System-Specific Static Code Analyzes for Complex Embedded Systems

Holger M. Kienle, Johan Kraft and Thomas Nolte
Mälardalen University, Västerås, Sweden
{holger.kienle, johan.kraft, thomas.nolte}@mdh.se

SQM 2010, Madrid, Spain
March 15, 2010

PROGRESS
A national Swedish Strategic Research Centre

MÄLARDALEN UNIVERSITY

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

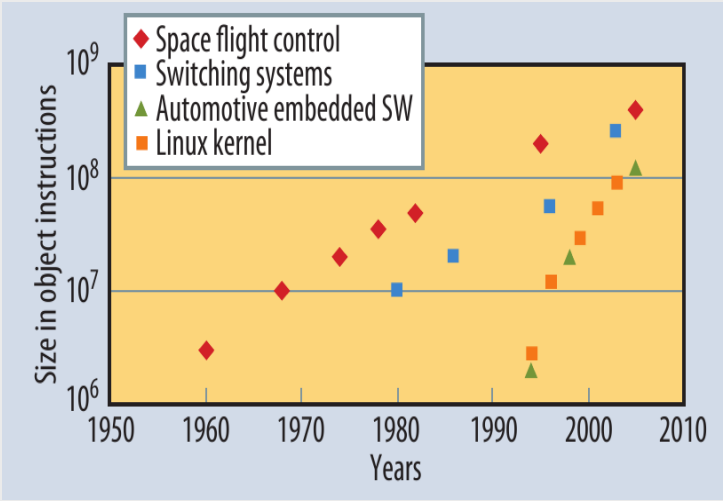
Swedish Foundation for Strategic Research

Introduction

- System-Specific Static Code Analyzes
  - Static analysis tailored to a specific system
  - Leverage from system-specific info
    - Architecture, naming conventions...
  - Complement to generic code checkers
  - Potentially high quality impact
- Complex embedded systems
  - High dependability requirements
  - Highly dynamic behavior

Embedded Systems

- Mechatronic control systems
- Real-time constraints
- Typically C/C++
- Often small (< 100 KLOC)
- Challenges
  - Correctness depends on *timeliness*
  - Dependability requirements
  - Adhere to (safety) standards
    - Increasing pressure from governments



Complex Embedded Systems

- Large (often > 1 MLOC)
- Long-lived (> 10 years)
- Safety- and/or business-critical
- Flexible - Highly dynamic runtime behavior
  - Event-triggered tasks (threads)
  - Priority-based online scheduling
  - Timing is not known at design time
    - An emerging system property

Example – ABB Robotics IRC 5

Industrial Robotics Control System

- A very advanced control system
  - Accurate (0,3 mm) while fast (3 m/s)
  - Controls up to 6 robots in parallel
- Large – around 3 million lines of C/C++
- Complex – over 60 processes (tasks)
- Robot performance depends mainly on SW

ABB

PROGRESS

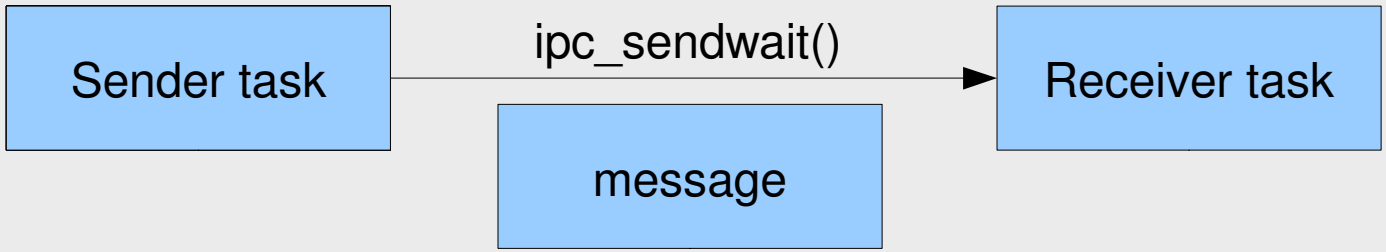
Experiences from ABB Robotics

- High software maintenance costs!
- Continuous feature growth and evolution over 15+ years gives legacy issues...
  - Increasing code size and complexity
  - Knowledge lost due to personnel turnover
  - Maintaining quality is a challenge
    - Code, architecture, documentation...
- High dependability requirements
  - Safety- and business critical
  - MTBF requirement of 8 years cont. operati
- Massive amounts of testing required
  - Still far from perfect sw quality...
- **Better development tool support needed**

