MODEL CHECKING

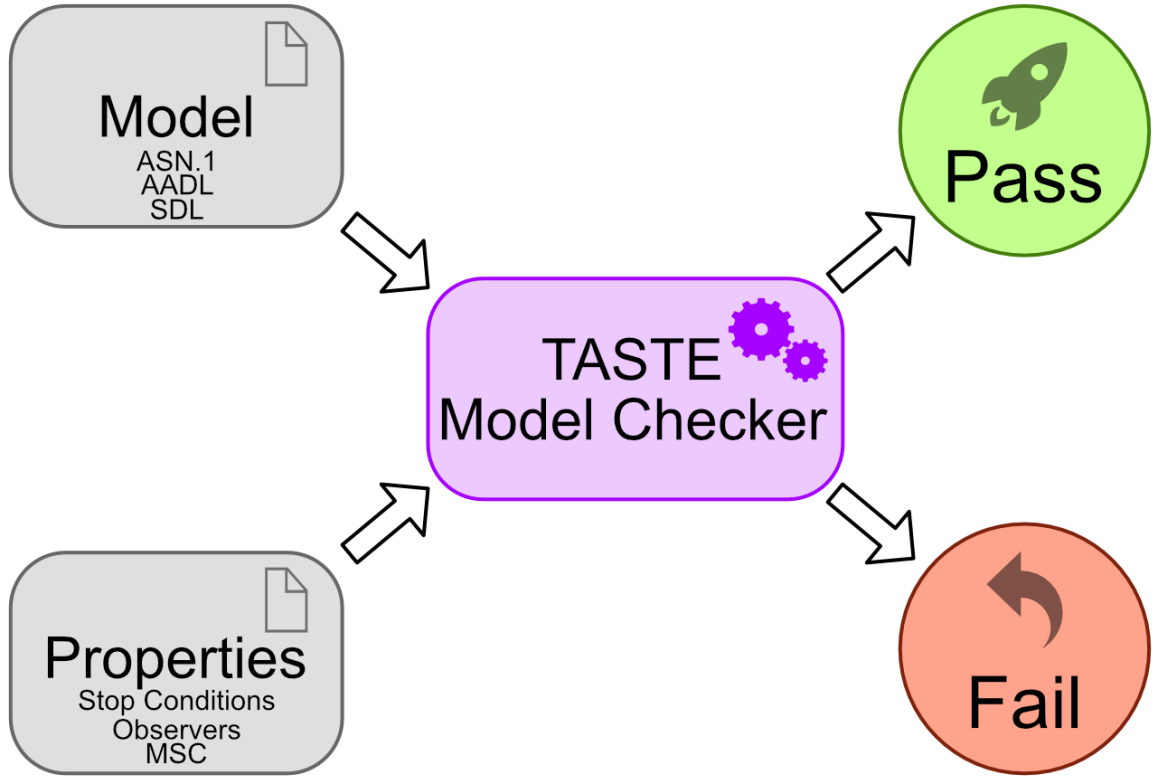
for Formal Verification of Space Systems

Model Based Space Systems and Software Engineering - MBSE2021

2021

Agenda

- Model Checking
- TASTE
- Promela/SPIN
- TASTE Model Checker
  - Introduction
  - Requirement specification languages
    - Stop Condition Language
    - Observer Language
    - Message Sequence Charts Language
  - Workflow
- Demonstration use cases
- Status and next steps



