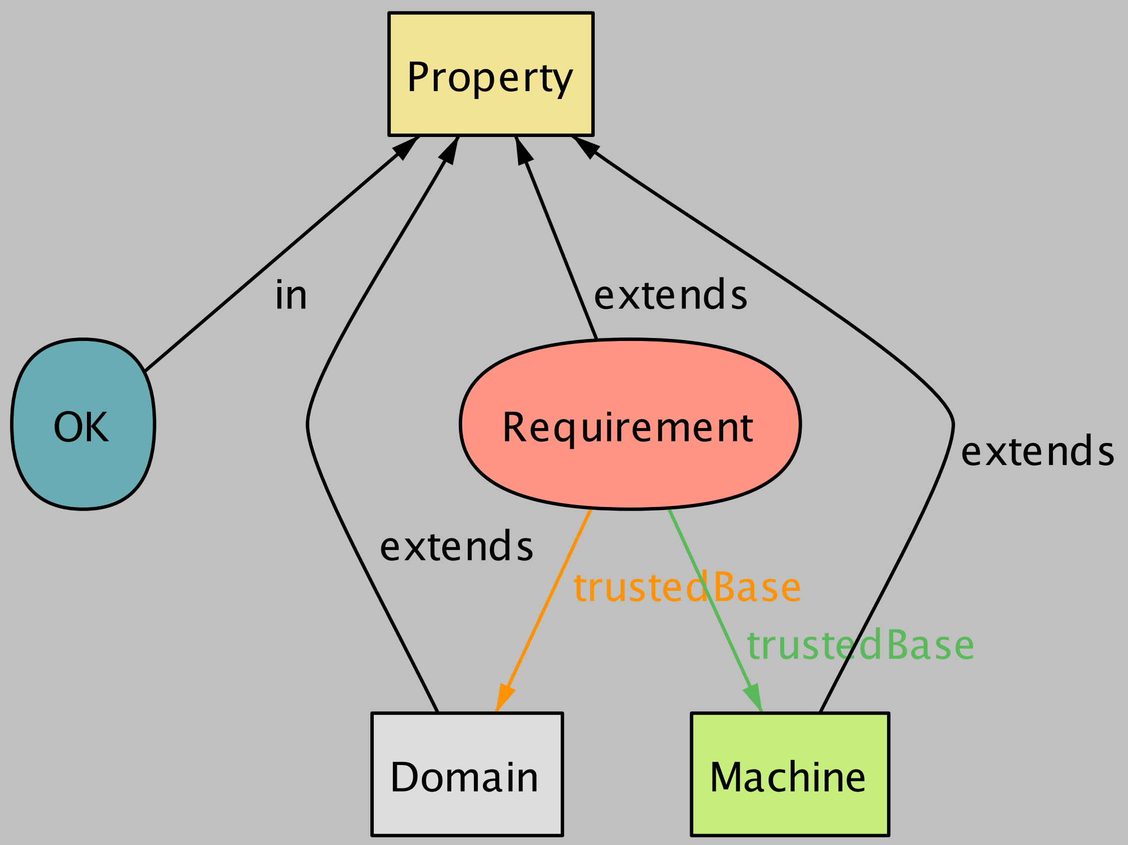
support analysis

*generate pictures like this!*

*overlay behaviour on system diagram*

# framework (1/6)

```
module framework

abstract sig Property {}
sig OK in Property {}

abstract sig Domain extends Property {}
abstract sig Machine extends Property {}
abstract sig Requirement extends Property {
	trustedBase: set Domain + Machine
	}
```



# ftp basics (2/6)

```
module ftp_shared
open framework

abstract sig Packet {}
sig Block, Hash extends Packet {}
sig File {blocks: set Block, hash: Hash}

fact Hashing {
	all f, f': File | f.hash = f'.hash iff f.blocks = f'.blocks
	}

sig Network extends Domain {inpackets, outpackets: set Packet} {
	all h: Hash & outpackets | h in inpackets or no f: File | f.hash = h
	this in OK iff inpackets = outpackets
	}

sig FileSystem extends Machine {file: File, client: Client} {
	this in OK iff (client.hash = file.hash and client.blocks = file.blocks)
	}

abstract sig Client extends Machine {hash: Hash, blocks: set Block, network: Network}
abstract sig Sender, Receiver extends Client {}
```

# architectural structure (3/6)

```
sig FTP_Requirement extends Requirement {
    from, to: FileSystem, sender: Sender, receiver: Receiver, network: Network
    }{
    from != to and no from.file & to.file
    sender = from.client and receiver = to.client
    network = sender.network and network = receiver.network
    }
```

