STITCHES
SoS Technology Integration Tool Chain for Heterogeneous Electronic Systems

Dr. Evan Fortunato

Abstract # 18869

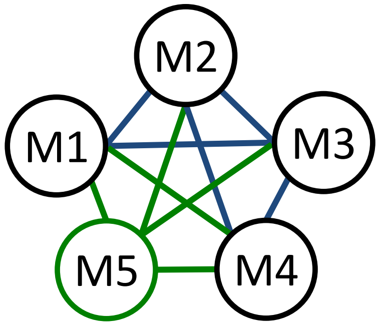
This Research was developed with funding from the Defense Advanced Research Projects Agency (DARPA)

The Goal: Composing Systems That Keep Up With The Times

- DoD has long assumed that homogeneous, fixed-configuration weapon systems are the only way to meet their goals of a superior military force
  - Must last a long time, so requirements are developed for 30+ years out.
  - Meeting 30 year out requirements with today's technology is hard
  - **Result is the best design possible with 20-30 year old technology** and updates are not efficient with respect to time or cost…
- Open Architectures Try to Solve this Problem
  - Requires enormous effort to reach a "global" consensus on the system architecture,
    - Even then, it is only a "local" version of "global"
    - Global standards have to work for everyone, so aren't optimized for your application
  - Result is **heterogeneous components in a homogeneous architecture** – which doesn't work because the architecture needs to evolve with the technology
  - Attempts to build flexibility into the architecture (to support heterogeneity) just result in overly complex infrastructures that still don't anticipate the new technologies
- What if Global Interoperability Didn't Require a Common Interface at ALL?

Understanding the Trade between Local and Global Message Standards…

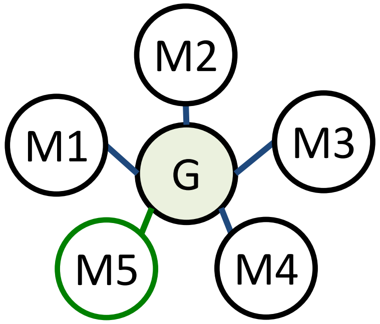
- Local Message Standards
  - Flexible – You Can Add New Messages Easily
  - Inefficient - Require N^2 Transforms (all pairs) for Interoperability



M# = Message #

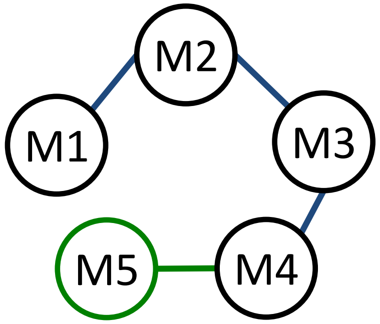
Transform M2 <- M1:
M2 = T21(M1)
Transform M5 <- M1:
M5 = T51(M1)

- Global Open Standards
  - Efficient – N Transforms to/from the Global Standard
  - Not Flexible – Can't change without tremendous effort



Transform M2 <- M1:
M2 = T2G(TG1(M1))
Transform M5 <- M1:
M5 = T5G(TG1(M1))

- Incremental Standards (STITCHES)
  - Efficient – ~N Transforms for Interoperability
  - Flexible – You Can Add New Messages Easily



Transform M2 <- M1:
M2 = T21(M1))
Transform M5 <- M1:
M5 = T54(T43(T32(T21(M1))))

Capabilities We Set Out to Develop

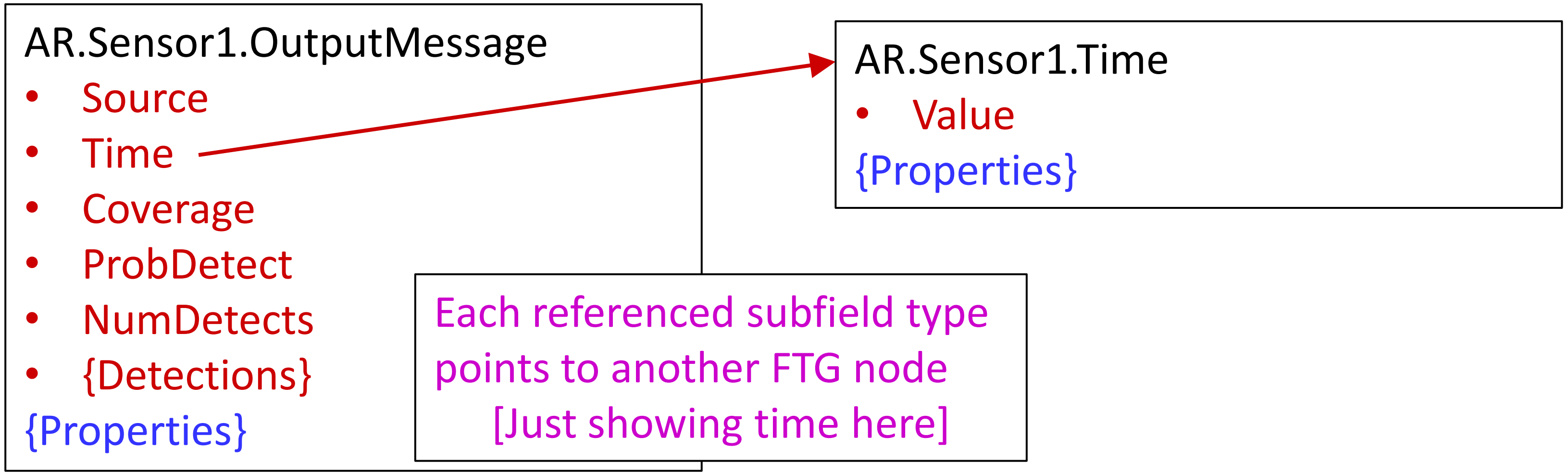
- Global Interoperability without Global Consensus on Interface Specification
  - Stateless Interactions (Message Transformations)
  - Stateful Interactions (Multiple Source Messages Required to Form Destination Message)
- Efficient Reuse in and Evolution of the Architecture
- Near Real-Time Construction of the SoS from Specification
- Optimized Implementation of Interfaces that are Small and Fast
  - Support for High Speed Packed Representations
- Allow Legacy Subsystems and Existing Open Architectures to Interoperate
- Cyber Defenses via Heterogeneity & Run-Time Execution Monitors
- Hierarchical and Resilient SoS Configurations

Key Innovation:
Field and Transform Graph (FTG)

- Fields are Nodes in the Graph and Contain:
  - A set of subfields (which are defined by other nodes in the graph)
  - A set of properties (mathematically precise specification of node properties)
  - Note: All node information is defined locally, no coordination required!
- Nodes are Connected by Links That Define the Transform from Source to Destination Nodes
  - Each link requires a pair wise human coordination between the source and destination
  - Transforms Expressed in a Domain Specific Language Built for this Purpose
  - Graph algorithms determine a composition of transforms (path through the FTG) that produce the destination message given a source message
- No Global Coordination Required to Update or Evolve Data in the FTG

Example of Building Out the FTG

Start with Sensor 1 (Order Doesn't Really Matter, but Details are Order Dependent)



AR.Sensor1.OutputMessage
- Source
- Time
- Coverage
- ProbDetect
- NumDetects
- {Detections}

{Properties}

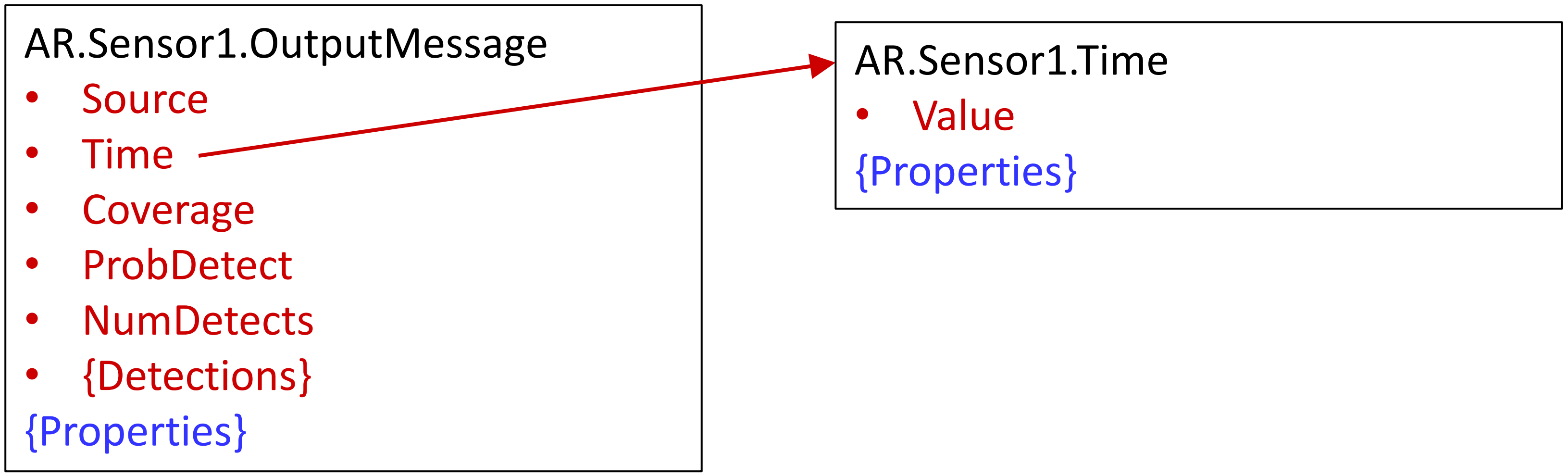
AR.Sensor1.Time
- Value

{Properties}

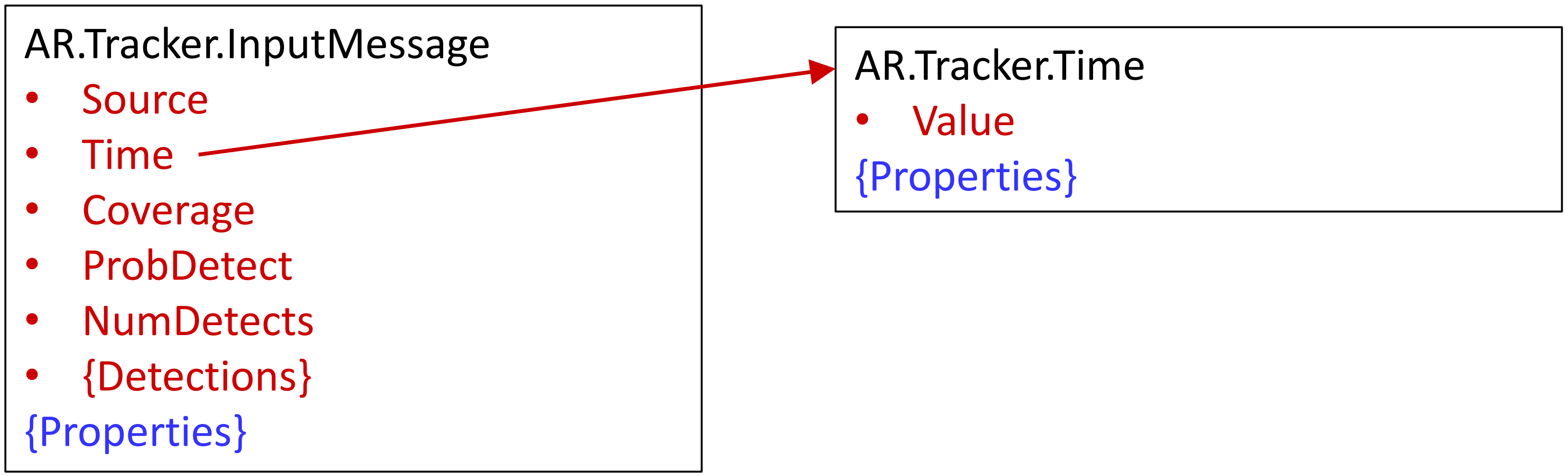
Each referenced subfield type points to another FTG node
[Just showing time here]

