The concept of a deeply-embedded domain specific language for modeling robotic systems in C++.

Koncepcja domenowego języka modelowania systemów robotycznych głęboko wbudowanego w język C++.

Tomasz Gawron

Katedra Sterowania i Inżynierii Systemów
Politechnika Poznańska

08.06.2016

Agenda

1. Introduction
2. Important properties of a software development environment
3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
4. A review of selected state of the art robotics/control middleware

Agenda

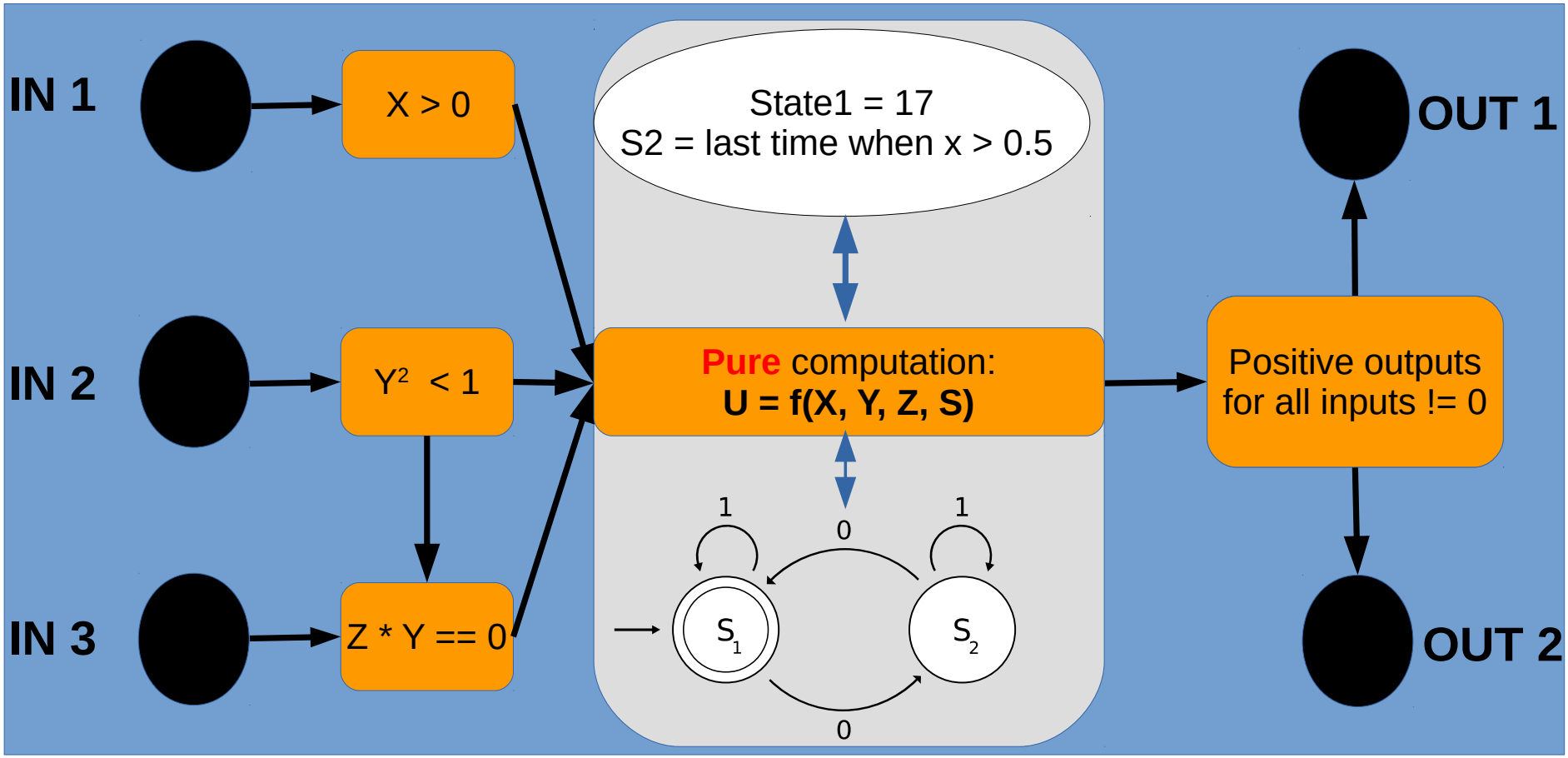
The long-term[1] goal of our current software development efforts

To provide a software development environment for robotic systems,
which shall be able to:

1. compose software systems using signal specifications and software components,
2. automatically generate proofs of their correctness and performance characteristics,
3. assist in development by inferring intermediate component specifications from source code,
4. intelligently generate repeatable code (the so-called boilerplate) required for operation.

[1] The short-term needs are currently still fulfilled by KSIS Framework.