System-Specific Static Analyses

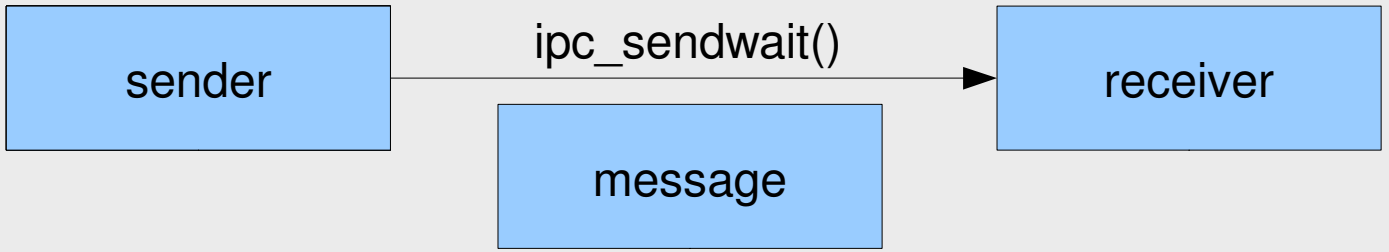
- Task interface
- IPC structure
- Task scheduling priority
- **IPC message in/out parameters**
- Visibility constraints
- Task and semaphore dependencies

IPC Message In/Out Parameters



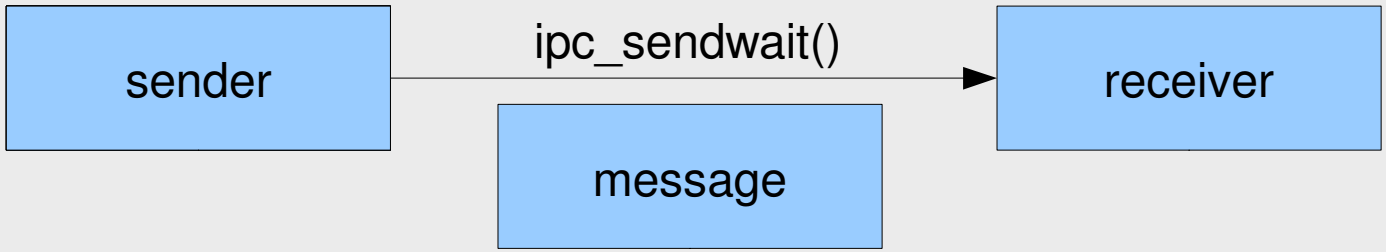

```
typedef struct
{
    IPC_HEADER header;
    int field1;                    /* In  */
    BOOL field2;                /* Out */
    STATUS field3;              /* Out */
} TASK_MSG_SERVICE42;
```

IPC Message In/Out Parameters (2)




```
STATUS task_service42 ( int mg ) {
        TASK_MSG_SERVICE42 msg;
        msg.field1 = mg;

        task_sendwait( … &msg … );

        for( i = 0 ; … ; i++ ) {
                … = msg.field2.state[i];
        }
        return msg.field3;
}
```

IPC Message In/Out Parameters (3)



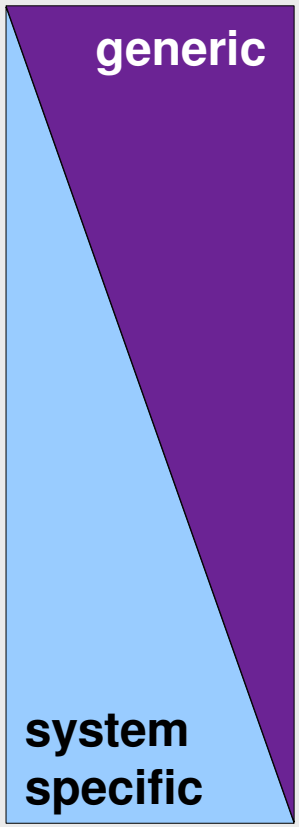

```
TASK_MSG_SERVICE42 *msg;
ipc_receive( … &msg … );

msg->field3 = do_computation (
                … msg->field1 …,
                &msg->field2 );

ipc_answer( … &msg … );
```