Model Checking

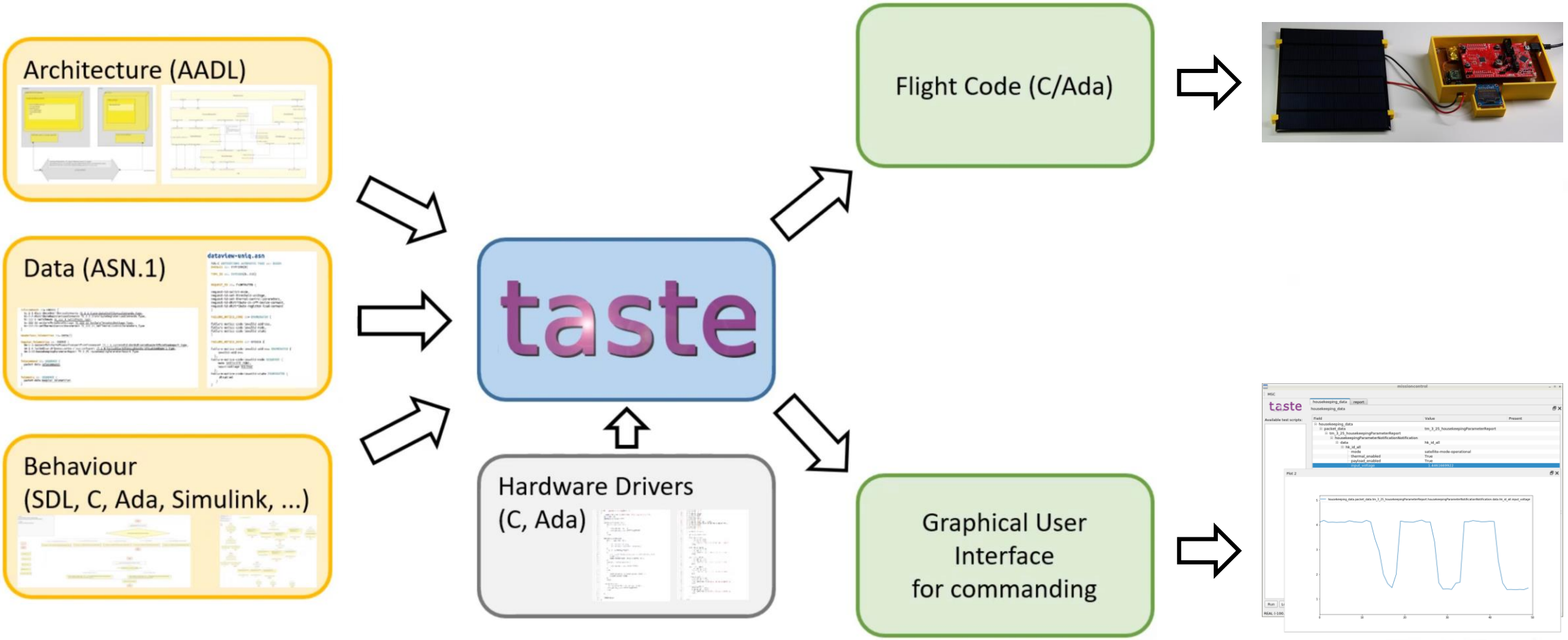
- Model checking is a method for verifying whether a finite-state model of a system meets a given specification – usually provided as invariants, as well as safety and liveness properties
  - Safety – nothing bad happens
  - Liveness – eventually something good happens
- Different approaches exist – we focus on explicit-state model checking
  - with explicit states (values) and with explicit transitions (changes of the states/values)
- In simple terms, it can be thought of as generalized testing:
  - "input vectors" are automatically generated (e.g., to exhaustively explore the system)
  - "output vectors" are verified using generic formulas (e.g., via Boolean expressions)
  - It borders between "static testing" (can be applied to the system design/specification) and "dynamic testing" (can "execute" the model, from which code can be derived)
- It does not replace testing, though it may complement it, and may be used to support test creation
  - Issues with platform, complexity, model/code equivalence, requirement formulation, etc.

TASTE

- "A tool-chain targeting heterogeneous embedded systems, using an MBSE development approach"
- Developed by ESA and based on standards:
  - AADL – logical and physical architecture description
  - ASN.1 (ACN) – data model specification (with additional encoding information)
  - SDL and MSC – for high-level behaviour description
  - But can also integrate other technologies – e.g., C, Ada, C++, Simulink...
- Includes a QtCreator based Integrated Development Environment - SpaceCreator
  - Providing graphical and text editors, build management, tool-integration, etc.…
- Uses Kazoo template processor for code generation
- It is open-source – free to use by companies and individuals
- Comes with a set of runtimes, consisting of middleware, templates and drivers
  - Available for x86, Leon, ARM and MSP430 targets, using Linux, RTEMS and FreeRTOS operating systems
- Please visit https://taste.tools/

TASTE

- TASTE can generate executable code for the embedded targets, **directly from the models**