The conceptual description of a component - visualization



The conceptual description of a component

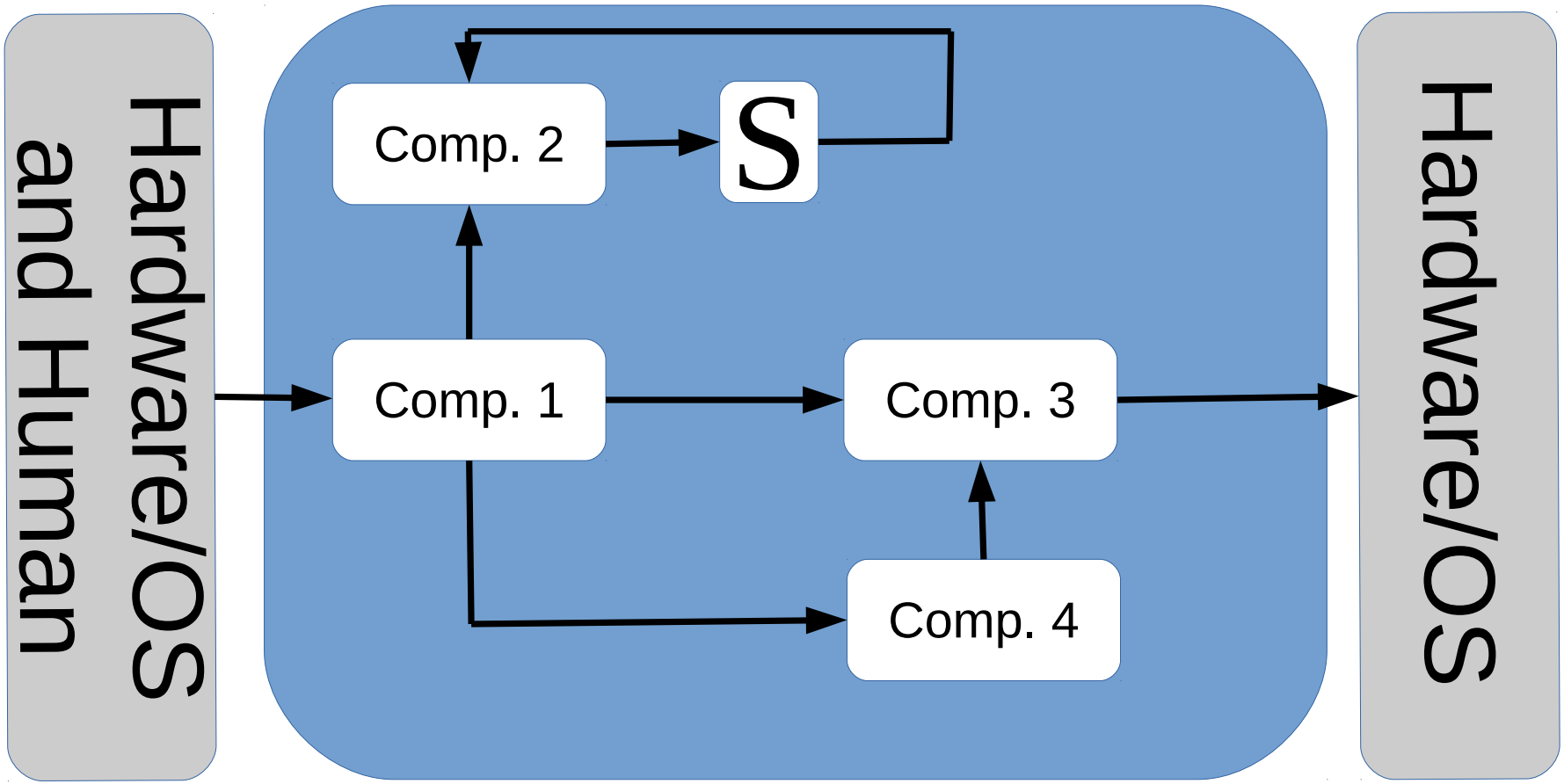
Component represents a thematically consistent transformation of input signals, which utilizes internal

- input *specification*,
- output *specification*,
- persistent state endowed with *invariants*,
- and possibly a (nested) *finite state automaton*.

Time can only be an input signal.
Other than that, a component is oblivious to the notion of time.

Signal *names* are irrelevant, composition is actually driven by their *types*, i.e. metadescription of the data they carry.

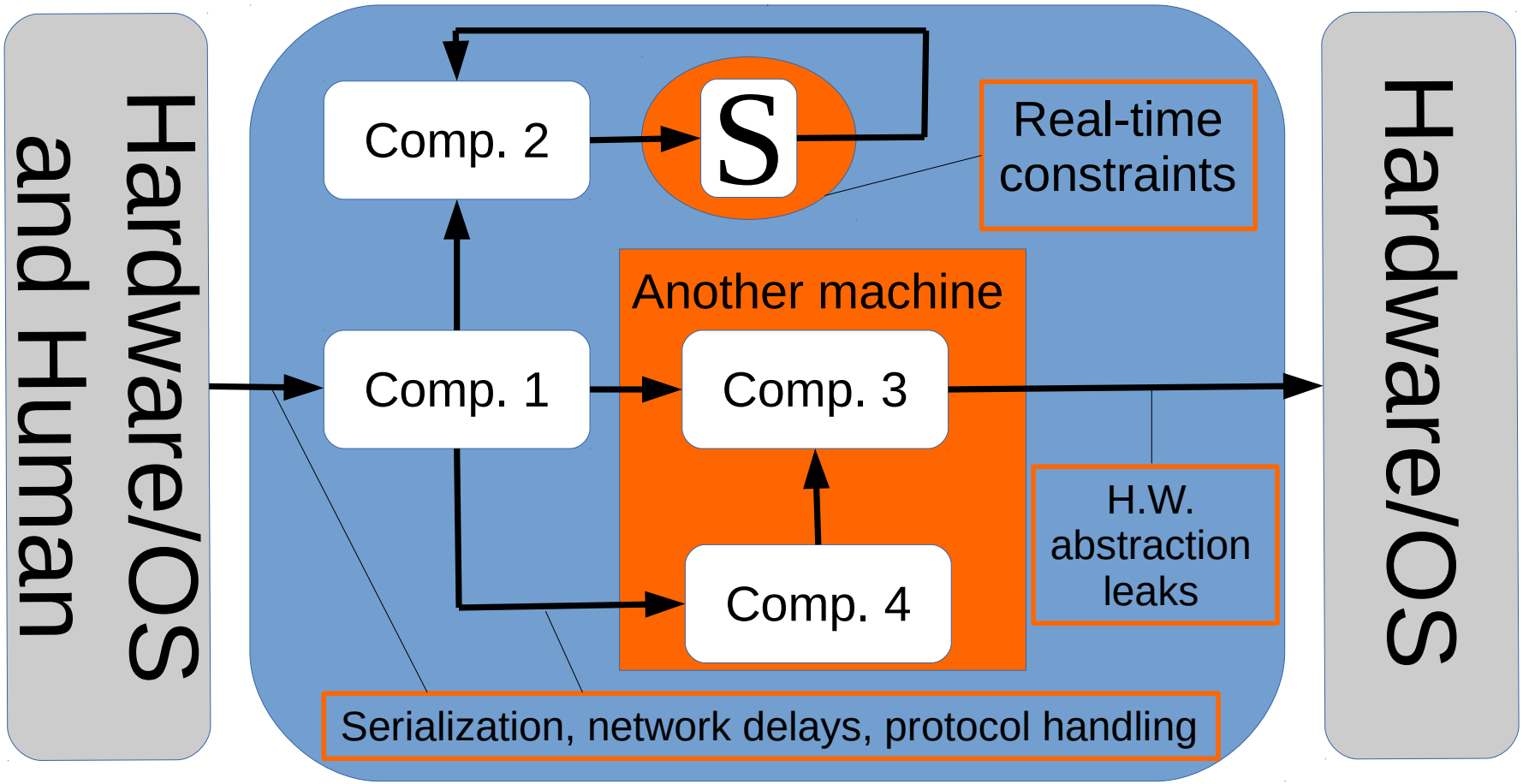
Software system as a composition of components - theory



Who determines connections? Can we prove that the system works?

Why this structure is lost at the implementation stage?

Software system in practice - the cross-cutting concerns



How can we predict time delays? How do we connect machines and processes?
Which components reside where? How do we separate logic and hardware?

Agenda

1. Introduction
2. Important properties of a software development environment
3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
4. A review of selected state of the art robotics/control middleware
   - Imperative programming approaches
   - Functional programming approaches

Modularity and scalability

- Are components truly separated? Is this separation forced?

Modularity and scalability

- Are components truly separated? Is this separation forced?
- Can the environment handle and *efficiently deploy* hundreds of components?

