Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support

Thomas Degueule
Centrum Wiskunde & Informatica (CWI)
Amsterdam, Netherlands
degueule@cwi.nl

Tanja Mayerhofer
TU Wien
Vienna, Austria
mayerhofer@big.tuwien.ac.at

Andreas Wortmann
RWTH Aachen University
Aachen, Germany
wortmann@se-rwth.de

***Abstract*—Domain-specific languages (DSLs) improve engineering productivity through powerful abstractions and automation. To support the development of DSLs, the software language engineering (SLE) community has produced various solutions for the systematic engineering of DSLs that manifest in language workbenches. In this paper, we investigate the applicability of the language workbenches GEMOC STUDIO and MONTICORE to the MDETools'17 ROVER challenge. To this effect, we refine the challenge's requirements and show how GEMOC STUDIO and MONTICORE can be leveraged to engineer a Rover-specific DSL by reusing existing DSLs and tooling of GEMOC STUDIO and MONTICORE. Through this, we reflect on the SLE state of the art, detail capabilities of the two workbenches focusing particularly on language reuse support, and sketch how modelers can approach ROVER programming with modern modeling tools.**

***Index Terms*—Software Language Engineering; Language Reuse; Language Workbenches; Rover; GEMOC Studio; MontiCore**

I. INTRODUCTION

Domain-specific languages (DSL) improve software engineering productivity and system quality through powerful domain abstractions and increased automation. To support DSL development, the software language engineering (SLE) [1] community has proposed various solutions for the systematic development, use, deployment, and maintenance of DSLs and related tools. These solutions manifest in language workbenches [2] that assist language engineers in all DSL engineering phases.

In this paper, we present the language workbenches GEMOC STUDIO [3] and MONTICORE [4] to demonstrate and compare their language engineering support on the MDETools'17 ROVER challenge [5]. GEMOC STUDIO is based on the Eclipse Modeling Framework (EMF) [6] and provides support for implementing executable DSLs and supporting tooling. This includes meta-programming approaches for defining DSL interpreters and generic components for efficiently developing DSL tools including model animators and debuggers. MONTICORE offers support for implementing textual DSLs with context-free grammars and provides a powerful infrastructure for developing analyzers, transformations, and code generators.

We present implementations of the RASPIROVER DSL, a DSL for defining the architecture and behavior of Raspberry Pi operated rovers, with both language workbenches. In these implementations, we aimed at reusing as many DSLs as possible to ease and speed up the DSL engineering process for the RASPIROVER DSL. We show *(i)* how to develop DSLs for software-controlled rovers with both language workbenches by reusing existing DSLs, *(ii)* the tool support that both workbenches offer to engineers for using the developed DSLs, and *(iii)* the individual strengths of GEMOC STUDIO and MONTICORE with regard to their support for engineering DSLs and supporting tools. We put a particular emphasis on language reuse [7] techniques offered by both language workbenches to facilitate DSL engineering.

In the following, Section II details the ROVER challenge studied in this paper. Section III and Section IV describe the solutions developed with GEMOC STUDIO and MONTICORE, respectively. Section V compares the language engineering support provided by the two studied language workbenches focusing particularly on language reuse. Finally, Section VI concludes.

II. ROVER LANGUAGE ENGINEERING CHALLENGE

Both GEMOC STUDIO and MONTICORE have been used in the past for implementing DSLs facilitating the development of software-controlled rovers. The ARDUINOML language [8] has been implemented using GEMOC STUDIO for modeling the architecture and behavior of Arduino-based systems, such as Arduino-operated rovers. The MONTIARCAUTOMATON (MAA) [9] architecture description language has been implemented using MONTICORE for modeling the architecture and behavior of robotics applications, such as the architecture and behavior of LEGO NXT operated rovers as well as powerful service robots [9]. While MAA is designed to be a platform-independent DSL for developing robotics applications running on any kind of platform, ARDUINOML is specific to the Arduino platform.