TASTE

- TASTE Models
  - Can be manually designed
  - Can be generated from other models, as a part of a broader or a hybrid MBSE workflow
    - See Capella-TASTE plugin, presented on MBSE2020
- These models enforce constraints (e.g., no pointer operations) and structure (interfaces)
  - Limiting for a human (which is both good and bad...)
  - Easier to analyse and transform (to and from)
- TASTE models can be used as a single source of truth, for an automated generation of:
  - the final code
  - the relevant documentation
  - other models used for e.g., design and behaviour verification

Promela/SPIN

- **Pro**cess/Protocol **Me**ta **La**nguage, used for describing concurrent processes in distributed systems
- **S**imple **P**romela **In**terpreter - a state of the art model checking engine for Promela
- Synchronous and asynchronous channels
- Primitive data types, arrays and structs
- Global and local (per process) data
- Priorities, atomicity
- "Non-deterministic" control flow
- Assertions
- Linear Temporal Logic support
- Unfortunately, no function calls and no real types
- However, there are C code blocks, if necessary

```
proctype Payload(chan cmd; chan tm)
{
    mtype command;
    end: do
        ::cmd?command ->
        if
        ::(command==payloadEnable) && (state==off) ->
            state = on;
            tm!payloadEnabled;
        ::(command==payloadDisable) && (state==on) ->
            state = off;
            tm!payloadDisabled;
        ::(command==payloadExecute) && (state==on) ->
            tm!payloadExecuted;
        ::else ->
            tm!failled;
        fi;
    od;
}
```

TASTE Model Checker

- TASTE Model Checker – tool to perform model checking on systems designed in TASTE, sourcing the information from the same models as the ones used for the code generation
- Assumptions, design decisions, limitations:
  - GUI, for end users, and CLI, for Continuous Integration and automation
  - Integrated with SpaceCreator (main IDE) and OpenGEODE (Observer editing)
  - Spin used as the model checking backend
  - Automated ASN.1, Interface View and SDL to Promela transformation
    - Only SDL, no C or Ada support
    - Only sporadic (asynchronous) interfaces between TASTE Functions (components)
  - Model properties (expressing requirements) to be defined via:
    - Stop Conditions (Boolean expressions extended with LTL)
    - Observers (state machines for advanced monitoring and model manipulation)
    - Verification Message Sequence Charts (required or forbidden messaging sequences)
  - Input vector reduction by per-interface ASN.1 data tailoring
  - State collapse by Observers and culling via Stop Conditions
  - Space-specific functionality – error injection, via Observers

