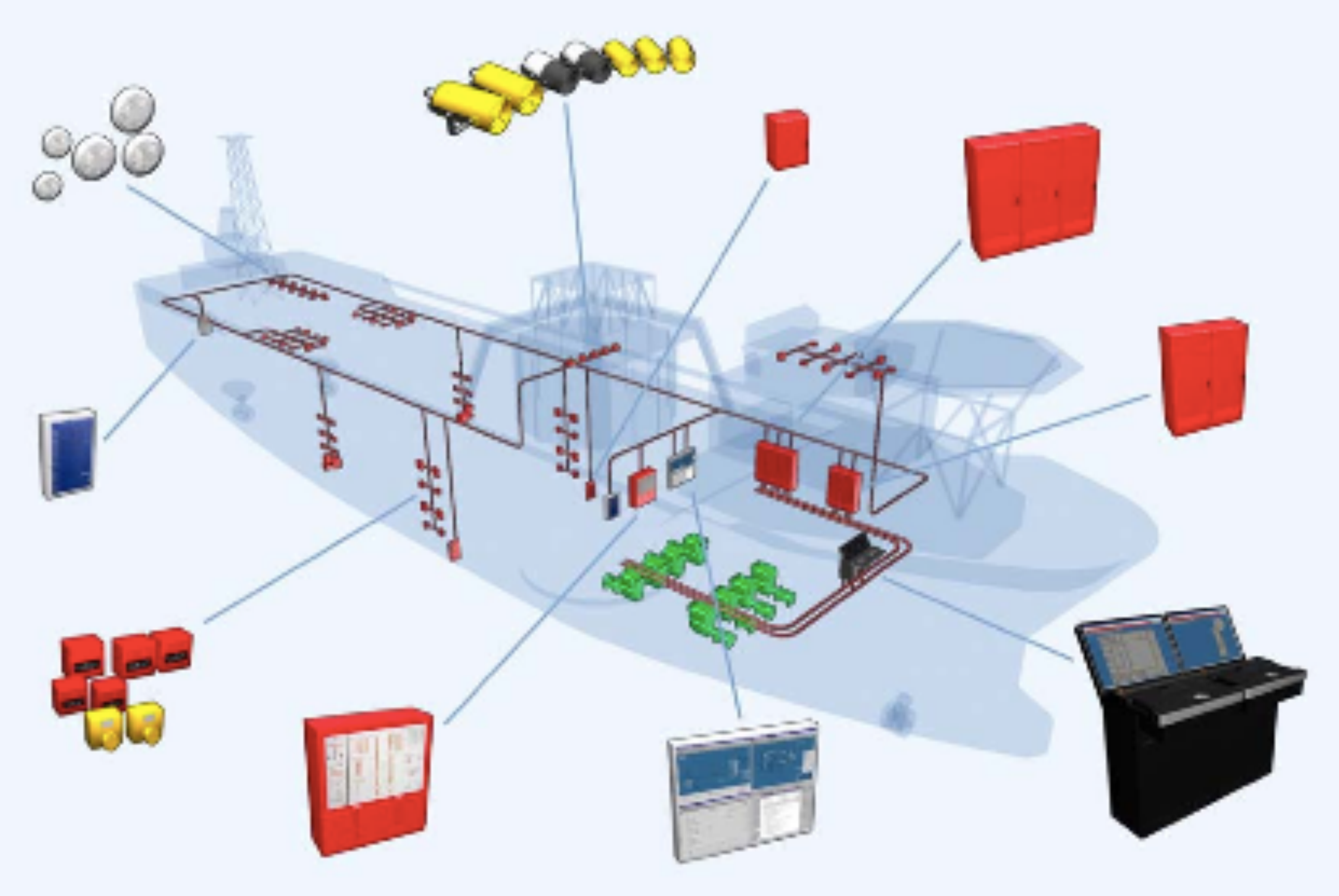
Assessment and Evolution of Safety-Critical Cyber-Physical Product Families