In this paper, we present a case study on reusing these and other existing DSLs to develop a new RASPIROVER DSL customized for the development of Raspberry Pi operated rovers, such as the Eclipse PolarSys Rover [10]. Our goal is to showcase the language reuse support offered by MONTICORE

[DMW17] T. Degueule, T. Mayerhofer, A. Wortmann:
Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support.
In: Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), 2017.
www.se-rwth.de/publications/

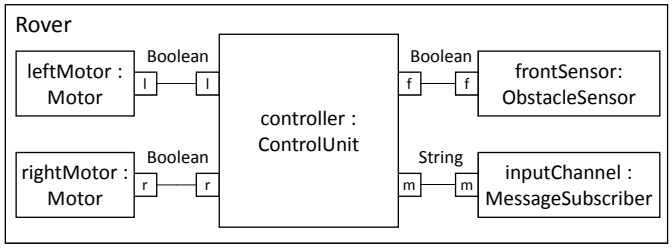

Fig. 1. Example rover hardware architecture defined with RASPIROVER DSL

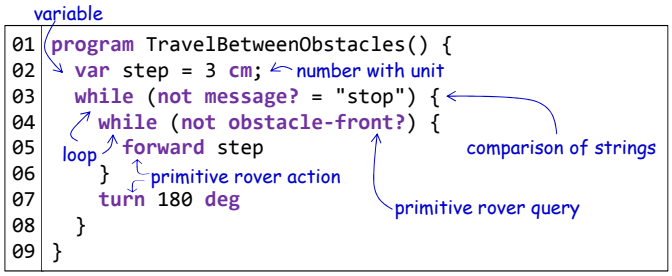

Fig. 2. Example rover control program defined with RASPIROVER DSL

and GEMOC STUDIO as well as their benefits for facilitating DSL engineering.

Before we present the implementations of RASPIROVER DSL with GEMOC STUDIO and MONTICORE, we first describe the requirements imposed on the development of RASPIROVER DSL regarding desired language features, reuse of existing DSLs, and DSL tooling.

A. Language Requirements

To set the language requirements for RASPIROVER DSL, we investigated several DSLs that have been used in the past to model rover systems. In particular, we investigated ARDUINOML and MAA mentioned before, as well as the UML-RT implementation of PAPYRUS-RT [11]. What the investigated modeling languages have in common is that they provide concepts for modeling both the hardware architecture of a rover and the behavior of the rover's control software. There are however considerable differences, in particular in how behavioral aspects are described. For instance, ARDUINOML provides an action language for defining the behavior, while PAPYRUS-RT relies on UML state machines with embedded C++ code. Based on our investigations, we decide to include in RASPIROVER DSL a hardware architecture description language and an action language as detailed in the following.

Rover Hardware Architecture Description Language: RASPIROVER DSL should enable the definition of the hardware setup of a Raspberry Pi operated rover. This includes defining hardware parts of the rover, such as the different actuators and sensors, as well as their connection to the rover's control unit through describing the mapping of the rover's board's pins to hardware modules. The primary rationale for including the hardware mapping into the DSL is to make this mapping explicit in a rover model, such that it can be leveraged in code generation and for interacting with the rover when executing a rover model.

Rover Control Language (RCL): For defining the behavior of rovers, RASPIROVER DSL should provide an imperative rover control language. We choose an action language because even if other behavioral languages (such as UML state machines) are used, an action language is still required to define fine-grained behaviors. Furthermore, the action language should abstract from a concrete programming language used to implement rover control software by providing high-level actions commonly supported by rovers. Therefore, RCL should comprise (1) *general control structures*, such as blocks, loops, and conditional statements as well as variables, primitive values, and variable assignments; (2) *rover-specific queries and actions*, in particular, dedicated queries for retrieving the temperature and humidity of a rover's environment, for obstacles in front of and behind the rover, and for messages posted to a central message board (*e.g.,* messages remotely sent to a rover), as well as actions for moving forward and backward, turning, stopping, logging, and sending messages; (3) *units* for defining distances to travel forward or backward, as well as the rotation angle for turning.

Fig. 1 shows an exemplary rover hardware description model that should be definable with RASPIROVER DSL. It features a control unit, two motors allowing the rover to drive forward, backward, and turn, an obstacle sensor on the front of the rover, as well as a message subscriber component allowing the rover to receive input commands. Fig. 2 shows an example of a rover control program written with RASPIROVER DSL defining that the rover should travel between two obstacles. In particular, it specifies that the rover should travel forward for 3 cm as long as no obstacle is detected in front of the rover (ll. 4-6). When an obstacle is detected, the rover should turn in the opposite direction (l. 7). This is repeated until a "stop" command is received via the message board (l. 3).

