Verification-driven Development of Icarous Based on Automatic Reachability Analysis

A Case Study

Marco A. Feliú
National Institute of Aerospace
Hampton, USA
marco.feliu@nianet.org

Camilo Rocha
Pontificia Universidad Javeriana
Cali, Colombia
camilo.rocha@javerianacali.edu.co

Swee Balachandran
National Institute of Aerospace
Hampton, USA
swee.balachandran@nianet.org

ABSTRACT

The *Integrated and Configurable Algorithms for Reliable Operations of Unmanned Systems* (Icarous) is a software architecture being developed for the robust integration of mission specific software modules and highly assured core software modules. This paper reports on the use of automatic reachability analysis during the development of Icarous, as a first step towards a broader formal verification effort of the software architecture. In particular, this paper explains how simulation based on state-space exploration and LTL model checking have been performed on a formal executable specification of the system in rewriting logic. Overall, this effort has unveiled issues such as deadlocks and undesired behavior, and have helped in improving the Icarous design and source code.

ACM Reference format:
Marco A. Feliú, Camilo Rocha, and Swee Balachandran. 2017. Verification-driven Development of Icarous Based on Automatic Reachability Analysis. In *Proceedings of 24th International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 2017 (SPIN'2017),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

One important goal in NASA's Unmanned Aerial System (UAS) Traffic Management project (UTM) is to support the safe operation of low-altitude UAS in uncontrolled airspace so that they can be used for civilian and public applications such as infrastructure monitoring and delivery of goods. In order to fulfill this goal, UTM requires unmanned UAS to be equipped with a minimum set of capabilities with conformance to safety criteria. The *Integrated and Configurable Algorithms for Reliable Operations of Unmanned Systems* (Icarous) [4] is an open source implementation in Java and C++ of a software architecture designed for supporting autonomous UAS missions. It uses airborne technology with the assistance of onboard "detection and avoid" systems available from the Daidalus project [6] in the form of verified algorithms with conformance to safety criteria. Given its critical nature and inherent complexity because of its high concurrency, due both to the internal interleaving of its modules and to the external environment it interacts with, a significant challenge for formal methods is to thoroughly analyze the composition of these algorithms in Icarous, in order to ensure its realibility in *real-world* situations.

SPIN'2017, Santa Barbara, CA, USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

The work reported in this paper is part of a broader effort in the development of techniques and tool support for automatic reachability analysis of Icarous. As a first step towards this goal, this paper explains how automatic reachability analysis has been performed with the help of an executable formal semantics of Icarous. Simulation based on state-space exploration and LTL model checking have unveiled issues such as deadlocks and undesired behavior, and have helped in improving the Icarous design and source code.

The formal specification of Icarous is a theory in rewriting logic [5] – a semantic framework unifying a wide range of models of concurrency – obtained from Icarous source code by an iterative refinement process. At each stage, this rewriting logic semantics of Icarous has been fully executable in Maude [3], thus benefiting from formal analysis techniques and tools such as state-space exploration an automata-based LTL model checking. The rewriting logic specification of Icarous has been found to be particularly well-suited for: (i) validating design decisions in Icarous architecture and implementation; (ii) understanding design and implementation issues after observing undesirable behavior of the actual system during flight-testing; and (iii) rapid prototyping and evaluation of new design features for Icarous.

2 OVERVIEW OF ICAROUS

Icarous is a software architecture being developed for the robust integration of mission specific software modules and highly assured core software modules. The set of software modules integrated in Icarous include formally verified algorithms that detect, monitor, and control conformance to safety criteria or that compute resolution and recovery maneuvers, mainly, available from the Daidalus project [6]. The architecture is flexible enough to support the integration of other algorithms.

Figure 1 depicts the distributed architecture of Icarous and the main software components. This system comprises three components that run concurrently and interact via shared variables:

- **COM:** it enables interaction with Icarous from a ground station via MAVLink messages. MAVLink is an open source protocol designed for communicating with small UASs [1]. This component also passes MAVLink messages streaming from the autopilot to the ground station and vice versa.
- **FMS:** it is responsible for making all flight related decisions, monitoring flight conditions for conflicts, and resolving conflicts when necessary. Conflict detection and resolution is established using hierarchical finite state machines that compose the verified algorithms available from Daidalus.

