AFRL-RI-RS-TR-2022-068

# MODEL-BASED INTENT-DRIVEN ADAPTIVE SOFTWARE (MIDAS)

VANDERBILT UNIVERSITY

*MAY 2022*

FINAL TECHNICAL REPORT

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

FOR THE CHIEF ENGINEER:

**/ S /**
STEVEN L. DRAGER
Work Unit Manager

**/ S /**
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

# REPORT DOCUMENTATION PAGE

| | | | |
|---|---|---|---|
| **1. REPORT DATE** MAY 2022 | **2. REPORT TYPE** FINAL TECHNICAL REPORT | **3. DATES COVERED** START DATE MAY 2020 | END DATE SEPTEMBER 2021 |

**4. TITLE AND SUBTITLE**
MODEL-BASED INTENT-DRIVEN ADAPTIVE SOFTWARE (MIDAS)

| | | |
|---|---|---|
| **5a. CONTRACT NUMBER** FA8750-20-C-0215 | **5b. GRANT NUMBER** N/A | **5c. PROGRAM ELEMENT NUMBER** 62303E |
| **5d. PROJECT NUMBER** | **5e. TASK NUMBER** | **5f. WORK UNIT NUMBER** R2Z6 |

**6. AUTHOR(S)**
Gabor Karsai, Alessandro Coglio, Abhishek Dubey

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

| | |
|---|---|
| **PRIME** | **SUB** |
| Vanderbilt University | Kestrel Institute |
| 110 21st Avenue South | 3260 Hillview Ave |
| Nashville TN 37203-2417 | Palo Alto CA 94304 |

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

| | |
|---|---|
| Air Force Research Laboratory/RITA | DARPA |
| 525 Brooks Road | 675 N. Randolph St |
| Rome NY 13441-4505 | Arlington VA 22203-2114 |

**10. SPONSOR/MONITOR'S ACRONYM(S)**
RI

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
AFRL-RI-RS-TR-2022-068

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

The increasing complexity of software systems makes the rapid propagation of requirement changes into the design and implementation code very problematic. The goal of the Intent-Driven Adaptive Software program was to develop technologies that assist developers in making changes to requirements and automatically propagating those changes to the design and implementation of software systems. The Model-based Intent-Driven Adaptive software project developed a vision for a comprehensive technology to achieve this goal by developing and implementing two components of that vision: a program specification and synthesis tool, and a domain-specific language and generators for the rapid configuration and adaptation of service-based architectures. These two results can serve as a foundation for the future implementation of the vision.

**15. SUBJECT TERMS**
Model Design Language, deferred concretization, formal specification and program synthesis, domain-specific modeling languages

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** |
|---|---|---|---|---|
| **a. REPORT** U | **b. ABSTRACT** U | **C. THIS PAGE** U | SAR | 23 |

| | |
|---|---|
| **19a. NAME OF RESPONSIBLE PERSON** STEVEN L. DRAGER | **19b. PHONE NUMBER** *(Include area code)* N/A |

## TABLE OF CONTENTS

## List of Figures

## 1 SUMMARY

The increasing complexity of software systems makes the rapid propagation of requirements changes into the design and implementation code very problematic. The goal of the Intent-Driven Adaptive Software (IDAS) program was to develop technologies that assist developers in making changes to requirements and automatically propagating those changes to the design and implementation of software systems. The Model-based Intent-Driven Adaptive Software (MIDAS) project developed a vision for a comprehensive technology to achieve this goal while developing and implementing two components of that vision: a program specification and synthesis tool, and a domain-specific language and generators for the rapid configuration and adaptation of service-based architectures. These two results can serve as a foundation for the future implementation of the vision.

## 2 INTRODUCTION

The complexity of software systems is increasing continuously, in parallel with the need for their rapid update and adaptation to changing requirements. The goal of the project was to develop a new approach to evolutionary software development and deployment that extends the results of model-based software engineering and provides an integrated, end-to-end framework for building software that is focused on growth and adaptation. The envisioned technology was based on the concept of a 'Model Design Language' (MDL) that supports the expression of the developer's objectives (the 'what'), intentions (the 'how'), and constraints (the 'limitations') related to the software artifacts to be produced. The 'models' represented in this language are called the 'design models' for the software artifact(s) and they encompass more than what we express today in software models. Software development is considered as a continuous process, as in the Development and Operations (DevOps) paradigm, where the software is undergoing continuous change, improvement, and extension; and our goal was to build the tools to support this. The main idea is that changes in the requirements will result in the designer/developer making changes in the 'design model' that will result in changes in the generated artifacts, or changes in the target system, at run-time, as needed. Such tool support is essential for developers as expensive, manual rework cannot be avoided without it.

The project has resulted in two prototype software development tools: (1) a program specification and synthesis environment based on a novel formal language, called Syntheto, as a front-end to a well-established program transformation system, and (2) a model-driven development tool based on a domain-specific language, called Open Unmanned Systems Autonomy Services–Model Driven Engineering (OpenUxAS-MDE), that enables the rapid configuration and adaptation of a software system for Unmanned Aerial Vehicle (UAV) Command and Control (C2) applications. The two languages are sub-languages of a future, more comprehensive Model Design Language and could serve as a starting point for an end-to-end tool-suite.