B. Reuse Requirements

Since we focus in this study on a comparison of language reuse support offered by the investigated language workbenches, we require that existing DSLs previously engineered with GEMOC STUDIO and MONTICORE should be reused wherever possible.

For implementing RASPIROVER DSL as specified above, we identify several DSLs that can be reused in GEMOC STUDIO and MONTICORE as shown in Table I. In particular, we aim to reuse ARDUINOML for the GEMOC STUDIO solution, as well as MAA and Java/P for the MONTICORE solution. The GEMOC STUDIO solution relies on a revised version of the unit domain model proposed in [12], while the units are implemented by hand in the MONTICORE solution. We manually implement the rover-specific queries and actions from scratch in both cases.

C. Tool Requirements

Regarding tooling support, we want to reuse any existing tool of the reused languages for the resulting RASPIROVER

TABLE I
LANGUAGES REUSED FOR DEVELOPING RASPIROVER DSL

Sub-language	GEMOC STUDIO	MONTICORE
Architecture Description	ARDUINOML	MAA [9]
General Action Language	ARDUINOML	Java/P [13]
Units	Reused from [12]	Newly implemented
Rover Actions	Newly implemented	

DSL language. This comprises, for instance, editors, model animators, model debuggers and code generators available for ARDUINOML and MAA. Other than that, we impose no further requirements, as the set of tools that can be developed with GEMOC STUDIO and MONTICORE is quite diverse. For instance, GEMOC STUDIO mostly supports graphical languages, while MONTICORE focuses on textual languages. As another example, GEMOC STUDIO specializes in language interpreters while MONTICORE specializes in code generators.

III. THE RASPIROVER DSL IN GEMOC STUDIO

The GEMOC STUDIO [3] is a language workbench for designing and using executable DSLs. It is built atop various modeling technologies of the Eclipse ecosystem (*e.g.,* EMF, Xtext, Sirius) and contributes new components, such as a model animator and a model debugger [14], along with customizable execution engines [15]. Following the GEMOC approach, the abstract syntax of a DSL is specified in Ecore [6], its concrete syntax in either Xtext or Sirius, and its execution semantics using various meta-languages (*e.g.,* K3 [16], ALE [17], or xMOF [18]). By specifying these artifacts, users of GEMOC STUDIO then enjoy various facilities, *e.g.,* language-specific debuggers and animators [15].

As one of the showcases of the GEMOC STUDIO, the authors designed ARDUINOML, a DSL with simulation and animation capabilities for designing programs that can be deployed on a given configuration of an Arduino board. The metamodel describing the abstract syntax of ARDUINOML consists of two main parts: a description of the hardware deployed on a given Arduino board (the set of modules and their associated pins), and a description of the scenario putting these modules into play with a simple action language providing basic control structures, variables, and expressions. The ARDUINOML language also comes with an associated interpreter.

Rather than starting the development of a new DSL for the PolarSys rover from scratch, one could reuse some parts of ARDUINOML and customize them to the specificities of the PolarSys rover, thereby obtaining a customized RASPIROVER DSL. To achieve this, GEMOC STUDIO includes MELANGE, a meta-language that allows language designers to reuse and compose various DSLs in the creation of new ones [19]. Using MELANGE, it is possible to prune the Arduino-specific parts of the ARDUINOML language, merge new concepts specific to the PolarSys rover (*i.e.,* rover actions and units as described in Section II-A), and customize the existing execution semantics (*i.e.,* override certain methods of the interpreter) to make the different parts fit together. Fig. 3 depicts the definitions of the various languages reused for implementing RASPIROVER DSL in MELANGE, along with the definition of the RASPIROVER DSL itself. The language implementations are available online.[1]

```
01 package rover
02
03 language RCL {
04   syntax "org.gemoc.rover.rcl/model/RCL.ecore"
05   with org.gemoc.rover.rcl.semantics.*
06 }
07 language Units {
08   syntax "org.modelexecution.units/model/units.ecore„
09   with org.modelexecution.units.semantics.*
10 }
11 language ArduinoML {
12   syntax "org.gemoc.arduino.sequential.model/model/arduino.ecore"
13   with org.gemoc.arduino.sequential.k3dsa.*
14 }
15
16 language RasPiRover {
17   slice ArduinoML on ["ArduinoBoard"]
18   renaming { "arduino" to "raspirover“
19            { "ArduinoBoard" to "RpiBoard" } }
20   merge RCL renaming { "rcl" to "raspirover" }
21   merge Units renaming { "units" to "raspirover" }
22   with rover.glue.*
23 }
```