Color Code:
Black is a Field Node that represents a Message
Red is a sub-Field Instance Name (referring to a Field Node as its type)
Blue is for Properties (not demonstrated here)
Green is for Transforms (used in later slides)
Purple/Pink is Voice Track Information

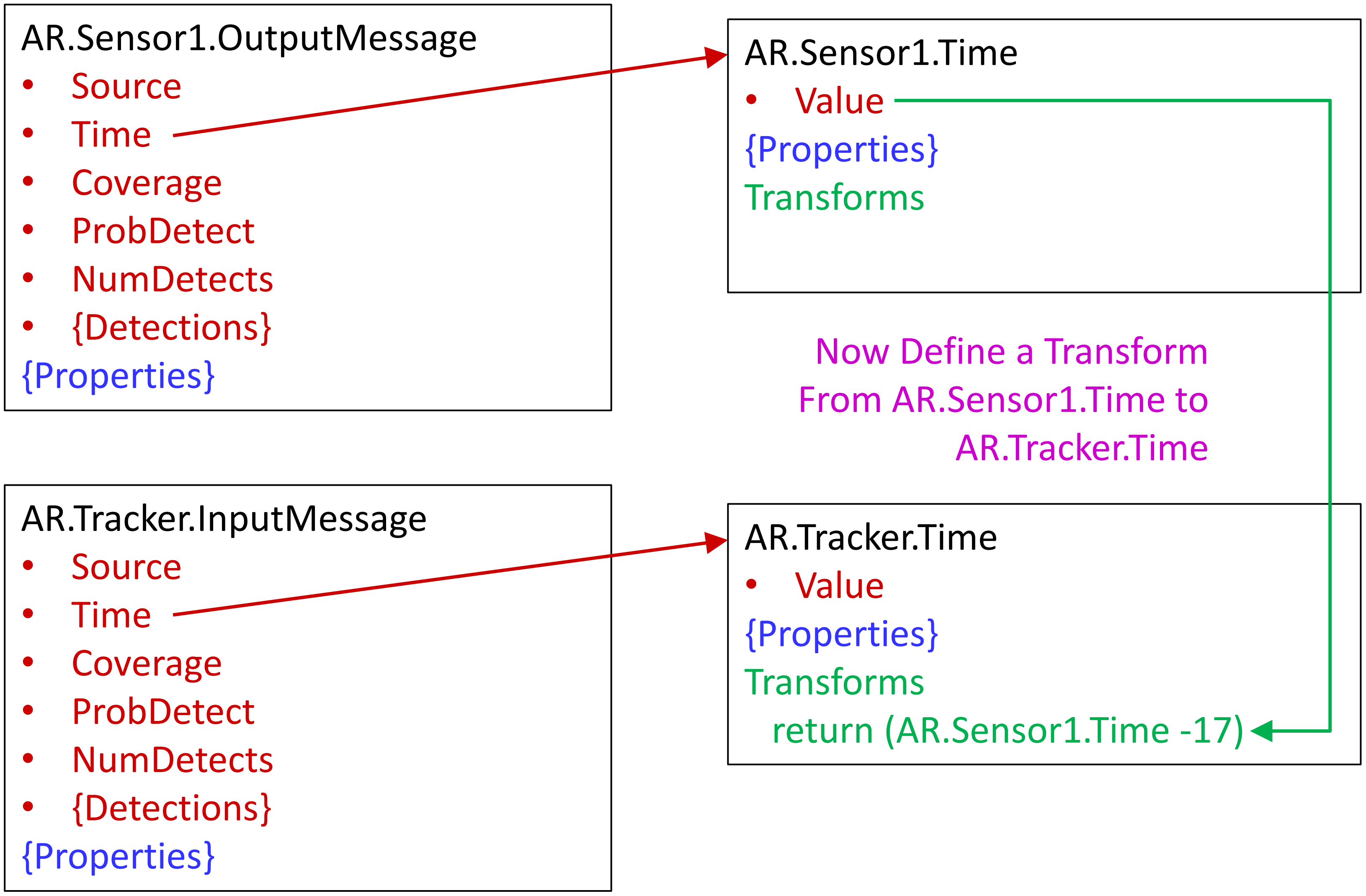
Add in a Tracker Message



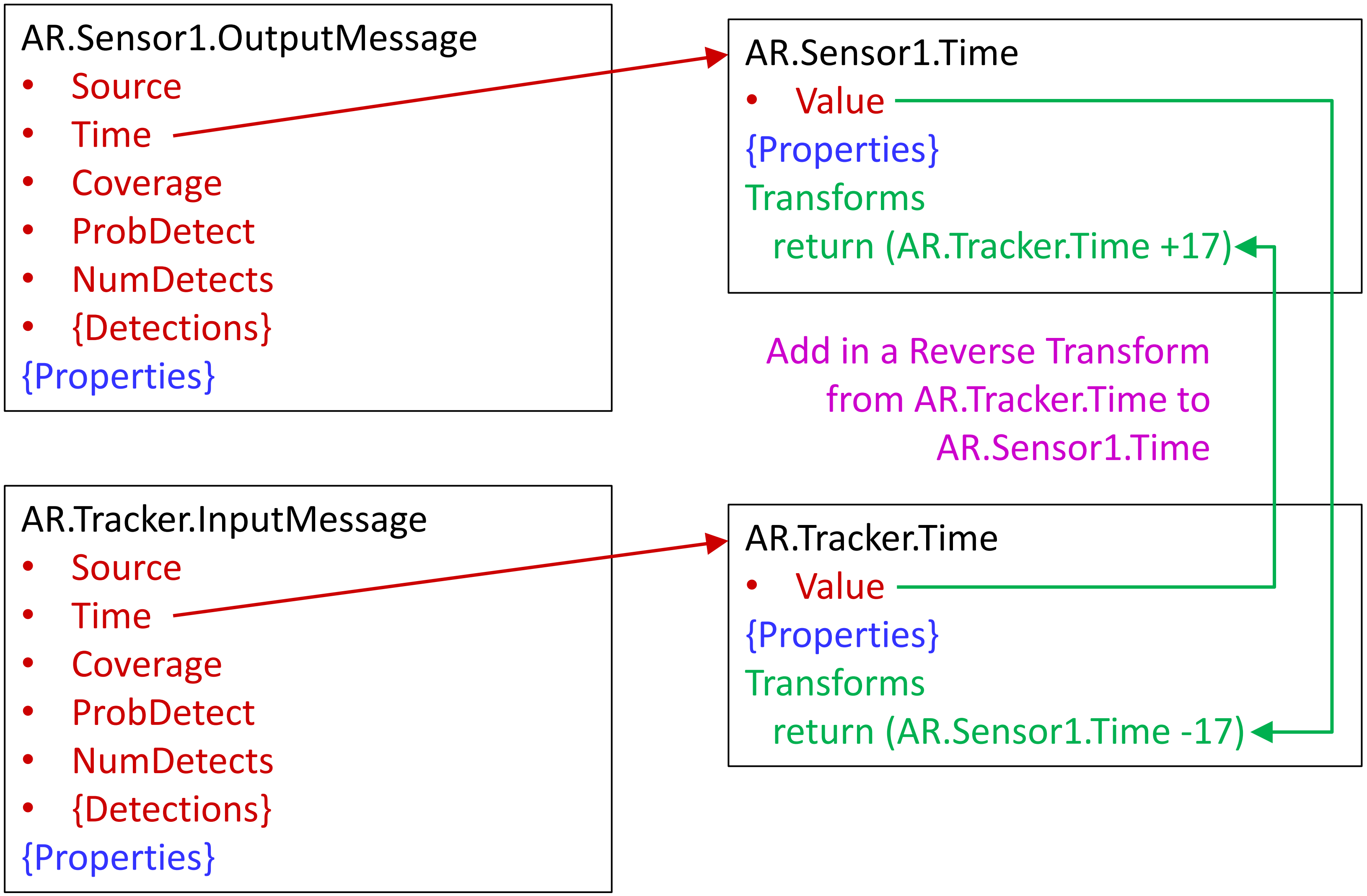
Add in the Tracker



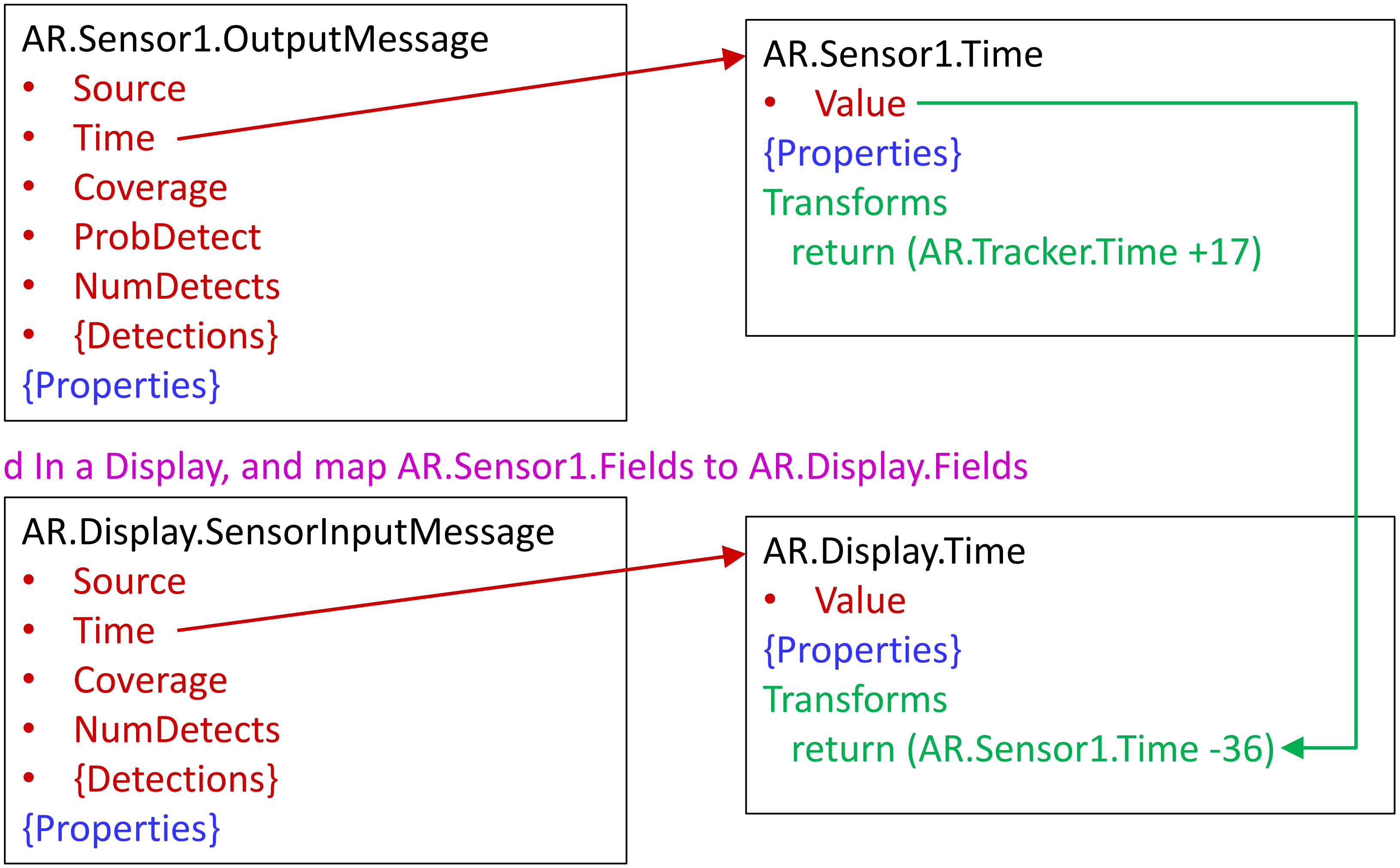
Define a Transform



Add in the Reverse Transform

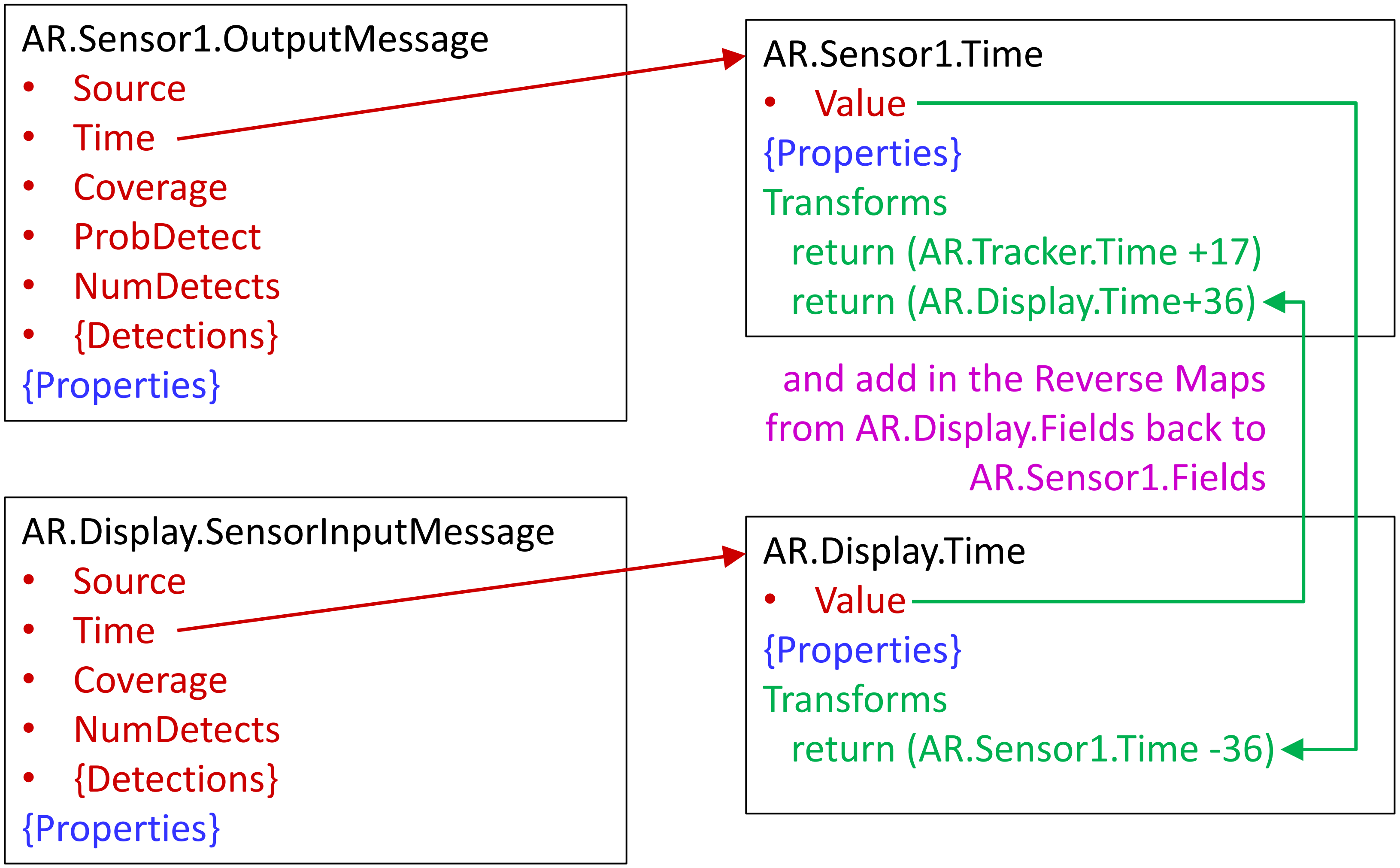


Add in a Display Message



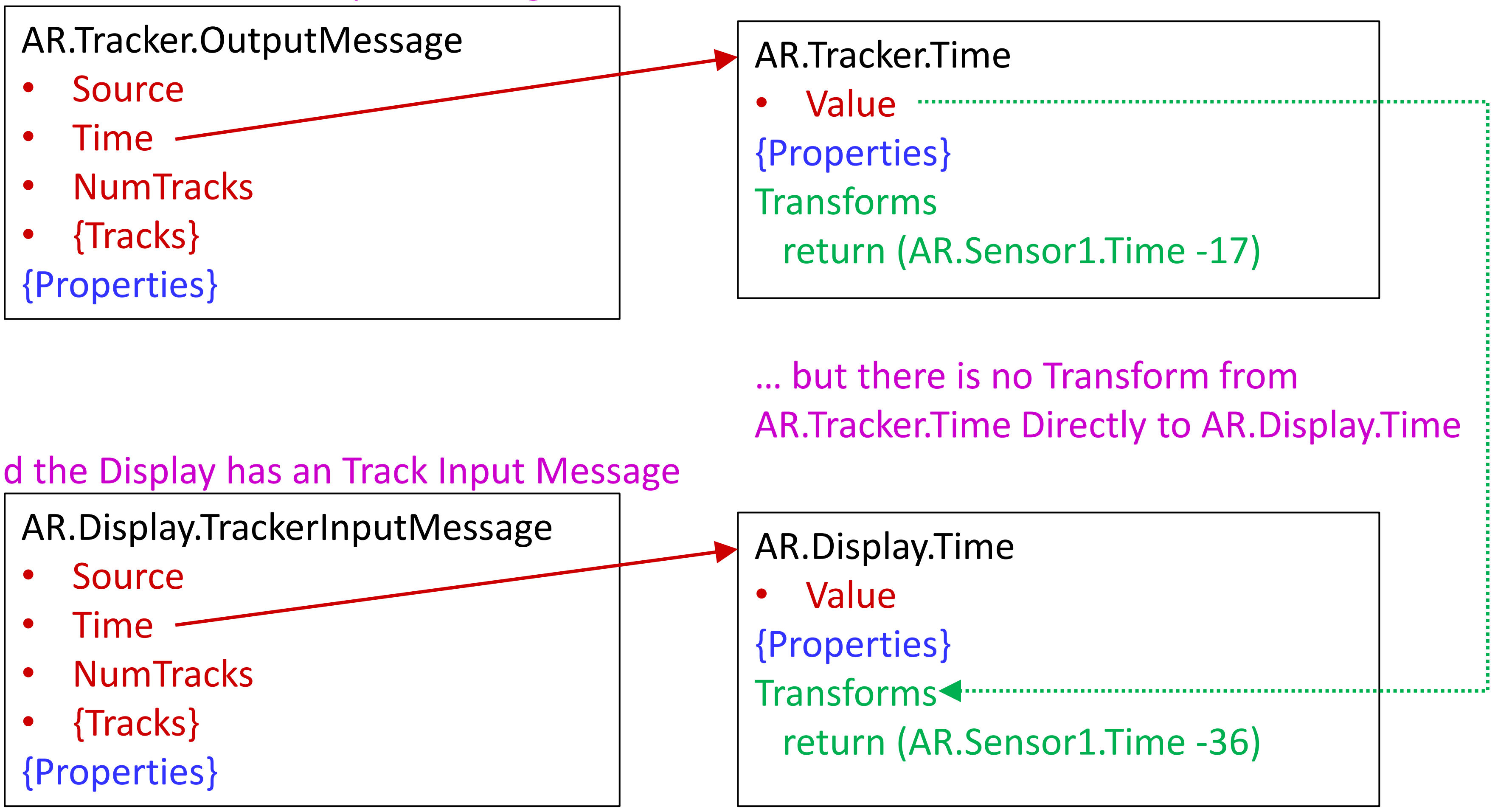