## 3 METHODS, ASSUMPTIONS, AND PROCEDURES

### 3.1 Challenges

The core challenge for the program was that of rapid adaptation of software systems to changing requirements. The problem is especially acute in complex, mission- and safety-critical applications that require extensive verification a nd v alidation b efore d eployment (or update) in the field. O f ten a s m all c h ange i n r e quirements l e ads t o m a jor, widespread changes in the implementation, i.e. the code base. Currently there is very little tool support for facilitating changes in an effective a nd p roductive manner.

Central to this problem are design and implementation decisions that are made early in the system development. Such design decisions (e.g. choosing a specific i mplementation for a critical data structure, without encapsulating it into an abstract data type) hamper down-stream adaptation of the code when the choice cannot be valid anymore. Hence, deferred concretization is needed, whereby low-level design choices are made late in the development process. Good software engineering practices, with disciplined developers can address this problem, but cannot completely eliminate it, especially on very large-scale systems.

The project described in this report aimed at solving this problem using two approaches: (1) formal specification a nd p rogram s ynthesis, a nd ( 2) d omain-specific mo de ling languages (DSML). Arguably, the approaches offer a l ternative p a ths t o s o lve t h e r a pid adaptation problem. Formal specification followed by (semi-)automatic synthesis facilitates productivity increases, because (1) changing the specification is simpler than changing a code base, and (2) the synthesis tool can not only (re-)verify the changed specification but c an a lso g enerate the implementation. On the other hand, a DSML can represent a complete system architecture with all variation points controlled by parameters, and the entire, detailed configuration of the architecture can be automatically generated from the DSML models.

However, both approaches have their own challenges. Existing formal specification and program synthesis approaches often require special skills and the knowledge of un-common languages, like Lisp. The use of domain-specific l anguages i n s oftware d evelopment i s (still) not a widespread practice; although their effectiveness a r e w e ll-recognized [4].The project described below addressed these issues by developing new technologies, as described below.

### 3.2 Overall approach

The overall approach and technical vision for the project is summarized in Figure 1. The lower part of the figure h i ghlights t h e c u rrent p r actice o f M o del-based S o ftware Engineer-ing (MBSE), while the full diagram shows the concept for the model-based, intent-driven adaptive development of software. Central to MIDAS is a ‘Model Design Language’: a higher-level design language that supersedes the existing MBSE approach in several ways, as listed below.

The goal of MDL is to represent as much knowledge as practically feasible about the software system, including its requirements, its environment, its target platform, as well as the process for synthesizing it (or its parts). This happens through a set of *models*, expressed either formally (in a precisely specified modeling language) or informally, in natural language.

A key feature of this language is that such models are inter-linked, and such links (i.e., dependencies) are tracked and maintained by tools. Listed below are the various models in MDL.

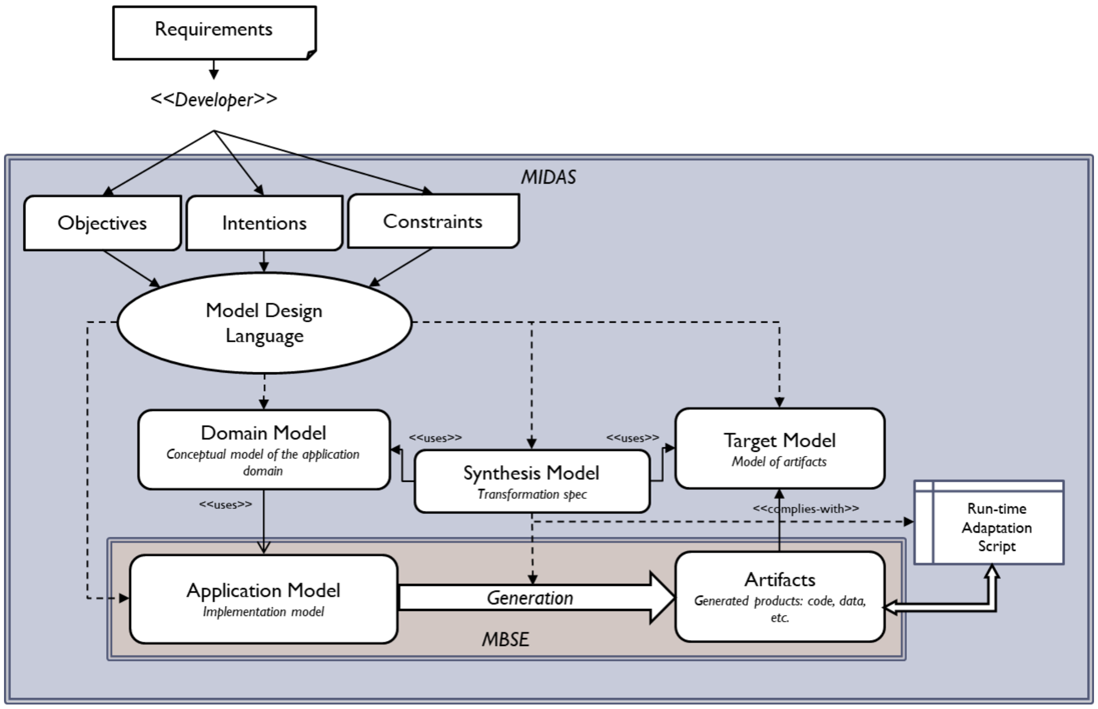


**Figure 1.** Overall technical approach.

1. Domain model (DM). The DM makes the modeling of the domain concepts explicit. Such models describe the conceptual structure of the 'world' the software artifact is operating in. Domain models are represented as hierarchically organized concepts that are linked to each other via various relationships, and have attributes capturing salient properties. These models capture the designer's understanding of the domain, without any relationship to implementation. The DM is built by developers who understand the domain well.
2. Application model (AM). The AM extends the traditional application model with (a) relating AM elements to DM elements, and (b) linking AM elements to explicitly expressed intentions and constraints. Note that not all DM elements are expected to have a corresponding AM, and vice versa, but all AM elements should be related to an intention. The AM is built by designers and developers.
3. Target model (TM). The TM makes the modeling of the target domain explicit. It is envisioned that the 'target domain' is going to be a software 'platform' for which compilable source code (or other data artifacts, like Makefiles) should be generated. Therefore, the TM will include models that are associated with parametrized code (or data) templates that will be filled out during the generation process. The TM is built