Modularity and scalability

- Are components truly separated? Is this separation forced?
- Can the environment handle and *efficiently deploy* hundreds of components?
- Do the component connection rules/scripts require changes when new components or abstractions are added?

Modularity and scalability

- Are components truly separated? Is this separation forced?
- Can the environment handle and *efficiently deploy* hundreds of components?
- Do the component connection rules/scripts require changes when new components or abstractions are added?
- Can we verify that system reconfiguration due to new components did not break its functionality?

Powerful abstractions

- Can we conveniently define and leverage component interfaces?
  e.g. Controller, Motor Interface, Motion Primitive

Powerful abstractions

- Can we conveniently define and leverage component interfaces?
  e.g. Controller, Motor Interface, Motion Primitive

- Can the system dynamically change depending on available *information* ?
  e.g. path generator adapting to the change of a path following controller

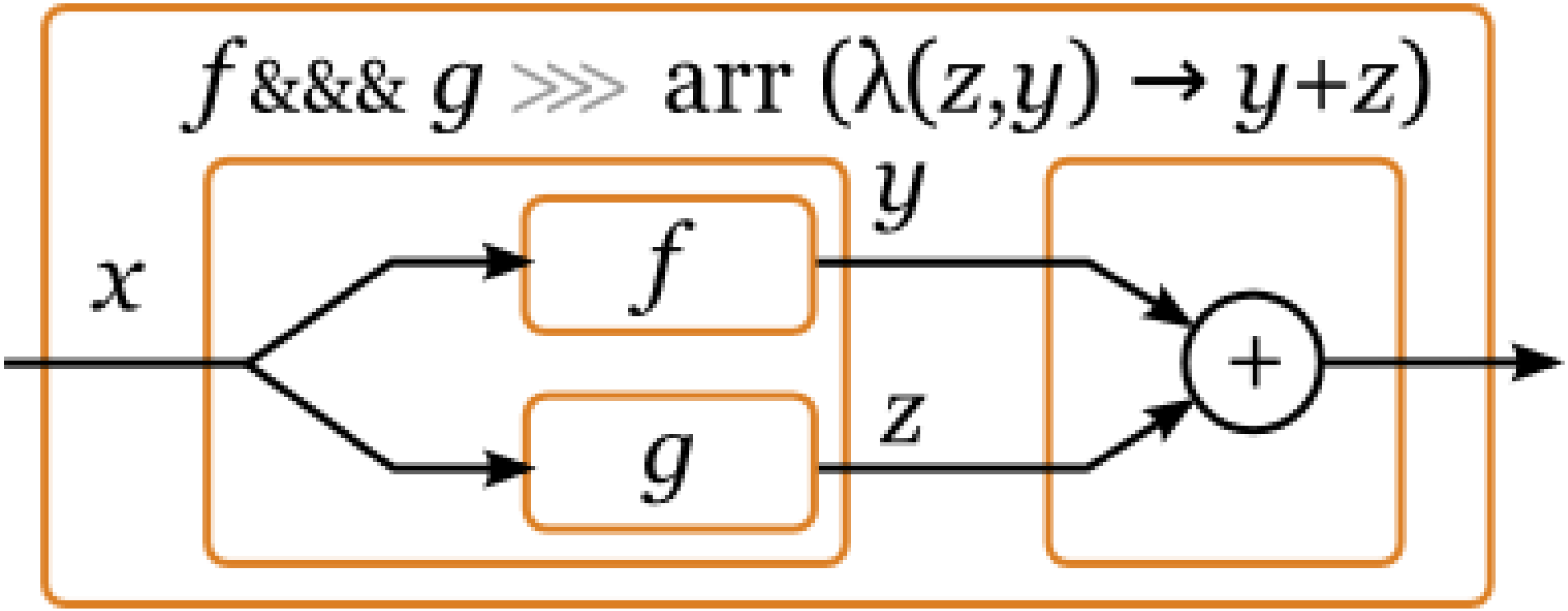
Powerful abstractions

- Can we conveniently define and leverage component interfaces?
  e.g. Controller, Motor Interface, Motion Primitive
- Can the system dynamically change depending on available *information* ?
  e.g. path generator adapting to the change of a path following controller
- Can we abstract component connection patterns?
  higher-order component functions such as feedback, multiplexing, automatons

Powerful abstractions

- Can we conveniently define and leverage component interfaces?
  e.g. Controller, Motor Interface, Motion Primitive
- Can the system dynamically change depending on available *information* ?
  e.g. path generator adapting to the change of a path following controller
- Can we abstract component connection patterns?
  higher-order component functions such as feedback, multiplexing, automatons
- Can we infer component interfaces from the structure of signals?
  the so-called *structural typing*

Powerful abstraction example - Haskell arrows[2]



[1] Haskell is a lazy purely-functional programming language. Image is taken from HaskellWiki article - "Understanding Arrows".

Ease of use, brevity and uniqueness of definitions

- As a programmer, do I have to repeat myself often, because of my tools?
  e.g. C header files, build configuration files, ROS package definitions

Ease of use, brevity and uniqueness of definitions

- As a programmer, do I have to repeat myself often, because of my tools?
  e.g. C header files, build configuration files, ROS package definitions

- Are there any useful default conventions so that we can stop making design decisions and focus on the *domain-specific* problem?

Ease of use, brevity and uniqueness of definitions

- As a programmer, do I have to repeat myself often, because of my tools?
  e.g. C header files, build configuration files, ROS package definitions
- Are there any useful default conventions so that we can stop making design decisions and focus on the *domain-specific* problem?
- As a system designer, can I observe how recent changes influenced system structures and where are the most mission-critical components?

Ease of use, brevity and uniqueness of definitions

- As a programmer, do I have to repeat myself often, because of my tools?
  e.g. C header files, build configuration files, ROS package definitions
- Are there any useful default conventions so that we can stop making design decisions and focus on the *domain-specific* problem?
- As a system designer, can I observe how recent changes influenced system structures and where are the most mission-critical components?
- Is the development environment documented with text and working examples? Are all the error messages clear, deterministic and documented?

Ease of use, brevity and uniqueness of definitions

- As a programmer, do I have to repeat myself often, because of my tools?
  e.g. C header files, build configuration files, ROS package definitions
- Are there any useful default conventions so that we can stop making design decisions and focus on the *domain-specific* problem?
- As a system designer, can I observe how recent changes influenced system structures and where are the most mission-critical components?
- Is the development environment documented with text and working examples? Are all the error messages clear, deterministic and documented?
- How long does compilation and deployment take?

Correctness certificates and real-time performance guarantees

- Is the system correct by construction? Can the environment verify desired properties of my components?