IPC Message In/Out Parameters (4)

- Check consistency between
  - Design (annotated in C comments)
  - Implementation (read/write-only struct fields)
- Exploit system-specific characteristics
  - Commenting convention
  - Coding style/patterns
    - Sender is contained in a single function

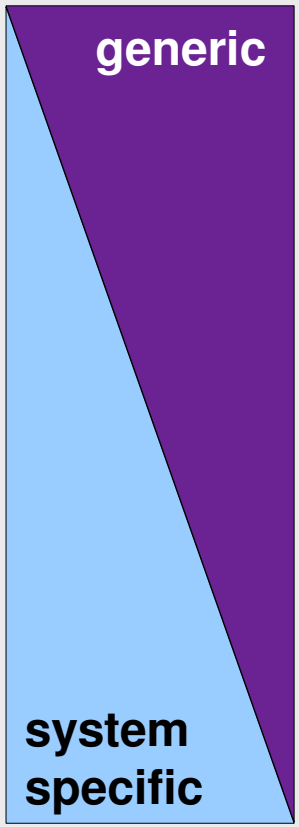
      → No inter-procedural analysis needed

Static Analyses Spectrum



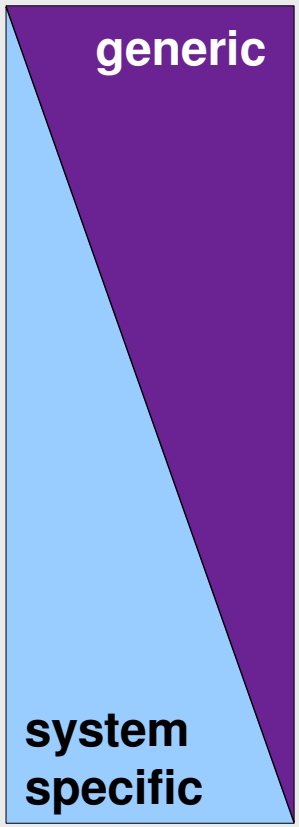
- size (bytes)
- size (LOC), diff
- code checkers (language-specific)
  - PC-lint for C/C++
  - MISRA C
- environment-specific analyses (e.g. COM/COM+ [Pinzger et al.])
- our analyses

System-Specific Static Analyses



- Approach
  - Develop analyses targeted at *one* system
  - Augment (not replace) more generic analyses
  - Focus on analyses that promise to have a high impact on key quality attributes

System-Specific Static Analyses (2)