by developers who understand the target domain and its intricacies well, but have experience with writing code templates and generation. Note that the TM may also include code templates for unit and system level tests.

4. Synthesis model (SM). The SM is for the implicit or explicit representation of how AM elements are mapped into code and data artifacts (that are compliant with the TM elements). It is envisioned that the SM will be built by skilled developers who are 'synthesis engineers'. Some SMs are prefabricated and re-used across many applications (e.g. a synthesis tool that generates code from a finite state machine model). Note that the SM may involve generation of many different artifacts: production code, data, test code, etc. The SM can also be used to generate changes to a running system: some of the requirement changes can be addressed by changing the configuration or properties of the deployed, active system. This could be facilitated by the 'Run-time Adaptation Script' shown on the right of Figure 1, which assumes a suitable adaptive software platform.
5. Objectives, Intentions, and Constraints (OIC). These models represent, at a high level of abstraction 'what' the system is expected to do, 'how' it is expected to do it, and under what 'restrictions'. These models may be somewhat unstructured with respect to the other models of the system, but they are to be linked to the other models, to provide traceability and an opportunity for analysis.

In summary, the envisioned overall approach relies on highly interlinked models of the objectives, intentions, constraints, domain, application, target platform, and synthesis process. The product of the development process is synthesized and/or instantiated executable code, plus other artifacts necessary for the further compilation and/or deployment of the system.

Two specific c o mponents o f t h e a b ove o v erall a p proach h a ve b e en d e veloped a n d implemented in the course of the project: (1) a formal methods based program synthesis environment, and (2) a domain-specific l anguage b ased m odel-driven d evelopment tool.

### 3.2.1 Program Synthesis in an Integrated Development Environment

The concept of deferred concretization that inspired the IDAS program is well addressed by the classic idea of program development by transformational stepwise refinement. However, this classic idea has been so far realized only in tools that required specialized expertise. An example is Kestrel's Automated Program Transformations (APT) toolkit [6], which is based on the 'A Computational Logic for Applicative Common Lisp' (ACL2) theorem prover [5] and requires users to be fluent i n t h e A C L2 t h eorem p r over. T h us, a m a jor c h allenge of this project has been to make this capability available to developers with less specialized expertise.

The approach of the project has been to take APT [6] as a starting point and making it more accessible. First, noticing that the ACL2 language, even leaving the theorem proving capabilities aside, is somewhat unfamiliar to developers due to the Lisp prefix syntax and the lack of static typing, a novel front-end language: Syntheto was designed and implemented with strong static typing and more infix syntax. There is a bi-directional translation between

Syntheto and ACL2. Syntheto can express not only ACL2 specifications, but also the invocation of APT transformations, thus providing complete coverage of the software synthesis process.

Syntheto's strong static typing helps automate certain classes of ACL2 theorems that capture type-like properties in ACL2's untyped language but that nonetheless require ACL2-specific expertise to formally verify. In other words, it is possible to automate a large number of (relatively simple) proof obligations, sparing users from explicitly dealing with them.

Another area where ACL2 and APT differ from more widely used programming languages is in the availability of Integrated Development Environments (IDEs). Good IDEs can make users much more productive, by enabling them to carry tasks faster and, more importantly, by providing immediate feedback and suggestions. Thus, an IDE for Syntheto was developed; the IDE provides a notebook interface, which is particularly useful for program transformations.

### 3.2.2 Model-driven Development for a Service-based Application Framework

Open Unmanned Systems Autonomy Services (OpenUxAS) [8] is an extensible software framework for mission-level autonomy for teams of unmanned systems, with a focus on UAVs. The framework includes several software services for high-level control of UAVs, route planning, plan execution, etc. that communicate via a software bus. The configuration of the software framework is done via a collection of eXtensible Markup Language (XML) files that supply parameters to the services, as well as determine the overall software architecture of the assembled services. The framework is interfaced to a multi-vehicle simulation environment Open Aerospace Multi-agent Simulation Environment (OpenAMASE) [7] that provides immediate feedback to the developer when experimenting with the framework. The primary use case for the framework is that of the development and testing of control and coordination algorithms for UAVs, with the eventual goal of executing these algorithms on-board the vehicles.

OpenUxAS is an example for a medium-complexity extensible software framework that can be configured for various missions. Changes in mission requirements need to be propagated to configuration changes, possibly indicating the need for new services. The current approach of using a suite of XML files for configuration is complex and error-prone, because one requirement change can lead to cascading changes in multiple files. Debugging configuration problems is difficult, especially if configuration information is implicit, in the implementation code itself. In a message-based architecture, like OpenUxAS, services publish and subscribe to messages that facilitate complex service interactions. But what message a service publishes and subscribes to is often hand-coded in the implementation (in C++, in this case). This leads to a few problems: (1) to understand how a configured instance of the framework works (for a specific mission), one needs to read the code base and manually map out these interactions and discover what messages are never generated and/or never consumed, (2) if a new service is to be added (or an existing one modified), a somewhat complex message handling loop needs to be developed (or an existing one needs to be updated). Clearly, a better approach is needed.