Correctness certificates and real-time performance guarantees

- Is the system correct by construction? Can the environment verify desired properties of my components?
- Does the environment force real-time safe coding, where necessary?
  Real-time safe communication and scheduling is not enough!

Correctness certificates and real-time performance guarantees

- Is the system correct by construction? Can the environment verify desired properties of my components?
- Does the environment force real-time safe coding, where necessary?
  Real-time safe communication and scheduling is not enough!
- Does the environment infer temporal, spatial and logical coupling between components to generate system implementation with optimal performance?

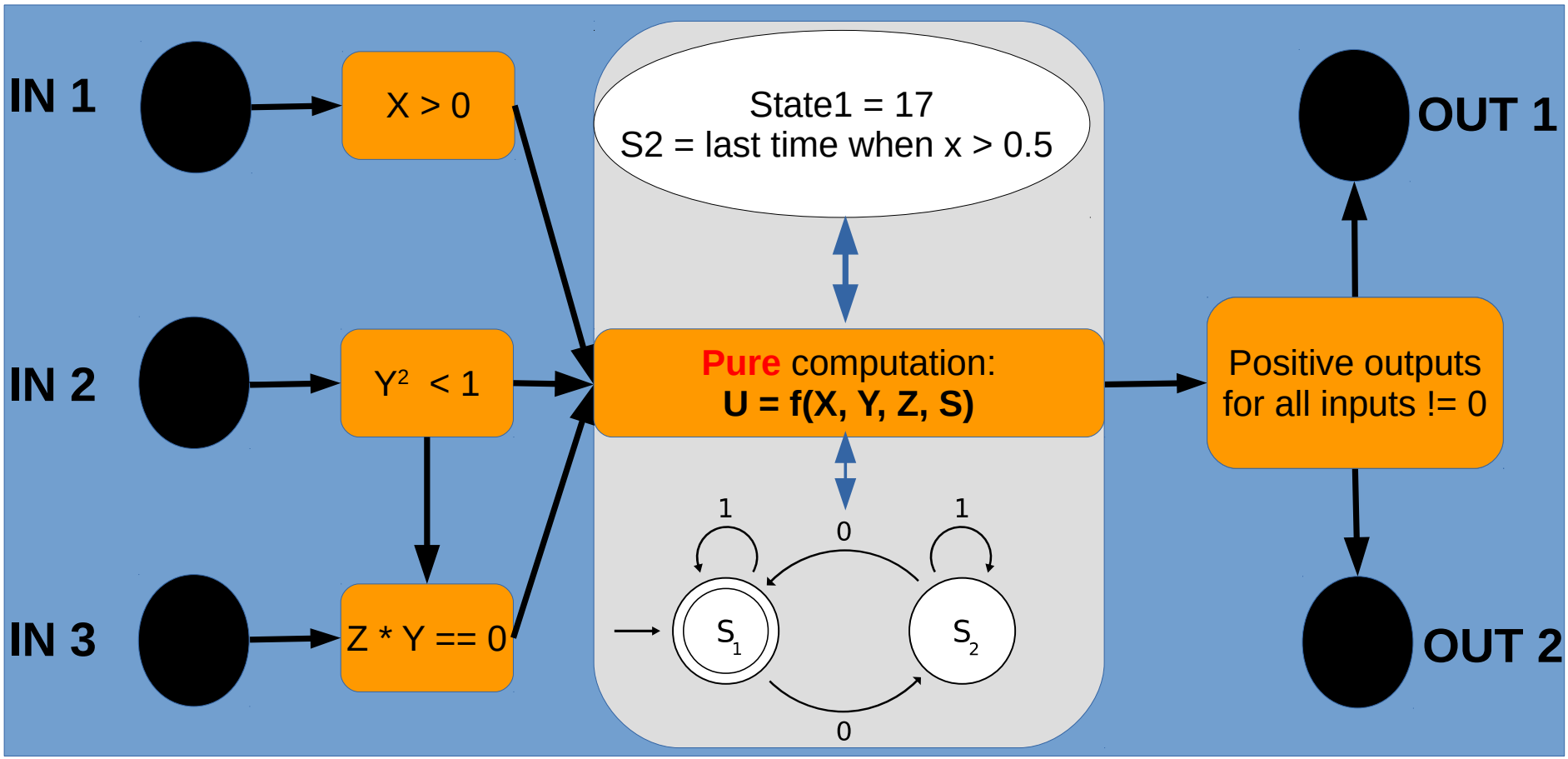
Agenda

1. Introduction
2. Important properties of a software development environment
3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
4. A review of selected state of the art robotics/control middleware
   - Imperative programming approaches
   - Functional programming approaches

Agenda

3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
   - Imperative programming approaches
   - Functional programming approaches

Component definition - concept



The developer just defines the above diagram as a C++ function...

Component definition syntax - whole definition is contained in one file

```
struct PlatformVelocity //an automatically exported and serialized signal
{
        Velocity V; //corresponds to Out1
        AngularVelocity Omega; // corresponds to Out2
};
//Strong typing, automatic SI units tracking and angle conversion
struct Pose2D
{       ContinuousAngle Theta; Coordinate X, Y; };
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guaranteed
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints
        //standard C/C++ output implementation
        o->Omega = -k_a*pose->Theta
}
```

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices
- Signals can be used like ordinary C variables and can be defined by C structures

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices
- Signals can be used like ordinary C variables and can be defined by C structures
- Invoking additional methods on signals creates their specifications

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices
- Signals can be used like ordinary C variables and can be defined by C structures
- Invoking additional methods on signals creates their specifications
- Signal types (in C++ sense) + specifications form a so called *liquid type* system,
  which allows us to prove program correctness at the compilation stage

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices
- Signals can be used like ordinary C variables and can be defined by C structures
- Invoking additional methods on signals creates their specifications
- Signal types (in C++ sense) + specifications form a so called *liquid type* system, which allows us to prove program correctness at the compilation stage
- Specifications form a contract determining possible component connections

Component definition syntax - comments

- Component is a C++ function, which is *pure* and recomputed on input change
  (i.e. output dependends only on function arguments)
- ASYCOS is designed to support embedded devices
- Signals can be used like ordinary C variables and can be defined by C structures
- Invoking additional methods on signals creates their specifications
- Signal types (in C++ sense) + specifications form a so called *liquid type* system,
  which allows us to prove program correctness at the compilation stage
- Specifications form a contract determining possible component connections
- Implementation failing to meet specifications result in a compilation error,
  because ASYCOS actually takes over the compilation process,

hence it is *deeply embedded in C++*

The power of deep embedding - the following will not compile

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        static int counter = pose->X;

        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure.HardRealTime(0.01, Critical); //r.t. constraints
        //standard C/C++ output implementation
        o->Omega = -k_a*pose->Theta
}
```

Error: Static declaration of counter from line x col y violates purity of Controller.

The power of deep embedding 2

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        int* buffer = new int[23];

        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints
        //standard C/C++ output implementation
        o->Omega = -k_a*pose->Theta;
}
```

Error: Heap allocation from line x col y violates HardRealTime spec of Controller.