Add In a Display, and map AR.Sensor1.Fields to AR.Display.Fields

Add in the Reverse Transform



What about the Gaps

Tracker Also Has An Output Message



AR.Tracker.OutputMessage
- Source
- Time
- NumTracks
- {Tracks}

{Properties}

AR.Tracker.Time
- Value

{Properties}

Transforms

  return (AR.Sensor1.Time -17)

… but there is no Transform from AR.Tracker.Time Directly to AR.Display.Time

And the Display has an Track Input Message

AR.Display.TrackerInputMessage
- Source
- Time
- NumTracks
- {Tracks}

{Properties}

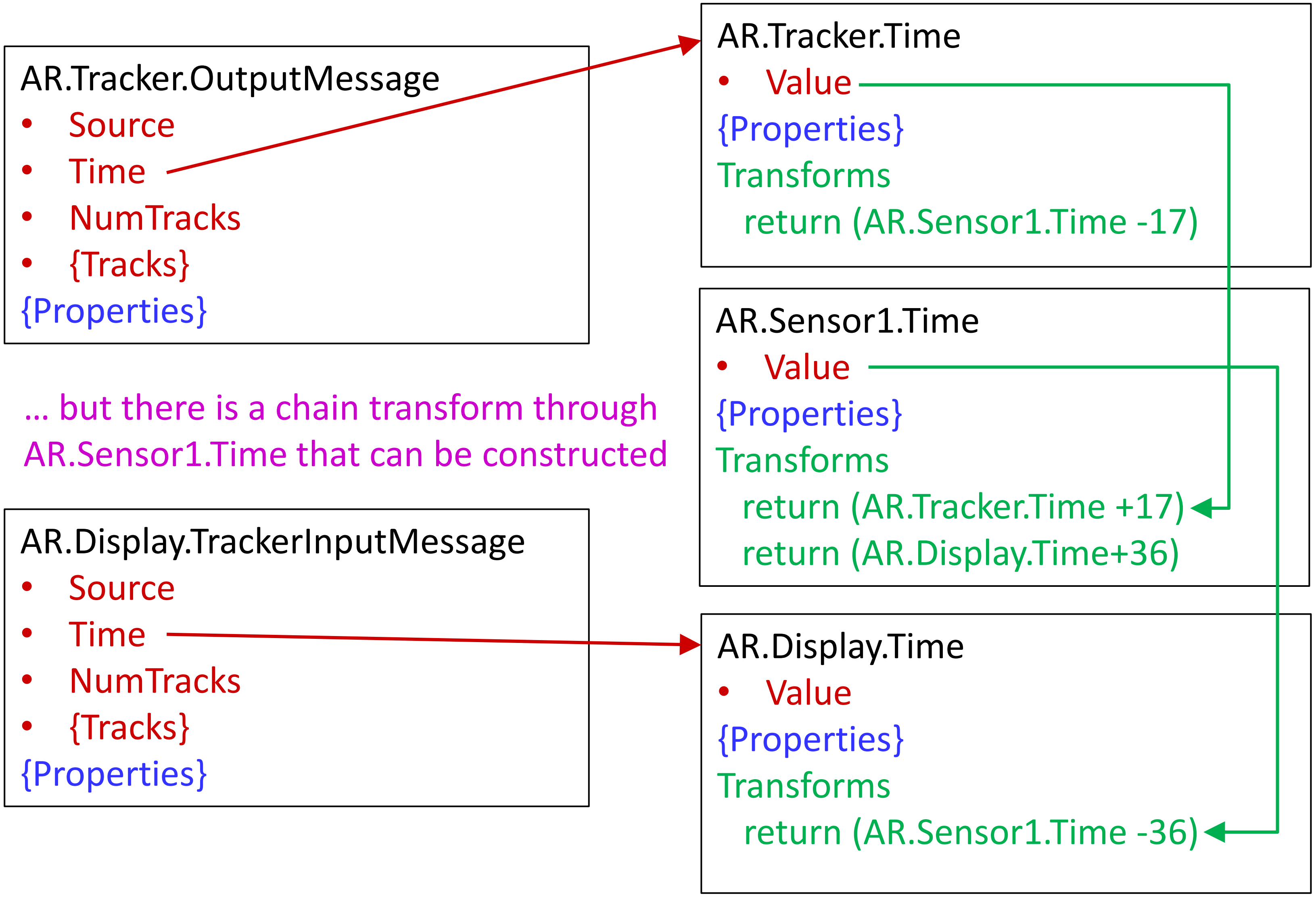
AR.Display.Time
- Value

{Properties}

Transforms

  return (AR.Sensor1.Time -36)