Fig. 3. Using MELANGE to compose the ARDUINOML, RCL and UNITS languages to form a new customized RASPIROVER DSL

First, the rover control language `RCL` is defined (ll. 3-6). It consists of its abstract syntax, defined in Ecore (l. 4), along with its execution semantics defined in K3 (l. 5). The `Units` and `ArduinoML` languages are defined in a similar way (ll. 7-10 and 11-14). Finally, the `RasPiRover` language composes the reused language components in a meaningful way. First, it slices [19] the `ArduinoML` language (l. 17) to extract the hardware-definition part and remove the Arduino-specific parts that should not be reused in the new language. Then, it merges the two other languages `RCL` and `Units` into the `RasPiRover` DSL (ll. 20-21). Finally, additional K3 aspects are woven to glue together the different components both syntactically and semantically (l. 22). As an example, Fig. 4 depicts the `ProjectToProgramGlue` and `OverridenProjectInterpreter` aspects that, respectively, insert a new containment reference from `ArduinoML`'s `Project` (l. 1) to `RCL`'s `RoverProgram` (l. 3), and override the execution semantics of `Project` to instead delegate to the interpreter of `RCL` (ll. 7-9). The **`renaming`** clauses (Fig. 3, ll. 18-21) simply ensure that all concepts of the three languages end up in the same logical package.

Overall, the MELANGE meta-language allows us to compose the three languages and reuse (parts of) their syntax and semantics (interpreter). As a result, the model execution capabilities of GEMOC STUDIO can be employed for executing RASPIROVER DSL models. The main limitation of the reuse support offered by MELANGE is that the composition operators

[1] https://github.com/tdegueul/gemoc-pirover/

```
@Aspect(className = Project)
class ProjectToProgramGlue extends ExecutableProject {
  @Containment RoverProgram program
}
@Aspect(className = Project)
class OverriddenProjectInterpreter extends ExecutableProject {
  @OverrideAspectMethod override void execute() {
    _self.program.run
  }
}
```

Fig. 4. Syntactically and semantically gluing ARDUINOML's `Project` with RCL's `RoverProgram` using K3 aspects

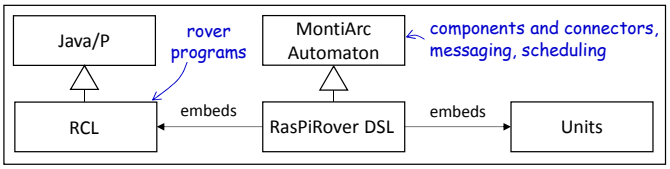

Fig. 5. Relationships of the languages reused to implement RASPIROVER DSL in MONTICORE

```
grammar RCL extends JavaDSL {
  ForwardAction       implements BlockStatement = "forward";
  BackwardAction      implements BlockStatement = "backward";
  LogAction           implements BlockStatement = "log" (msg:String);
  MessageQuery        implements Literal = "receive?";
  TemperatureQuery    implements Literal = "temperature?";
  // additional actions and queries
}
```

```
grammar RasPiRover extends MontiArcAutomaton, RCL {
  start MACompilationUnit;
  RCLEmbedding implements Element = "behavior" Name "{" Statement+"}";
}
```

from MontiArcAutomaton — from RCL

Fig. 6. Grammars `RCL` and `RasPiRover` realizing the definition and integration of RASPIROVER DSL into MAA

do not cope with concrete syntax, *i.e.,* the concrete syntax of the resulting language must be defined by hand to benefit from the animation and debugging facilities of GEMOC STUDIO. For more information on the reuse support offered by MELANGE, we refer the interested reader to [19], [20].