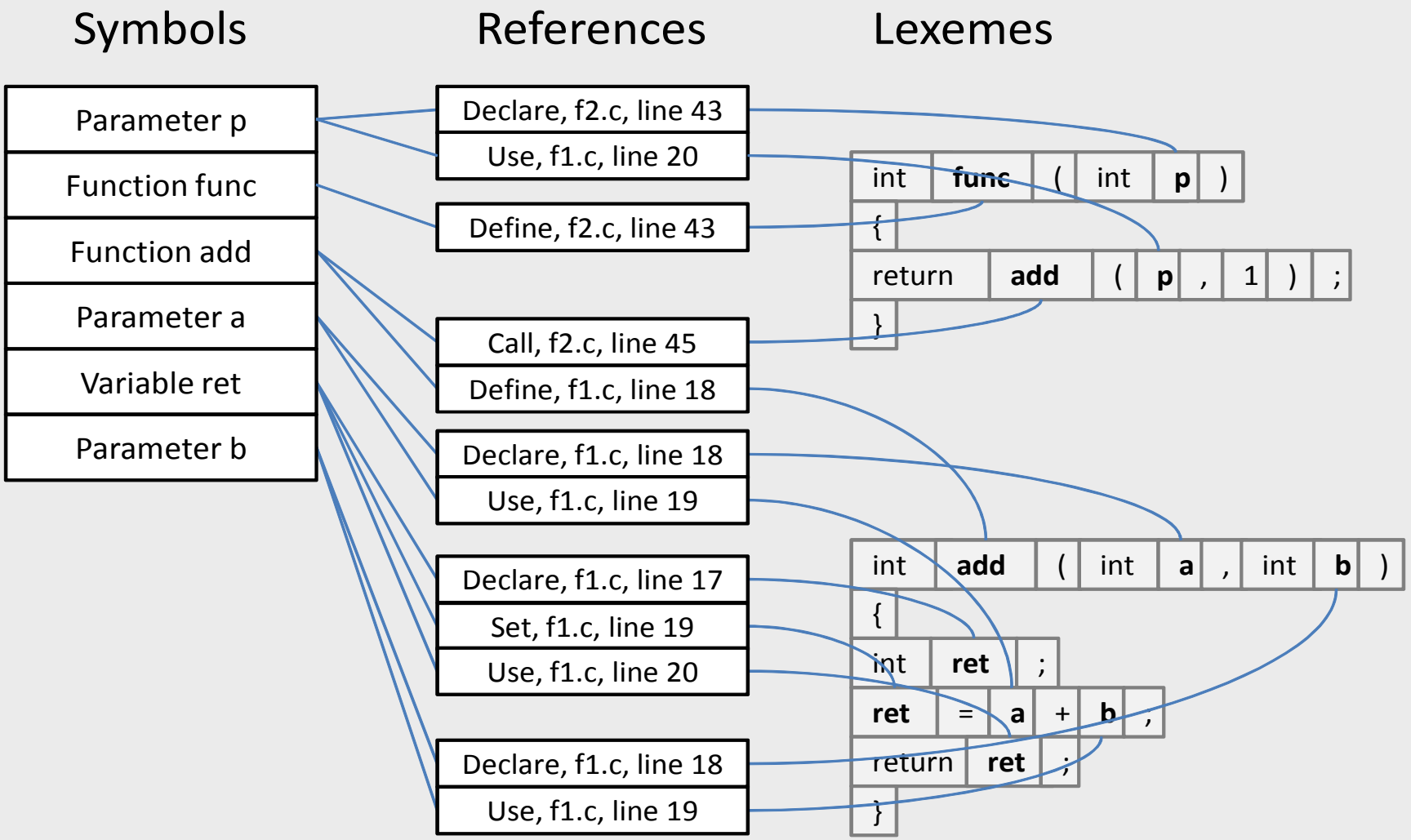
- Benefits
  - Less false positives
    - 30% not uncommon for generic analyses
    - Increases developers' trust
  - Better diagnostic messages
  - Can be incrementally introduced (experience-based)
  - Can take advantage of simplifying assumptions

Enabling System-Specific Static Analyzes

- An analysis platform is needed
  - Several candidates exists...
- Main candidate: “Understand for C++” (1)
  - Creates a “symbol database” over the code
    - “Entities”: Files, Functions, Variables...
    - “References”: Calls, Uses, Assignments...
  - Scales to large systems
    - Parses 1,3 MLOC in 149 seconds
  - Has an interesting API, in Perl and C
    - Well documented, allows for custom extensions

(1) Scientific Toolworks, Inc.

“Understand” Database Structure



Understand API Example (Perl:ish)

- Sub exploreCallGraph(func)
  - $visited{$func->id()} = $func->name();
  - foreach my call (func.refs("call"))
    - If (not visited(call.ent().id())) {
      - print(func.name() . " calls " .
        call.ent().name() . " at " .
        call.file().name() . " " . call.line());
      - exploreCallGraph($call->ent()); }