A better approach is to use a simple domain-specific language that supports:

- The generation of all related XML configuration files for configuring the framework to support specific missions.
- The checking of the framework configuration for architectural completeness, i.e. that the message publishers and consumers are 'aligned': every message produced by at least one service, every message consumed is produced by at least one service.
- The generation of skeleton code for service implementation that can be extended with 'business logic'. As the service specification may change frequently during development, the generation should be done in such a way that existing additions are not lost during the re-generation process.

In the course of the project a Domain Specific Language (DSL) has been developed that solve the above problems. This language and the corresponding generators are a concrete example for a synthesis model, mentioned above.

## 4 RESULTS AND DISCUSSION

### 4.1 Syntheto

#### 4.1.1 Language summary

Syntheto was designed and developed as a novel language that provides a front-end to ACL2 and APT. Syntheto is a statically strongly typed language, whose type system has the following features:

- Primitive types such as booleans, integers (of arbitrary size), characters, etc.
- Collection types for finite sets, sequences, and maps.
- An option type for either a value of a type or a special marker for none.
- User-defined product types (i.e. records).
- User-defined sum types (i.e. disjoint unions, such as Rust-style enums).

Since ACL2 is a functional language, so is Syntheto: it has expressions, but not statements as such. However, certain expression construct have a surface syntax that looks like statements. Syntheto expressions include:

- Boolean, character, string, and integer literals.
- Variables.
- A range of unary and binary expressions.
- Conditional expressions.
- Function calls.
- Bindings of variables to expressions. (The surface syntax of bindings resembles variable assignment statements.)
- Product type constructions, deconstructions, and updates.
- Sum type constructions, deconstructions, updates, and tests.

At the top level, Syntheto supports the following constructs:

- Type definitions, which may be singly and mutually recursive.
- Function definitions, which may be singly and mutually recursive.
- Functions defined with universal and existential quantifiers.

- Function specifications, which are second-order predicates.
- Theorems, which express expected properties of the defined and specified functions.
- Transformation calls.

Function specifications, as mentioned, are second-order predicates, which characterize (the input/output behavior of) the acceptable functions to be synthesized. This specification approach is based on [2], and is very general. However, in some cases a user can specify a function by defining it, in a clear but possibly inefficient way: transformations can be used on defined functions to turn them into more efficient versions.

The transformation calls supported by Syntheto provide an interface to an initial selection of APT transformations. More APT transformations could be supported in the future.

The Appendix contains the Syntheto grammar expressed using the XText[1] syntax.

### 4.1.2 Interface with ACL2

The types in the Syntheto type system described above are represented in ACL2 using a library to emulate structured types in ACL2's untyped language [9]. Syntheto expressions are represented as ACL2 expressions, and Syntheto functions are represented as ACL2 functions; this is also the case for the second-order predicates mentioned above, despite the fact that ACL2's language is first-order, using a library to represent (limited) second-order in ACL2 [3].

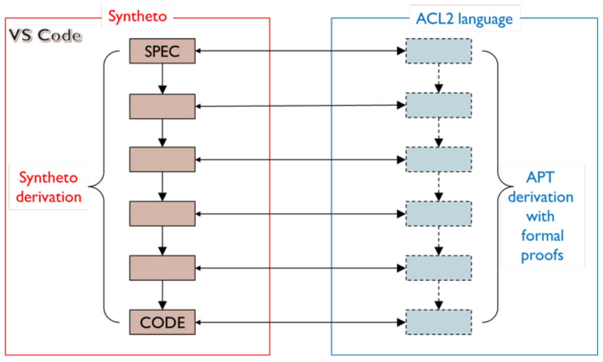


**Figure 2.** Syntheto-ACL2 interoperation.

Syntheto transformations correspond to APT transformations. A challenge is that, besides the forward translation from Syntheto to ACL2, needed to represent all the Syntheto constructs in ACL2, it was necessary to develop a backward translation from ACL2 to Syntheto, for at least Syntheto expressions and functions. The reason is that, when a transformation is applied to a function, it yields a new function (in some cases multiple functions, but this is not important for the present discussion), but it is an ACL2 function, because the Syntheto transformation call is turned into an ACL2 transformation call that operates on the ACL2 representation of the the Syntheto target function. The resulting ACL2 function must be translated to a Syntheto function, because the whole point of Syntheto is to "hide" the ACL2 details and provide a Syntheto-centric functionality. In other words, an "illusion" is created that transformations operate directly on Syntheto, even though there is a round-trip to ACL2 behind the scenes. Figure 2 shows the interoperation of Syntheto and the ACL2 system. The challenge with the backward translation is that ACL2 is untyped while Syntheto is strongly typed: this means that type information must be reconstructed in ACL2. The issue does not arise in the forward translation, because it is easy to "forget" types than to "reconstruct" them.

The ACL2 Bridge is used to realize a bidirectional communication with ACL2. The ACL2 Bridge is a tool that enables starting up an ACL2 process that acts as a server listening to a socket. The approach taken here is to start up this process, load all the Syntheto support into it, and have the IDE (described in more detail below) exchange messages through the socket.