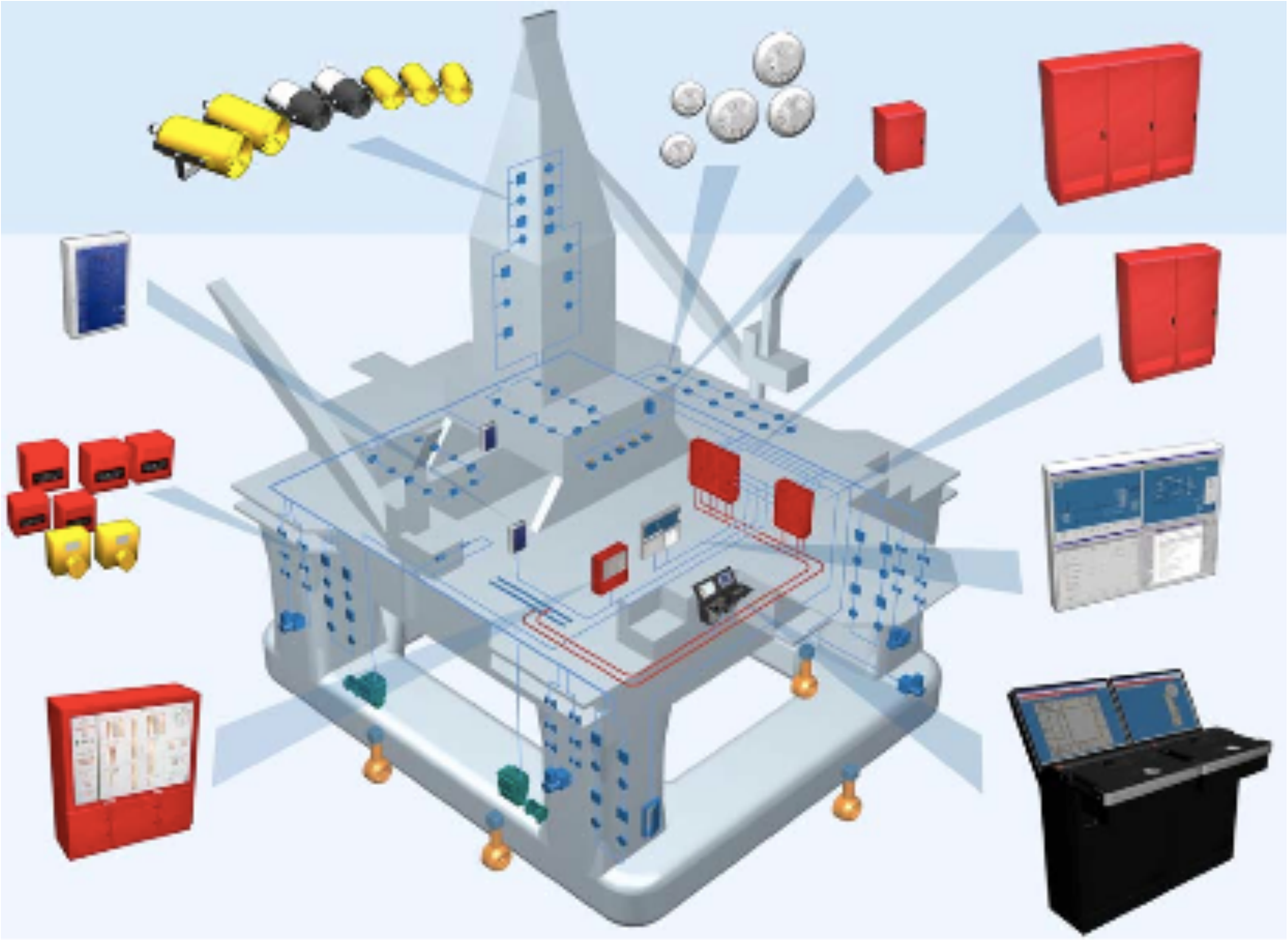
Leon Moonen

SATToSE 2014

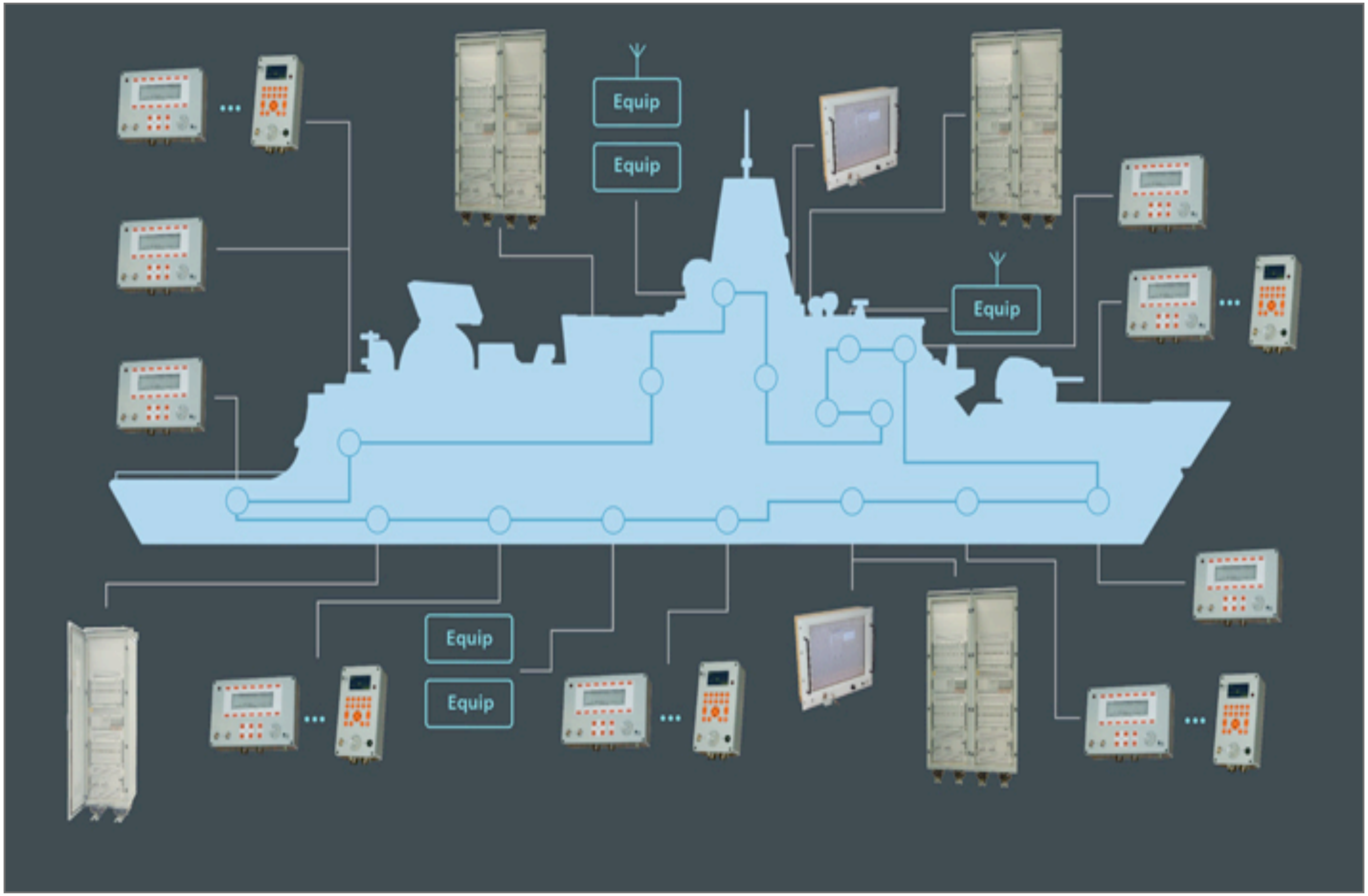
Safety monitoring and control systems



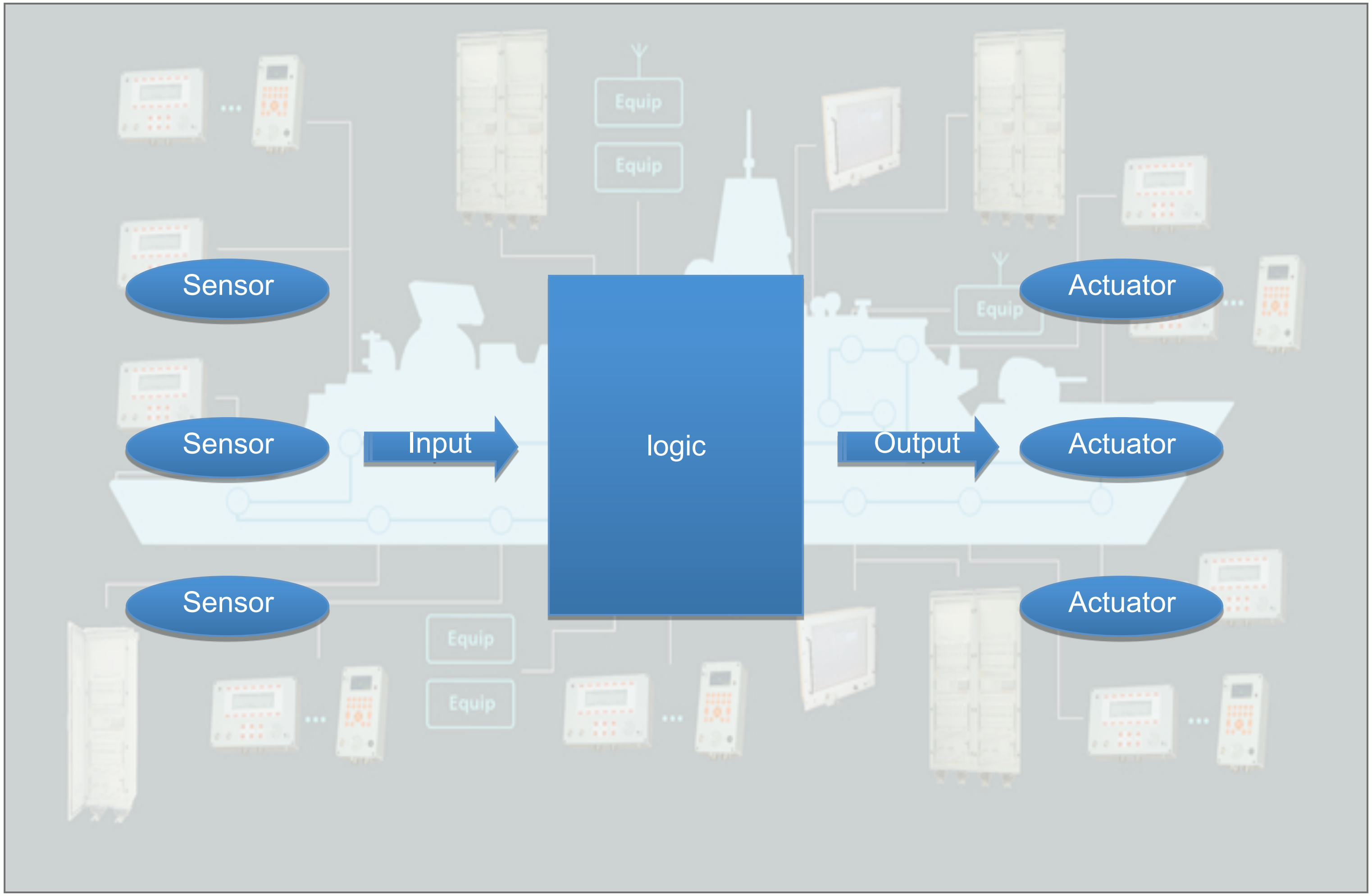
Safety monitoring and control systems



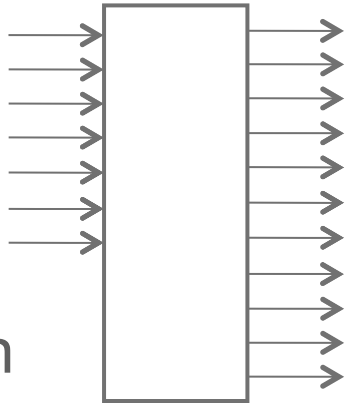
Safety monitoring and control systems



Safety monitoring and control systems



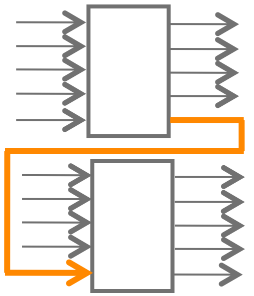
Component-based development



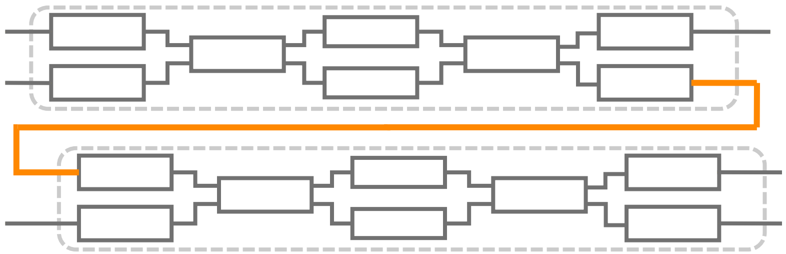
- *product family,* mostly “one off” products
- compose safety logic for particular installation by configuring a network of standard modules
- clear separation of concerns, well-defined interfaces
- proprietary component composition framework
  - runtime environment for communication/synchronization etc.
  - “statically” configured using XML files that describe component instantiation, initialization and interconnections
- other characteristics:
  - components: MISRA-compliant C code
  - developed over 15-20 years

Evolving requirements…

- customer specific options add *crosscutting* control logic
  - inhibit, override, acknowledgements, manual operation via screens …
- additions to scale up the safety logic:
  - cascading modules to handle more input or output ports than originally foreseen



  - cascading configurations to connect the safety logic of related hazard areas



Problem statement

increasingly complex configurations make it hard to understand and reason about system behavior

can we provide source based evidence that a given actuator is triggered by the correct sensors?

Tracking information flow

"find source based evidence that a given actuator is triggered by the correct sensors?"

⇔ is there *information flow* from the desired sensors to the selected actuator?

⇔ are the desired sensors (input ports) part of the *backward program slice* for the selected actuator (output port)?

Program slicing

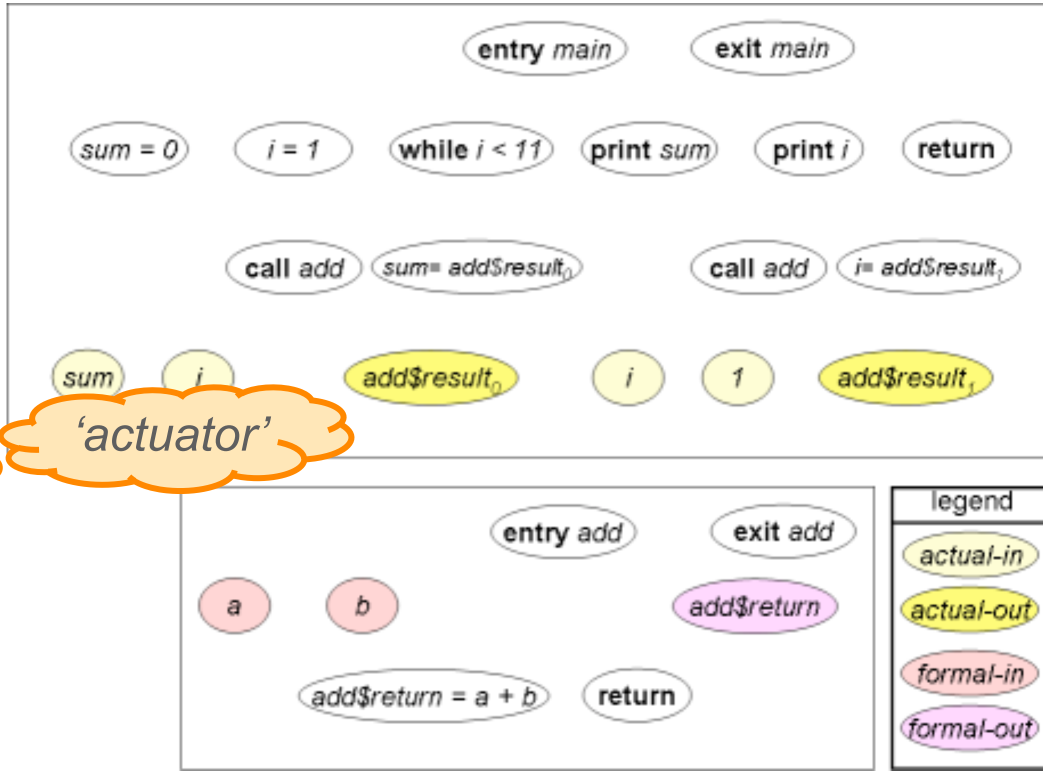
- program slice: set of programs points ('statements') that may affect values at point of interest (aka *slicing criterion*)

```
void main() {
   int sum, i;
   sum = 0;
   i = 1;
   while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
   }
   printf("sum = %d\n", sum);
   printf("i = %d\n", i);
}

static int add(int a, int b){
   return a + b;
}
```