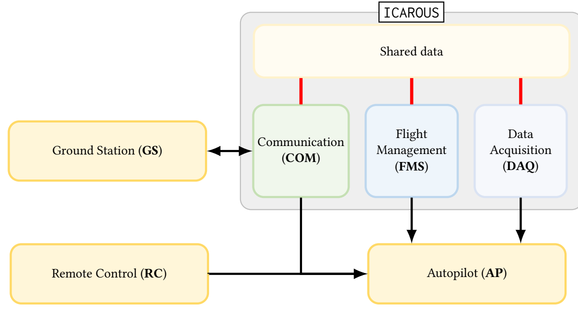

Figure 1: ICAROUS distributed architecture, components, and external entities.

DAQ: it reads data from a MAVLink compatible autopilot. Read data is stored in a shared data structure that can be accessed by all other components.

ICAROUS can interact with external entities such as: a ground station (**GS**) from which users can feed commands, send, or receive information from the UAS; an autopilot (**AP**) that automatically guides the UAS when in autonomous mode; and an optional remote control device (**RC**) that enables manual guide.

ICAROUS supports mission execution with autonomous, safety-based navigation and decision making while monitoring UTM procedures and mission-specific criteria. There is an ongoing project that uses ICAROUS for the development of beyond visual-line-of-sight autonomous power lines inspection missions. For more details the reader is referred to [4].

3 REWRITING LOGIC SEMANTICS

Rewriting logic [5] is a semantic framework that unifies a wide range of models of concurrency. Specifications in rewriting logic are called rewrite theories and they can be executed in the rewriting logic implementation Maude [3]. By being executable, they benefit from a set of formal analysis tools available to Maude, such as state-space exploration an automata-based LTL model checking. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \uplus B, R)$ with: (i) $(\Sigma, E \uplus B)$ an order-sorted equational theory with signature Σ, E a set of equations over the set T_Σ of Σ-terms, and B a set of structural axioms over T_Σ for which there exists a finitary matching algorithm (e.g., associativity, commutativity, and identity, or combinations of them); and (ii) R a finite set of rewrite rules over T_Σ. Intuitively, $\mathcal{R}$ specifies a concurrent system whose states are elements of the set $T_{\Sigma/E \uplus B}$ of Σ-terms modulo $E \uplus B$ and whose concurrent transitions are axiomatized by the rewrite rules R according to the inference rules of rewriting logic [2]. In particular, for $t, t' \in T_\Sigma$ representing states of the concurrent system described by $\mathcal{R}$, a transition from t to t' is captured by a formula of the form $[t]_{E \uplus B} \rightarrow_{\mathcal{R}} [t']_{E \uplus B}$; the symbol $\rightarrow_{\mathcal{R}}$ denotes the binary rewrite relation induced by R over $T_{\Sigma/E \uplus B}$.

The rewriting logic semantics of ICAROUS is a rewrite theory $\mathcal{R}$ with *topsort* System, meaning that concurrent transitions in the system are over the set $T_{\Sigma,\mathtt{System}}$ of Σ-terms of sort System. Some equations are used for defining internal deterministic transitions whose intermediate states can be ignored for the purpose of formal verification. In this rewriting logic semantics, each one of the ICAROUS components and the external entities (see Section 2) is specified as an object-like term that is part of an object configuration. In particular, each one of the verified algorithms available from DAIDALUS that are part of the system is specified as a 'black-box' whose behavior obeys its design contract. Communication between components is specified by shared variables that are accessed under mutual exclusion. Communication between internal components and the external entities is modeled by message passing in an actor-model-like scenario. Overall, the rewriting logic semantics of ICAROUS comprises 80+ rewrite rules, 30+ equations formally specifying the concurrent behavior of the system, and approximately 1000+ lines of Maude code.

The reference for specifying the rewriting logic semantics of ICAROUS and the external entities has been, mainly, the state machines defining the high-level dynamics of the internal components and the source code.