Transform Chains Can Resolve the Gaps



AR.Tracker.OutputMessage
- Source
- Time
- NumTracks
- {Tracks}

{Properties}

AR.Tracker.Time
- Value

{Properties}
Transforms
  return (AR.Sensor1.Time -17)

… but there is a chain transform through AR.Sensor1.Time that can be constructed

AR.Sensor1.Time
- Value

{Properties}
Transforms
  return (AR.Tracker.Time +17)
  return (AR.Display.Time+36)

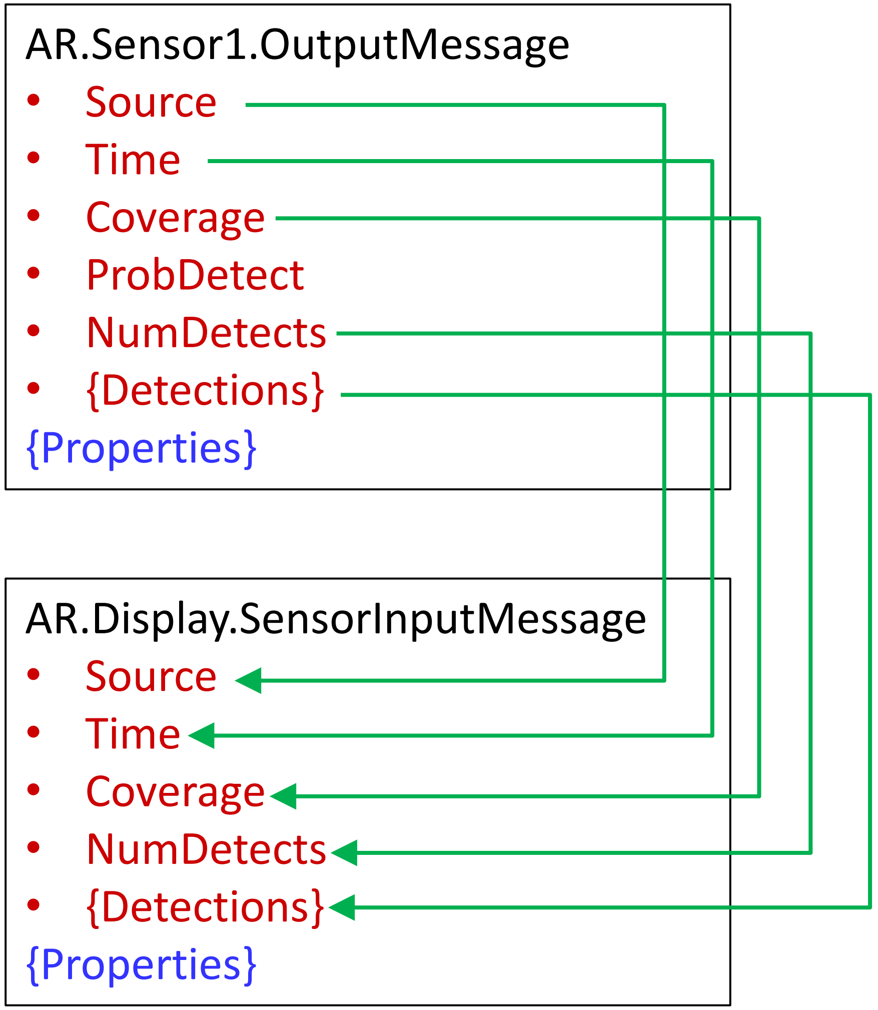
AR.Display.TrackerInputMessage
- Source
- Time
- NumTracks
- {Tracks}

{Properties}

AR.Display.Time
- Value

{Properties}
Transforms
  return (AR.Sensor1.Time -36)

Assign Fields Between Messages



Explicitly Map Fields (via Assign operator) from AR.Sensor1.OutputMessage to AR.Display.SensorInputMessage using the field transforms define in the FTG

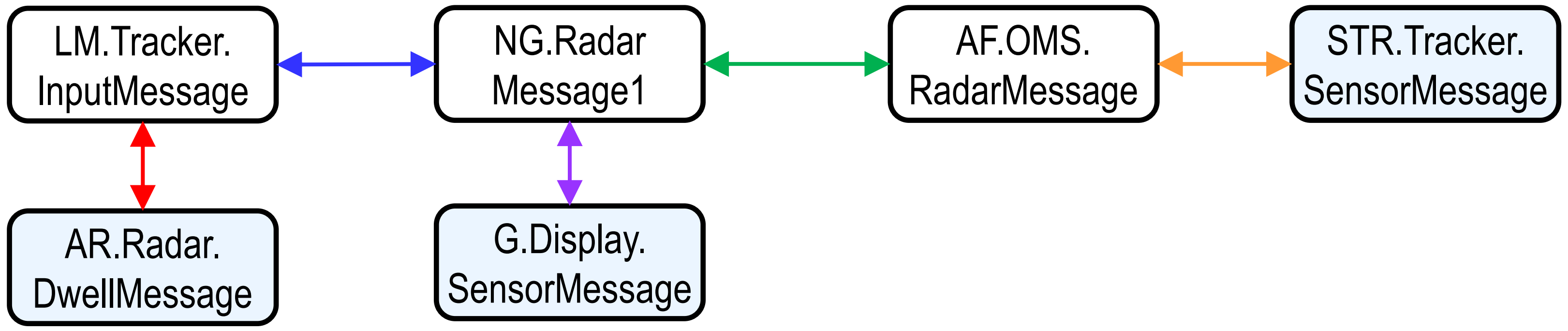
A Simple Example To Illustrate STITCHES

- A Number of Subsystems Have Been Developed Independently
  - Sensors (Radars, EO Cameras, WAMI, etc.), Trackers and Fusers, Display Systems, Sensor Resource Managers, etc.
  - No common standard for message definitions
  - Information content is **compatible**, so they "should" be able to interoperate
- Want to Build Mission Configurations
  - Goal: Supply specifications of mission configurations and have the system autogenerate the code to make the mission work
  - Assumes that the specified mission is supported by the subsystems
- Consider a Simple 3 Subsystem, ISR Configuration Specification
  - Radar: Produces AR.Radar.DwellMessage
  - Tracker: Consumes: STR.Tracker.SensorMessage; Produces: STR.Tracker.TrackMessage
  - Display: Consumes: G.Display.SensorMessage, G.Display.TrackMessage
  - Desired Connections:
    - Radar -> Tracker; Radar->Display; Tracker->Display

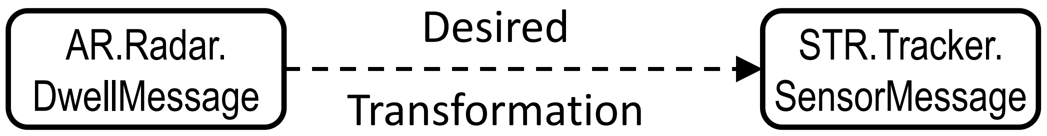
EO: Electro-Optical
WAMI: Wide Area Motion Imagery
ISR: Intelligence, Surveillance, Reconnaissance

Resolving the (Notional) FTG to Form Composite Transformations

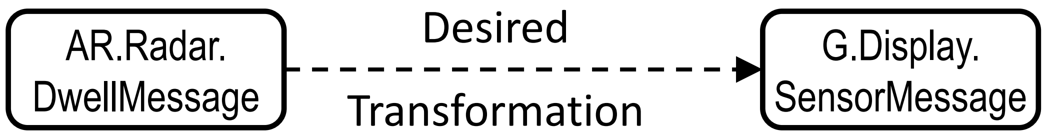
Top Level FTG Relevant for this Example:



Connection 1:
(Radar to Tracker)

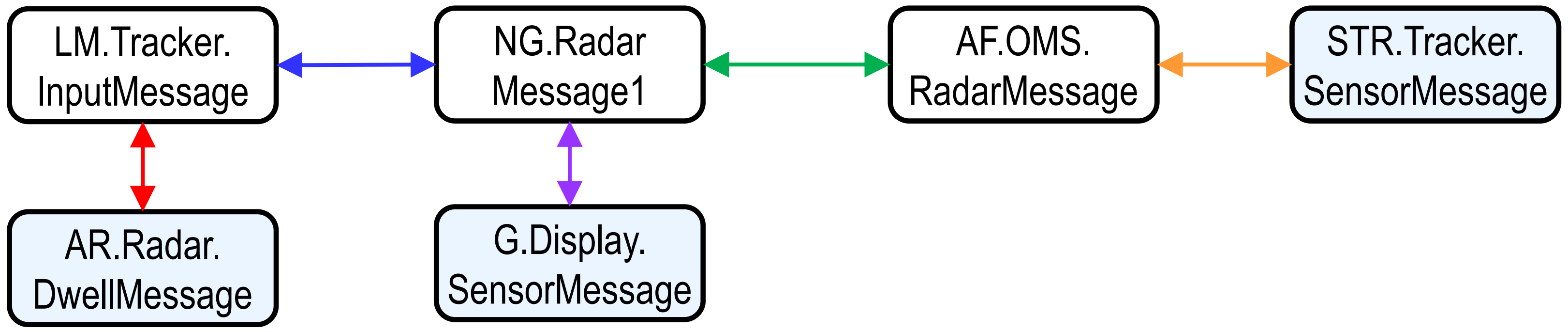


Connection 2:
(Radar to Display)

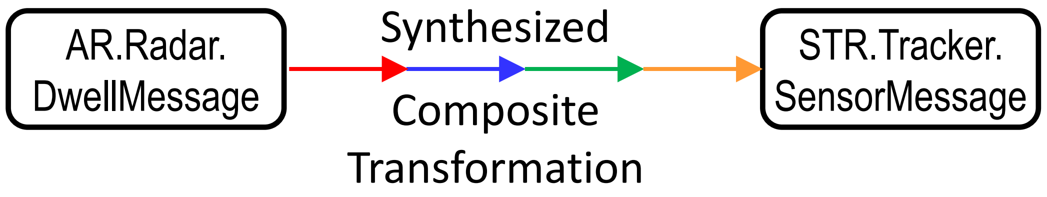


Resolving the (Notional) FTG to Form Composite Transformations

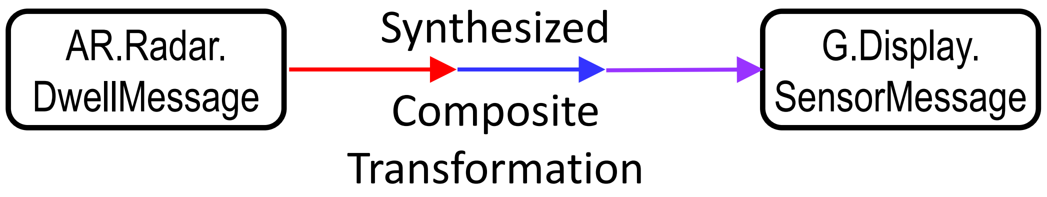
Top Level FTG Relevant for this Example:



Connection 1:
(Radar to Tracker)



Connection 2:
(Radar to Display)