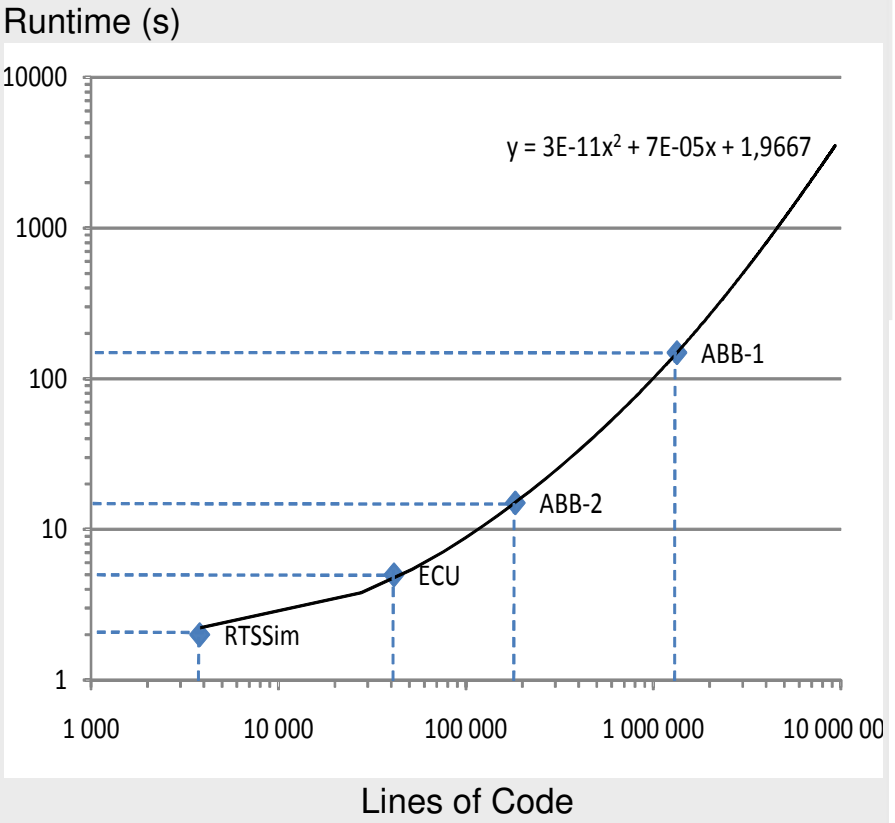
- my mainFunc = db.lookup("main","function");
- exploreCallGraph(mainFunc);

Why “Understand”

- Several similar tools exists
  - CodeSurfer, Imagix 4D, Rigi ...
- A handfull were evaluated in 2005
- Understand was selected, since
  - Superior processing speed
    - Imagix and CodeSurfer chokes on 500 KLOC
  - Well-documented API
    - in Perl and C
  - Relatively affordable at the time, $500.

“Understand” Performance

- Code parsing
  - 183 KLOC: 15 sec
  - 1,3 MLOC: 149 sec
- Fitted curve (x^2)
  - The x^2 factor is small
    - “linear” for < 1 MLOC
  - Predicted:
    - 5 MLOC: ~20 min
    - 10 MLOC: ~60 min
- Scales to large systems!

