Welcome to the 15th Workshop on Domain-Specific Modeling

Preface

Domain-Specific Modeling (DSM) languages provide a viable and time-tested solution for continuing to raise the level of abstraction, and thus productivity, beyond coding, making systems development faster and easier. When accompanied with suitable automated modeling tools and generators it delivers to the promises of continuous delivery and devops.

In Domain-Specific Modeling (DSM) the models are constructed using concepts that represent things in the application domain, not concepts of a given programming language. The modeling language follows the domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts. Together with frameworks and platforms, DSM can automate a large portion of software production. This automation is possible because of domain-specificity: both the modeling language and code generators fit to the requirements of a narrowly defined domain, often inside one organization only.

The 15th workshop on Domain-Specific Modeling will provide a forum for presenting research work, experience reports and language demonstrations. This year we received 17 papers, of which we accepted 12. Each paper was reviewed by three persons. We would like to thank program committee for their help and contribution during the review process. The accepted papers are organized in the program into four categories: experiences on language engineering, code generation, language evolution and use, and language engineering perspectives.

Following the workshop theme we also have interactive workgroup discussions. Participants choose the topics like identify new research questions or focus on more detail on topics presented earlier in the workshop.

The DSM workshop is one of longest running series of workshops at SPLASH/OOPSLA, this being the 15th anniversary of the series and we plan a little celebration the workshop.

October 2015
Pittsburgh, Pennsylvania
15th WORKSHOP ON DOMAIN-SPECIFIC MODELING

27 October 2015, Pittsburgh, Pennsylvania, United States

Program Committee

Ankica Barisic, Universidade Nova de Lisboa, PT
Christian Berger, University of Gothenburg, SE
Filipe Correia, Universidade do Porto, PT
Davide Di Ruscio, Università degli Studi dell'Aquila, IT
Michalis Famelis, University of Toronto, CA
Oystein Haugen, Østfold University College, NO
Steven Kelly, MetaCase, FI
Thomas Kuchne, Victoria University of Wellington, NZ
Tanja Mayerhofer, Vienna University of Technology, AT
Bernhard Rumpe, RWTH Aachen University, DE
Kari Smolander, Lappeenranta University of Technology, FI
Yu Sun, Cal Poly Pomona, US
Naoyasu Ubayashi, Kyushu University, JA
Timo Wegeler, independent, DE

Organizing Committee

Jeff Gray, University of Alabama, US
Jonathan Sprinkle, University of Arizona, US
Juha-Pekka Tolvanen, MetaCase, FI
Matti Rossi, Aalto University School of Economics, FI
# Table of contents

| Domain Specific Modelling for Clinical Research | 1 |
| Jim Davies, Jeremy Gibbons, Adam Milward, David Milward, Seyyed Shah, Monika Solanki and James Welch |
| CHARIOT: A Domain Specific Language for Extensible Cyber-Physical Systems | 7 |
| Subhav Pradhan, Abhishek Dubey, Aniruddha Gokhale and Martin Lehofer |
| Experience Report: Constraint-Based Modeling of Autonomous Vehicle Trajectories | 13 |
| Kenneth McKeever, Yevgenya Zelieke, Matt Bunting and Jonathan Sprinkle |
| Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures | 19 |
| Katrin Hölldobler, Pedram Mir Seyed Nazari and Bernhard Rumpe |
| Automating Engineering with a Domain-Specific Language and a Code Generator | 25 |
| Al Niessner, Oh-Ig Kwoun, Belinda Randolph and Honghanh Nguyen |
| Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table | 31 |
| Pedram Mir Seyed Nazari, Alexander Roth and Bernhard Rumpe |
| Mixed Generative and Handcoded Development of Adaptable data-centric Business Applications (demo) | 36 |
| Pedram Mir Seyed Nazari, Alexander Roth and Bernhard Rumpe |
| Reusing Legacy DSLs with Melange (demo) | 39 |
| Thomas Degueule, Benoît Combemale, Arnaud Blouin and Olivier Barais. |
| Supporting Users to Manage Breaking and Unresolvable Changes in Coupled Evolution | 42 |
| Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio |
| Towards Improving Software Security using Language Engineering and mbeddr C | 49 |
| Markus Voelter, Zaur Molotnikov and Bernd Kolb |
| Extensible Visual Constraint Language | 57 |
| Brian Broll and Ákos Ledeczi |
| Systematic Evaluation of Three Data Marshalling Approaches for Distributed Software Systems | 63 |
| Hugo Andrade, Federico Giaimo, Christian Berger and Ivica Crnkovic |
Domain Specific Modelling for Clinical Research

Jim Davies, Jeremy Gibbons, Adam Milward, David Milward, Seyyed Shah, Monika Solanki, and James Welch
Department of Computer Science, University of Oxford
firstname.lastname@cs.ox.ac.uk

Abstract
The value of integrated data relies upon common data points having an accessible, consistent interpretation; to achieve this at scale requires appropriate informatics support. This paper explains how a model-driven approach to software engineering and data management, in which software artefacts are generated automatically from data models, and models are used as metadata, can achieve this. It introduces a simple data modelling language, consistent with standard object modelling notations, together with a set of tools for model creation, maintenance, and deployment. It reports upon the application of this approach in the provision of informatics support for two large-scale clinical research initiatives.