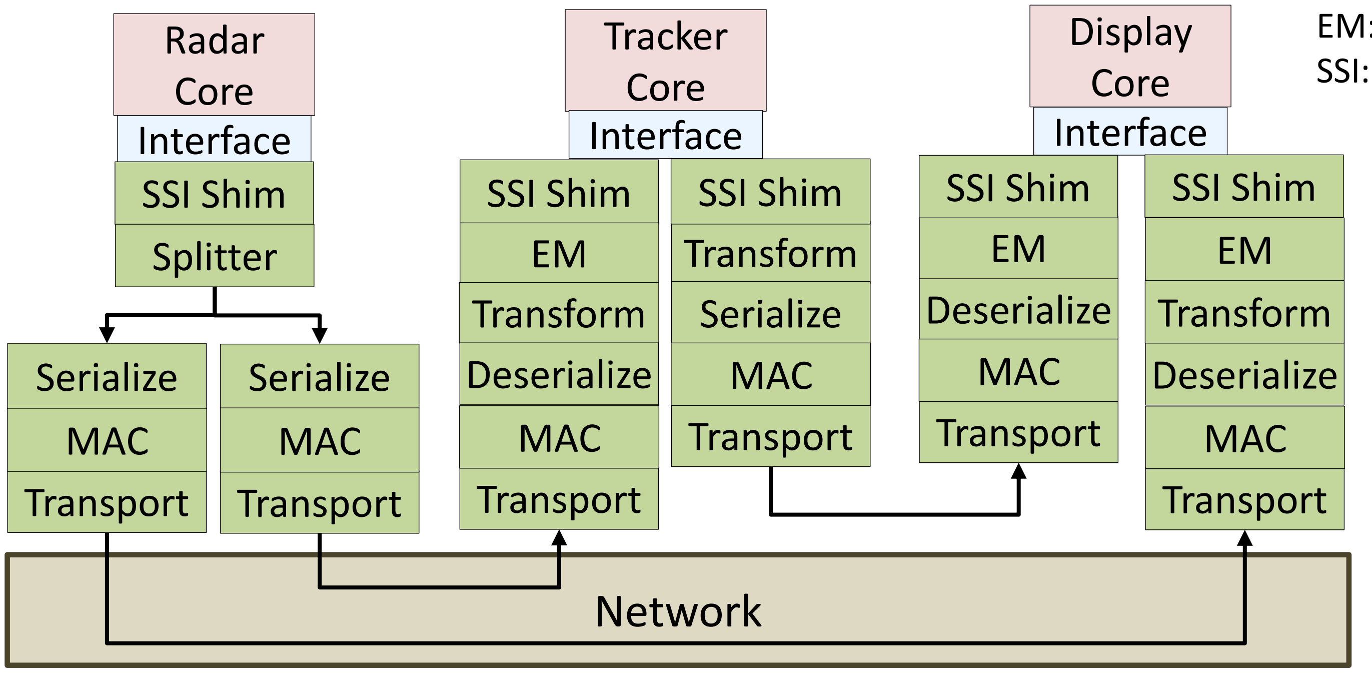


Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

What Does STITCHES Produce?

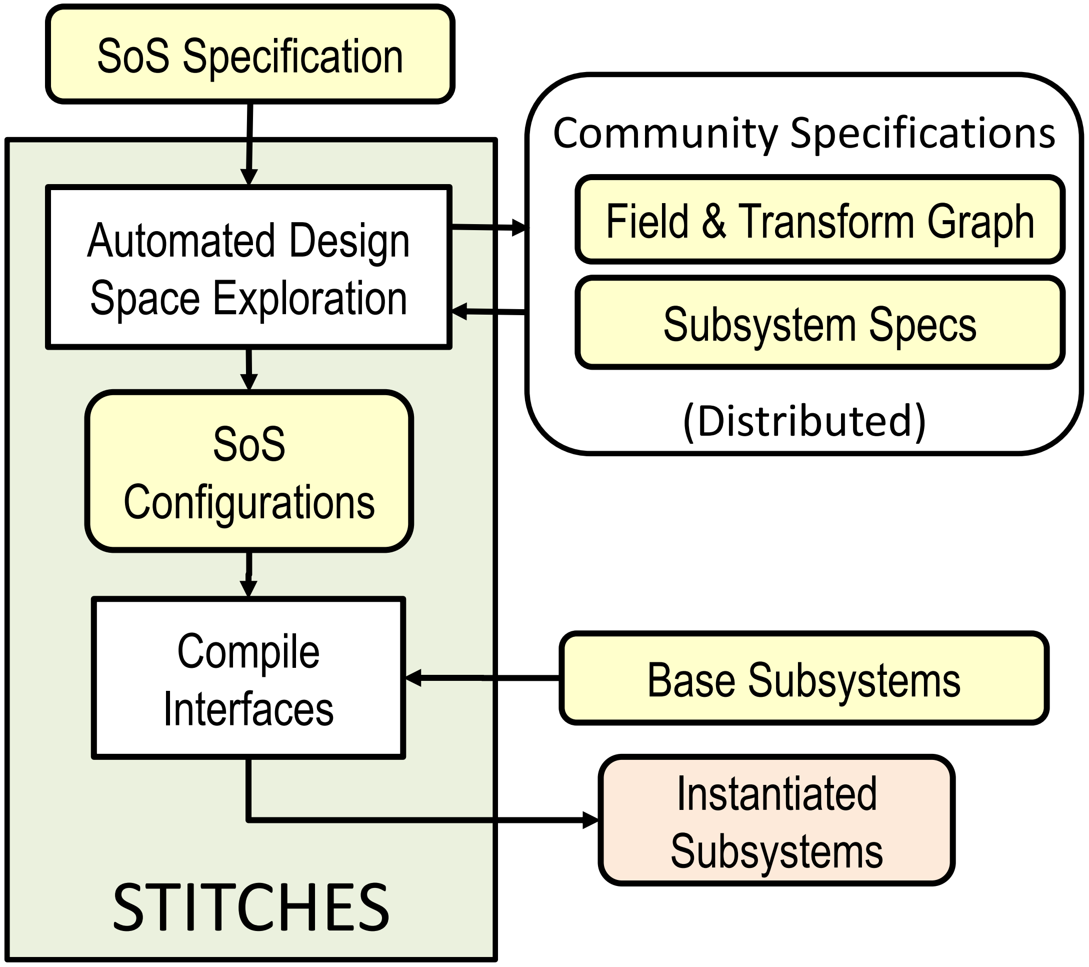
Subsystem Core	Developed by Subsystem Engineers
Interface	Developed by Subsystem Engineer with STITCHES Autogenerated Libraries Developed once per Core Version, Works for all SoS Configurations
HCAL	Autogenerated by STITCHES; Tailored to Each SoS Configuration

HCAL: Heterogeneous CAL
CAL: Critical Abstraction Layer
MAC: Message Authentication Code
EM: Execution Monitor
SSI: Subsystem Interface



STITCHES is Focused on Implementing a Scalable Approach to Building SoS Capabilities

- Design Space Exploration
  - Process FTG to Construct Transformation Chains
  - Specify HCAL Stack by forming & solving optimization problems
- Compiler
  - Construct HCAL Stack Structure
  - Optimize Transforms for this Instance of the Interface
  - Provide Structural Cyber Security through heterogeneity and whitelist property enforcement
  - Generate C++/Java Code & Compile into binaries



Result: High Performance Interfaces Optimized For Each Application

SS and SoS Specs Define the Remaining Elements of the System in an Efficient Way

SubSystem Specifications

For Each SubSystem:

- Input Interfaces
  - Messages (FTG Node)
- Output Interfaces
  - Messages (FTG Nodes)
- Resources
  - Supported Languages
  - Supported Transports
  - Supported Serialization
  - Computational Resources

SoS Specification

Defines Each SubSystem(SS):

- SS Instance Name & Type

Defines Each Connection between SSes

- Source Information (1 or more)
  - SS Instance Name
  - SS Interface
- Destination Information
  - SS Instance Name
  - SS Interface
- Message Flows (1 or more)
  - Source Message(s)
  - Destination Message

Walking through the Steps…

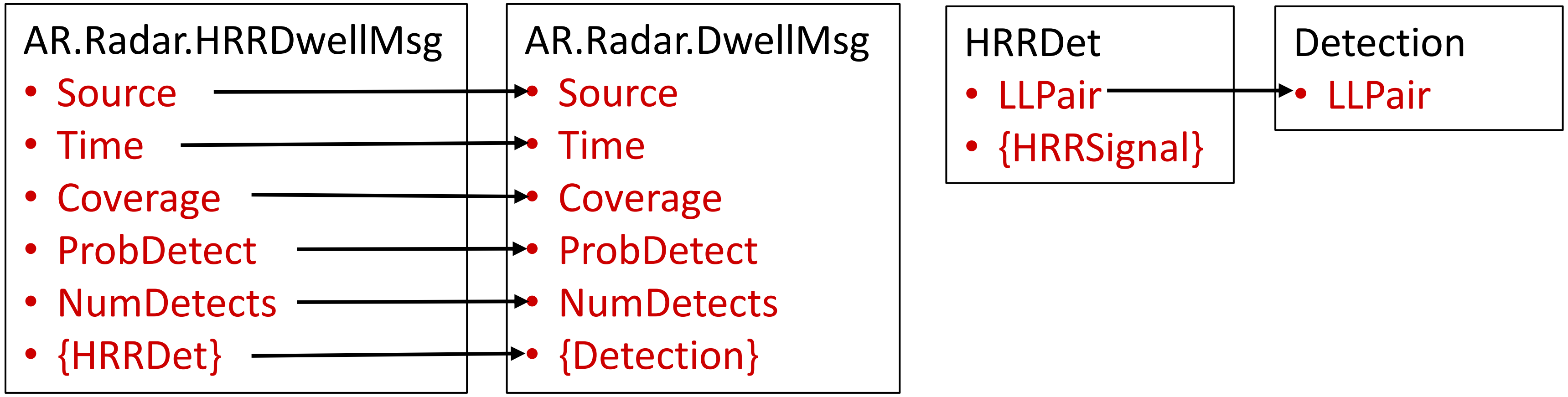
- Create a STITCHES Enabled Subsystem (Time: ~Days; Freq: Once per SS version)
  - Start with an existing Subsystem (SS) core
  - SS Engineer Models the interface in the FTG (creates nodes)
  - STITCHES auto-generates SS Interface (SSI) Skeleton; SS engineer completes SSI
  - SS Engineer adds FTG links to other nodes to connect to a community
  - Check in the FTG
    (including annotations and verification with tools such as Rockwell's AGREE)
- Create a New SoS (Time: ~Hours; Frequency: Once per SoS)
  - SoS engineer defines the SoS Specification (SSes and their connections)
  - If needed, SoS engineer must add any missing required links in the FTG
  - Check in New FTG including incremental verification (with tools such as AGREE)
  - SoS engineer runs STITCHES on the SoS Specification to Build the SoS (generate code)
- FTG Provides Re-Use without Common Interfaces
  - An FTG Node is re-used in many messages
  - Messages are re-used in many subsystem interfaces
  - Subsystems are re-used in many SoSes

Next We Walk Through a Few of the More Interesting Features in STITCHES

1. Architectural Evolution (both Backwards & Forwards Compatibility)
2. Compiler Converts a Specification to Running Systems
3. Compile Time Performance
4. Run Time Performance
5. Integrating Legacy Systems that Can't be Changed
6. Cyber Resiliency via AutoGenerated Runtime Enforcement of White List Property
7. Synchronizing Transformations
8. Hierarchical Definitions Including Resilient Backup Configurations

Evolution of the Architecture: Backwards Compatibility

- Let's Remember Our Simple 3 Subsystem ISR SoS
  - Radar: Produces AR.Radar.DwellMsg
  - Tracker: Consumes: STR.Tracker.SensorMsg; Produces: STR.Tracker.TrackMsg
  - Display: Consumes: G.Display.SensorMsg, G.Display.TrackMsg
  - Connections: Radar -> Tracker; Radar->Display; Tracker->Display
- Now Let's Add in an Upgraded Radar (Includes an HRR Signature)
  - Construct Transform That Assigns Each Subfield from the new to the old Message