### 4.1.3 IDE with Notebook Interface

A challenge in developing an IDE for Syntheto is that an ACL2 process must interface with a Java process; this is because the IDE is written in Java, but the same would apply to IDEs written in other languages (there are no modern IDEs written in ACL2). The issue is one of inter-language interfacing: the ACL2 Bridge provides a working approach to handle the connection, but there are Java data structures on one side and ACL2 data structures on the other side. Thus, Java code was developed to perform bidirectional translations between Java and ACL2 data structures; these are only a (syntactical) portion of the semantic bidirectional translation between Syntheto and ACL2.

The project has developed an "extension" for the Visual Studio Code (VS Code), an open source Microsoft IDE, that provides an integration with a conventional development environment. The VS Code extension provides the following services:

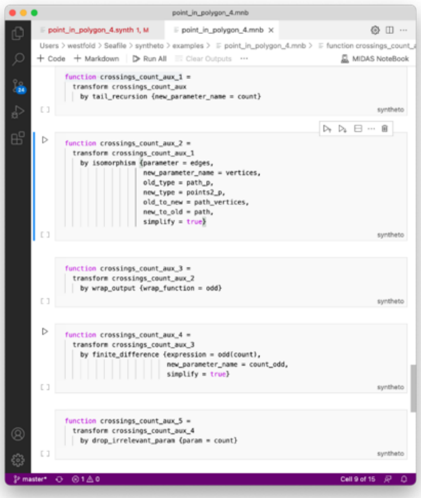
- Syntax-driven editing of Syntheto specifications. This editor enforces the syntactical rules of Syntheto on the user input, and performs simple semantic checks.
- Transforming the specifications via multiple transformation steps into ACL2 syntax and sending it to the ACL2 Bridge.
- Receiving responses from the Bridge, separating them into error messages and synthesized code (in ACL2/Lisp), translating the latter back into Syntheto syntax, and rendering it.

The first and the last step uses a 'notebook' interface that enables a workflow in the style of incremental editing, submission, and response viewing. Figure 3 shows an example screen of the interface.

### 4.1.4 Syntheto implementation

The Syntheto environment will be available as follows:

- The backend at `https://github.com/acl2/acl2`, the ACL2 community library, under the directory `https://github.com/acl2/acl2/tree/master/books/kestrel/syntheto`.
- The frontend at `https://github.com/KestrelInstitute/syntheto-frontend`.


point_in_polygon_4.mnb
point_in_polygon_4.synth 1, M
point_in_polygon_4.mnb
Users > westfold > Seafile > syntheto > examples > point_in_polygon_4.mnb > function crossings_count_
+ Code
+ Markdown
Run All
Clear Outputs
MIDAS NoteBook
syntheto
master*
Cell 9 of 15

```
function crossings_count_aux_1 =
  transform crossings_count_aux
    by tail_recursion {new_parameter_name = count}
```

```
function crossings_count_aux_2 =
  transform crossings_count_aux_1
    by isomorphism {parameter = edges,
                    new_parameter_name = vertices,
                    old_type = path_p,
                    new_type = points2_p,
                    old_to_new = path_vertices,
                    new_to_old = path,
                    simplify = true}
```

```
function crossings_count_aux_3 =
  transform crossings_count_aux_2
    by wrap_output {wrap_function = odd}
```

```
function crossings_count_aux_4 =
  transform crossings_count_aux_3
    by finite_difference {expression = odd(count),
                          new_parameter_name = count_odd,
                          simplify = true}
```

```
function crossings_count_aux_5 =
  transform crossings_count_aux_4
    by drop_irrelevant_param {param = count}
```

**Figure 3.** Syntheto notebook interface.

## 4.2 OpenUxAS-MDE

The OpenUxAS-MDE language was developed to assist in the generation of all XML configuration files for the OpenUxAS framework. The modeling language has the following features:

- Message specifications represent message data types. The named message types then can be used in the service definitions.
- Service definitions represent the parameters, and the messages produced and consumed by the service.
- Vehicle definitions specify vehicle types and their parameters, and vehicle declarations specify concrete instances of vehicles to be used in the mission.
- Task definitions specify task types and their parameters, and task declarations specify concrete instances of tasks to be performed in a the mission. Scenario specifications assemble task declarations and represent a complex scenario, with multiple tasks.
- Application specifications (that incorporate all the above elements) provide the complete specification of a mission.

The above specifications and definitions are used in generation process as follows.

- Message specifications are used in generating XML definitions for the supporting messaging framework.
- Service definitions are used in generating service skeletons for implementation (in C++).
- Vehicle definitions and declarations are used to generate configuration files for the simulator, as well as the software framework services.
- Task definitions and declarations, and scenario specifications are used to configure the simulation framework.