3.1 States

A system state in the rewriting logic semantics has sort System and corresponds to a multiset (sort Configuration) of object-like terms (sort Object) and messages (sort Msg):

```
op {_} : Configuration -> System .
sorts Object Msg Configuration .
subsort Object Msg < Configuration .
op <_|_> : Oid AttributeSet -> Object .
op none  : -> Configuration .
op __ : Configuration Configuration -> Configuration
                                    [assoc comm id: none] .
```

The sort Oid represents object identifiers. Thus, an object in this signature is a term with an identifier and a set of named attributes. A specific structure for messages is not enforced. Multiset union for objects and messages is represented by juxtaposition and it is associative and commutative, and it has identity element none.

For example, a system state can be represented as a term of sort System having the form:

```
{ < ST | ... > < FMS | ... > < COM | ... > < DAQ | ... >
  < AP | ... > <  RC | ... > <  GS | ... > }
```

This configuration of objects represents the state of internal components and of the external entities: ST represents the shared memory; FMS, COM, and DAQ represent, respectively, the **FMS**, **COM**, and **DAQ** components; and AP, RC, and GS represent, respectively, the **AP**, **RC**, and **GS** external entities. For instance, the following term represents an internal state of the **FMS**:

```
< FMS | state: flight,     uasState: takeoff(waitLast(msg)),
        conflict: noop,    prevConflict: noop,
        usePlan: true,     resolutionState: noop
        flightPlanSize: 4, nextFlightPlanWaypoint: 0 >
```

In this case, the **FMS** is in flight mode and the UAS is performing a takeoff maneuver while waiting for the last message from the ground station. There is not an active nor a past conflict – thus no resolution has been done by the corresponding algorithm – and a flight plan with 4 waypoints is in execution; the next waypoint in this plan is the first one in the sequence, meaning that the flight is about to begin.

3.2 Transitions

Internal and external communication, and behavior of the components and entities are specified by equations and rules in the rewriting logic semantics.

3.2.1 Internal communication. It takes place via the shared memory represented by object ST; all shared variables in the system are attributes of this object. Shared variables can be of basic types such as integer or real numbers, but also lists and queues can be found there. The ST object uses a semaphore for controlling the concurrent read-write access to the shared variables. In particular, this object has a lock attribute indicating when an object has exclusive access to the shared memory; initially, this attribute is set to none indicating that no component has exclusive access to the shared variables. The following rule specifies how the **FMS** acquires the lock of the shared memory:

```
rl { < FMS | uasState: takeoff(waitLast(msg)), AtS >
     < ST  | lock: none, AtS' > Cnf }
=> { < FMS | uasState: takeoff(waitLast(msg)), AtS >
     < ST  | lock: FMS,  AtS' > Cnf } .
```

The left-hand side (before =>) indicates that this rule can be applied to a state in which the **FMS** is in flight mode, the UAS is in a takeoff maneuver, and the lock of the shared memory is not taken. The rest of the system state, captured by variable Cnf of sort Configuration, can be in any state. The effect of applying this rule is specified by its right-hand side (after =>): the lock is acquired by the **FMS** by setting attribute lock in ST to FMS. In this way, mutual exclusion for internal communication through shared variables is specified in the rewriting logic semantics of Icarous.

3.2.2 External communication. External communication is enforced by message passing consisting in the transit of messages produced by an object into the input buffer of another object. A 'send' message can be represented generically as follows:

```
op send : MAVLinkMsg Oid -> Msg .
```

In a message of the form send(m,o), the parameter m is a MAVLink message (sort MAVLinkMsg) representing the message being sent and o (sort Oid) the object identifier of the recipient. Once created, messages are put in the system state as part of the object configuration. The following rule specifies how a DO-SET-MODE(Guided) message is sent from the **FMS** to the **AP**:

```
rl { < FMS | uasState: takeoff(start), state: flight, AtS >
     Cnf }
=> { < FMS | uasState: takeoff(...),   state: flight, AtS >
     Cnf send(DO-SET-MODE(Guided),AP) } .
```

The reception of a message by an object is specified equationally by putting the message last in the input buffer of the corresponding object. The use of equations in this case indicates that such are internal atomic transitions that are invisible in terms of concurrency.

```
eq { < AP | in:[MSGs],      AtS > Cnf send(aMSG,AP) }
 = { < AP | in:[MSGs,aMSG], AtS > Cnf               } .
```

Rules tuning the reliability of the communication channels with each external entity are included in the rewriting logic semantics. They consider loss or reordering of messages, and can be enabled with some *flags* when executing the semantics.