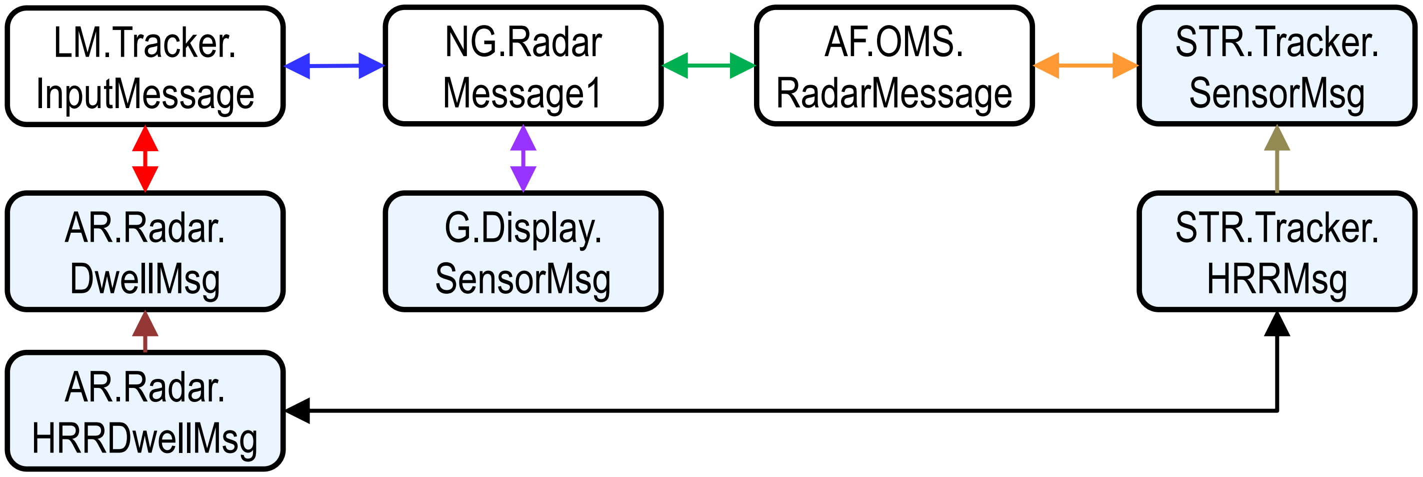
  - Now can use the new Radar anywhere that you can use the old radar (but won't get access to the HRR signal)

Evolution of the Architecture: Forwards Compatibility

- Now Let’s Add in an Upgraded Tracker (That can use HRR Signatures)
  - Construct Transform That Assigns Each Subfield from the HRR Source to HRR Destination

AR.Radar.HRRDwellMsg		STR.Tracker.HRRMsg
Source	→	Source
Time	→	Time
Coverage	→	Coverage
ProbDetect	→	ProbDetect
NumDetects	→	NumDetects
{HRRDets}	→	{HRRDets}

  - Now the New Tracker can use the HRR Signals from the new Radar

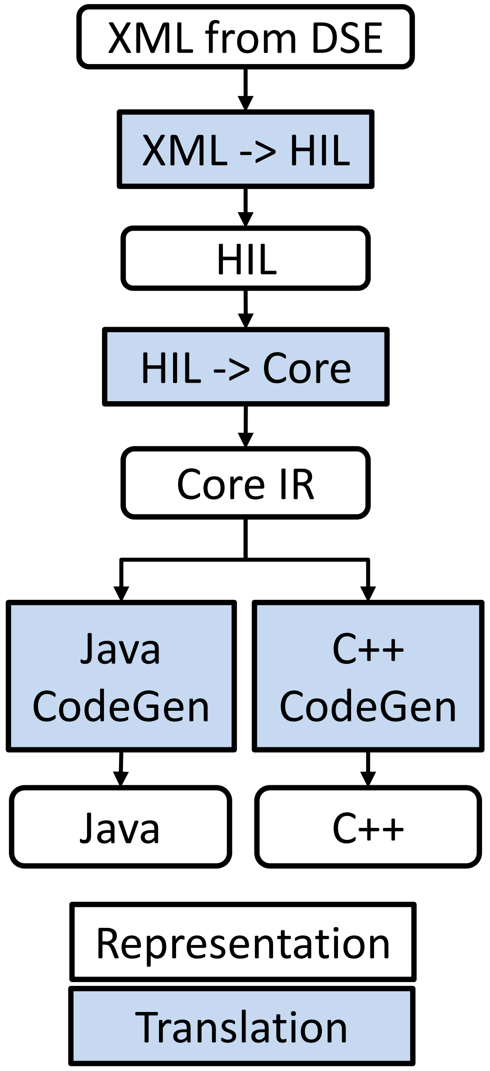


The STITCHES Compiler, Motivation

- In general, Design Space Exploration (DSE) chooses the HCAL's functionality – the what not the how:
  - "Transform from MsgA to MsgB using this chain", …
  - "Then check these properties with an Execution Montior", …
  - "Then serialize using Google Protocol Buffers", …
  - "Then send to Destination via ZeroMQ"
- Compiler Transforms This Description into Executable Implementations (How)
  - For any HCAL, there are many possible implementations
  - Objective implementations are secure, performant and interoperate with any STITCHES compiler
- A more traditional approach is to use a framework (common super-classes, generic interfaces, etc)
  - Because we generate the code after we see the optimized SoS Config, we don't need a framework!
  - Enables significantly more compile time validation and optimization

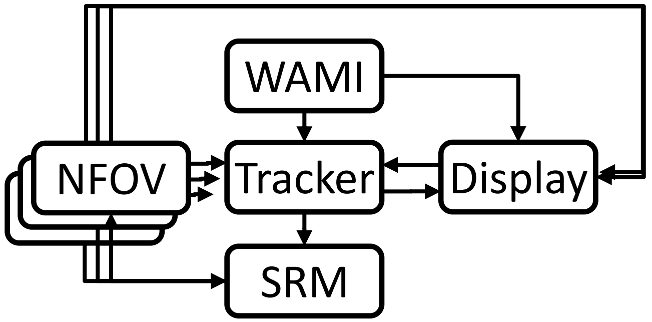
The Compiler Transforms the Specification Into Running Code

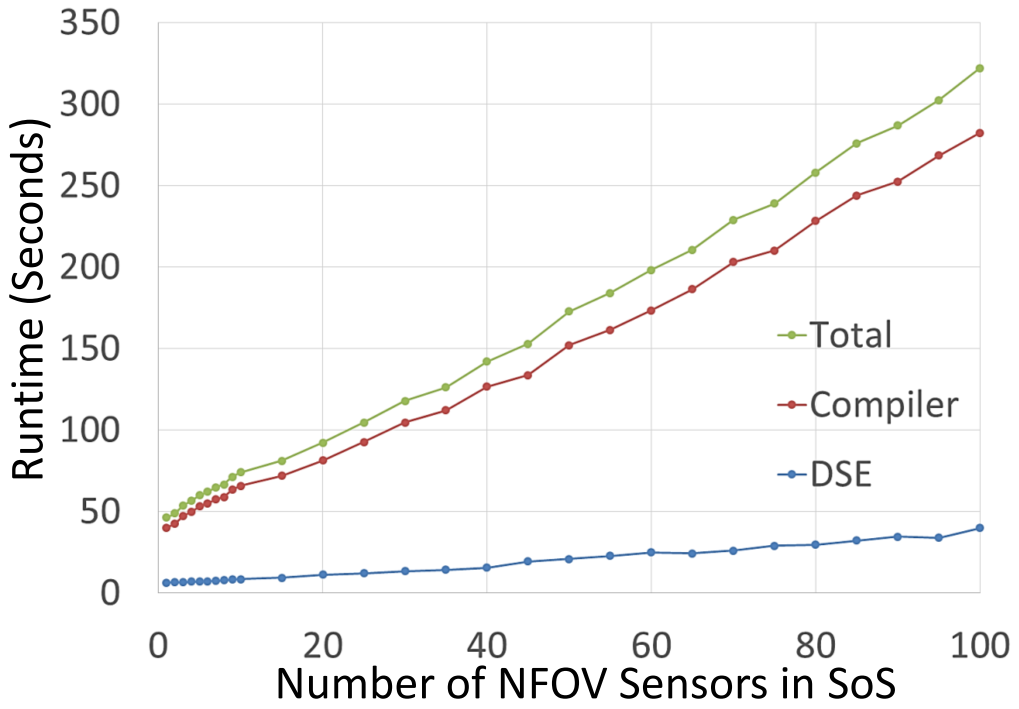
- HCAL Intermediate Language (HIL) is designed for human use to fully specify stacks and transformations
  - Provides high-level functions, allows complex nested expressions, and provides syntactic sugar to simplify use
  - Creates fully-defined type system allowing compile-time validation and reduced run-time errors
  - Parsed from a format that is human readable and editable, to allow for inspection, debugging and testing
- Core Internal Representation (IR) is designed for machine analysis (optimization and code generation)
  - Creates an explicit representation that is more easily reasoned over for the purposes of optimization
  - Adds variable names to intermediary products to ease code generation in both Java and C++
- Target Languages of Java and C++ allow for wide applicability of the result



Does STITCHES Scale to Relevant Sized SoSes?

- Simple ISR SoS
  - WAMI, Tracker, Display, SRM and Variable Number of NFOV Sensors
  - Connections:
    - WAMI->Display; WAMI->Tracker
    - Tracker->Display;
    - Display->Tracker; Tracker->SRM;
    - SRM->NFOV
    - NFOV->Tracker; NFOV->Display
- Complexity Grows with N(# NFOVs)
  - # Subsystems = N + 4
  - # Connections = 3N + 5
- Note: Execution Monitors Are Disabled in These Runs

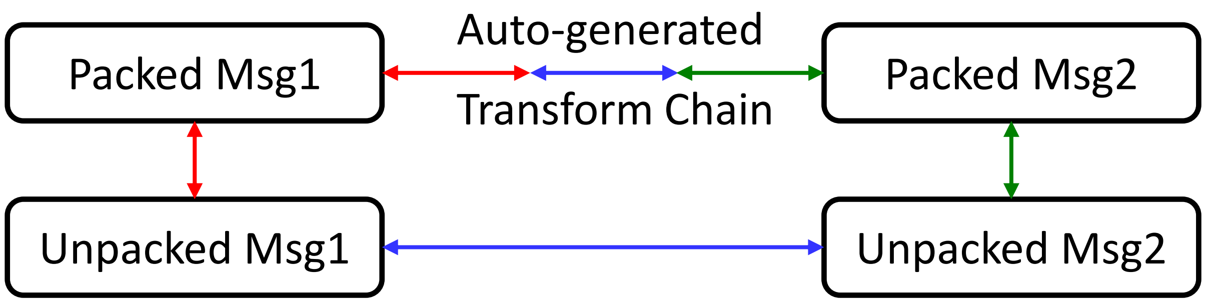




WAMI = Wide Area Motion Imagery
NFOV = Narrow Field of View

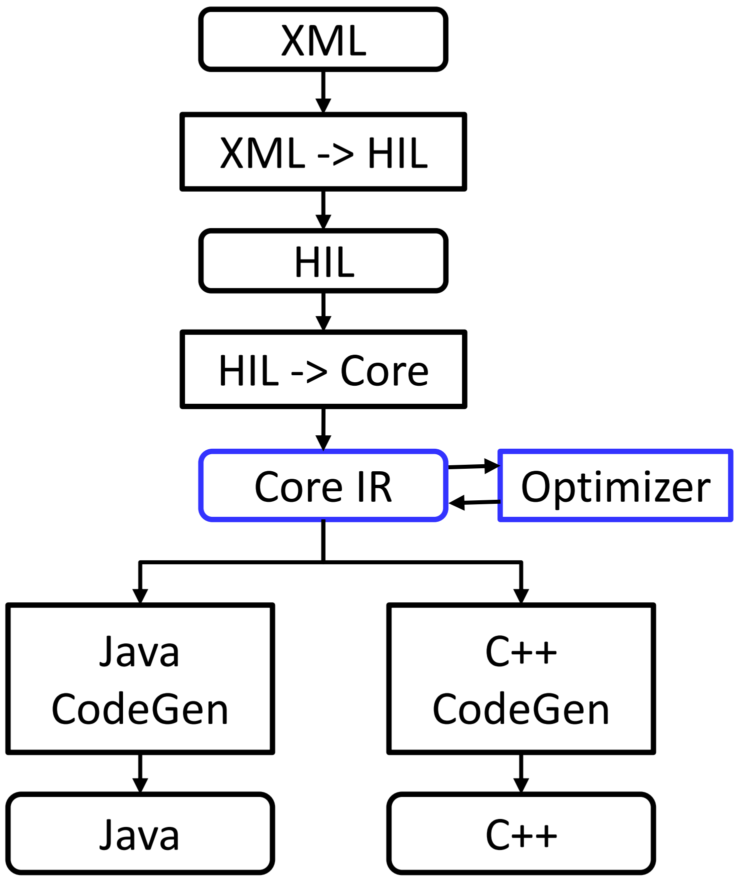
Handling Packed Representations

- Many real systems mix their interface definition with their implementation
  - Result is a serialized (Packed) form of the interface that can represent multiple different interface messages (e.g., STANAG 4607) with descriptor words for run time resolution
  - Packed messages are often used for run-time efficiency - they tend to be the big / high rate messages in the system. So don't want to unpack if not necessary
- Mirrored Unpacked Nodes Provide an Effective and Efficiency Solution
  - Create a Second Unpacked Node that Contains a Structured Version of the Interface
  - Create Transforms between the Unpacked and Packed Nodes
  - Interact with other Interfaces via their Unpacked Representations
  - Auto-generate the Desired (high performance) Packed-to-Packed Transforms