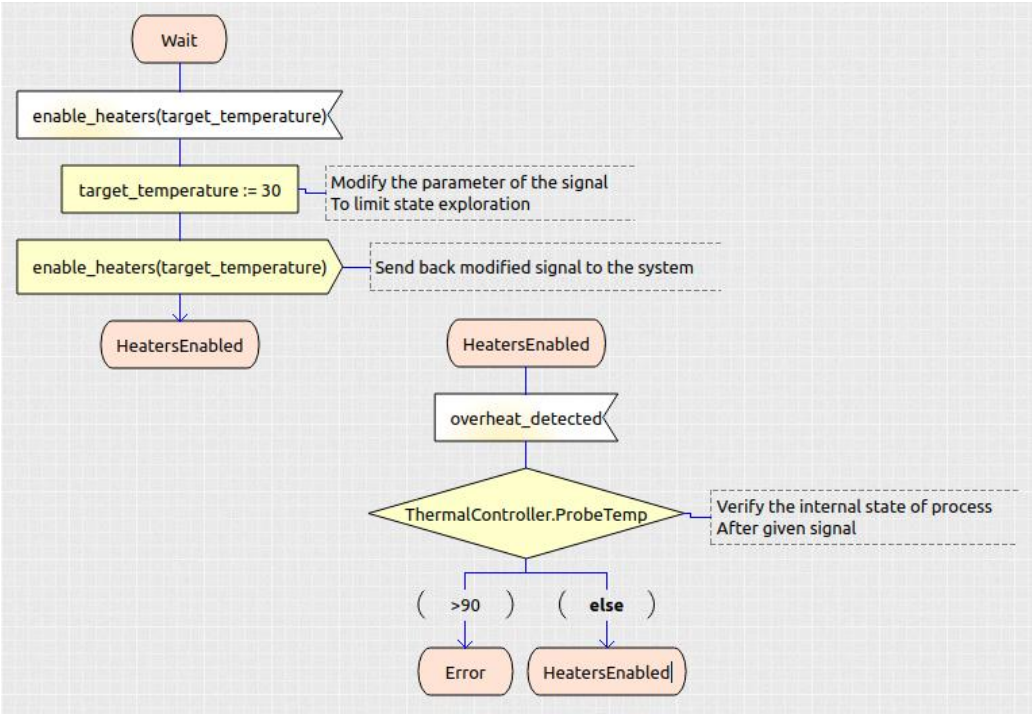
Stop Condition Language

- Stop Condition Language is a textual requirement specification language, which allows to:
  - Define safety requirements using *always* and *never* clauses
  - Define liveness requirements using *eventually* clause
  - Reduce state exploration using *filter_out* clause by cutting off potentially irrelevant states
- Stop Condition Language allows to define a set of Boolean expressions, which express requirements about the following elements of the system:
  - Internal state of SDL processes in the system including nested states and parallel states
  - Internal data of SDL processes in the system
  - Values of signal's parameters
- Stop Condition Language will be translated into Promela LTL clauses

```
-- The state of ExampleProcess shall eventually be Initialized
eventually state(ExampleProcess) = Initialized;

-- The internal variable exampleVar of process ExampleProcess shall always
-- be smaller than 100
always ExampleProcess.exampleVar < 100;

-- When the state of ExampleProcess is Initialized then
-- the value of exampleVar shall never be smaller than 0.
never state(ExampleProcess) = Initialized and ExampleProcess.exampleVar < 0;

-- The Model Checker Engine shall not investigate states
-- when variable anotherVar in ExampleProcess is equal to false
filter_out ExampleProcess.anotherVar = false;
```

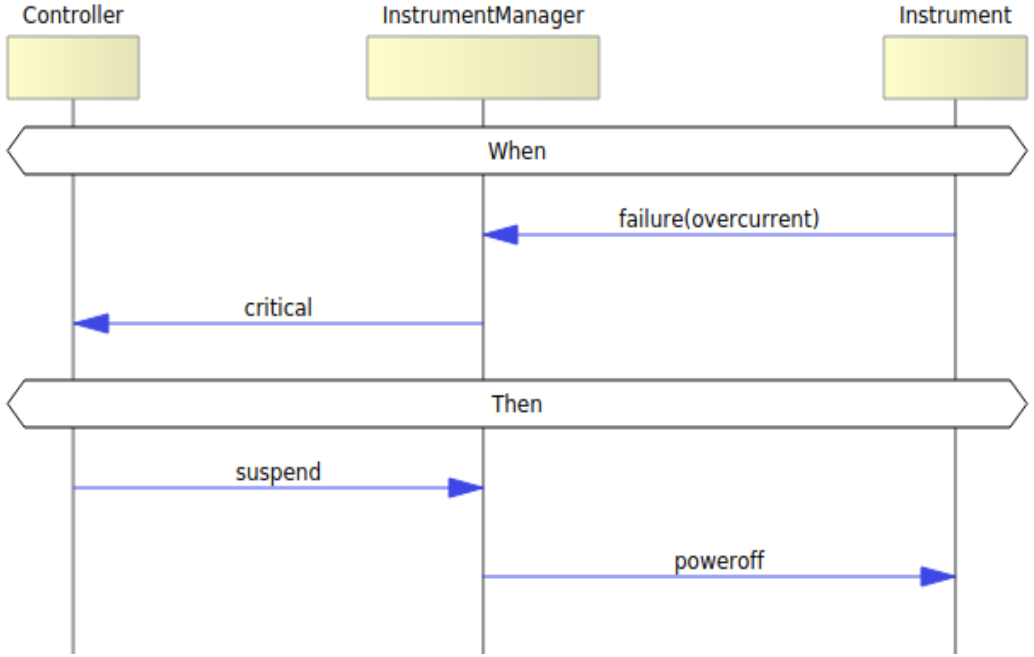
Observer Language

- Observer language is a graphical and textual language based on SDL, which allows to define special SDL processes with an ability to observe the entire system and optionally modify its state
- An Observer can react to different situations in the system, including:
  - Signal emission or reception by any process
  - Signal loss in any process
  - Change of internal state or variable of any process
- An Observer can report detection of an error by entering state marked as a special *Error state*
- An Observer can modify the state of the system or a signal's parameter which allows to:
  - Guide state exploration to meaningful scenarios
  - Constrain state exploration
  - Inject errors into the system
- Observers will be translated into Promela code