3.2.3 Component behavior. The behavior of each component is specified by abstracting the source code of the implementation and by validation against the state machines defining the high-level semantics of each component. For example, for specifying how Icarous lands an UAS, the following abstraction can be used as a simplified form of the actual source code:

```
case LAND:
  if (!landStart) {
    SetMode(4); // Set mode to guided
    SendCommand(0, 0, MAV_CMD_NAV_LAND, ...);
    landStart = true; }
  if (currPosition.alt() < 6.0) state = TERMINATE;
```

The first if construct checks if the **FMS** is in the first iteration of the landing state: if this is the case, it sequentially sends two commands and sets a flag for ensuring that the next iteration will not be the first one. This in turn induces the following intermediate land states in an execution of the specification: land(start), land(sent(DO-SET-MODE(Guided))), and land(sent(NAV-LAND)). The second if construct checks if the altitude of the UAS is below a certain value: if this is the case, then the **FMS** transitions to a terminal state. These transitions can be captured by a rule that can be fired non-deterministically:

```
rl { < FMS | uasState: land(sent(NAV-LAND)), AtS > Cnf }
=> { < FMS | uasState: terminate,            AtS > Cnf } .
```

3.2.4 External entity behavior. The **RC** and **GS** are specified to behave at will by enabling the reproduction of several combinations of interactions, including those corresponding to human intervention in the system. The **AP** is a piece of software with certain standard behavior, which is part of the rewriting logic semantics.

For example, Icarous waits for an acknowledgment from the **AP** after sending a MAVLink message. In turn, the **AP** processes an incoming message and sends an acknowledgment message if necessary. Moreover, Icarous waits for other messages during a flight. For instance, a MISSION-ITEM-REACHED message is sent from the **AP** each time the UAS arrives at a destination. In order to support all these scenarios, the AP object has a buffer with messages, describing such a behavior, that are sent in order and executed with every possible interleaving.

```
rl { < AP | behaviour:[aMSG,MSGs], out:[MSGs'], AtS > Cnf }
=> { < AP | behaviour:[MSGs], out:[MSGs',aMSG], AtS > Cnf } .
```

This can introduce spurious behavior in the rewriting logic semantics. However, so far there has not been a case requiring the restriction of this behavior, mainly, because it can be filtered out for verification purposes. Moreover, this generality has become an actual feature in the semantics because it provides an abstraction of the **AP** that can model less than ideal flight situations.

4 AUTOMATIC VERIFICATION

This section presents some examples of how automatic verification has helped in detecting issues in the design or implementation of

Icarous. The verification of reachability properties uses the automatic search-space and LTL model checking capabilities available to Maude specifications.

The rewriting logic semantics presented in Section 3 has been useful for automatically analyzing reachability properties, including LTL formulas, in a number of ways. In particular, during the development of Icarous, its rewriting logic semantics has been used for both *preemptive* and *non-preemptive* (i.e., reactive) verification. In this context, preemptive verification refers to the more conventional way of using formal methods techniques for proving properties of the system before it is deployed. In contrast, reactive verification has been key for understanding design and implementation issues after observing undesirable behavior of the actual system during flight-testing. This has been particularly useful for finding scenarios that are hard to reproduce during flight-testing thanks to the search-space capabilities available from Maude.

Maude's state-space search capabilities have been used to automatically identify potential deadlocks in the implementation. The deadlocks found so far have never taken place during flight-testing. A Maude command like the following search query can detect *any* execution trace leading from an initial state `start` to a state that can not be further rewritten and which is not an expected final state (i.e., a deadlock), as shown below:

```
search start =>! St:State such that isNotFinal(St:State) .
```

For example, a deadlock was detected in the **FMS** when the UAS was in state `takeoff(waitLast(Ack(?, NAV-TAKEOFF)))`. This is because the **FMS** can wait indefinitely for a `NAV-TAKEOFF` message confirmation from the **AP** that can be lost during communication.