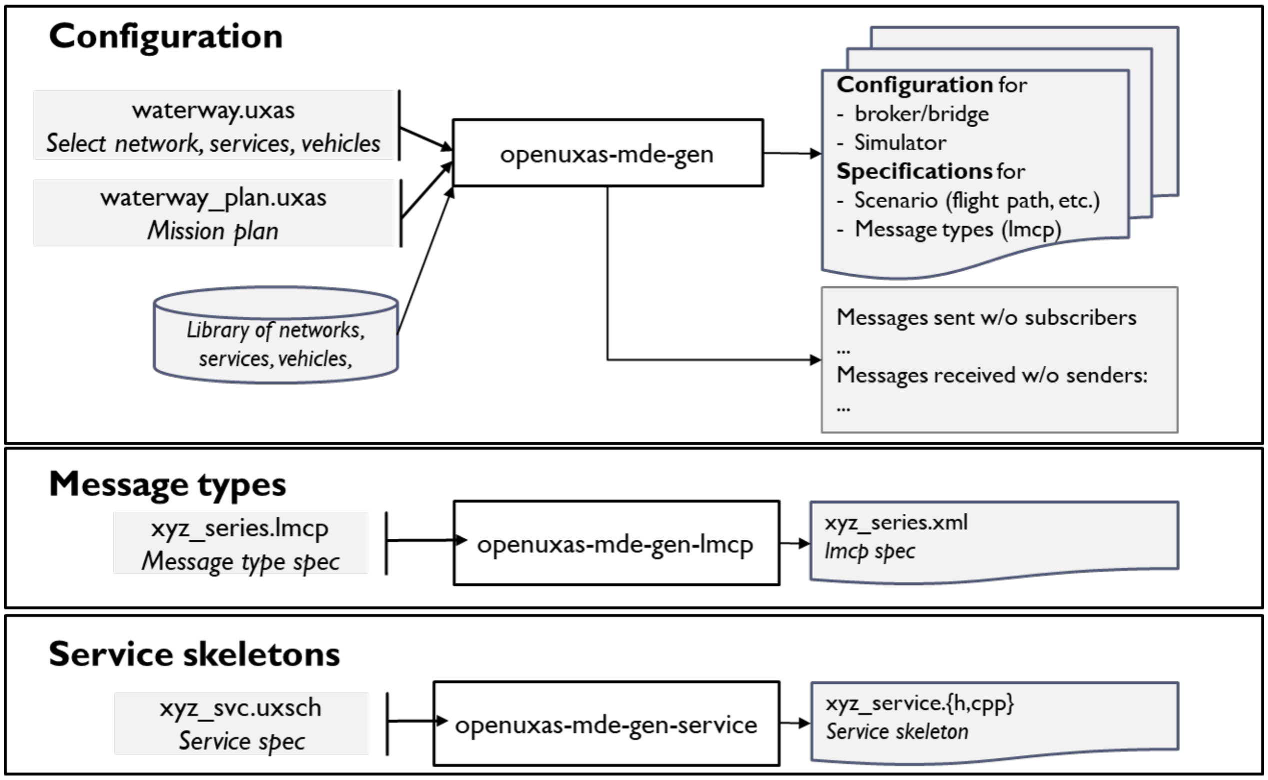


**Figure 4.** OpenUxAS-MDE DSL and generators.

- Application specifications (through the inclusion of the previous elements) are used to generate all the configuration files.

The textual definitions, declarations and specifications expressed in the DSL is the single source of truth for configuring all aspects of the mission, both the simulation, as well as the software framework. Changes in the inputs are trivially propagated to the generated configuration files. The DSL is more compact than the XML files, on one example application about 100 lines of DSL code resulted in 1285 lines of XML.

The DSL has been used to develop other components beyond XML generators. A code generator has been developed that produces the service skeleton implementation code. This code has 'protected sections' where developer-provided, business logic code can be added. These additions are preserved across runs of the generator such that code added is never lost. Another tool analyzes the application architecture and matches up the producers and consumers of messages. The developer is informed about messages that are expected but not produced, and about messages produced but not consumed. This check allows the early detection of architectural mismatch. Figure 4 shows the various elements of the DSL and the generated artifacts.

### 4.2.1 OpenUxAS-MDE implementation

The implementation is available from `https://github.com/MbIDAS/OpenUxAS-MDE`.

## 5 CONCLUSIONS

The project has provided insights into developing complex software systems that could undergo frequent adaptation due to changes in requirements.

Regarding formal specifications and program synthesis the observation is that the front-end language should be similar to languages with which average developers are already familiar. Languages with strong typing (unlike Lisp), with infix syntax, well-chosen keywords and strict syntactical structure are highly advantageous. Formal tools need to fit seamlessly into conventional IDE-s, and attention needs to be paid to the presentation and user interaction. Translating error messages and synthesis results coming back from the synthesis tool into a familiar form is essential for acceptance.

Regarding the use of DSLs, the lesson learned is that developing a DSL *after* a complete software framework has been designed requires significant effort, and thus there are advantages to co-developing the DSL with the framework. DSLs can provide productivity improvements not only by generating code artifacts, but by supporting analysis on the final application software. DSLs specifications are higher-level, mostly declarative programs that can potentially influence many components of a software system. Explicit traceability from specifications to derived (or dependent) artifacts is essential so that the impact of changes becomes clear.

In summary the project has produced two artifacts that can contribute to the overall approach outlined earlier. However, further research and development is needed, especially in the area of traceability from requirements to implementation and back, as well as the deeper integration of model-based, formal methods based, and conventional development techniques.

## 6 REFERENCES

[1] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, page 7, 2008.

[2] Alessandro Coglio. Pop-refinement. *Archive of Formal Proofs*, July 2014. `http://afp.sf.net/entries/Pop_Refinement.shtml`, Formal proof development.

[3] Alessandro Coglio. Second-order functions and theorems in ACL2. In *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2015)*, pages 17–33, October 2015.