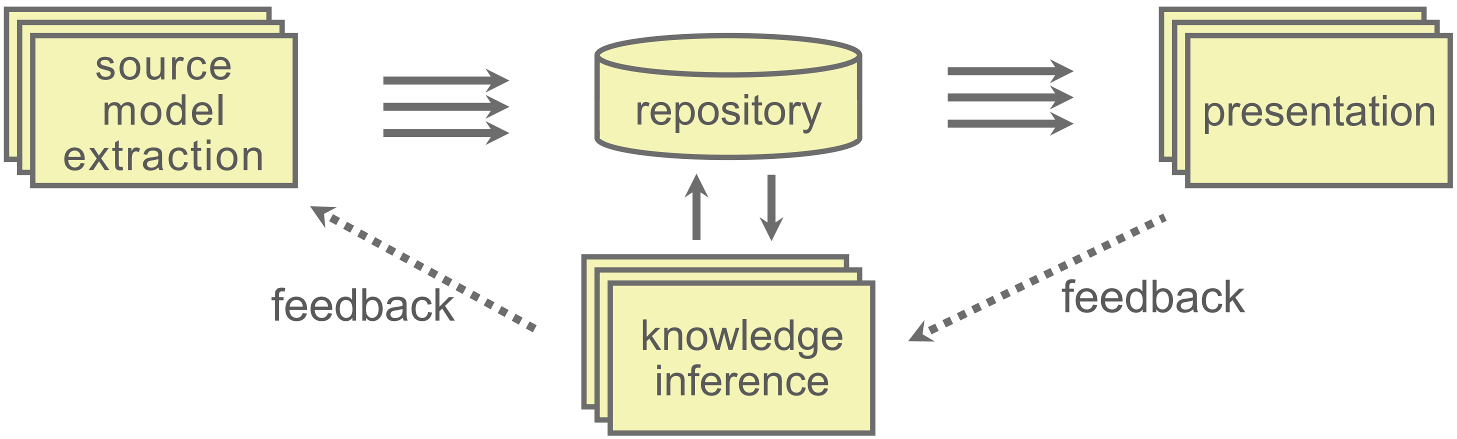
'sensor'

'actuator'



program dependence graph [src: CodeSurfer help]

Overall approach



create program dependence model from artifacts

track information flow trough system using program slicing

visualize at appropriate level for our users

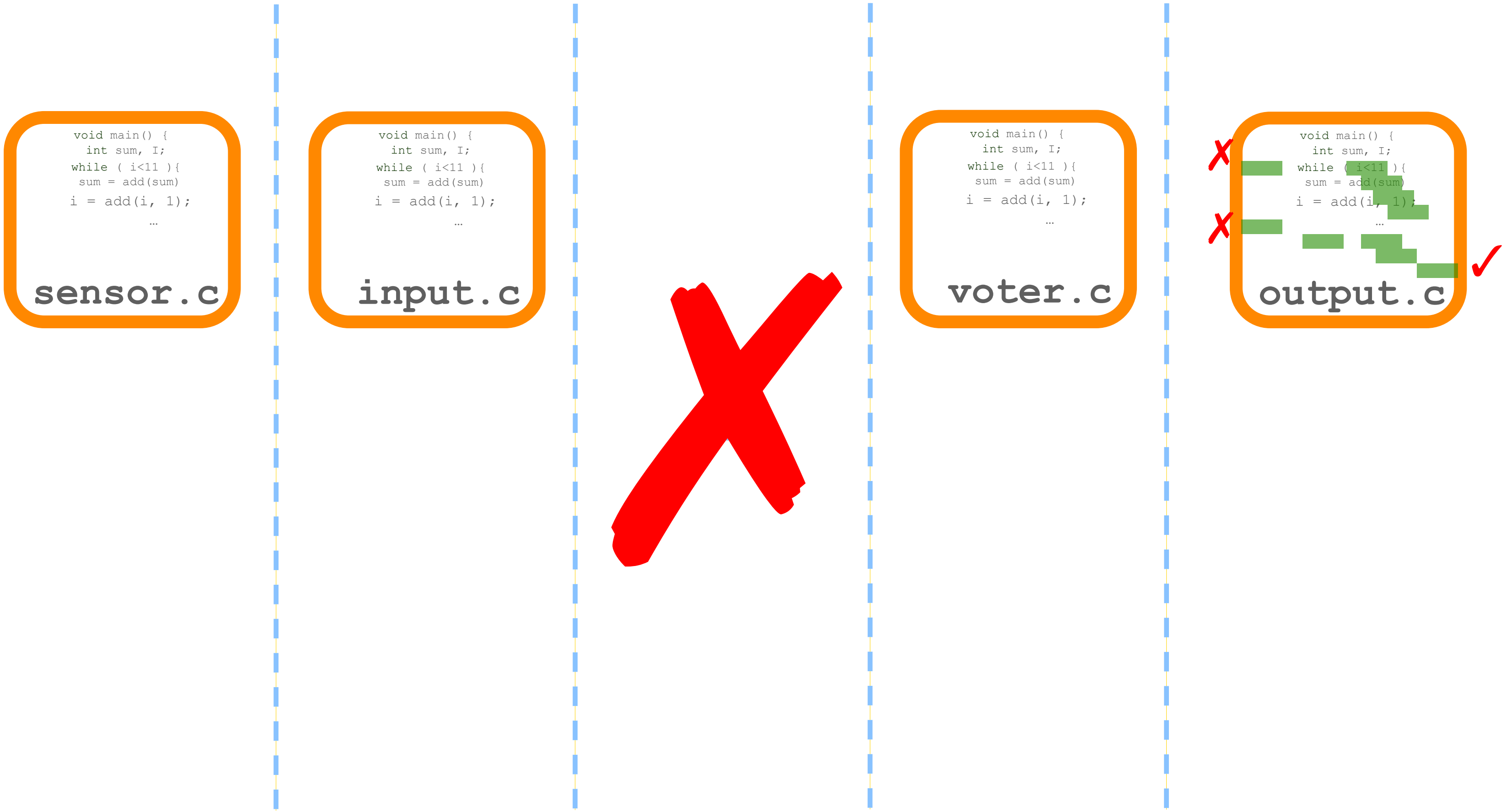
Challenge: heterogeneous systems

- systems are not just set of components
  - actual behavior depends on composition & configuration
  - literature focuses on analysis of homogeneous systems
    - little work that crosses language boundaries / incorporate information from composition or coordination technology in analysis

⚡ existing technology is programming language specific

⚡ no support for “external” artifacts

Challenge: heterogeneous systems



Challenge: heterogeneous systems

- system is not just set of components
  - actual behavior depends on composition & configuration
  - literature focuses on analysis of homogeneous systems
    - little work that crosses language boundaries / incorporate information from composition or coordination technology in analysis

⚡ existing technology is programming language specific

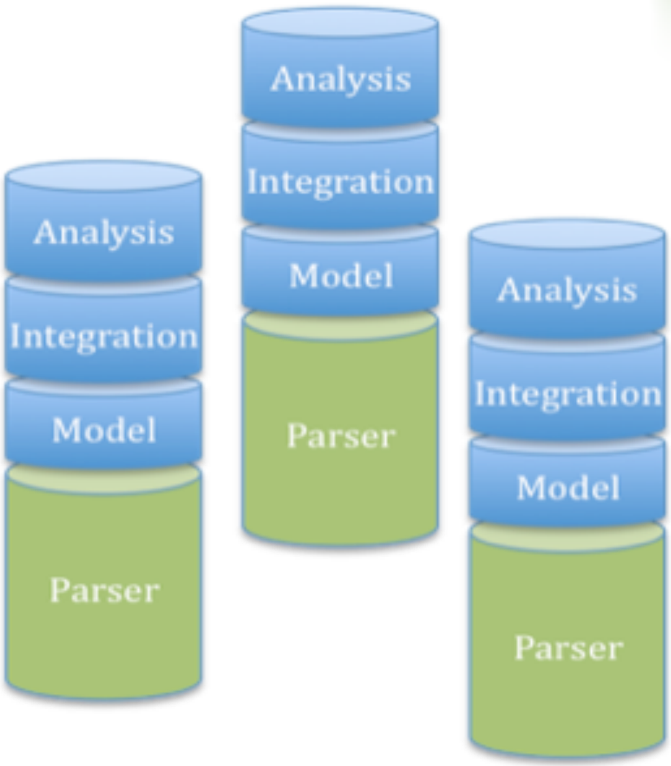
⚡ no support for “external” artifacts

➞ our solution: reverse engineer *one system-wide model* from the various source and configuration artifacts