Message Sequence Charts Language

- MSC is a language used to capture a system's operational scenarios in a formal way
- The restricted subset of this language was chosen to specify requirements for modelled system
- This requirement specification language allows to define *required* and *unwanted* behaviours in an unambiguous manner
  - The *When* condition is used to define a trigger for a *requirement* applicability
  - The *Then* or *Then Not* is used to define the actual *requirement*
- The following elements of MSC are supported:
  - Instance
  - Message
  - Coregion
  - Timer
- The MSC will be translated into Observers



Workflow

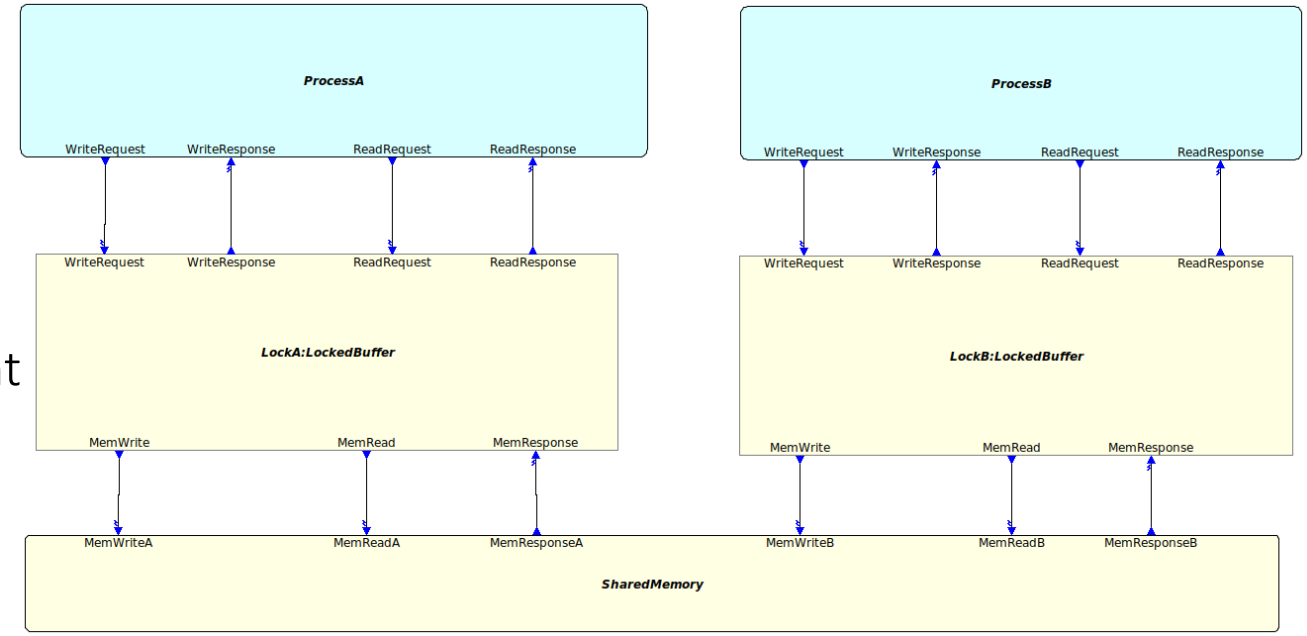
- A simplified version of the proposed workflow is as follows:
  - Define an iteration of the model structure (e.g., signals, states, data structures)
  - Formalize the requirements by translating the informal ones into Stop Conditions, Observers and MSCs
  - Define an iteration of model behaviour using SDL
  - Define a model checking scenario by:
    - selecting the relevant subset of the model under verification
    - selecting the applicable requirements
    - [optionally] refining input vector generation
    - [optionally] preparing additional Observers transitioning the model into the desired initial state
- Meant to be iterative, performed concurrently with the system design and implementation
- It is assumed that it can and will be tailored by the end users for their needs