Categories and Subject Descriptors D.2.12 [Interoperability]; D.3.3 Programming Languages: Specialized application languages

Keywords domain-specific modeling languages, interoperability, mapping, code generation, model transformation, meta-modeling

1. Introduction
To obtain the evidence required to support the development and introduction of a new treatment, or a new diagnostic tool, we need to consider the results of detailed, clinical observations of a large number of individuals. These observations need to be made, and the results recorded, in a consistent fashion.

The usual way of achieving this is through prior agreement upon a study protocol: a detailed specification of the information required, and an account of the proposed analysis. Clinical staff receive training and support to ensure that the collection proceeds according to the protocol, and data is recorded using a single set of 'case report forms'.

There are two problems with this. The first is the cost of manual data collection, of additional training, and of bespoke systems development. The second is a lack of a guarantee of consistency across studies: even where two studies require what is essentially the same information, differences in specifications may mean that the data collected is incompatible.

For example, in breast cancer, 'histological type of tumour' is an common piece of information. However, one study might record this against an enumeration such as

- in-situ ductal only
- tubular/cribriform
- ductal grade unknown
- mixed

whereas another might offer a choice of

- invasive ductal or no specific type
- tubular
- mucinous
- invasive cribriform

Even if we assume that the clinical staff have the same interpretation of these technical terms, the resulting data cannot be combined without additional effort and some loss of information.

We can reduce the cost of new studies by re-using data already collected: in earlier studies, or in the clinical information systems used to support patient care. We can reduce the cost of bespoke systems development by generating case report forms, queries, databases, and workflows from the detailed specifications in the protocol. We can increase consistency across studies, and facilitate re-use of the data, by coordinating the design of specifications.

To do this in practice, and at scale, requires effective, domain-specific modelling. We need models that describe study data: the detailed data specifications mentioned above. We need models that describe the relevant contents of clinical information systems. We need models that describe forms, queries, databases, and workflows used for data collection, transmission, and integration.

We need also a mechanism for relating the declarations of individual data items in different models. We need to be able to record the fact that two items, declared in different models, represent the same information: that any value assigned would have the same interpretation in both contexts. This is precisely what is needed if we are to re-use data from different systems, or combine data from different studies.

In this paper, we describe the notion of a model catalogue: an application that stores and presents models, links data declarations, and supports the generation of artifacts such as case report forms and data schemas. We introduce the domain-specific language that describes the catalogue contents. We then report upon the experience of deploying the catalogue, and the domain-specific modelling language, in the development of national infrastructure for clinical research.

2. Data Models
2.1 Data sets and data standards
A data set definition for clinical research study will consist in a number of different parts, each of which declares a set of related data items. Typically, this will be a set of data items that would be collected together: the results of a particular kind of observation, or the account of a particular kind of intervention. These parts may be 'repeating': the same kind of observation may be made many times of a single study participant.

A data item declaration should explain not only the name under which values are to be stored, but also the type of those values. If the type is numeric, then the unit of measurement should be given. If the type is an enumeration, then the intended interpretation of each value should be explained. Finally, the parts of the dataset may be connected or related to one another, and these relationships may have constrained multiplicities.

It should be clear that a dataset definition can be represented as a class diagram or object model. Data items can be introduced as attributes, parts of the model as classes, and relationships between classes as associations—complete with multiplicities. Data types and enumerations can be used to support attribute declarations.
A data set definition may apply to more than one study. It may also be used as a data standard for communication between information systems used in healthcare. The UK National Laboratory Medicines Catalogue, for example, provides a set of standard definitions for pathology reports, to facilitate safe, effective data transfer across different systems.

2.2 Studies, forms, schemas, and databases

A data set definition will not contain enough information to completely characterise a study. The study protocol document will contain precise information about study timetables, workflows, and procedures, as well as a considerable amount of free text explanation. A domain-specific modelling language for studies would be more expressive than a domain-specific language of data set definitions.

Similarly, a domain-specific modelling language for case report forms will support the description of form structures, sections, and 'skip logic'; for example, 'if yes, then go to Question 5'. A modelling language for XML schemas will support the description of schema structures, choices, and complex types, and a modelling language for databases will include information about queries and constraints.

A model of a case report form will contain a number of data item declarations, each with the same information context as a data item definition in an abstract data model. We might be forgiven, then, for thinking that we might not need an abstract data modelling language: we could simply consider, relate, and re-use data definitions from models of 'real' artefacts: studies, forms, schemas, and databases.