The power of deep embedding 3

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints
        //standard C/C++ output implementation
        o->Omega = -k_a*pose->Theta

        o.NoDefaults();
        //oops forgot to set o->V
}
```

Error: V is not always assigned in Controller - NoDefaults spec violated.

Such specifications can be also forced globally.

The power of deep embedding 4

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints
        //standard C/C++ output implementation

        //a long-running planning process violating r.t. constraints
        auto res = ompl::RRT::Plan(pose);
        o->Omega = -k_a*pose->Theta + res;
}
```

Error: Controller halting not verified due to function call from line x col y.

The power of deep embedding 5

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints

        int i = 10;
        while(i > 0)
        {
                i = i - 1;
                if(i == 0) i = 1;
        }
}
```

Error: Controller halting not verified due to execution cycle originating from line x col y.

Unfortunately, sometimes ASYCOS needs help - halting problem is undecidable.

The power of deep embedding 6

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose.AlwaysAvailable(); //input updates guarantees
        o.Pure().HardRealTime(0.01, Critical); //r.t. constraints

        o.Predicate(L->V > 0); //L is for lambda expr. (optional)
        o->V = sqrt(square(L->X) + square(L->Y));
}
```

Error: Controller output predicate from line x col y was not verified.

Unfortunately, currently this works in a very controlled environment due to *theoretical* obstacles.

The power of deep embedding 7

```
void Controller(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PlatformVelocity> o, //a SINGLE output
        Config<float> k_a, //configuration parameters (any POD C++ type)
        )
{
        //verifiable signal specifications
        pose->AlwaysAvailable(); //input updates guarantees
        o->Pure()->HardRealTime(0.01, Critical); //r.t. constraints

        pose.Predicate(L->X > 0 || L->Y > 0);
        o.Predicate(L->V > 0); //L is for lambda expr. (optional)
        o->V = sqrt(square(L->X) + square(L->Y));
}
```

No error during component compilation, the input predicate will be a *constraint for system composition*.

System composition

```
int main(int argc, char *argv[]) {
        Application app(argc, argv, "demo application");
        //Intra-subsystem component compositions will be preferred
        Subsystem sys1, sys2; //Application can be split into subsystems
        //Components can be added using namespaces
        sys1.AddNamespace("ksis", !L->Pure());
        //Components can be added using basic predicates
        sys2.AddNamespace("ksis", L->IsHardRealTime());
        app.AddSystems(sys1, sys2);
        //Components can be added using relative paths
        app.AddComponents("/controllers/*");
        //Choosing main signals to guide system composition
        app.Output<WheelVelocities>(L->HardRealTime(0.01, Critical));
        //An exemplary ROS interop. signal driving the composition
        app.Input<ros:sensor_msgs:LaserScan>(L->ExternalName("/scan"));
        return app.Run(); //Initialize and run to completion
}
```

The system designer chooses groups of modules to be *intelligently composed* by ASYCOS according to system and component specifications.

System composition - developer feedback

```
int main(int argc, char *argv[]) {
        Application app(argc, argv, "demo application");
        app.AddComponents("/VFO/*", "/TF/*", "Core/*");
        //Fixing the composition error
        app.NeverConnect(VFOController, SkidCompensator);
        return app.Run(); //Initialize and run to completion
}
```

The above code fixes the following error:
Ambiguous composition: VFOController -> SkidCompensator,
SkidCompensator requires unique input.

Component sets yielding ambiguous compositions (i.e. contradictions) give errors.

System designer *iteratively inspects the graph of composed sytem*
and *adds composition hints*,
hence designer-ASYCOS feedback is formed.

System composition - spatial coupling

```
int main(int argc, char *argv[]) {
        Application app(argc, argv, "demo application");
        //Multiple machines configured in separate files
        Machine robot("robot"), remote("remote"), sim("sim");
        Subsystem sys1, sys2;
        sys1.PinToMachine(remote);
        //Soft-sync. replication useful for debugging
        sys2.PinToMachine(robot).ReplicateOn(sim);
        //Specs work across systems and machines, but can be local

        //Form an ad-hoc (spatial) subsystem in a new thread
        sys2.Spec<VFOController>(L->SeparateThread());
        //Totally separate a component in a new process
        app.Spec<Visualizer>(L->ExclusiveProcess());
        return app.Run(); //Initialize and run to completion
}
```

ASYCOS is designed to attempt *optimal task scheduling and spatial distribution* given its component knowledge.

The system designer can assist ASYCOS as shown above.

ASYCOS workflow - a programmer's perspective

1. Run ASYCOS daemon process in project directory
   (automatic CMake support will be ported from KSIS Framework)
2. Add files and edit components code in Your favorite IDE/Editor
   (Seamless IDE integration by creation of a fake compiler executable is possible, but not planned)
3. Inspect ASYCOS component compilation errors and go to 2.
4. Edit Your system composition hints in native C++
5. Inspect ASYCOS composition errors and go to 2. or 4.

ASYCOS is designed to work continuously, it is also prepared for incorrect code.

The idea is to rapidly prototype (i.e. implement, simulate, visualize and deploy) algorithms and drivers directly using ASYCOS[3].

[3] Obviously, some online communication with MATLAB can be still useful. It will be supported in later stages of the project.

No boilerplate code and no build configuration

1. The programmer implements only component function definitions and local signal definitions in .cpp files

2. ASYCOS auto-generates appropriate header files, serialization code, synchronization code, etc.

3. When components explicitly reference each other's signals, they include those auto-generated headers

Every directory of .cpp files lands in its own static library.

CMake scripting is only required to add external libraries.