Demonstration Use Cases

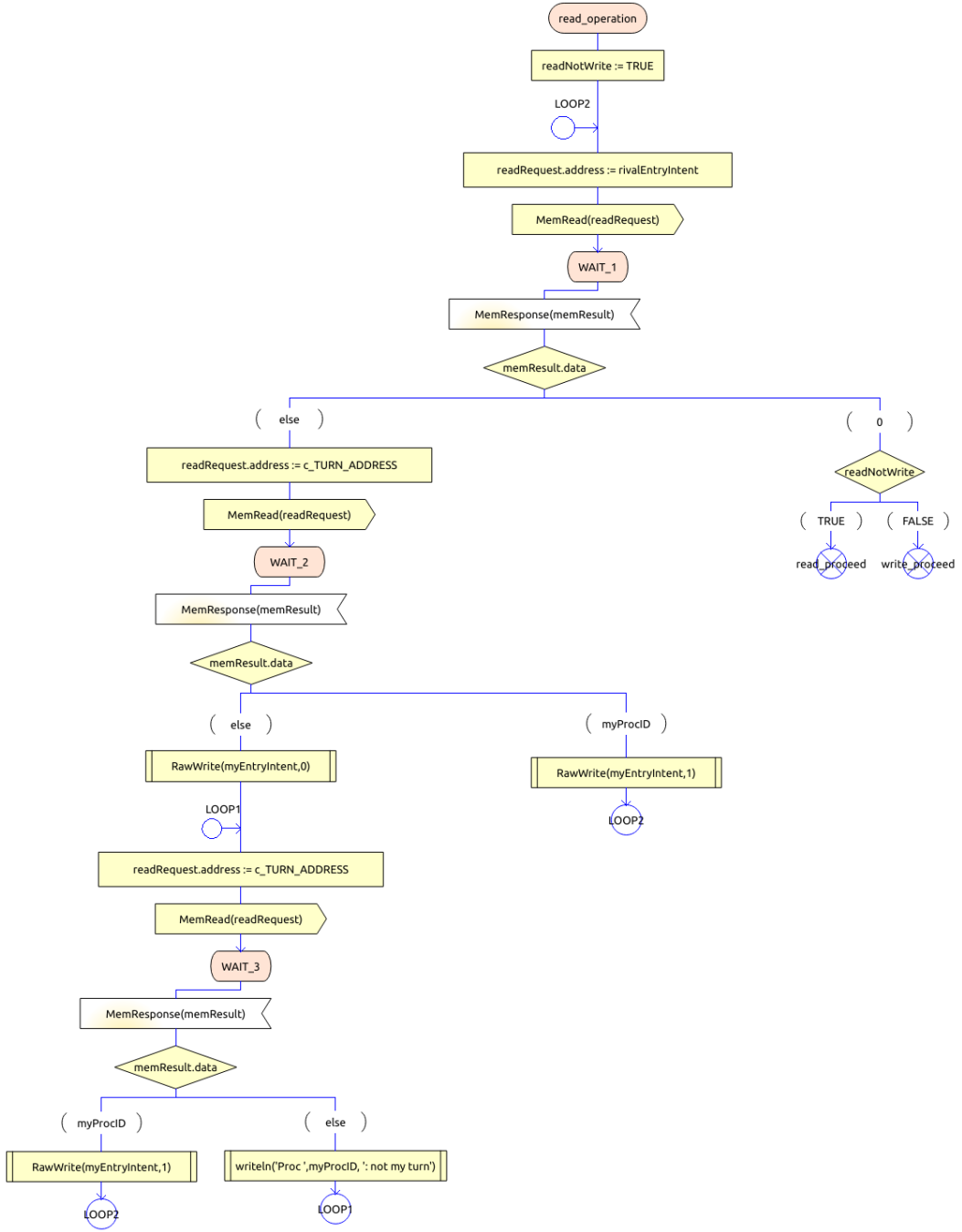
- Goals
  - Validate the model checker and explore its capabilities and limits
  - Provide representative practical examples that demonstrate the applicability of the model checking technique
- Cases in development
  - Toy Model
    - Small model with well-understood behaviour
  - Subsystem Model
    - Larger model corresponding to an isolated subsystem with potentially complex internal behaviour
  - Application Model
    - System-level model demonstrating interacting subsystems with commanding from external sources