- incremental approach, model merging to combine parts

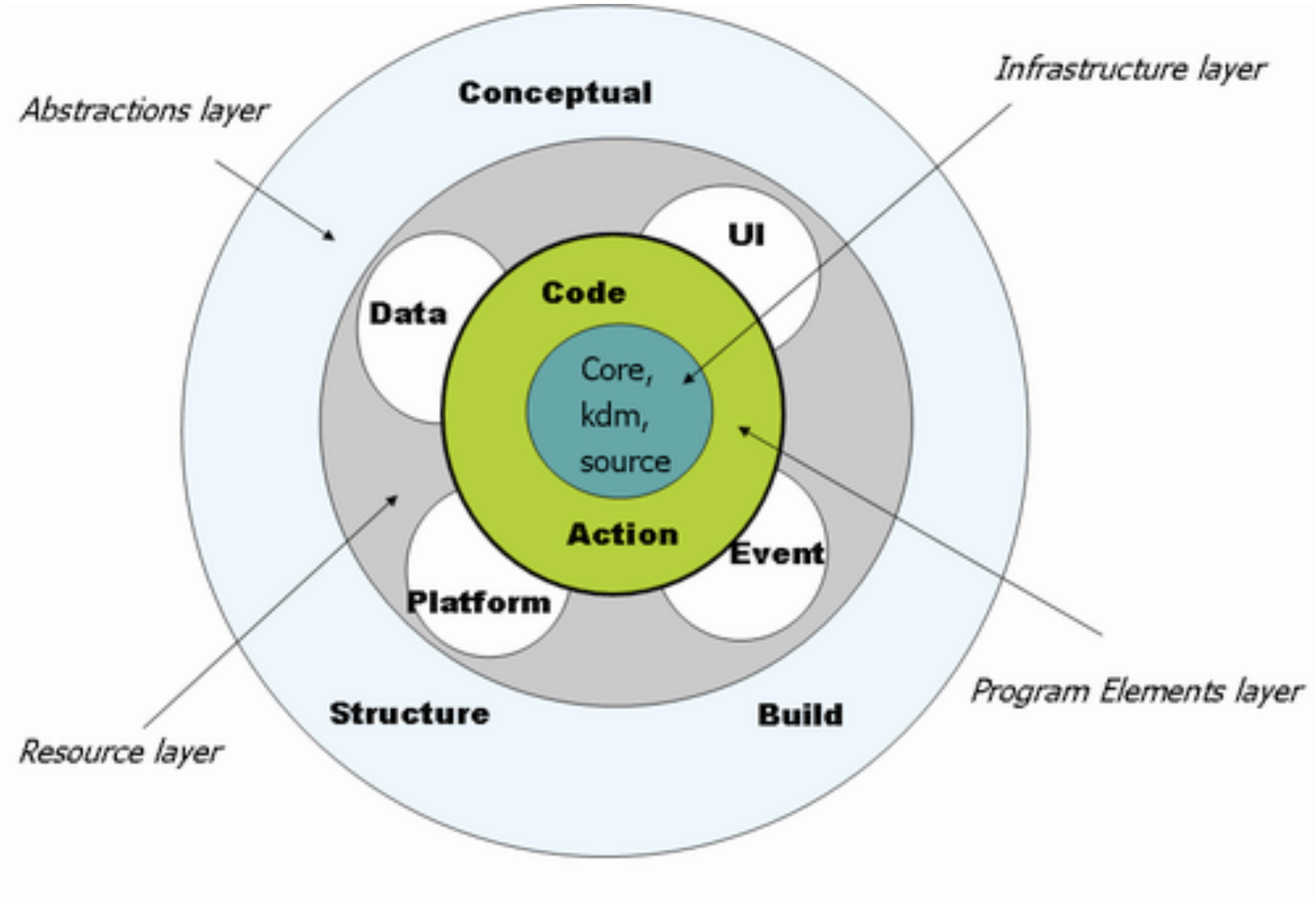
Build on OMG's ADM and KDM

- Architecture Driven Modernization
  - extend model driven architecture to existing software systems
  - set of standards for exchanging (meta-)data about existing systems
  - use for analysis, visualization, refactoring and transformation
- "traditionally"
  - many independent tools / techniques
  - unique strengths, also unique gaps
  - no choice but to use several tools
  - language & platform dependencies