Temporal properties of a rewrite theory $\mathcal{R} = (\Sigma, E \uplus B, R)$ can be expressed and automatically verified using Maude's LTL Model Checker [3]. A set of state predicates Π for $\mathcal{R}$ can be equationally-defined by an equational theory $\mathcal{E}_\Pi$; the theory $\mathcal{E}_\Pi$ defines the LTL semantics for any $p \in \Pi$ as follows: it is said that $p(t)$ holds for a term t representing a system state iff $p(t) =_{\mathcal{E}_\Pi} true$. For the rewriting logic semantics $\mathcal{R}$ of Icarous, this defines a Kripke structure $(T_{\Sigma,\mathtt{System}}, \rightarrow_{\mathcal{R}}, L_\Pi)$ with labeling function L_Π defined for each $t \in T_{\Sigma,\mathtt{System}}$ and $p \in P$ by $p \in L_\Pi(t)$ iff $p(t)$ holds. The user is referred to [3] for more details and examples on LTL model checking for rewrite theories in Maude.

Some issues that have been observed during flight-testing of Icarous have been analyzed by using LTL model checking on the rewriting logic semantics. As an example, consider a situation observed during flight-testing where the UAS behaved erratically when the **RC** was being used. At some point, the **AP** was taking control of the aircraft, without any apparent reason, voiding the interaction with the **RC**. The **AP** can receive commands from Icarous or from external entities such as the **RC**, and it behaves in different ways depending on the source of the commands. The specific design problem was that the mode on which the **AP** operates could be changed at any time, regardless of the completion status of the maneuver being taken, thus it could enter an undesired loop: executing under the control of the **RC**, entering autonomous mode, being interrupted by the **RC**, ..., without ever completing the maneuver from either the **RC** or the autonomous commands.

In the following LTL formula in the syntax of Maude, `~`, `U`, and `[]` denote, respectively, the negation, the 'until', and the 'always' temporal operators, while `onMode(RC)` is a state predicate that is true precisely in those states in which the **AP** is under control of the **RC**. The following formula states that the **RC** eventually gains control of the **AP** and, once it happens, it will keep controlling it:

```
~ onMode(RC) U [] onMode(RC)
```

When model checking this property from an initial state in which the UAS was flying without any conflicts, different violating traces were found. The technical issue was that, even when the **RC** was controlling the **AP**, the **FMS** would send messages changing the operation mode of the **AP** resulting in the **RC** losing control of the UAS. This behavior was identified and corrected in Icarous.

5 CONCLUDING REMARKS

Icarous is a software architecture being developed under NASA's UTM project for the robust integration of mission specific and highly assured software modules. This paper presents a high level rewriting logic semantics of this software architecture. The semantics is written in Maude and, therefore, is fully executable. Together with automatic reachability analysis techniques, the rewriting logic semantics has been useful both in preemptive and non-preemptive verification, discovering issues with the design and implementation of Icarous. The verification techniques employed in the verification task include space-state search and LTL model checking. In particular, this verification-driven development, which is based on automatic reachability analysis, has helped in discovering new errors, replicating errors offline that have been encountered during flight-testing, and preserving desired properties during iterations of this development.

Additional work remains to be done. The rewriting logic semantics of Icarous will continue to provide formal support by evolving in parallel with its implementation. For such a purpose, the semantics can be especially valuable to explore new features of the software architecure. A symbolic rewriting logic semantics, based on the rewriting modulo SMT technique [7], is being developed for providing symbolic execution and symbolic reachability analysis for Icarous.

REFERENCES

[1] MAVLink: Micro air vehicle communication protocol. http://qgroundcontrol.org/mavlink/start. Accessed: 2017-02-01.

[2] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[4] M. Consiglio, C. Muñoz, G. Hagen, A. Narkawicz, and S. Balachandran. ICAROUS: Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC 2016)*, Sacramento, California, US, September 2016.

[5] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[6] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, and M. Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.

[7] C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, 86(1):269 – 297, 2017.