Agenda

3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
   - Imperative programming approaches
   - Functional programming approaches

Finite state machines for free

```
void ControllerS1(
        In<Pose2D> pose, //one or more inputs (any C++ type)
        Out<PV> o, //a SINGLE output
        State<SData1> s //SData1 and SData2 are any POD C++ types
        )
{ /*Control law for state 1*/ }
//A transition function
State<SData2> S1ToS2(In<Pose2D> pose, Out<PV> o, State<SData1> s)
{
        pose.Predicate(L->x > 3);
        State<SData2> newState; //Setup a new state
        newState->someData = s->X;
        return newState;
}
void ControllerS2(In<Pose2D> pose, Out<PV> o, State<SData2> s)
{ /*Control law for state 2*/ }
```

- Defined in a decoupled manner, to allow for extensibility and nesting
- A natural consequence of our component system
- Analogously, one can define complex, real-time safe component switching patterns

Arrow-like component combinators and initialization context

```
PlatformVelocity PureFunction(const Pose2D& pose)
{ /* implementation */ }

//Every .cpp file can contain init function
//used to generate components available for composition
void Init(Subsystem& sys)
{
        //In this file, component functions will not be detected
        sys.NoAutoDetect();

        //Lift a pure function to component
        sys.AddComponent( Lift(PureFunction).Spec.HardRealTime(0.01));

        //Make a component that is invoked only once
        sys.AddComponent(OneShot(AComponent));
}
```

- Combinator is a function of components abstracting connection patterns
- Combinators can be used both in composition and component definition
- Other useful combinators: Random, FirstAvailable, Differentiate (symbolic)

On-demand structural typing and metadata access

```
struct Pose3D
{ Coordinate X, Y, Z; ContinuousAngle Theta; }

void Controller(In<Pose2D> pose, Out<PV> o)
{
        //Matches any type containing
        //the same members as Pose2D
        pose.StructuralMatching(false);
        //Setting true in the above
        //matches strictly the same set of members

        //One can also access true signal metadata
        //in non real-time code
        auto name = pose.Members()[0].Name();
        /* Implementation */
}
```

- Pose3D will be auto-converted to Pose2D
- Useful for combinators and generic transformations (i.e. showing the name of a signal)

Instruction budgeting with automatic compile-time verification

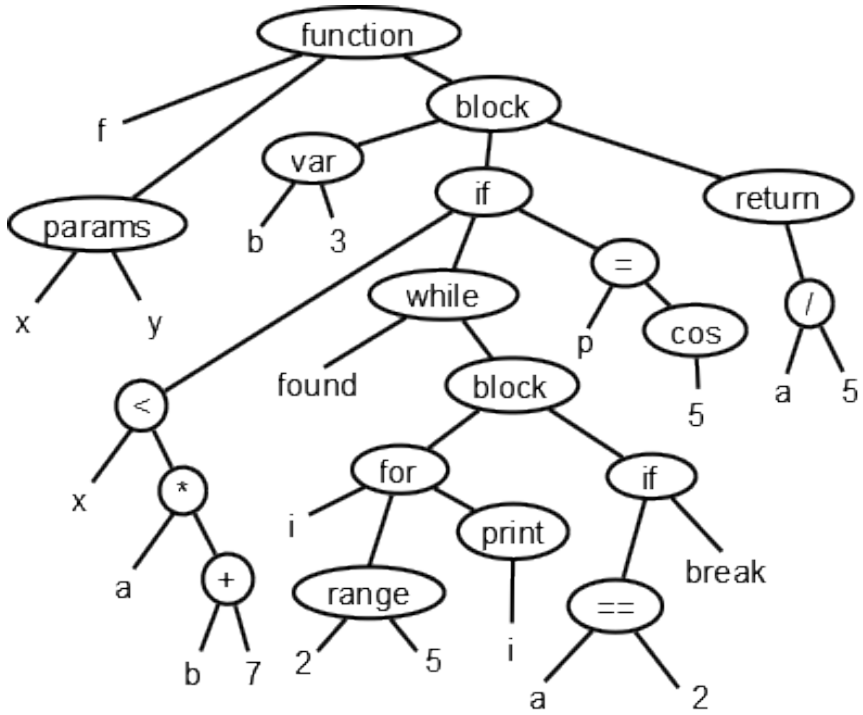
```
void Controller(In<Pose2D> pose, Out<PV> o)
{
        out.InstructionBudget(100);
}

//Instruction cost definitions are extensible like so
void Init(Subsystem& sys)
{
        sys.InstructionCost(sin, 30);
}
```

- The above component will fail to compile if its implementation takes more than 100 instructions
- One can use ”MaxCyclomaticComplexity” to constrain the number of execution paths in component
- To avoid critical dependencies on uncertain components use ”MaxCriticalityCoefficient”, which corresponds to number of connections the component participates in

Agenda

3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
   - Imperative programming approaches
   - Functional programming approaches

How does ASYCOS reason about code?



Code generation and verification is performed in terms of lazy graph processing algorithms.

Initially source code is viewed as a n-ary tree
with even-level nodes being operations and odd-level nodes being their arguments.

The compilation process

- We select appropriate .cpp files and invoke Clang parser

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations
- Final component graph of the system is generated and presented to the user

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations
- Final component graph of the system is generated and presented to the user
- C++ code is generated in memory

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations
- Final component graph of the system is generated and presented to the user
- C++ code is generated in memory
- We invoke a second pass of Clang parser and include generated code

Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations
- Final component graph of the system is generated and presented to the user
- C++ code is generated in memory
- We invoke a second pass of Clang parser and include generated code
- Later compilation stages (optimization passes, assembly generation) ensue

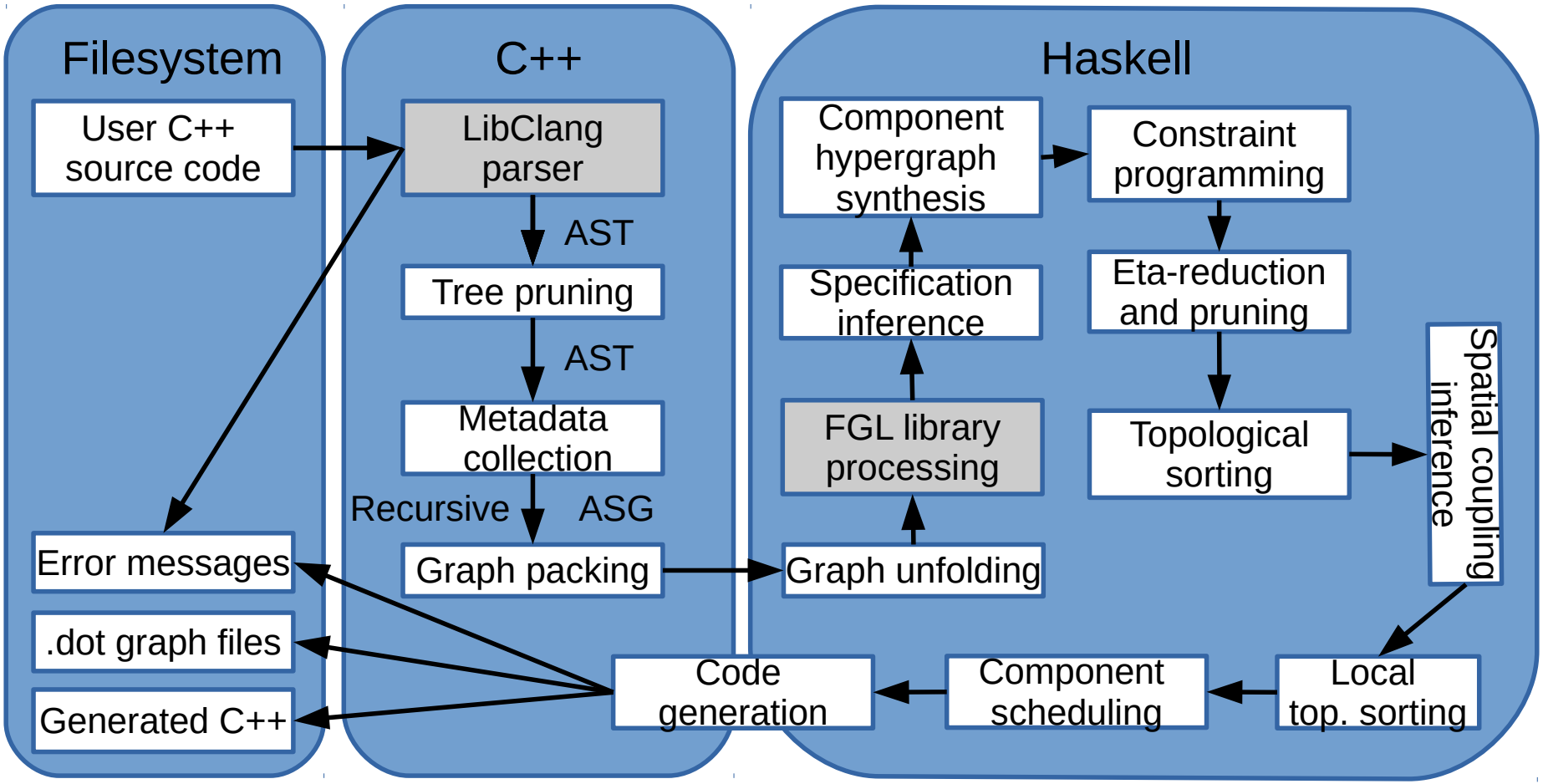
Some of the above steps happen in concurrently.