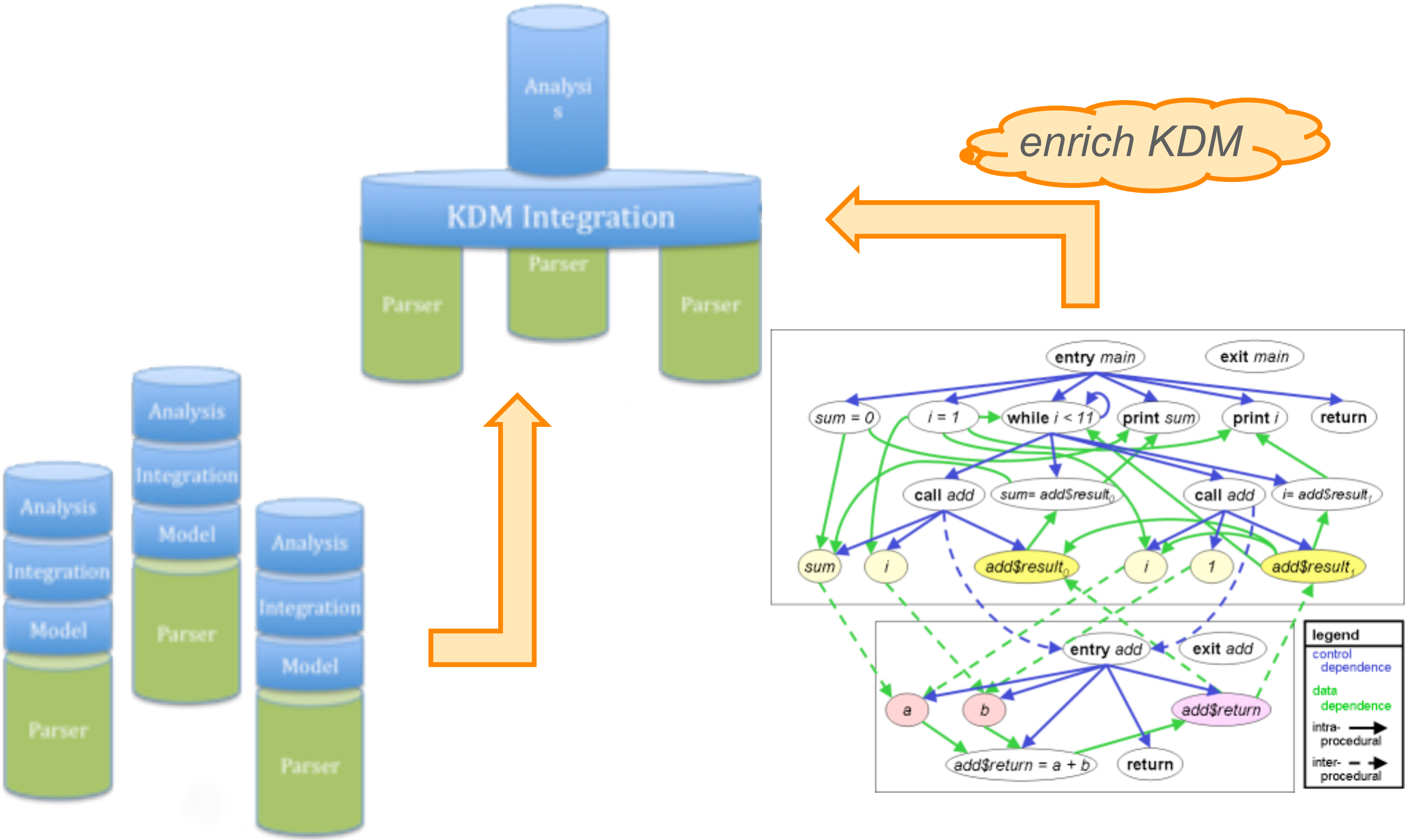


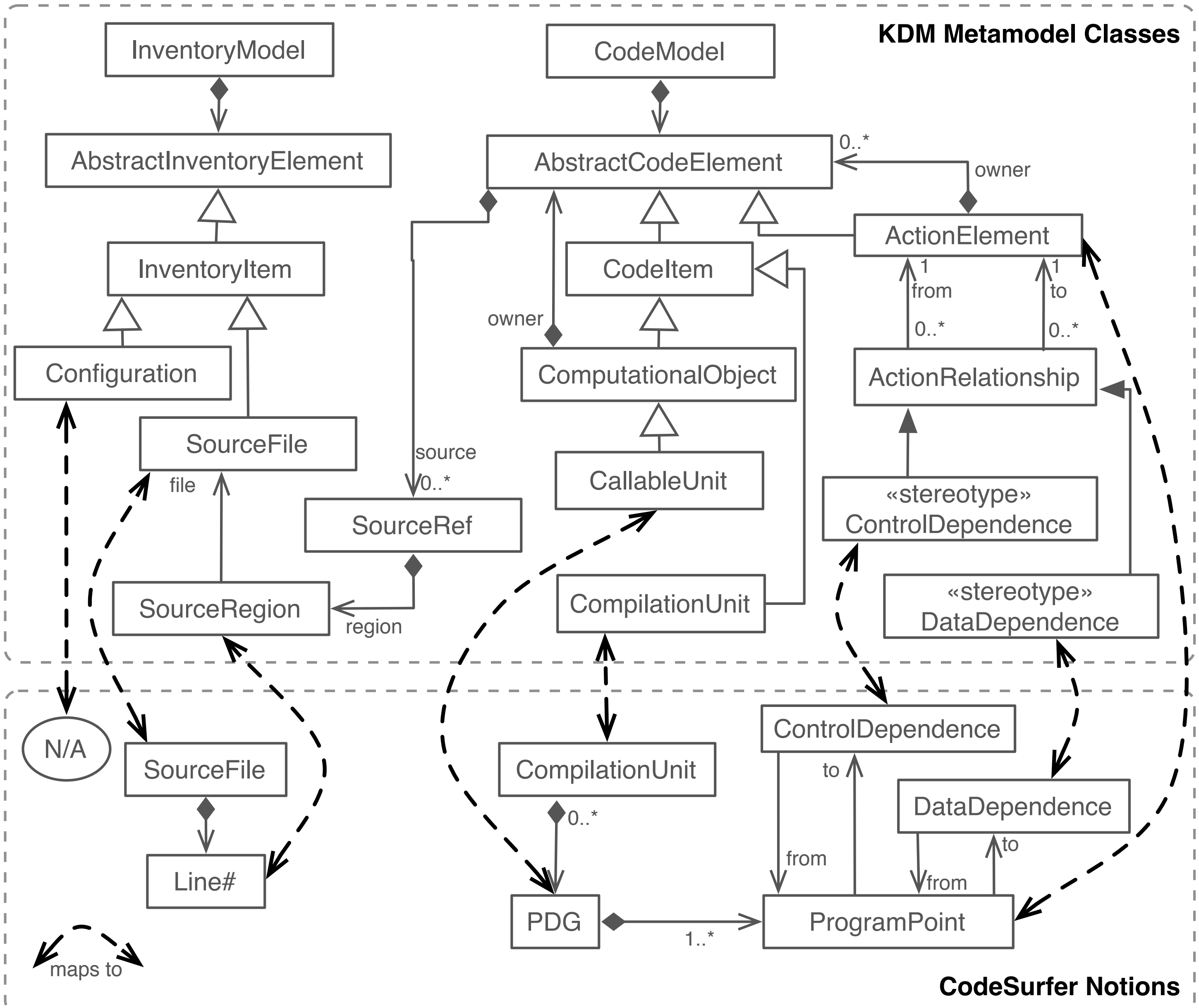
disjoint analysis islands;
result of *silo* tool design

Knowledge Discovery Metamodel (KDM)

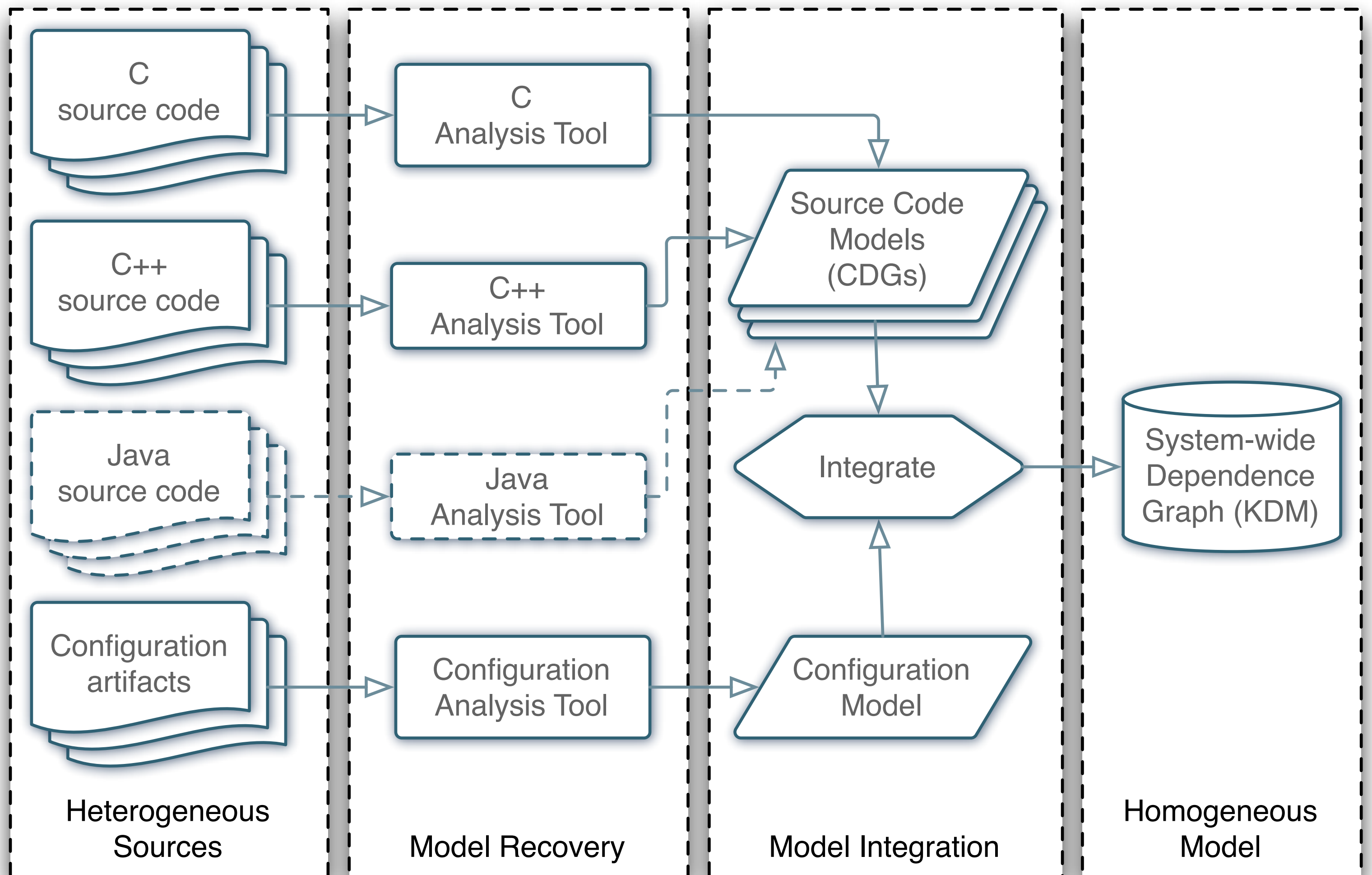


"From silo solutions to KDM ecosystems"

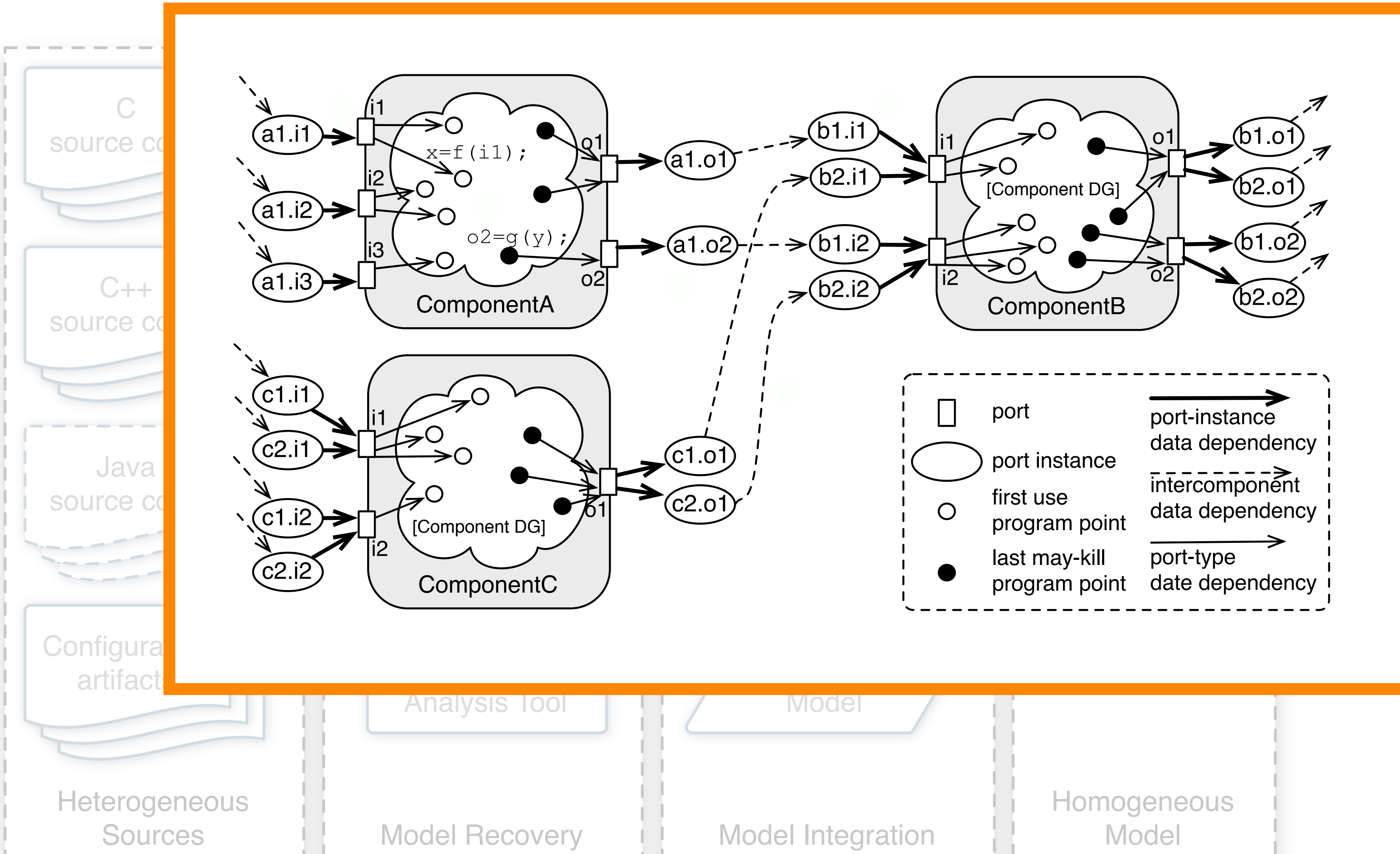



KDM Metamodel Classes
InventoryModel
AbstractInventoryElement
InventoryItem
Configuration
SourceFile
file
SourceRegion
region
SourceRef
source
0..*
CodeModel
AbstractCodeElement
0..*
owner
owner
CodeItem
ComputationalObject
CallableUnit
CompilationUnit
ActionElement
1
from
1
to
0..*
0..*
ActionRelationship
«stereotype»
ControlDependence
«stereotype»
DataDependence
N/A
SourceFile
Line#
CompilationUnit
0..*
PDG
1..*
ProgramPoint
ControlDependence
to
from
DataDependence
to
from
maps to
CodeSurfer Notions