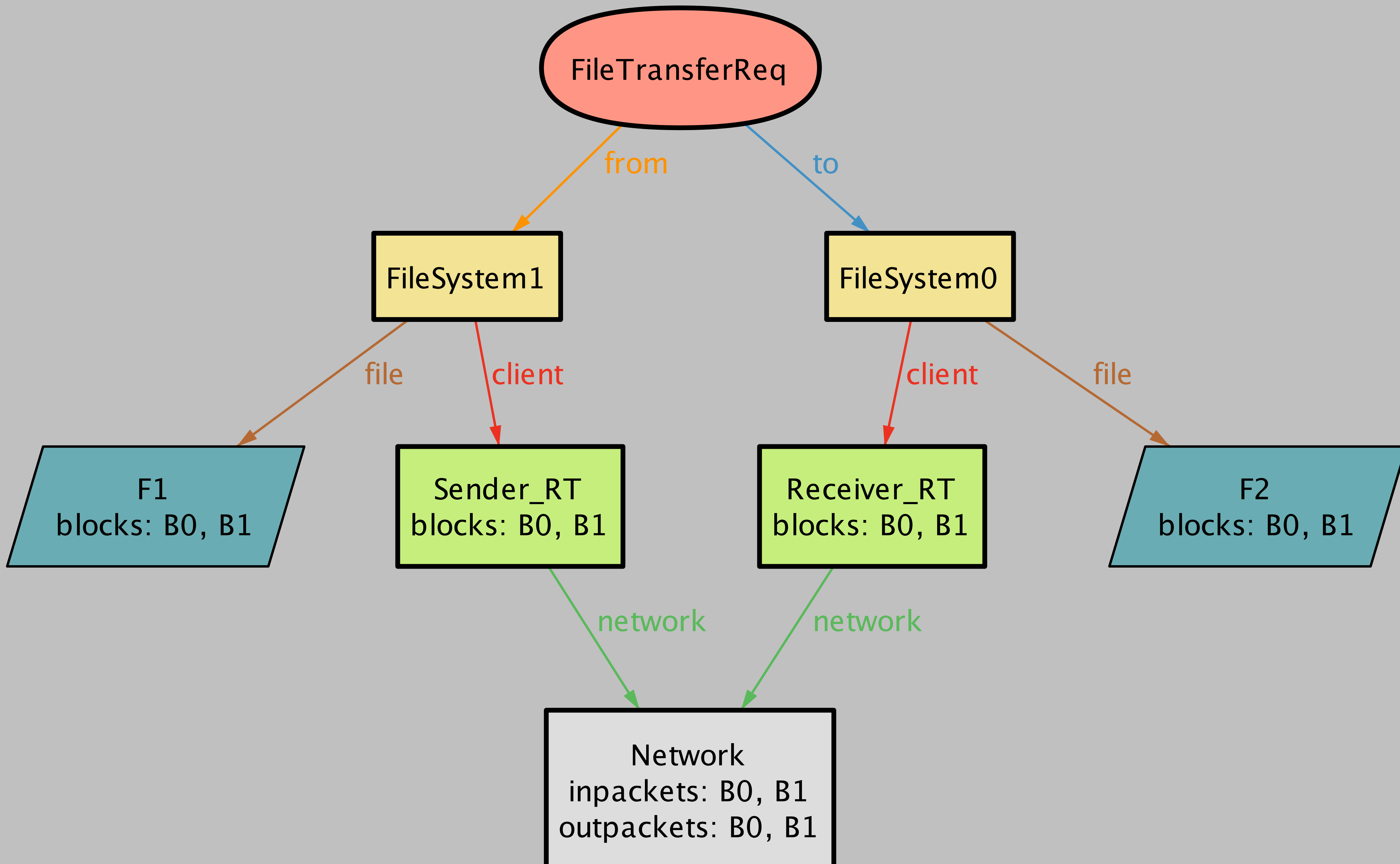
# version 1: reliable transport (4/6)

```
module ftp_reliable_transport
open ftp_shared

sig Sender_RT extends Sender {} {
	this in OK iff network.inpackets = blocks
	}

sig Receiver_RT extends Receiver {} {
	this in OK iff network.outpackets = blocks
	}

sig FileTransferReq extends FTP_Requirement {} {
	this in OK iff from.file.blocks = to.file.blocks
	}

fact {
	FileTransferReq.trustedBase = Sender + Receiver + FileSystem + Network
	}
```