The compilation process

- We select appropriate .cpp files and invoke Clang parser
- We explore abstract syntax trees generated by Clang
- Relevant trees are searched for metadata (e.g., function names and types)
- Information recovered from the trees is packed in Haskell-readable format
- We operate using the concept of infinitely unfolding graphs of operations
- Final component graph of the system is generated and presented to the user
- C++ code is generated in memory
- We invoke a second pass of Clang parser and include generated code
- Later compilation stages (optimization passes, assembly generation) ensue
- Error messages are collected, augmented with ASYCOS errors and presented to the user

Some of the above steps happen in concurrently.

Internals of ASYCOS



Current implementation is less involved than this full implementation schematic[4].

[4] AST - abstract syntax tree, ASG - a nested graph of ASTs

Agenda

3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
   - Imperative programming approaches
   - Functional programming approaches

Current stage of the project - proof of concept

1. Done: API design (but we are open to suggestions)
2. Done: High-level algorithm design
3. Current: Efficient AST conversion and transfer to Haskell
4. Current: Generalization of AST transformations in Haskell
5. Current: Correct handling of all the above examples (minor glitches)
6. Future: Extensive testing
7. Future: QT-based GUI
8. Future: Proper Z3 SMT solver integration for constraint programming
9. Future: A small exemplary system implementation
10. Future: KSIS Framework porting

ETA of a limited prototype for KSIS usage: end of July
ETA of a first complete (public) version: end of year 2016

We did enough to prove, that a fully-fledged implementation is possible.
However, a lot of major technical issues must be solved to make ASYCOS usable.

Current development team members

KSIS members[5]:

- T. Gawron
  (concept, API design, design of the algorithms, all of the current source code)
- D. Pazderski
  (API design feedback, feature requests)

Student members (A. and R., 6th semester):

- P. Gała
  (learning internals of Haskell and compiler infrastructure, future focus: AST processing)
- J. Wojtkowiak
  (learning Haskell and HsQML, future focus: responsive GUI for ASYCOS users)
- A. Bykowski
  (learning high-level design, future focus: synchronization and coupling inference)

[5] We'd also like to thank M. Michałek, M. Michalski and T. Jedwabny for discussions and observations relevant to this work.

We are very open to suggestions and discussion of our opinions

---

Thank You

Agenda

1. Introduction
2. Important properties of a software development environment
3. ASYCOS - toolkit for Automated SYnthesis of COntrol Software
   - Our development philosophy and design decisions
   - Examples demonstrating interesting features of ASYCOS
   - A sketch of the implementation
   - Current status of the implementation
4. A review of selected state of the art robotics/control middleware
   - Imperative programming approaches
   - Functional programming approaches

Agenda

- Our development philosophy and design decisions
- Examples demonstrating interesting features of ASYCOS
- A sketch of the implementation
- Current status of the implementation

4. A review of selected state of the art robotics/control middleware
   - Imperative programming approaches
   - Functional programming approaches

ROS - Robot Operating System

Component representation: a process (node)

- Defacto standard - well documented and maintained
- Network communication and message routing for free
- Lots of drivers and library interfaces
- Command line tools and visualization support

- High syntactic noise and repeated definitions
- No component synchronization
- Slow compilation and problematic deployment
- Unverifiable, XML-based configuration
- Fragile parameter system
- Helps only at the interprocess scope
- No real-time perf. and slow communication

```
std_msgs::String input;
void inputCB(const std_msgs::String::ConstPtr& msg)
{
        input.data = msg.data;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "component");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = \\
        n.advertise<std_msgs::String>("output", 1000);
  ros::Subscriber sub = n.subscribe("in", 1000, inputCB);
  ros::Rate loop_rate(100);

  while (ros::ok())
  {

    //component computes output
    //using the global input variable
    std_msgs::String output;
    output.data = "Computation result...";

    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }

  return 0;
}
```

OROCOS - Open Robot Control Software

Component representation: a thread (object)

- Real-time performance if *used correctly* with Xenomai and low latency messaging
- Limited verification of composition correctness
- Component synchronization and triggering
- Finite state machines support

- Relatively small community
- Complicated installation and deployment
- Syntactic noise worse than ROS
- Relatively few supporting tools
- Often component = Xenomai thread, leading to scheduling overhead
- ROS integration is not scalable
- Forces object oriented programming

```
class Controller : public RTT::TaskContext {
protected:
        double Kp;
        OutputPort<double> steer;
        InputPort<double> target;
        InputPort<double> sense;
public:
        Controller(const std::string& name)
        : TaskContext(name, PreOperational),
        Kp(50.0)
        {
                //practical, long port specs. omitted
                provides()->addProperty("Kp", Kp);
                provides()->addPort( "Steer", steer );
                provides()->addPort( "Target", target );
                provides()->addPort( "Sense", sense );
        }
        bool configureHook()
        {
                return Kp > 0.0;
        }
        void updateHook()
        {
        //component computes output here
        }

//rest of the setup code omitted
```