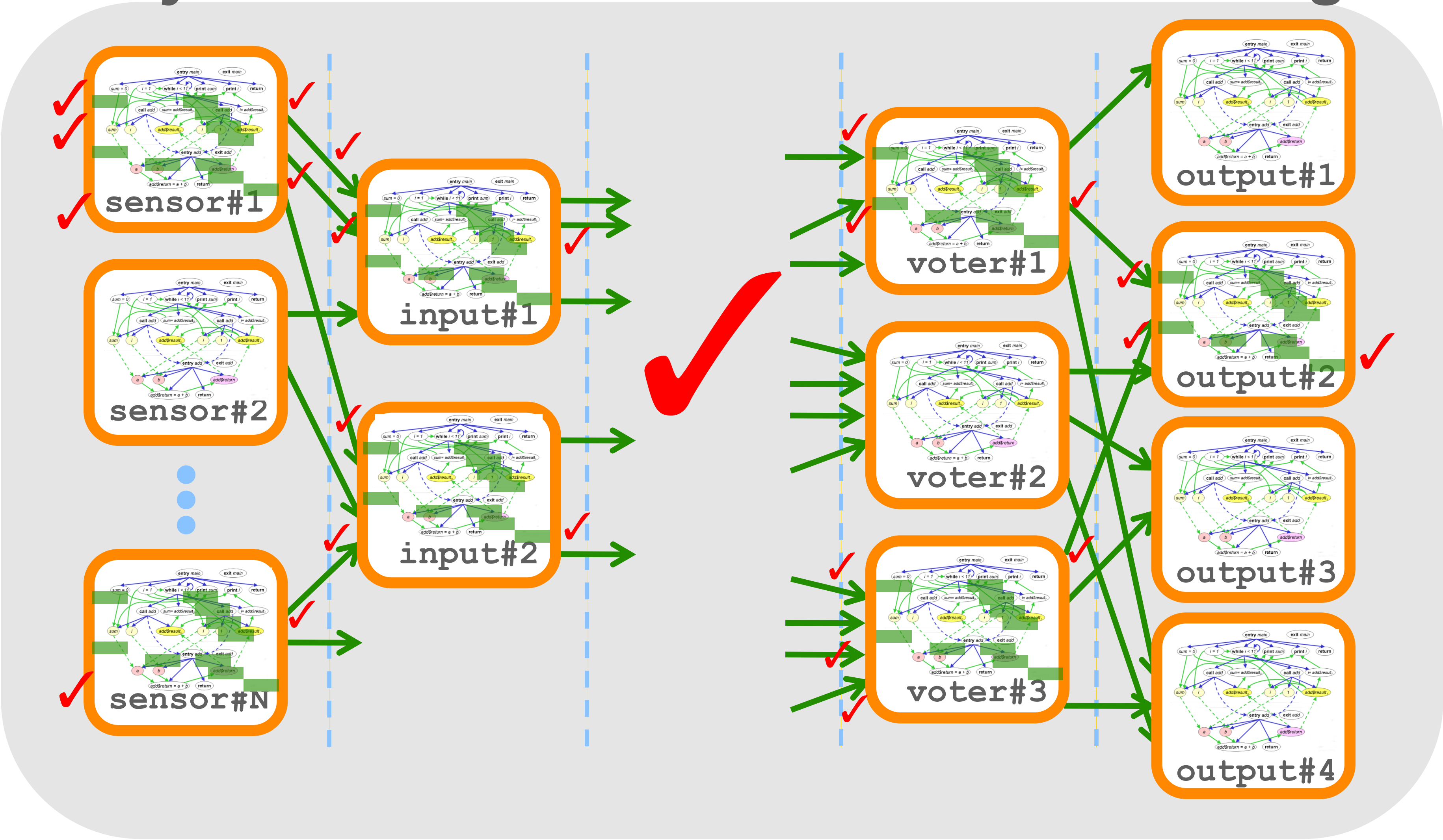
Model reconstruction approach



Model integration (merging)



System-wide information flow tracking

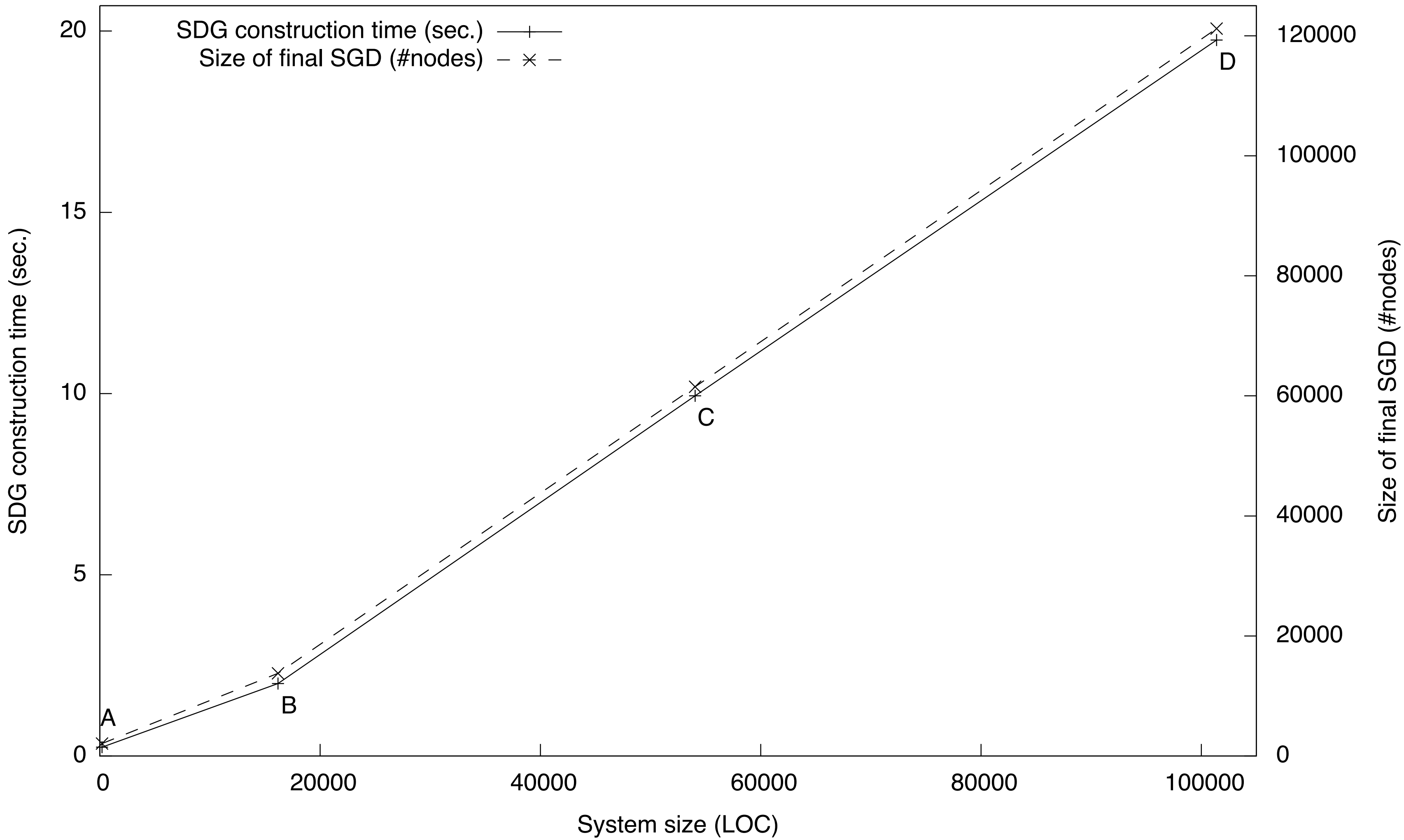


Precision and scalability

- precision: identical results as CodeSurfer
  - created identical component based and integrated versions
  - random selection of slicing criteria, compared slices
- linear scaling w.r.t. LOC

System	A	B	C	D
# Components	4	6	30	60
LOC	207	16181	54053	101393
$\sum$CodeSurfer CDG generation times (sec.)	3.181	13.064	65.022	132.381
Model transformation time (sec.)	0.246	1.996	9.938	19.755
# Nodes (KDM SDG)	2074	13787	61507	121197
# Dependencies (KDM SDG)	3784	46276	216956	431042

Precision and scalability

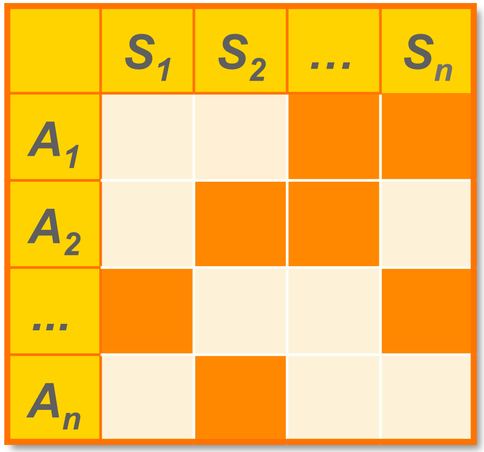


Using information flow for software certification and comprehension

- *information flow* can be computed from dependence graph using *graph traversal* (cf. *program slicing*)
- raw information flow is too detailed
- need to present at appropriate level of detail for users:
  - *safety domain experts*: need system level and inter-component views but treat components as black boxes
  - *developers*: need inter- and intra-component abstractions that allow them to drill down to relevant source code