[4] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. Dsls: the good, the bad, and the ugly. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 791–794, 2008.

[5] Matt Kaufmann and J Strother Moore. The ACL2 theorem prover: Web site. `http://www.cs.utexas.edu/users/moore/acl2`.

[6] Kestrel Institute. APT (Automated Program Transformations). `http://www.kestrel.edu/home/projects/apt`.

[7] Derek Kingston, Steven Rasmussen, and Laura Humphrey. Automated uav tasks for search and surveillance. In *2016 IEEE Conference on Control Applications (CCA)*, pages 1–8, 2016.

[8] Steven Rasmussen, Derek Kingston, and Laura Humphrey. A brief introduction to unmanned systems autonomy services (uxas). In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 257–268, 2018.

[9] Sol Swords and Jared Davis. Fix your types. In *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2015.

## A APPENDIX

### A.1 Syntheto Grammar

```
grammar edu.vanderbilt.isis.midas.Syntheto with org.eclipse.xtext.common.
    Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate syntheto "http://www.vanderbilt.edu/isis/midas/Syntheto"

Program returns Program:

TopLevelConstruct:
     TypeDefinition | FunctionDefinition | FunctionSpecfication | Theorem |
    TransformDefinition;

TypeDefinition:
     SumTypeDefinition | ProductTypeDefinition | SubTypeDefinition;

Pragmadirectives:
     '%' (BridgeConnectionDirective | ProcessModelDirective) (';')?;

BridgeConnectionDirective:
     'use' 'acl2' ('host' | '@') host=Ipaddress ('port' | ':') port=INT;

Ipaddress hidden():
     (first=INT '.' second=INT '.' third=INT '.' fourth=INT);

ProcessModelDirective:
     check?=('check');

PrimitiveType:
     boolean?=('bool') | char?=('char') | int?=('int') | string?='string';

TypeElement returns TypeElement:
     Option;

Option returns TypeElement:
     Sequence | {Option} ('opt' '<' element=TypeElement '>');

Sequence returns TypeElement:
     Set | {Sequence} ('seq' '<' element=TypeElement '>');

Set returns TypeElement:
     Map | {Set} ('set' '<' element=TypeElement '>');

Map returns TypeElement:
     PrimaryTypeElement | {Map} ('map' '<' domain=TypeElement ',' range=
    TypeElement '>');
```

```
PrimaryTypeElement returns TypeElement:
    {PrimaryTypeElement} (primary=PrimitiveType | typeref=[TypeDefinition]);

ProductTypeDefinition:
    {ProductTypeDefinition} (struct?='struct')? productID=ID ('{' element+=
   TypedVariable (',' element+=TypedVariable)*
    ('|' invariant=Expression)? '}')?;

TypedVariable:
    name=ID ':' type=TypeElement;

Alternative:
    {Alternative} (product=ProductTypeDefinition);

SumTypeDefinition:
    'variant' name=ID '{' alternatives+=ProductTypeDefinition (','
   alternatives+=ProductTypeDefinition)* '}';

SubTypeDefinition:
    {SubTypeDefinition} 'subtype' name=ID '{' element=TypedVariable '|'
   invariant=Expression '}';

FunctionSpecfication:
    ('specification') name=ID '(' 'function' funcName=ID '(' (param+=Param (',
   ' param+=Param)*)? ')' ('returns' '('
    returnlist+=Param (',' returnlist+=Param)* ')')? ')' '{' expr=
   BlockStatement '}';

Theorem:
    'theorem' name=ID (('forall') '(' foralltag+=TypedVariable (',' foralltag
   +=TypedVariable)* ')')? ('|')?
    expression=Expression;

FunctionDefinition:
    ('function') name=ID '(' (param+=Param (',' param+=Param)*)? ')'
    ('assumes' assumes=Expression)? ('returns' '('
    returnlist+=Param (',' returnlist+=Param)* ')')? (('ensures' ensures=
   Expression)?) ('measure' measure=Expression)?
    ('{' ((expr=BlockStatement)) '}');

TransformDefinition:
    ('function') name=ID '=' 'transform' transformed_fn=[TransformableType] '
   by' transformation=transformation_type;

TransformableType:
    FunctionDefinition | TransformDefinition;

transformation_type:
    tail_recursion | remove_cdring | flatten_param | isomorphism |
   finite_difference | drop_irrelevant_param
```

```
    | wrap_output | rename_param |  simplify;

finite_difference:
    'finite_difference' '{' 'expression' '=' expression=Expression ',' '
   new_parameter_name' '=' new_parameter_name=ID ','
    'simplify' '=' simplify=BooleanLiteral '}';

flatten_param:
    'flatten_param' '{' 'old' '=' old=ID ',' 'new' '=' '[' newlist+=ID ',' (
   newlist+=ID)+ ']' '}';

wrap_output:
    'wrap_output' '{' 'wrap_function' '=' identifier=ID '}';

drop_irrelevant_param:
    'drop_irrelevant_param' '{' 'param' '=' identifier=ID '}';

tail_recursion:
    'tail_recursion' '{' 'new_parameter_name' '=' identifier=ID '}';

isomorphism:
    'isomorphism' '{' 'parameter' '=' parameter=ID ',' 'new_parameter_name' '=
   ' new_parameter_name=ID ',' 'old_type' '='