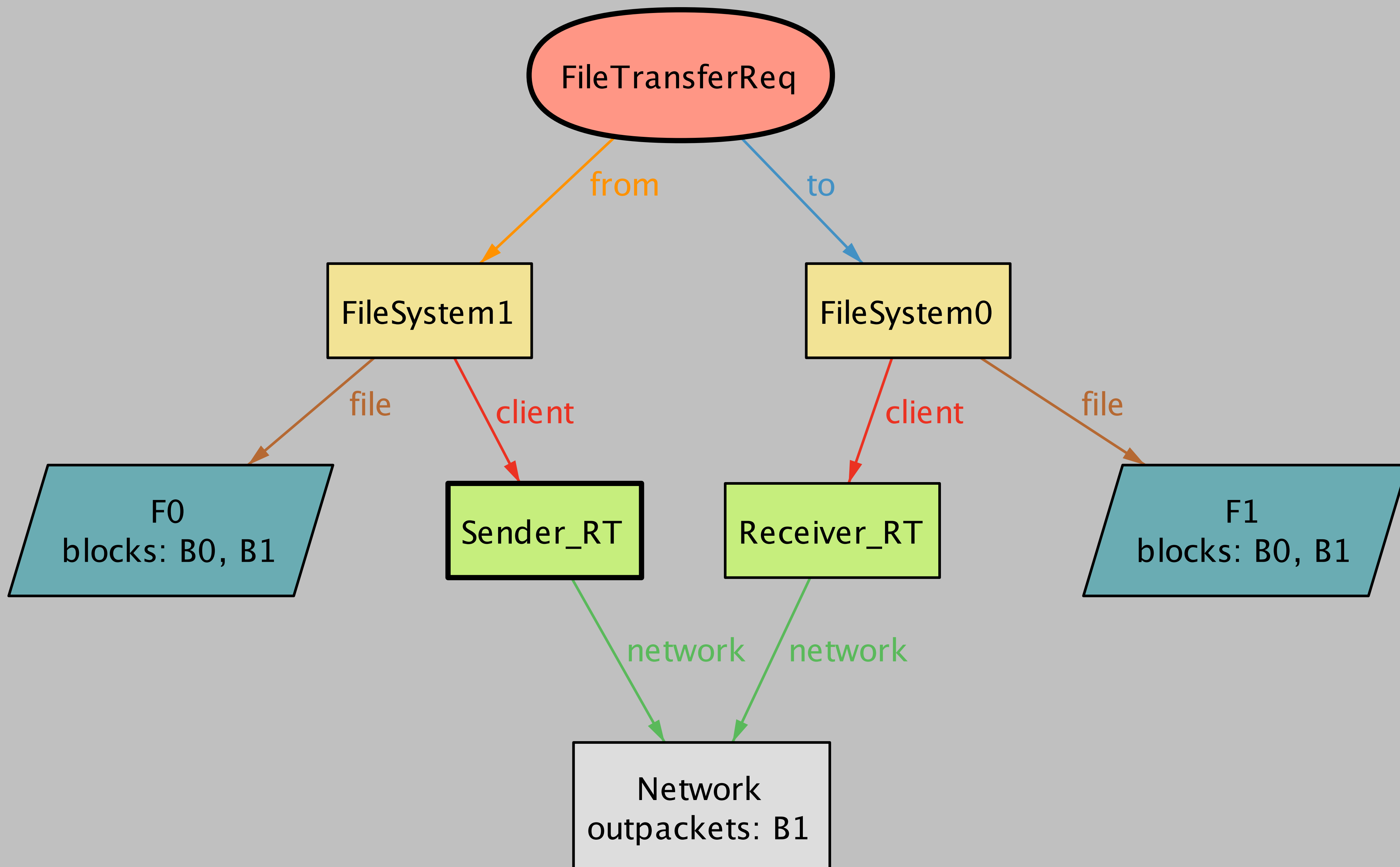
# analysis (5/6)

```
module ftp_analysis
open ftp_reliable_transport

check TrustedBaseSuffices {
	FileTransferReq.trustedBase in OK implies FileTransferReq in OK
	} for 3 but exactly 1 Requirement, 2 FileSystem, 2 Client, 1 Network

run AllWorking {
	Property in OK
	}

run WorkingDespiteFailure {
	FileTransferReq in OK
	some Property - OK
	}

run WorkingDespiteBadNetwork {
	FileTransferReq + Client + FileSystem in OK
	Network not in OK
	}
```