Interlude: capturing safety knowledge

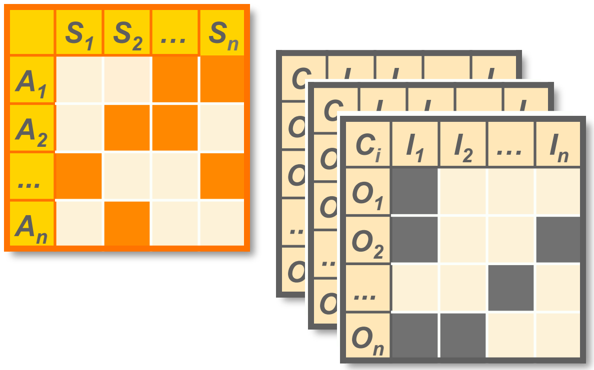
- at highest lever, the *desired* overall safety behavior for system is recorded as so called *cause and effect matrix*



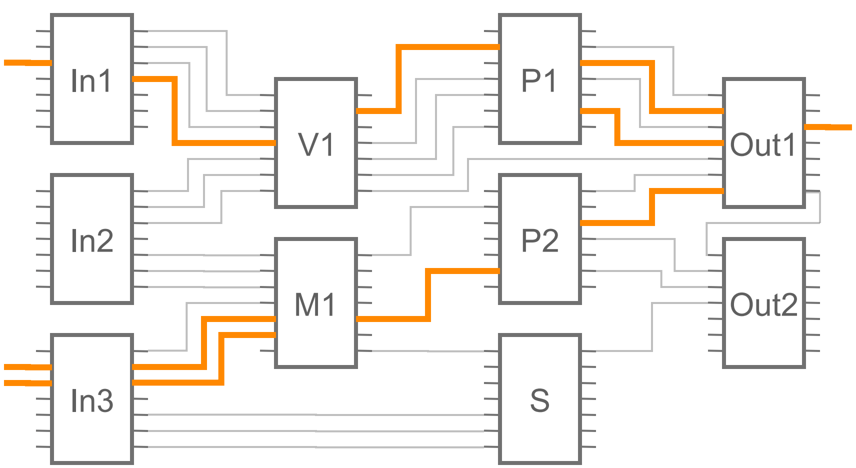
  - based on discussions between customer and safety expert (variant on requirements elicitation)

Show information flow to safety experts

- dependency matrices at system and component level
  - provides survey info
  - system level should correspond to cause and effect matrix used by safety expert to specify desired behavior

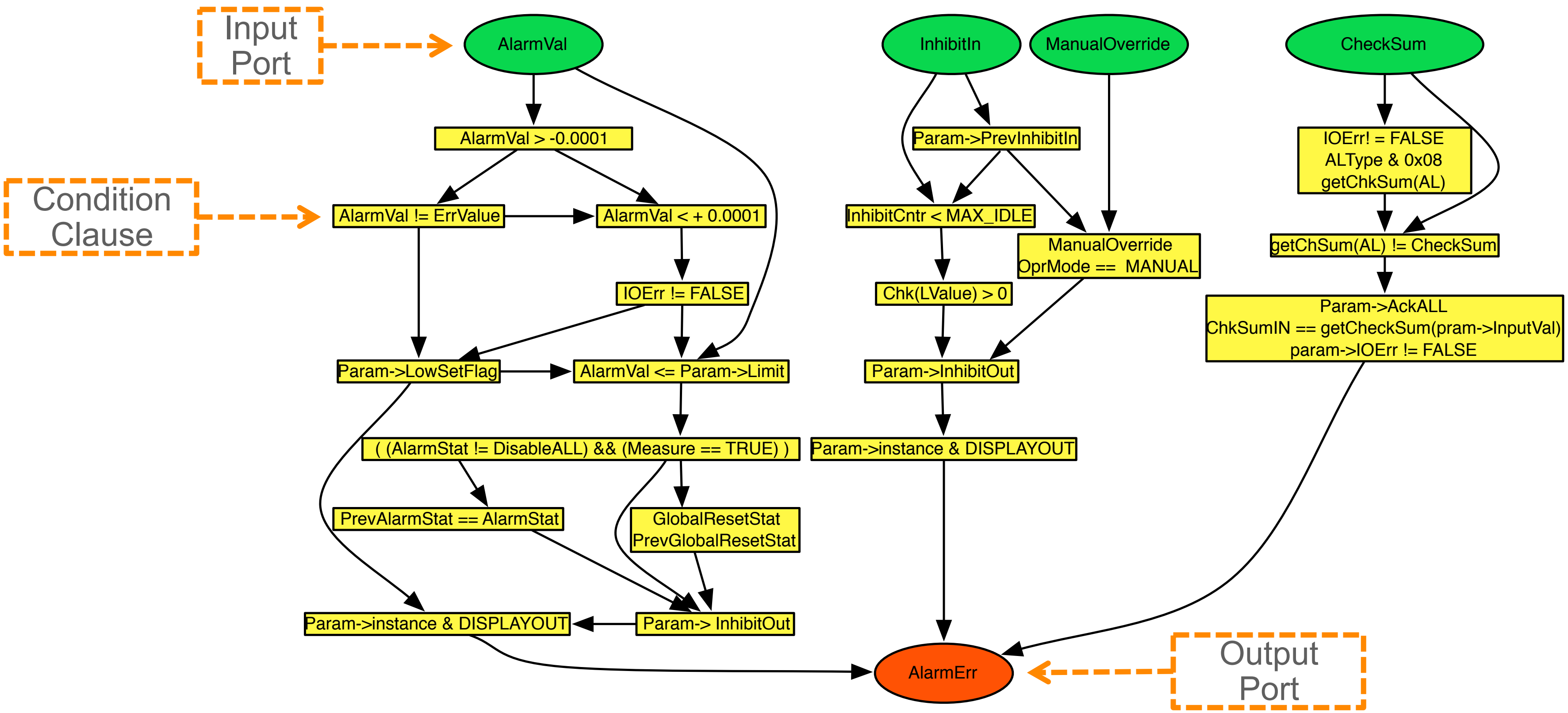


- inter-component information flow
  - "slice through system" to show which sensor signals trigger given actuator
  - detail for safety expert, survey info for developer



Show information flow to developers

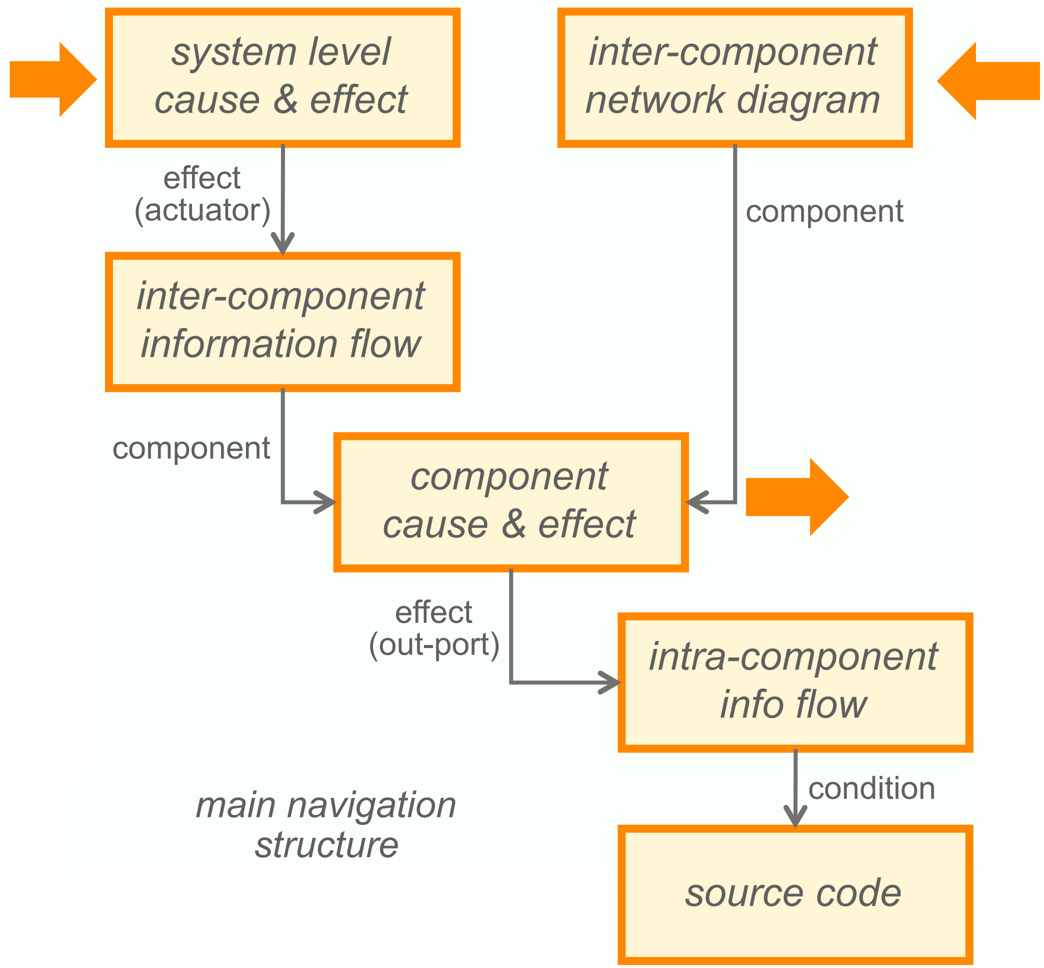
- intra-component information flow
  - “slice through component”, shows conditional flow to output port



Five task-specific, interconnected, layers of abstraction:

```
#include <stdio.h>
#include "system_def.h"

int main (void) {
    while ( under(NDA) )  {
        printf("nothing to see here\n");
    }
    return(0);
}
```



Genericity

- reverse engineered system-wide dependence graph can be used for *all analyses based on PDG/SDG*
- configuration analysis specific to Kongsberg Maritime component framework configuration artifacts (XML)
  - mostly parsing, also implemented Java / Spring version
- our slicer is specific to KDM-based SDGs, not application
  - planned experiment with injecting our SDG back into CodeSurfer
- information flow visualization aimed at KM tasks

User evaluation

- *exploratory*, *qualitative* study
  - 6 participants: developer / system integrator / safety expert
- *structured interview* with each participant (60-90min)
  - 30 Likert-scale questions and 6 open questions
  - researcher-administered, to stimulate discussion and Q&A
  - transcribed & analyzed using open and axial coding
- overall feedback *positive*: intuitive, low learning curve
- various suggestions for refinement and extensions
- system integrator and safety experts: "what we actually need is *impact analysis* on complete *product family*"
  - *retrofitting team:* "backporting" changes to existing installations

What's next? Guiding evolution!