KSIS Framework

Component representation: a module (execution fiber)

- Zero-copy data flow and deterministic computation scheduling
- Automatic system composition with user hints
- Minimal syntactic noise (no config files)
- Low-overhead nested finite state machines and voluntary preemption mechanism
- Components can suspend execution of everything connected to their outputs
- One command to install, two commands to deploy
- Seamless integration with KSIS robots and ROS

- Community of < 10 people and 1 maintainer
- Abysmal documentation
- Slow compilation and debugging complexity

```
POLY_MODULE(Demo, Controller)()
    {
        UpdatePeriodically(100);
        GoToRunning();
    }

    CONST(real, gain, 2);

    OUT(real, output);
    IN(real, input);

    STATE(Running)
    {
                out() = in()*gain();
                if(in() > 10)
                        GoToStopping();
    }

    NSTATE(Stopping, Running)
    {
        out() = 0;
        Propagate();
    }
END
```

MIT Drake and Simulink

Component representation: a system (Matlab object)

- Full power of Matlab always available
  which might also encourage bad coding practices
- Mostly sound system composition with limited symbolic manipulation
- Good visualization and simulation support

- Syntactic noise due to Matlab type system limitations
- Small community
- Seemingly hard to add new algorithms
- Very hard deployment (Matlab instances on the robot, S-function dependency hell, etc.)
- No real-time guarantees, high memory overhead and space leaks
- Potential code generation problems inherited

```
classdef SimpleDTExample < DrakeSystem
  methods
    function obj = SimpleDTExample()
      % call the parent class constructor:
      obj = obj@DrakeSystem(...
          0, ... % number of continuous states
          1, ... % number of discrete states
          0, ... % number of inputs
          1, ... % number of outputs
          false, ... % because the output does not depend
                 on u
          true);  % because the update and output do not
                 depend on t
    end
    function xnext = update(obj,t,x,u)
      xnext = x^3;
    end
    function y=output(obj,t,x,u)
      y=x;
    end
  end
end
```

Agenda

- Our development philosophy and design decisions
- Examples demonstrating interesting features of ASYCOS
- A sketch of the implementation
- Current status of the implementation

4. A review of selected state of the art robotics/control middleware
   - Imperative programming approaches
   - Functional programming approaches

ROSHask and Frob

Component representation: an arrow (algebraic)

- Correct by construction thanks to the Functional-Reactive Programming concepts and arrow algebra
- Arrows are Simulink-like blocks and arrow combinators are block connection patterns
- The most powerful abstractions known to computer science (symbolic differentiation, dependent typing, etc.)
- Legendary brevity thanks to Haskell's friendliness to domain-specific language design

- Very steep learning curve
  (proper understating of Haskell can take years)
- No real-time guarantees due to Haskell internals
- Potential memory leaks - a known issue with FRP
- Haskell dev. environments are not mature

```
sayHello :: Topic IO S.String
sayHello = repeatM (fmap mkMsg getCurrentTime)
  where mkMsg = S.String . ("Hello world " ++) . show


showMsg :: S.String -> IO ()
showMsg = putStrLn . ("I heard " ++) . view S._data


main = runNode "controller"
       $  runHandler showMsg =<< subscribe "chatter"
       $ advertise "chatter" (topicRate 1 sayHello)
```

Galois Copilot

Component representation: behavior description

- Hard real-time guarantees and machine-generated correctness proofs
- Low level memory manipulations available
- Possible portability between h.w. platforms
- Potential zero-copy data flow through STM
  i.e. Software Transactional Memory
- Usable in an industrial setting (generates C)

- Very steep learning curve
- Focused on embedded platforms
- Programmer is very constrained
- No network support
- “High-level“ language yielding “assembler-like“ implementations in practice

```
{-
  "If the majority of the engine temperature probes
        exeeds 250 degrees, then the
  cooler is engaged and remains engaged until the
        majority of the engine
  temperature probes drop to 250 or below.  Otherwise,
        trigger an immediate
  shutdown of the engine."  -}

engineMonitor :: Spec
engineMonitor = do
  trigger "shutoff" (not ok) [arg maj]

  where
  vals     = [ externW8 "tmp_probe_0" two51
             , externW8 "tmp_probe_1" two51
             , externW8 "tmp_probe_2" zero]
  exceed   = map (> 250) vals
  maj      = majority exceed
  checkMaj = aMajority exceed maj
  ok       = alwaysBeen ((maj && checkMaj) ==> extern "
       cooler" cooler)

  two51  = Just $ [251, 251] P.++ repeat (250 :: Word8)
  zero   = Just $ repeat (0 :: Word8)
  cooler = Just $ [True, True] P.++ repeat False

engineExample :: IO ()
engineExample = interpret 10 engineMonitor
```

ScaPS - Scala Program Synthesis

Component representation: an assembler instruction

- In theory, an arbitrary control system can be generated automatically by means of genetic programming
- Real-time guarantees and correctness by construction
- Almost no implementation effort required
- Developed actively in Poland by prof. K. Krawiec but not exactly for the purposes of robotics

---

- One must understand both functional programming in Scala and basics of genetic program synthesis
- In practice a program of length > 200 instructions can be generated for years
- The developer has no way to help the generator and utilize domain-specififc knowledge
- If generation fails, most of the time we will not know why
- ScaPS community might be even smaller than KSIS Framework community

Comparison matrix (subj. scale $1 - 10$)

	Scalability	Modularity	Abstractions	Ease of use	Brevity	Dev. feedback	Correctness certs.	RT and performance	Deployment speed	Cont. integration
ROS	6	7	2	7	2	5	1	3	6	3
OROCOS	7	7	3	4	4	6	2	3	5	3
KSIS Framework	8	8	5	6	9	4	3	6	4	4
MIT Drake	6	8	5	4	7	6	4	2	2	3
Simulink	5	6	4	8	6	6	5	3	2	1
Xenomai/QNX	8	6	2	6	7	3	1	6	5	6
FreeRTOS	4	5	3	5	5	3	1	6	4	2
DSP BIOS	3	5	3	5	6	3	1	7	4	2
ROSHask	7	7	6	6	8	6	3	2	4	3
Frob	8	7	7	5	7	7	4	2	6	3
Copilot	5	4	6	2	7	6	9	8	3	7
ScaPS	2	1	2	9	10	1	8	5	1	1

Conclusions from the comparison matrix

- There is no clear winner, when all of the properties are considered
- Continuous integration, which is essential for rapid prototyping is mostly non-existent
- Purely-functional programming tools lack real-time performance and tend to waste memory
- The lack of structure and abstractions in conventional solutions hinders derivation of both automatic and manual correctness proofs
- In many cases ease of use decreases when one moves to purely-functional solutions, which are otherwise very attractive
- Current solutions focus *either* on embedded or PC platforms