# example: all working



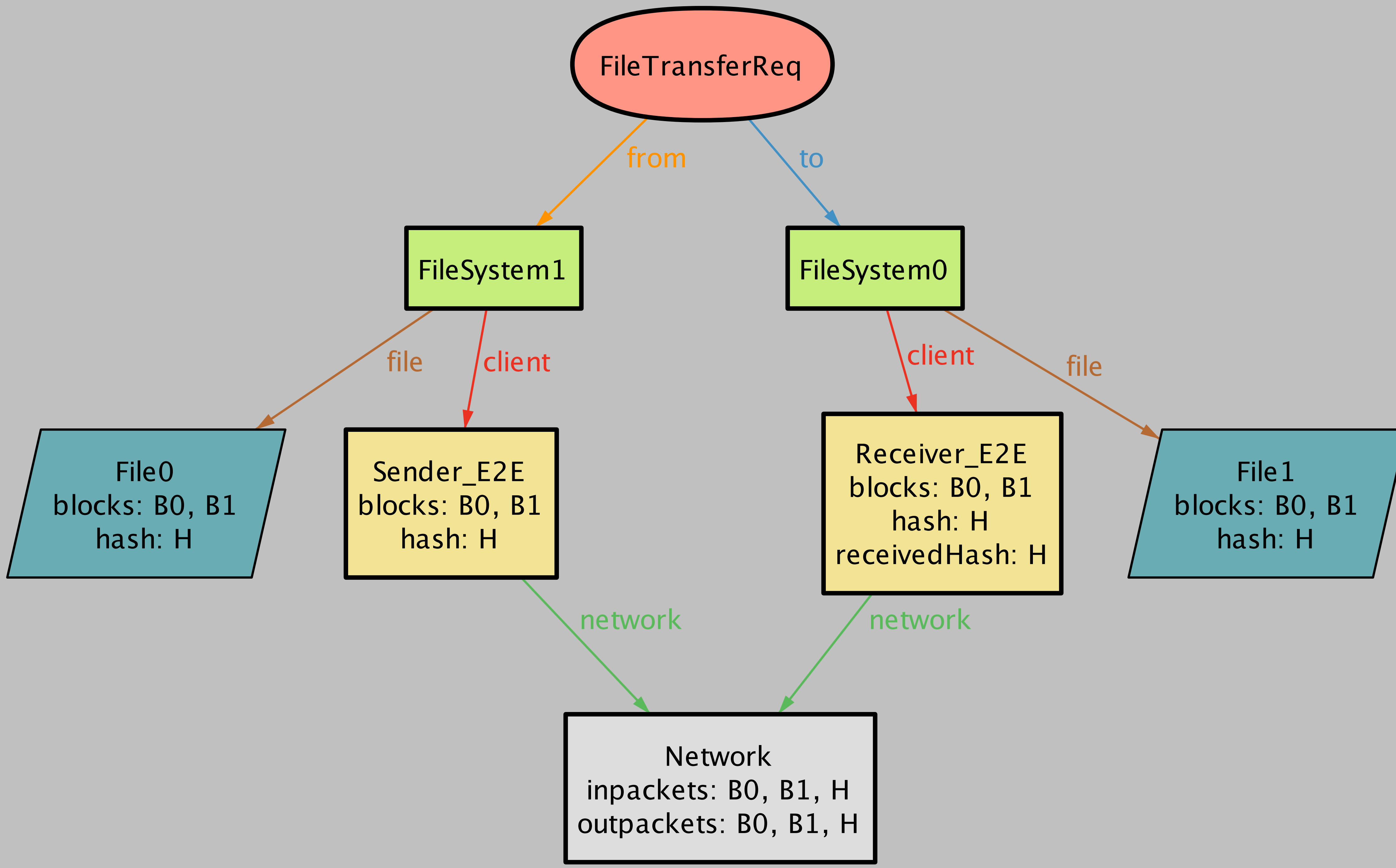
# example: working despite failure



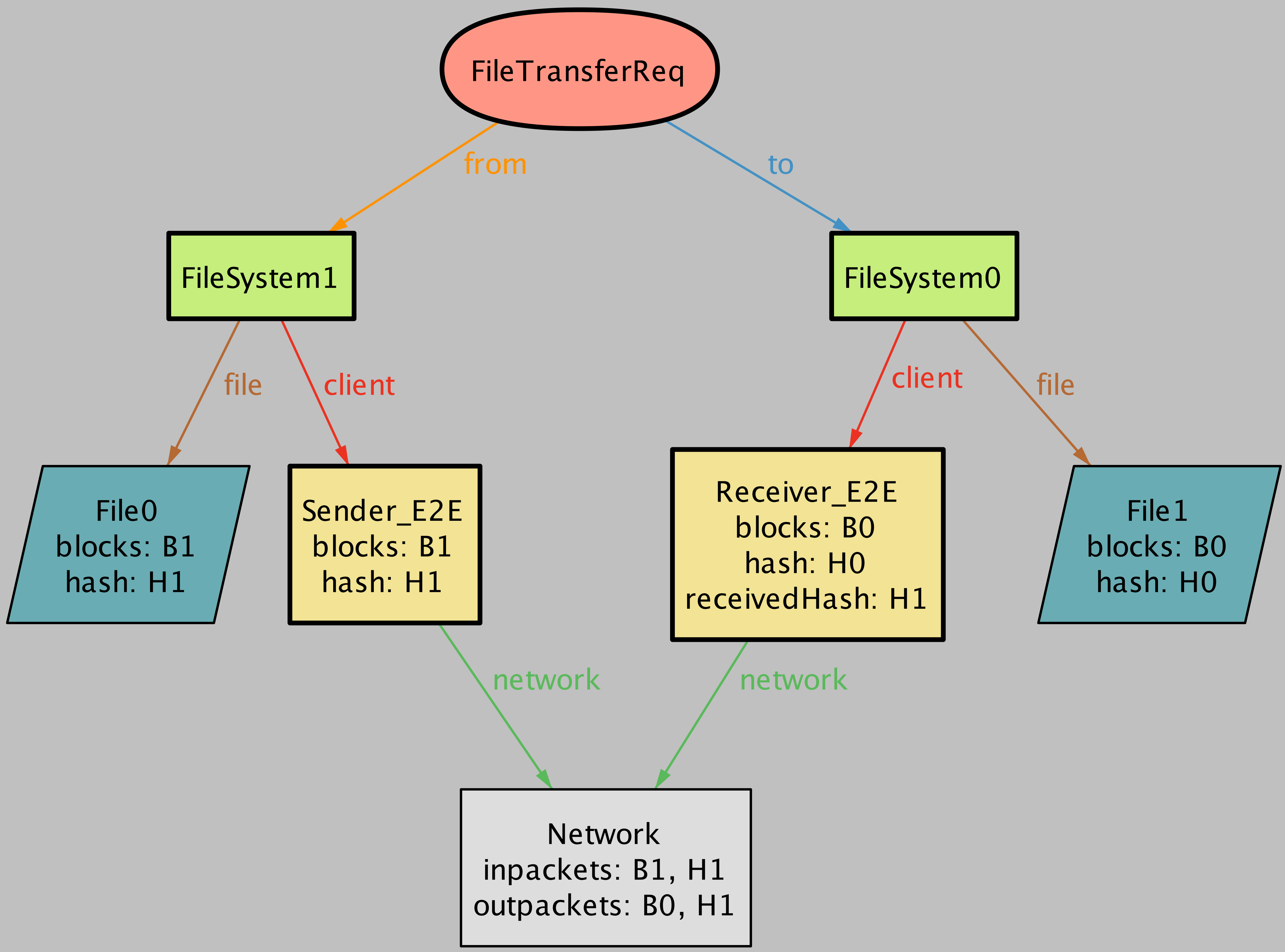
# version 2: end to end (6/6)

```
module ftp_end_to_end
open ftp_shared

sig Sender_E2E extends Sender {} {
	this in OK iff network.inpackets = blocks + hash
	}

sig Receiver_E2E extends Receiver {receivedHash: Hash} {
	this in OK iff network.outpackets = blocks + receivedHash
	}

sig FileTransferReq extends FTP_Requirement {} {
	this in OK iff (from.file.blocks = to.file.blocks or to.client.receivedHash != to.client.hash)
	}

fact {
	FileTransferReq.trustedBase = Sender + Receiver + FileSystem
	}
```

# example: all working



# example: working despite bad net



# conclusions

# summary

design for dependability

*small trusted bases*
*for most critical properties*

formal method support

*to clarify properties*
*to compose elements of case*
*to check code against specs*

any spec language would do

*but some features of Alloy help:*
*subtypes, visualization, solving*

# research avenues

analysis

*compute trusted base with unsat core*

design

*catalog of dependable designs*
*design transformation rules*

case studies

*Cambridge, MA voting system*
*proton therapy*

# related work

goal-based approaches

*goal-based decomposition [KAOS]*

*goal-based argument structure [GSN]*

module dependency diagrams

*uses relation [Parnas]*

*design structure matrix [Lattix]*

problem frames

*frame concerns [M. Jackson]*

*requirements progression [Seater]*

*architectural frames [Rapanotti et al]*

There probably isn't a best way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have a clear division of responsibilities among the parts.

Butler Lampson
*Hints for computer system design* (1983)