IV. THE RASPIROVER DSL IN MONTICORE

MONTICORE [4] is a workbench for the development of modular, textual languages based on extended context-free grammars, Java context conditions and translational realizations of semantics using template-based code generation. It enables reusing (parts of) languages via inheritance, embedding, and aggregation [21]. *Inheritance* enables reusing productions from the inherited grammar for which MONTICORE generates parsers, abstract syntax classes, context condition checking infrastructures, and code generation infrastructures. This supports specializing or extending languages while reusing existing tooling from the inherited language. With *embedding*, extension points in the host grammar are filled with productions from embedded grammars. This enables, for instance, reusing languages for well-defined concerns, such as expression languages. *Aggregation* loosely combines languages for joint analysis. To this effect, elements used in models of one language that reference elements of models of another language (such as references to data types in an architecture language) are interpreted specific to the integration. This integration is external to both languages and, hence, does not require participating languages to be aware of the integration.

A. *Reusable* MONTICORE *Languages*

Many of the languages required for realizing RASPIROVER DSL are already available in MONTICORE: MAA [9] enables describing software components and can thus be reused for the hardware architecture description language of RASPIROVER DSL through language inheritance. For realizing the rover control language RCL of RASPIROVER DSL, JAVA/P, the action language of UML/P [22] can be refined and extended as required. For this, we can first restrict JAVA/P to feature only variable assignments, statements, conditionals, and while-loops. This is achieved by inheritance, i.e., RCL inherits from JAVA/P, and an adds additional context condition that prevents instances of unsupported JAVA/P abstract syntax classes from being used in RASPIROVER DSL models. Then we add new primitives dedicated to movement, sensing, and communication to RCL. Furthermore, the UNITS language is embedded into RCL. Consequently, the overall language composition of RASPIROVER DSL is as depicted in Fig. 5. MONTICORE combines the parsers and abstract syntax classes of the languages accordingly.

B. *Combining Syntaxes for the* RASPIROVER DSL

Leveraging JAVA/P through inheritance enables the integration of new primitives for rover actions and queries via its various interfaces. To this end, rover actions, which resemble statements in JAVA/P, are designed to implement JAVA/P's `Statement` interface. This enables the usage of the new actions wherever statements are supported, *e.g.,* loops and conditionals. Queries, which resemble literals, are analogously designed to implement MONTICORE's `Literal` interface, enabling their use wherever literals are supported. With this, the complete RCL grammar is only 17 lines of code (15 new productions) as illustrated in Fig. 6 (top). In MONTICORE, embedding is a specific usage of grammar inheritance that links productions of the grammar to be embedded into extension points of the host grammar. For embedding RCL into RASPIROVER DSL, we leverage MONTICORE's multiple inheritance to realize embedding as linking one interface from the host grammar `MAA` to productions from the embedded grammar `RCL`. This is depicted in Fig. 6 (bottom). With this, we obtain a textual concrete syntax as illustrated in Fig. 7.

Well-formedness of RCL programs is checked by new context conditions that ensure, for instance, that results of `receive?` queries (which return strings) are not compared to numbers. To this end, adapters between the abstract syntax classes generated from MAA's ports and RCL's primitives enable interpreting the latter as method calls of the appropriate return types, *i.e.,* whenever MONTICORE looks up what `receive?` in the context of JAVA/P is, the adapters return

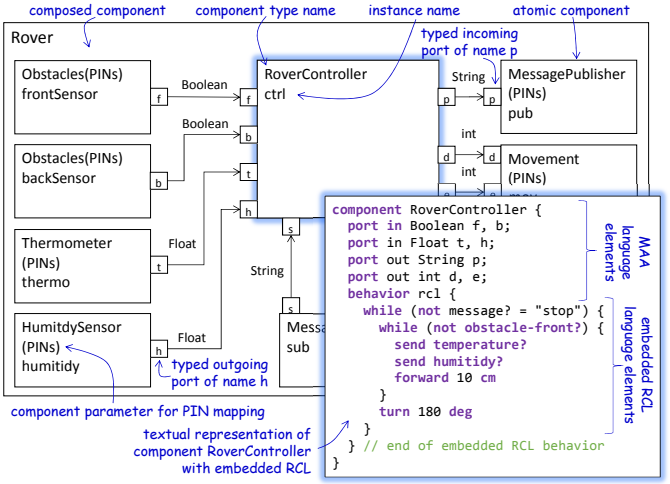

Fig. 7. Example RASPIROVER DSL model defined in MONTICORE

a method of return type string. With this in place, all JAVA/P well-formedness rules are applied automatically.