Toy Model

- Mutual exclusion using Dekker's algorithm
- Two instances of an SDL buffer locking process
- Shared memory is also modelled as an SDL process
  - Memory written/read by exchange of messages
- Application processes are represented by environment
  - May issue read/write requests in any order
  - All orderings visited by the model checker
- Properties to be proven : exclusion, liveness, fairness
- Also examine known defective implementations



```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
```

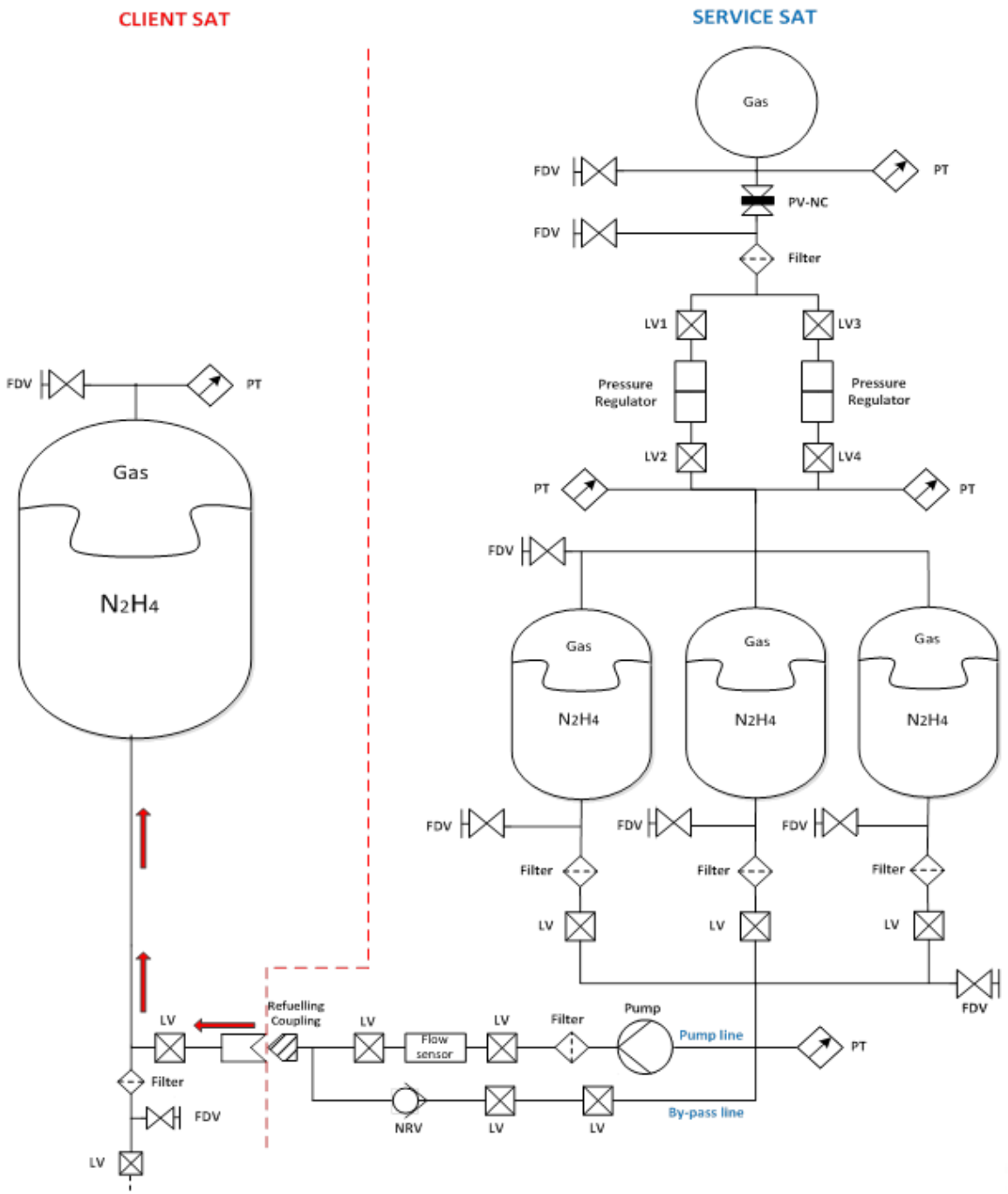


Subsystem Model

- The selected subsystem model is a CANopen network using the ECSS-E-ST-50-15C protocol
- Starting from a minimal subset, explore models of progressively higher complexity
  - Find the capacity limits of the model checker
  - Detect protocol defects
- Use MSCs to define wanted/unwanted message sequences
- Use Observers to cause message loss / corruption
- Use the complexity reduction features of the model checker