STITCHES Uses Post-Composition Optimization

- STITCHES assembles sequences of transforms that may involve:
  - Frequent copying
  - Duplicated computation
  - Inverted computations: e.g., `ToBytes(ToDouble(input[:800]:bytes));`
  - Inefficient extra looping over the same data
- Current Optimizations
  - **Simplification** to remove assignments
  - **De-duplication** of computation
  - **Peephole Optimizations** to replace code sequences with faster equivalents
  - **Loop fusion** to combine operations on the same data over multiple loops (not included in performance results on next slide)
  - Note: the optimizer currently only optimizes in the transform layers



Optimized Performance:

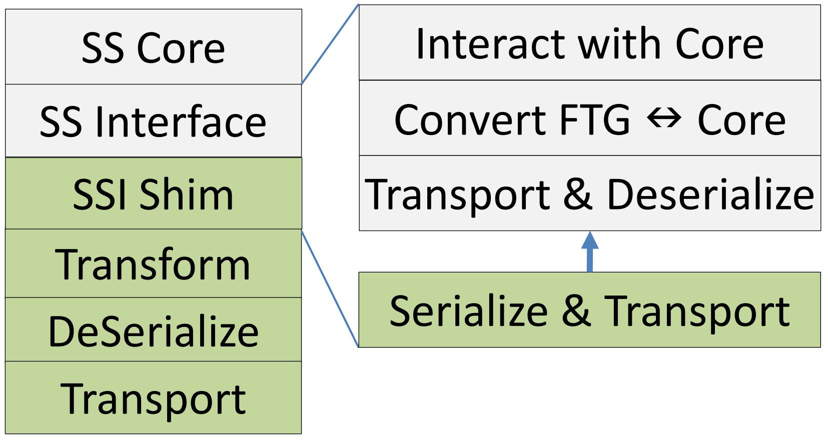
[Packed → Unpacked → Unpacked → Packed] vs. [Packed → Packed]

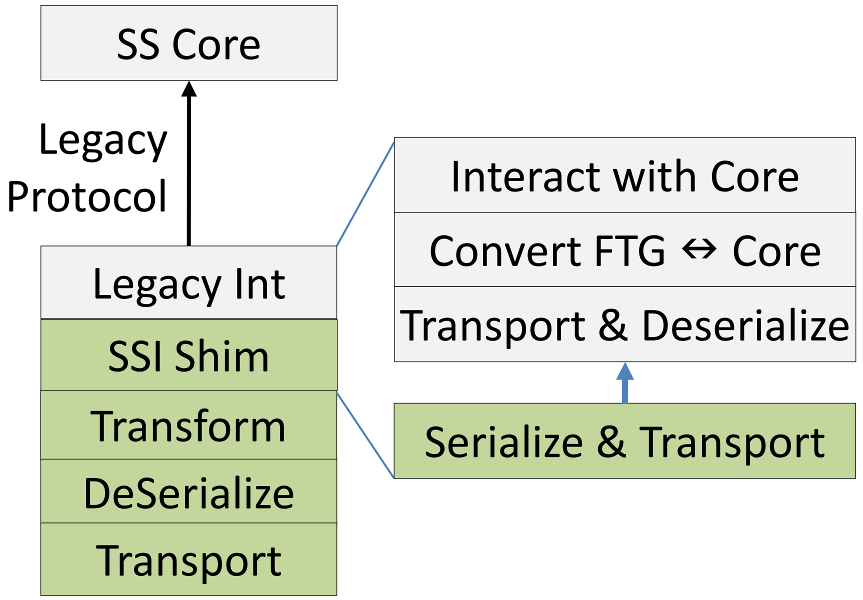
Connection	PUUP vs PP	Java HCALS		C++ HCALs	
		Speed Mbps	Latency ms	Speed Mbps	Latency ms
R1 -> T1 (No Transform)	PUUP	3000±35	1.1±0.1	2889±52	0.7±0.1
R1 -> T1 (No Transform)	PP	3005±18	1.0±0.1	2897±38	0.7±0.1
R1 -> T2 (Only Change Time)	PUUP	1972±18	1.1±0.1	2891±38	0.7±0.1
R1 -> T2 (Only Change Time)	PP	1967±22	1.2±0.1	2889±53	0.7±0.1
R1 -> T3 (Switch Order Lat, Lon)	PUUP	1100±9	1.5±0.1	1035±32	1.1±0.1
R1 -> T3 (Switch Order Lat, Lon)	PP	1058±9	1.6±0.1	1042±25	1.2±0.1
R1 -> T4 (Change All Fields)	PUUP	685±5	2.0±0.1	963±23	1.2±0.1
R1 -> T4 (Change All Fields)	PP	755±7	1.9±0.1	898±21	1.3±0.05

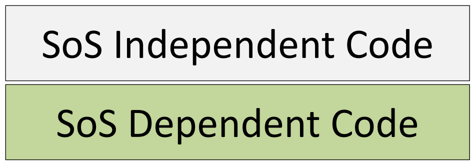
MAC and Execution Monitors are Disabled for these Performance Runs
All interactions via localhost, so no network latencies are involved
Data Gathered on a Standard Quad Core Workstation

Unifying Legacy and STITCHES Enabled SSes

- What Makes a Subsystem STITCHES Enabled?
  1. Implements an SS Interface (SSI):
     - Converts between core and FTG interface
     - Provides FTG messages to an HCAL process (currently through a named pipe)
  2. Provides a Local Computational Resource to Host an HCAL Locally
- Legacy Systems Can't be Changed
  1. No Issue Implementing an SSI
     - Interaction with the Core is more complicated (via legacy protocol)
     - Otherwise it is the same
  2. Need to Host HCAL Somewhere Else

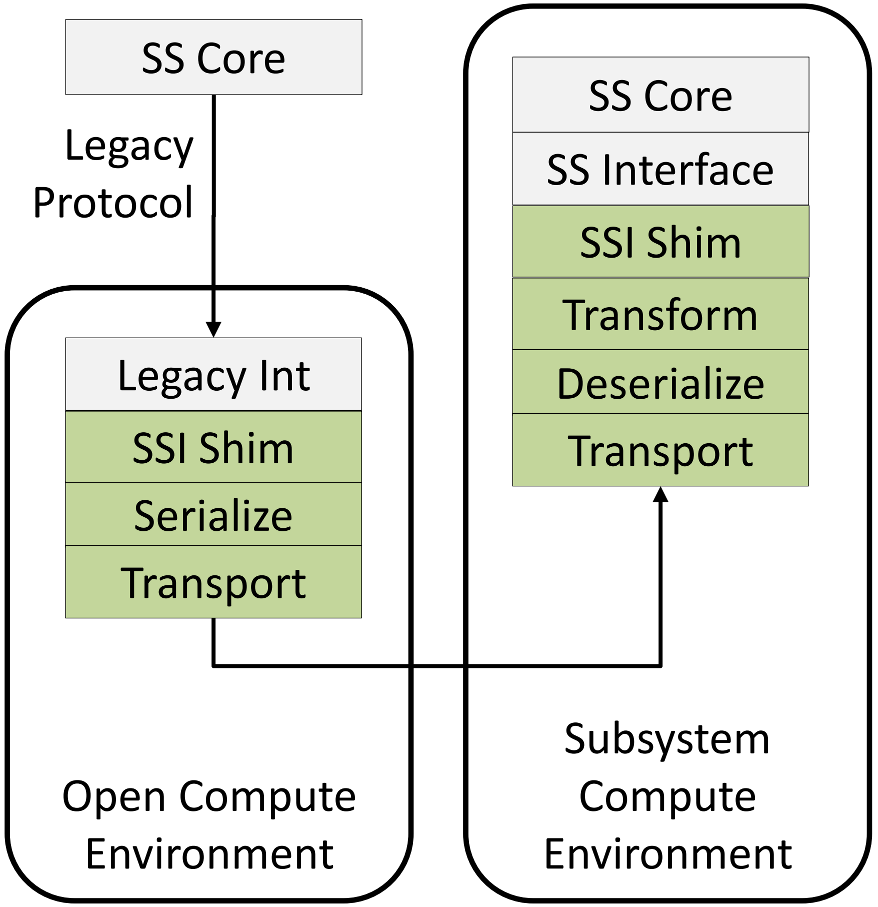






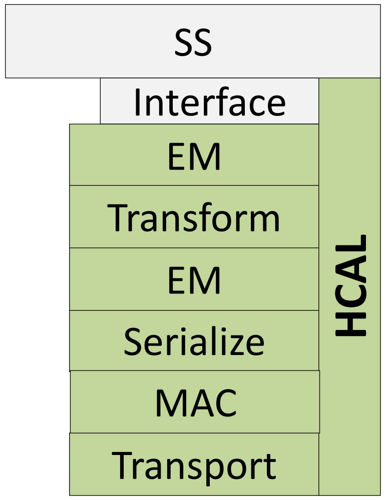
Clean Solution is to Host the HCAL on an Open Compute Environment (OCE)

- Legacy Interface (LINT) Now Acts as a Remote SS Core
  - Connections with the SSI Shim are the same as if it were local (pipes)
  - Can Split/Join Feeds in the HCAL if multiple Connections are Required
- Further Optimization is Possible
  - Move the LINTs to the Subsystem
  - Remove the Extra Serialize, Transport, Transport, Deserialize Layers
- Optimization Would Break A Key Assumption in Current Architecture
  - LINTs act like a SS Interface – which only exist in one place
  - Resilient SoS Config Sometimes Require LINTs Feeds to be in multiple HCAL Stacks
  - Note, equivalent to the optimization of letting software only SSes live on another HCAL



Cyber Resiliency

- Use Detailed Information on Interface Properties to Improve System Cyber Defenses
  - Auto generate distributed whitelist enforcement (EMs) of field, message and SoS properties based this specific composition
  - Provide a minimal, unpredictable attack surface via heterogeneous implementations of the minimal interface needed for this composition's interactions
- Execution Monitors Provide a Framework for Loading White List Property Checks Into the HCAL
  - Property checks written into the FTG nodes
  - Automatically loaded into the EM based on available resources
  - If Any Property Check Fails, the Message is Suppressed
- Current Version of STITCHES Also Supports MAC (Message Authentication Codes) for Crypto Signatures



HCAL: Heterogeneous CAL
CAL: Critical Abstraction Layer
MAC: Message Authentication Code
EM: Execution Monitor

Synchronizing Transforms Handle the Stateful Logic for Asynchronous Message Inputs

- Case 1: Combine Messages From Multiple Sources
  - Example: Tag EO Messages with GPS Time
  - Messages are still independent, so no requirement on message delivery
  - First Argument is FIFO (First In, First Out) Processed, All Others are Port Sampled (Most Recently Received)
- Case 2: Fragmentation Logic Differences Between Source and Destination
  - Example: STANAG 4607 (Dwell is sent as a sequence of Header, Data[], Trailer)
  - Must Impedance match between different fragmentation standards
  - Messages are dependent – If you drop one, the entire dwell is invalid
  - Interface is defined by the Logical Message (Dwell), but implemented “Virtually” by a sequence of messages (subfields of Dwell)
  - Only the first argument of a Synchronizing Transform can be implemented “Virtually”
- Synch Transforms Use a Simple State Machine Construct with Three Actions
  - Init Block: Construct Persistent Context Variables
  - Inc Block: Increment the current state (mutation!) based on the new message
  - Term Block: Finish processing the current context and then close it