C. Combining Code Generators for the RASPIROVER DSL

Integrating the behavior of MAA (scheduling, message passing) and RCL (commanding the rover to perform actions) requires (1) the integration of their template-based code generators, and (2) a mapping from rover primitives to sending/receiving messages in the architecture (*e.g.,* mapping the primitive `forward` to sending a messages to a motor).

The MAA code generator requires that for embedded behavior models (in our case RCL programs), a Java class is generated that implements a specific interface of MAA's common Java run-time environment [9]. This interface prescribes a `compute()` method that takes a set of named values (read from incoming ports) as input and returns another set of named values (that are assigned to outgoing ports) as output. Every time a component (e.g., a rover's control unit) should execute its behavior, this method is invoked by MAA. The body of the `compute()` method is generated by the code generator of RCL. For this, the RCL generator can reuse the JAVA/P generator templates for translating control structure, conditionals, *etc.* The translation of RCL primitives into sending and receiving messages is specific to the RCL generator and must be implemented accordingly. To take into account the connection among rover components, the generator takes as additional input a map from RCL primitives to port names. Based on this, the generator translates actions to sending messages, and queries to receiving messages.

D. Summary

Overall, MONTICORE facilitates reusing the syntax and semantics of independently developed DSLs. The central artifacts for this are MONTICORE grammars, Java context conditions, the symbol table, and template-based code generators, out of which only the grammars require learning a specific meta-language while the rest is implemented in Java. Hence, MONTICORE focuses on programmatic language engineering. This, however, confronts language engineers with the complexities of Java. Moreover, there is no support for producing graphical editors for MONTICORE languages. The language implementations are available online.[2]

TABLE II
COMPARISON OF REUSE SUPPORT IN GEMOC STUDIO AND MONTICORE

Reuse Support	GEMOC STUDIO	MONTICORE
Language Components:		
Abstract Syntax	●	●
Concrete Syntax	○	●
Semantics	● (interpreters)	● (generators)
Transformations	●	●
Mechanisms:		
Removing Concepts	● (slicing)	◐ (context condition)
Adding Concepts	● (merging/inheritance)	● (inheritance)
Changing Concepts	◐ (weaving/inheritance)	◐ (inheritance)
Language Embedding	○	●

V. DISCUSSION

As illustrated, the language reuse capabilities of GEMOC STUDIO and MONTICORE differ in the language constituents that can be reused as shown in Table II. For both workbenches, reusing abstract syntax, semantics, and transformations[3] is supported. MONTICORE also supports the reuse of concrete syntax as the concrete syntax integrated into the grammars defining the abstract syntax as well. However, the reuse mechanisms of GEMOC STUDIO and MONTICORE differ in their expressiveness (see Table II). While GEMOC STUDIO supports removing abstract syntax elements through slicing, MONTICORE only supports removing abstract syntax elements through well-formedness rules that actively prevent their instantiation. Adding new abstract syntax elements is supported through inheritance by both language workbenches. GEMOC STUDIO also supports merging two metamodels on joint classes. MONTICORE also supports inheriting concrete syntax. Through inheritance, both workbenches also support to change language concepts, whereas embedding, *i.e.,* specification and binding of dedicated language extension points is specific to MONTICORE.

We also investigated the impact of reuse in GEMOC STUDIO and MONTICORE with respect to the reuse that could be achieved in the implementations of RASPIROVER DSL. For GEMOC STUDIO, we measured the numbers of reused metamodel elements (classes, features, operations) and lines of code of the existing interpreters. For the new artifacts, we considered the size of the new RCL language (metamodel

[2] http://www.se-rwth.de/materials/rcl/

[3] Please note that we did not explicitly discuss the reuse of transformations in this paper.