- support evolution of the product family (taking existing product installations into account)

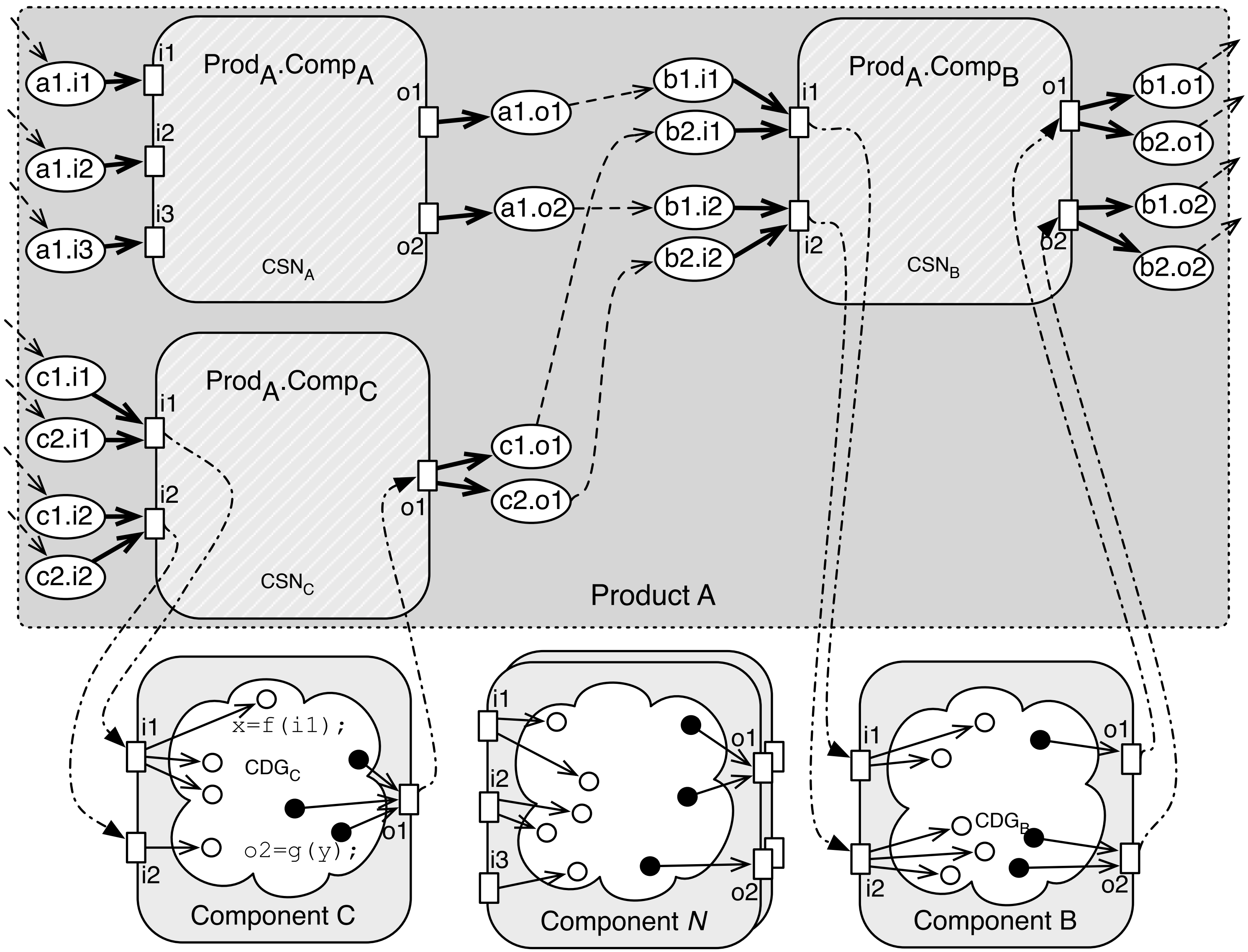
✧ re-certification of modified components is costly
  - a *cost-effective* evolution strategy minimizes the amount of re-certification needed

➢ need an objective way to compare different evolution 'scenarios' *before* actually making the changes

Towards evidence-based evolution recommendations

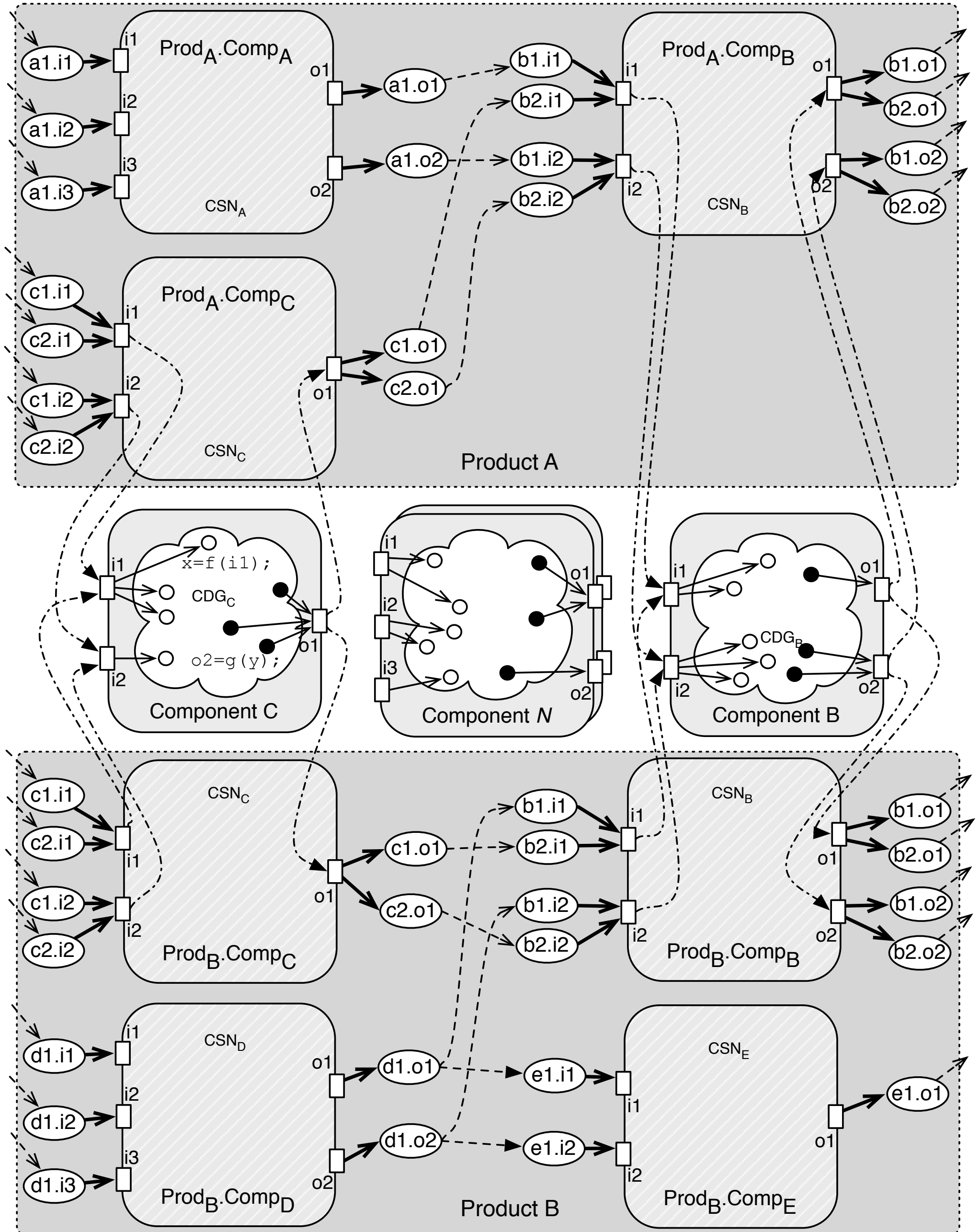
1. reverse engineer *dependence models* representing products and families (mega-modelling)

System-wide product dependence graph (SPDG)



Family Dependence Graph (FDG)

- combine SPDGs for all products in family
  - share components
- enrich with component summary edges to 'cache' component level information flow
- annotate with *attributes* (e.g., slice size)



*Towards **evidence-based** evolution recommendations*

ongoing work

1. reverse engineer *dependence models* representing products and families (mega-modelling)
2. define *scalable* and *precise* impact analysis of change scenarios (managing safe approximations is challenge)
3. develop method to *quantify and compare impact* (working assumption: use slice size as quantifier)
4. use *constraint programming* to select evolution strategy that minimizes impact (hence *re-certification* efforts)

➢ *recommendation engine* for evolution

Questions & Discussion

Leon Moonen
leon.moonen@computer.org
http://leonmoonen.com/