A Simple Example: Defragmenting a Dwell

APOGEE RESEARCH

```
AR.Radar.Message3
• header: AR.Radar.Header
• dets[:32]: AR.Radar.Data
• trailer: AR.Radar.Trailer
```

```
AR.Radar.Header
• Source: AR.Radar.Source
• Time: AR.Radar.Time
• Cov: AR.Radar.Coverage
• PD: AR.Radar.PD
```

```
AR.Radar.Data
• dets[:16]:AR.Radar.Det
```

```
AR.Tracker.Message1
• Source: AR.Tracker.Source
• Time: AR.Tracker.Time
• Coverage: AR.Tracker.Coverage
• ProbDetect: AR.Tracker.ProbDetect
• NumDetects: AR.Tracker.NumDetects
• Dets[:512]: AR.Tracker.Detections
```

```
Source Msg: AR.Radar.Message3 (Virtual)
Destination Msg: AR.Tracker.Message1

Context Variable
  detections[:512]:AR.Radar.Det;
Init Bind on AR.Radar.Header
  Source = Assign(in.Source);
  Time = Assign(in.Time);
  Coverage = Assign(in.Cov);
  ProbDetect = Assign(in.PD);
Inc Bind on AR.Radar.Data
  Append(detections, in);
Term Bind on AR.Radar.Trailer
  Dets=Assign(detections);
  NumDetects = AR.Tracker.NumDetects
   {Value = Len(detections)};
  send out;
```

Special Variables:

in: message just received

out: instance of Destination message

Hierarchical SoSes

Systems

System of Systems applies hierarchically

- Components into an electronic "box"
- Boxes connected via an avionics bus
- Avionics buses on an aircraft
- Aircraft in a flight

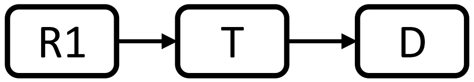
Every System of Systems is a System **and** is Composed of Systems

- We use System of Systems (SoS) for the composed system
- We use SubSystem (SS) for the systems being composed

- Incorporate SSes that are Actually Implemented as SoSes (SSS)
- What Makes a Subsystem a Subsystem (to STITCHES)?
  - Provides a Set of Interfaces for SoS Composition
  - We Don't Need to Understand Where Those Interfaces are Implemented (Abstraction of Interfaces Allows Efficient Hierarchical SoS Composition)

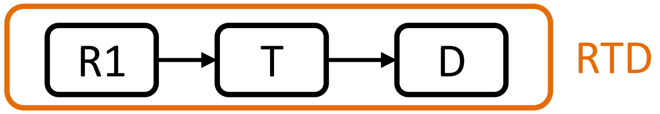
Consider a Subsystem Defined by a Simple SoS Configuration with (Pseudo) Spec

- Consider a Simple 3 SS SoS
  - Radar feeds a Tracker
  - Tracker feeds a Display
- Define a Single Config SoS SS (SC-SSS)
  - Internal Connections Define How the SC-SSS is Wired Up
  - External Interfaces Define How Other SSes can Interact with it
  - External Interfaces are Implemented by an Internal Interface
- RTD Can Now Be Used as a SS in Other SoS Configurations
  - Note: A SS can only be directly used in a Single SoS



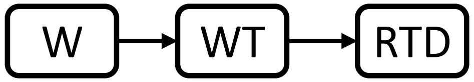
```
Subsystems:
  R1: Int1:R1.Msg
   T: Int1:T.In; Int2:T.Out
   D: Int1:D.In
Connections:
  R1:Int1:R1.Msg->T:Int1:T.In
   T:Int2:T.Out->D:Int1:D.In
```

```
RTD: Single Config SSS
 Subsystems: R1; T; D
 Internal Connections:
  R1:Int1:R1.Msg->T:Int1:T.In
  T:Int2:T.Out->D:Int1:D.In
 External Interfaces:
  Int1:D.In <- D:Int1:D.In
```

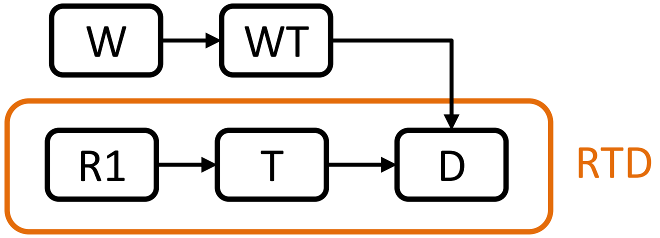


Composing a SSS with Other SSes

- Consider a Simple 3 SS SoS
  - WAMI feeds a WAMI Tracker
  - WAMI Tracker feeds an RTD
- DSE Will Solve for RTD, then Compose
  - Mitigate Sub-optimality by Maximizing Margin on SSes that Implemented SSS Interfaces (D in this case)
  - Assembles HCAL constraints on D from both RTD Composition and W-T-RTD Composition
  - Early prototype of real-world functionality to support SSes used in Multiple SoSes



```
Subsystems:
    W: Int1:W.Msg
   WT: Int1:WT.In; Int2:WT.Out
  RTD: Int1:D.In
Connections:
   W:Int1:W.Msg->WT:Int1:WT.In
  WT:Int2:WT.Out->RTD:Int1:D.In
```

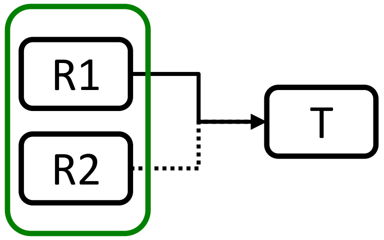


Handling Resilient Configurations is a State Dependent Transform

- STITCHES Uses Small Optimized Interfaces Tailored for Specific Configuration
  - Transforms remove information that the destination doesn’t need
  - Interfaces that aren’t used are blocked to reduce attack surface and minimize waste network traffic
- Traditional OSA Operates at the Other End of the Spectrum
  - Translate all SS interfaces/messages to global standard for open integration
  - Result is flexibility in real-time updates, but with sub-optimal performance
- Augment STITCHES to Generate HCALs with Configuration Switch Layers
  - Filter messages across the stacks and to/from SSes based on the determined Configuration State
  - Operate as a State Dependent Synchronizing Transform
  - Can Optimize the Sub-Stacks Based on the Specific State Configuration
- Note: STITCHES Does Not Reason Over What the State Should Be
  - This is a Hard Problem and Out of Scope of Our Effort
  - Configuration State is Managed by External Source (Mission Management Software, User, etc.) and Provided Via State Management Messages
  - Current Version Doesn’t Manage Transients Robustly

Resilient Configurations as SSS

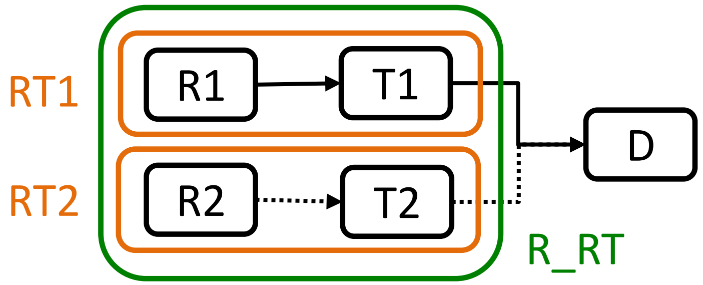
- Consider a Simple 3 SS Resilient SoS
  - Radar 1 (R1) feeds a Tracker
  - Radar 2 (R2) is a backup feed for Tracker
  - R1 & R2 interfaces don't need to be the same but must be "equivalent"
- Define a Resilient SoS SS (RC-SSS)
  - Multi-Mode, with a different SS "active" in each Mode
  - Define set of external interfaces
  - Define which SS implements the External Interfaces in Each Mode
- R_R Can Now Be Used as a SS in Other SoS Configurations
  - Which Radar is Active will Change with Mode Switch Message



```
Subsystems:
  R1: Int1:R1.Msg
  R2: Int1:R2.Msg
   T: Int1:T.In; Int2:T.Out

R_R: Resilient Config SSS
 Modes:
   Mode1: R1
   Mode2: R2
 External Interfaces:
   RROut:CatRROut
    Mode1Impl:R1:Int1:R1.Msg
    Mode2Impl:R2:Int1:R2.Msg

Connections:
 R_R:RROut:CatRROut->T:Int1:T.In
```

Resilient Config SSS Can Be Composed of Multiple Single Config SSSes

```
Subsystems:
  R1: Int1:R1.Msg
  R2: Int1:R2.Msg
  T1: Int1:T.In; Int2:T.Out
  T2: Int1:T.In; Int2:T.Out
   D: Int1:D.In;

RT1: Single Config SSS
 Subsystems: R1; T1
 Internal Connections:
  R1:Int1:R1.Msg->T1:Int1:T.In
 External Interfaces:
  Int1:T.Out <- T1:Int2:T.Out

RT2: Single Config SSS
 Subsystems: R2; T2
 Internal Connections:
  R2:Int1:R2.Msg->T2:Int1:T.In
 External Interfaces:
  Int1:T.Out <- T2:Int2:T.Out
```



```
R_RT: Resilient Config SSS
 Modes:
  Mode1: RT1
  Mode2: RT2
 External Interfaces:
  R_RTOut:CatRTOut
   Mode1Impl:Int1:T.Out
   Mode2Impl:Int1:T.Out

Connections:
 R_RT:R_RTOut:CatRTOut->D:Int1:D.In
```

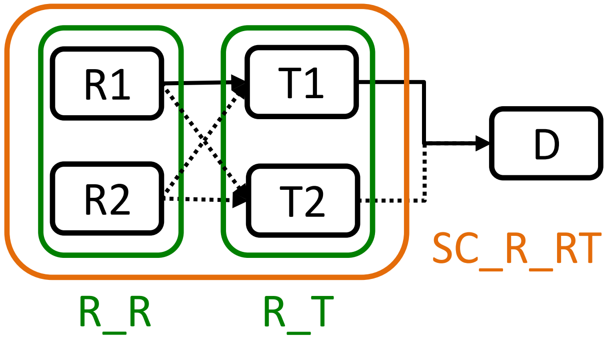
Mode Switch Messages Will Switch btw

Mode 1: R1->T1->D

Mode 2: R2->T2->D

Single Config SSS Can Be Composed from Multiple Resilient Config SSSes

```
Subsystems: (Same as Before)

R_R: Resilient Config SSS
 Modes:
  Mode1: R1
  Mode2: R2
 External Interfaces:
  RROut:CatRROut
   Mode1Impl:Int1:R1.Msg
   Mode2Impl:Int1:R2.Msg

R_T: Resilient Config SSS
 Modes:
  Mode1: T1
  Mode2: T2
 External Interfaces:
  RTIn:CatRTIn
   Mode1Impl: Int1:T.In
   Mode2Impl: Int1:T.In
  RTOut:CatRTOut
   Mode1Impl: Int2:T.Out
   Mode2Impl: Int2:T.Out
```



```
SC_R_RT: Single Config SSS
 Subsystems: R_R; R_T
 Internal Connections:
  R_R:RROut:CatRROut->R_T:RTIn:CatRTIN
 External Interfaces:
  Int1:CatRTOut <- R_T:Int2:CatRTOut

Connections:
 SC_R_RT:Int1:CatRTOut->D:Int1:D.In
```

Mode Switch Messages Will Switch btw

Mode 11: R1->T1->D; Mode 12: R1->T2->D

Mode 21: R2->T1->D; Mode 22: R2->T2->D

Demonstrated Capabilities

- Global Interoperability without Global Consensus on Interface Specification
  - Stateless Interactions (Message Transformations)
  - Stateful Interactions (Multiple Source Messages Required to Form Destination Message)
- Efficient Reuse in and Evolution of the Architecture
- Near Real-Time Construction of the SoS from Specification
- Optimized Implementation of Interfaces that are Small and Fast
  - Support for High Speed Packed Representations
- Allow Legacy Subsystems and Existing Open Architectures to Interoperate
- Cyber Defenses via Heterogeneity & Run-Time Execution Monitors
- Hierarchical & Resilient SoS Configurations that simplify complexity of large SoSes