    old_type=ID ',' 'new_type' '=' new_type=ID ',' 'old_to_new' '=' old_to_new
   =ID ',' 'new_to_old' '=' new_to_old=ID ','
    'simplify' '=' simplify=BooleanLiteral '}';

rename_param:
    'rename_param' '{' 'old' '=' old=ID ',' 'new' '=' new=ID '}';

simplify:
    {simplify}'simplify';

remove_cdring:
    'remove_cdring' '{' 'simplify' '=' simplify=BooleanLiteral '}';

Param:
    tag=TypedVariable;

BlockStatement:
    LetExpression | IfExpression | WhenExpression | UnlessExpression |
   CondExpression | BlockExpression;

BlockExpression:
    (isreturn?='return')? expr=Expression ';';

CondExpression:
    'cond' '{' branches+=CondBranches+ '}';

CondBranches:
    test=Expression '{' thenexpr=BlockStatement '}';

LetExpression:
    ('let' (vars+=TypedVariable (',' vars+=TypedVariable)*) '=' first=
   BlockStatement second=BlockStatement);

WhenExpression:
    'when' '(' test=Expression ')' '{' thenexpr=BlockStatement '}' elseexpr=
   Elseexpr;

UnlessExpression:
    'unless' '(' test=Expression ')' '{' thenexpr=BlockStatement '}' elseexpr=
   Elseexpr;

IfExpression:
    {IfExpression} 'if' '(' test=Expression ')' (('{' (thenexpr=BlockStatement
   ) '}')) ('else' elseexpr=Elseexpr);

Elseexpr:
    {Elseexpr} '{' (elseexpr=BlockStatement) '}';

Expression returns Expression:
    Implies_expr;

Implies_expr returns Expression:
    Or_expr ({Implies_expr.left=current} (implies?='==>' | implied?='<==' |
   iff?='<=>') right=Or_expr)*;

Or_expr returns Expression:
    And_expr ({Or_expr.left=current} '||' right=And_expr)*;

And_expr returns Expression:
    Compare_expr ({And_expr.left=current} '&&' right=Compare_expr)*;

Compare_expr returns Expression:
    Math_expr ({Compare_expr.left=current} (geq?='>=' | leq?='<=' | eq?='==' |
    neq?='!=' | gt?='>' | lt?='<')
    right=Math_expr)*;

Math_expr returns Expression:
    Multiplication (({Plus.left=current} '+' | {Minus.left=current} '-') right
   =Multiplication)*;

Multiplication returns Expression:
    Unary_expr (({Multi.left=current} '*' | {Div.left=current} '/' | {Modulo.
   left=current} '%') right=Unary_expr)*;

Unary_expr returns Expression:
    PrimaryExpression | {Unary_expr} op=OpUnary operand=Unary_expr;
```

```
OpUnary:
    not?="!" | negation?="-";

PrimaryExpression returns Expression:
    '(' Expression ')' | {LiteralValue} value=Literal | {FunctionCall} func=[
   FunctionDefinition] ('(' (args+=Expression
    (',' args+=Expression)*)? ')') | {VariableAssignment} (variable=
   ElementTagQualifier);

ElementTagQualifier returns ElementTagQualifier hidden():
    child=[Subelement] (({ElementTagQualifier.left=current} '.') (subelement=[
   Subelement]))*;

Subelement:
    ProductTypeDefinition | TypedVariable | SumTypeDefinition;

Literal:
    BooleanLiteral | NumberLiteral | StringLiteral | ProductLiteral |
    SingleValueBuiltins | TwoValueBuiltins ;

TwoValueBuiltins:
    {TwoValueBuiltins} builtin=('member' | 'remove_first' | 'add' | 'append')
   '('
    operand =Expression ','element =Expression ')';

SingleValueBuiltins:
    builtin=('length' | 'is_empty' | 'first' | 'rest' | 'last') '(' element=
   Expression ')' | empty?='empty' ('(' ')')? ;

BooleanLiteral:
    {BooleanLiteral} ('false' | isTrue?='true');

NumberLiteral:
    {NumberLiteral} value=INT;

StringLiteral:
    {StringLiteral} value=STRING;

ProductLiteral:
    {ProductLiteral} (product=[ProductTypeDefinition]) '(' assignment+=
   ProductAssignment (','
    assignment+=ProductAssignment)* ')';

ProductAssignment:
    left=[TypedVariable] '=' right=Expression;

SeqLiteral:
    {SeqLiteral} '[' (elements+=Expression (',' elements+=Expression)*) ']';
```

## LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

| | |
|---|---|
| ACL2 | A Computational Logic for Applicative Common Lisp |
| AM | Application model |
| APT | Automated Program Transformations |
| C2 | Command and Control |
| DevOps | Development and Operations |
| DM | Domain model |
| DSL | Domain-specific Language |
| DSML | Domain-Specific Modeling Language |
| IDAS | Intent-Driven Adaptive Software |
| IDE | Integrated Development Environment |
| MBSE | Model-based Software Engineering |
| MDL | Model Design Language |
| MIDAS | Model-based Intent-Driven Adaptive Software |
| OIC | Objectives, Intentions, and Constraints |
| OpenAMASE | Open Aerospace Multi-agent Simulation Environment |
| OpenUxAS | Open Unmanned Systems Autonomy Services |
| OpenUxAS-MDE | Open Unmanned Systems Autonomy Services–Model Driven Engineering |
| SM | Synthesis model |
| TM | Target model |
| UAV | Unmanned Aerial Vehicle |
| VS Code | Visual Studio Code |
| XML | eXtensible Markup Language |