TABLE III
COMPARISON OF REUSE ACHIEVED WITH GEMOC STUDIO AND MONTICORE (LOC: LINES OF CODE; PROD: PRODUCTIONS; ELEM: ELEMENTS)

Artifact	GEMOC STUDIO		MONTICORE	
	Size of New Artifacts	Size of Reused Artifacts	Size of New Artifacts	Size of Reused Artifacts
Syntax	141 elem. (RCL metamodel) 24 LoC (K3 glue)	361 elem. (metamodels)	21 LoC / 18 Prod. (grammars) 4 LoC / 2 Prod. (glue)	1,267 LoC / 246 Prod. (grammars)
Semantics	281 LoC (RCL interpreter) 50 LoC (K3 glue)	643 LoC (interpreters)	439 LoC (generators) 1 LoC (Java glue)	1,266 LoC (generators)
Reuse-Specific	19 LoC (Melange)	-	-	-

and interpreter), the glue code, and the MELANGE file. For MONTICORE, we measured the lines of code in the new grammars and in the reused grammars, as well as their size in terms of new and reused productions. For MONTICORE's generators, we also measured the lines of code of new and reused code. The results are depicted in Table III. As can be seen from these results, we could reuse a large portion of RASPIROVER DSL from existing languages and only needed to implement RCL and some glue code from scratch in both language workbenches.

VI. CONCLUSION

We presented the quintessential constituents and activities required for implementing RASPIROVER DSL, a DSL for defining the hardware architecture and control software behavior of rovers, with GEMOC STUDIO and MONTICORE. For both language workbenches, we could rely on existing DSLs to build the new rover-specific DSL. Through both case studies, we showed how the different language reuse mechanisms are applied and highlighted how they differ. We hope that this supports practitioners in creating custom DSLs with minimal effort.

REFERENCES

[1] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2008.

[2] S. Erdweg, T. van der Storm, M. Völter *et al.*, "The State of the Art in Language Workbenches," in *Proceedings of the 6th International Conference on Software Language Engineering*. Springer International Publishing, 2013, pp. 197–217.

[3] The GEMOC Initiative, "The GEMOC Studio," 2017. [Online]. Available: http://gemoc.org/studio.html

[4] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: a framework for compositional development of domain specific languages," *STTT*, vol. 12, no. 5, pp. 353–372, 2010.

[5] Model-Driven Engineering Tools Challenge (MDETools'17), "Rover Challenge Problem." [Online]. Available: http://mase.cs.queensu.ca/mdetools/index.php?id=rover

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2008.

[7] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language Composition Untangled," in *Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications*. ACM, 2012, pp. 7:1–7:8.

[8] The GEMOC Initiative, "Arduino Modeling GitHub Project." [Online]. Available: https://github.com/gemoc/arduinomodeling

[9] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems," *Journal of Software Engineering for Robotics*, vol. 6, no. 1, pp. 33–57, 2015.

[10] PolarSys, "PolarSys Rover." [Online]. Available: https://www.polarsys.org/projects/polarsys.rover

[11] The Eclipse Foundation, "Papyrus Real Time." [Online]. Available: https://eclipse.org/papyrus-rt/

[12] T. Mayerhofer, M. Wimmer, and A. Vallecillo, "Adding Uncertainty and Units to Quantity Types in Software Models," in *Proceedings of the 9th International Conference on Software Language Engineering*. ACM, 2016, pp. 118–131.

[13] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, ser. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[14] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs," in *Proceedings of the 8th International Conference on Software Language Engineering*. ACM, 2015, pp. 137–148.

[15] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the GEMOC Studio (Tool Demo)," in *Proceedings of the 9th International Conference on Software Language Engineering*. ACM, 2016, pp. 84–89.

[16] Inria DiverSE, "K3 – Executable Metamodeling," 2017. [Online]. Available: http://diverse-project.github.io/k3/

[17] M. Leduc, T. Degueule, B. Combemale, T. van der Storm, and O. Barais, "Revisiting Visitors for Modular Extension of Executable DSMLs," in *Proceedings of the 20th International Conference on Model-Driven Engineering Languages and Systems*. ACM, 2017.

[18] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs based on fUML," in *Proceedings of the 6th International Conference on Software Language Engineering*, ser. Lecture Notes in Computer Science, vol. 8225. Springer, 2013, pp. 56–75.

[19] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A Meta-language for Modular and Reusable Development of DSLs," in *Proceedings of the 8th International Conference on Software Language Engineering*. ACM, 2015, pp. 25–36.

[20] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J. Jézéquel, "Safe model polymorphism for flexible modeling," *Computer Languages, Systems & Structures*, vol. 49, pp. 176–195, 2017.

[21] A. Haber, M. Look, P. M. S. Nazari, A. N. Perez, B. Rumpe *et al.*, "Composition of Heterogeneous Modeling Languages," in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Springer International Publishing, 2015, pp. 45–66.

[22] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.