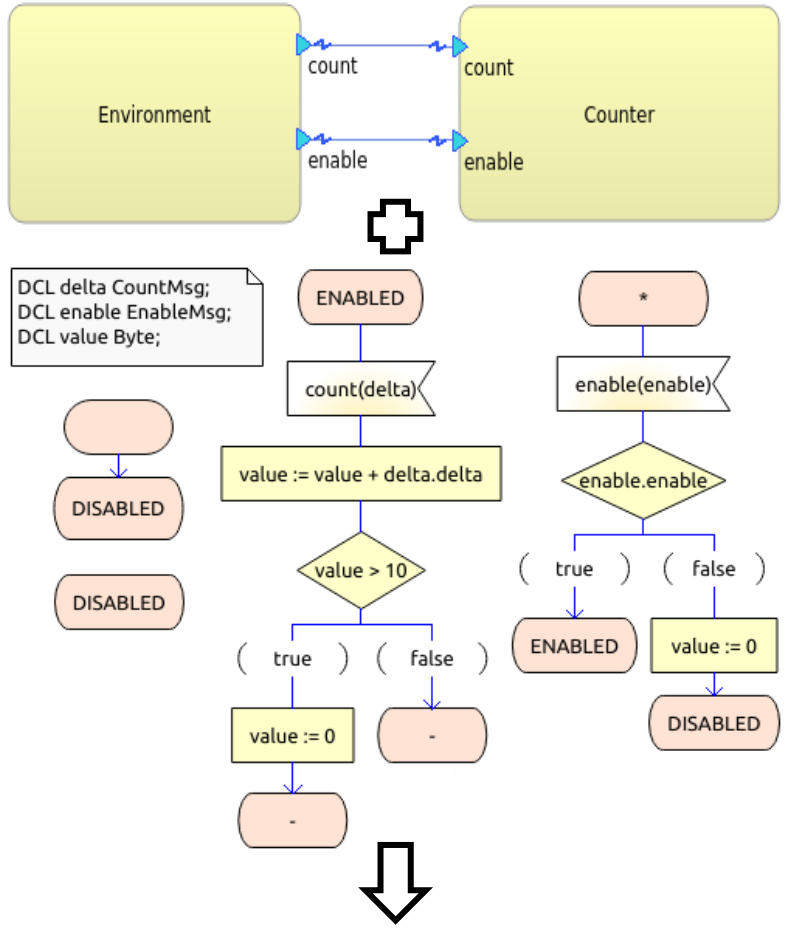
Application Model

- In-orbit refuelling of a client spacecraft by a service spacecraft
  - Two independent systems, separately commanded
  - Feared events can be caused by incorrect commanding
- Examine system response to combinations of component failures



Status and next steps

- TMC requirements are consolidated
- Requirement specification languages are defined
- Workflow is proposed
- Design is defined
- Validation use-cases are selected
- **Proof-of-concept** has been prepared
  - based on a trivial system designed in TASTE
  - manually coded in Promela
  - successfully exercised in Spin
- Implementation is in progress
  - Early prototype expected in a few months
  - Validation will begin next year




```
active proctype Counter_Count() priority 2 {
atomic{
        do
        ::count?counterCountMsg ->
            CountMsgAssign(model.counter.delta, counterCountMsg);
            if
                ::(model.counter.state == STATE_COUNTER_DISABLED) ->
                    Counter_DISABLED_Count(model.counter.delta);
                ::(model.counter.state == STATE_COUNTER_ENABLED) ->
                    Counter_ENABLED_Count(model.counter.delta);
            fi
            Observer_WAIT_Count_postdelivery(counterCountMsg);
        od
    }
}
```

Thank you for your attention

Michał Kurowski
mkurowski@n7space.com

Rafał Babski
rbabski@n7space.com

+48 22 299 20 50
www.n7space.com

Steve Duncan
stephen.duncan@thalesaleniaspace.com

+33 (0) 1 57 77 80 00
www.thalesgroup.com