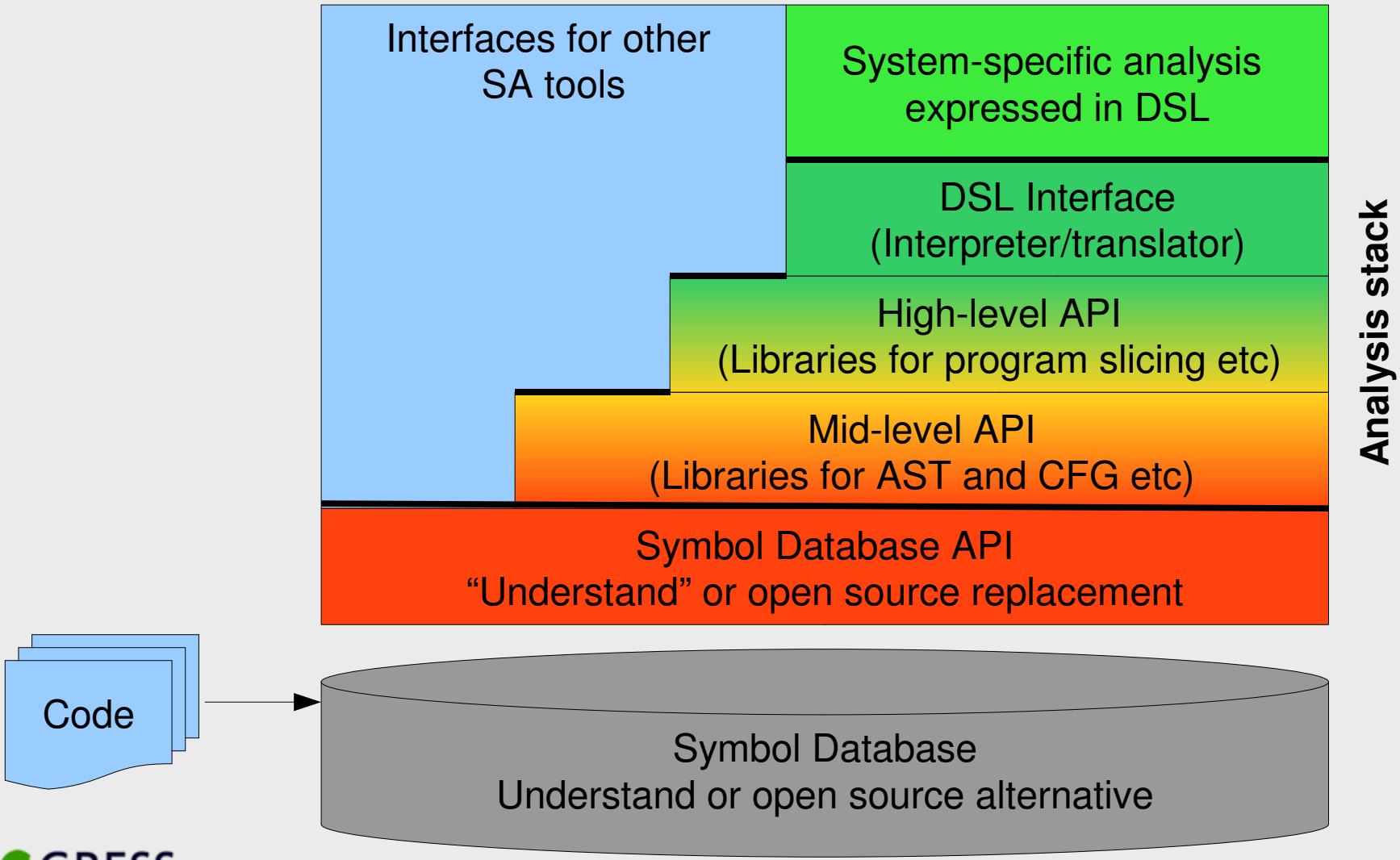


PROGRESS

Drawbacks of Understand

- Relatively simplistic program model
  - No AST:s or CFG:s
  - Probably a reason for the fast processing
  - Lexeme level analysis still possible
    - AST and CFG libraries can be added on top
- No advanced analyzes built-in
  - API allows for own extensions
    - Example: Program slicing recently implemented
- Proprietary and expensive ($2000/lic.)
  - **Replace with similar open source solution**

Outlined Solution



Summary

- System-specific static analyzes as complement to generic code checkers
  - Leverage on system-specific information
  - High potential, especially for large systems
  - Easy to integrate in development process
  - Potentially high impact on software quality
- Enabling technology
  - An open solution similar to Understand
  - Additional libraries (analysis stack)
  - A notation for analysis specifications (queries)

System-Specific Static Analyses

- Task interface
- IPC structure
- **Task scheduling priority**
- IPC message in/out parameters
- **Visibility constraints**
- Task and semaphore dependencies

Task Scheduling Priority

- Generally each task has a fixed priority
  - Determined at design time
  - Not changed during run-time
- However, there are exceptions...

```
bytes = ipc_receive ( … IPC_NODELAY … );

if (bytes <= 0 ) {
       (void) os_change_priority( task_id,
                                  max_priority );
       prio_change = TRUE;
}
```

Task Scheduling Priority (2)

- Check that code changes do not introduce new dynamic priority changes
  - **`os_change_priority`**`()` calls
- Important because such a change requires a detailed impact analysis

Visibility Constraints

- Functions are tagged as
  - PUBLIC (anybody can call them)
  - PRIVATE (within translation unit)
  - INTERNAL (only within *subsystem*)

```
#define PRIVATE static
#define PUBLIC
#define INTERNAL

PRIVATE void foo();
```

- Check function calls for violations of the INTERNAL visibility constraints