However, the fact that a model corresponds to a particular artefact, or even a particular kind of artefact, provides additional context for the data definitions it contains. Whether or not there are additional, explicit constraints upon a data item, the fact that any data collected will have been entered into an implementation of the form tells us strictly more about it—narrowing the interpretation of the data definition.

2.3 Models as metadata

A domain-specific model used in the generation or documentation of an artefact represents valuable metadata about that artefact, and also about any data that the artefact is used to collect or produce. A domain-specific data model—or a data component of any model, for that matter—can be used as metadata about other models. In this way, we can relate artefacts described by different models, and hence the data collected by different studies, forms, or schemas.

To use a model as metadata for an artefact, we have only to create a link between the artefact and a published instance of the model, held in a repository or model catalogue. This link may be created automatically if the artefact is generated from the model, or if the model is generated from the artefact: for example, we may generate a more abstract data model from the schema of a relational database.

To use a model as metadata for another model, we create links between the two models. Typically, these will be links between individual data items: for example, an attribute in a model of a form, labelled height, could be linked to an attribute in a model of a data set or data standard, labelled patient's height in cm, measured without shoes, to indicate that form attribute has all of the properties described in the data set definition.

As we argued above, the form attribute will be further constrained by the remainder of the form model, so the relationship between the two is asymmetric. For this reason, we refer to such a link as an 'implements' relationship. If a pair of attributes are 'implements' of each other, then we refer to the pair of links as a 'same as' relationship.

In general, we do not expect to find 'same as' relationships between attributes. Different models will add different constraints to the definition of a data item. Instead, consistency of data definitions between studies, forms, or schemas will be represented by 'implements' links to the same data set or data standard. The data definitions are not identical, but are consistent as far as the constraints of the data standard are concerned.

3. Implementation

3.1 Generic data modelling language

A model for a generic data modelling language is shown in Figure 1. All data classes, data elements, and data types are declared and managed within Models. A Class may contain many Elements, and may have other classes as components—corresponding to the UML concept of composition. An element has a unique Type, which may be reference-valued, a Primitive type, or an Enumeration. Reference types correspond to class names within this or some other model. EnumValue, enumeration values, are managed as separate, identified items.

A model may be declared as a new version of an existing model. Any of the items within a model may be declared as a refinement of an existing item. This indicates that its interpretation or semantics should be seen as an extension of those associated with that other item. Typically, this will correspond to the author of the model recording that a particular data class or data element is intended to conform to some existing, published standard.
3.2 Domain-specific data models

The current model catalogue implementation supports the creation, storage, and management of data models in the language of Figure 1. Domain-specific models of studies, forms, and schemas are generated from these models, using different sets of heuristics, but are not themselves managed as catalogue items. A more comprehensive approach would involve persisting, managing, and editing these models alongside the generic data models that they correspond to: the generation process could then work in both directions.

Figure 2 shows the relationship between domain-specific data models—in particular, models of form designs—and generic data models. It shows also the implementations derived from the form designs using a model-driven approach. In the terminology of MDA [18], we may see the form model as a platform-independent entity (at the M1 metamodeling level) and the form implementation as a platform-specific entity (also at M1).

The data model language and the form model language are both entities at the M2 level. The catalogue would support both of these as instances of a data metamodeling language (or model metamodel) at the M3 level. The relationship between the generic data model and the corresponding domain-specific model would be one of data refinement, in the sense of [19]: in the case of a form model, this would be a simple correspondence between classes, elements, and datatypes; for a workflow or process model, the ways in which data is exposed through transactions and events would need to be considered.

The advantage of this more comprehensive approach is that aspects of form design and implementation can be introduced and managed directly, through editing of Form Models, rather than being encoded as options in a generation pipeline. The corresponding generic data model may be abstracted automatically from the form model; alternatively, the form model may be partially (re)generated from the generic data model, in the sense of [6].

An alternative approach is shown in Figure 3, in which the generic data model is used as a metamodel for domain-specific data modelling languages. Existing language and tool support for the use of models as metamodels in this context does not allow for the definition and maintenance of the ‘instance of’ relationships in the diagram, and the catalogue implementation under development follows the approach of Figure 2. However, this alternative approach would remove the need to maintain separate, generic data models, and it remains the subject of active investigation.

3.3 Catalogue Implementation

In designing the catalogue, we paid considerable attention to the ISO/IEC 11179 standard for metadata registration, which sets out a design for metadata catalogues. Some difficulties have been encountered in the practical application of ISO/IEC 11179 at scale: see, for example [13] and [14].

The principal complaint is that there is no structuring mechanism for data definitions: data items can be associated only at the conceptual level. As a result, each item has to be defined separately: there is no opportunity to add the same information to several data item definitions at once, whether this is within the model, or as a link to another model.

The approach taken in our implementation of the model catalogue is more general: the use of tagging supports multiple classification schemes, and allows the representation of relationships as well as simple taxonomies. However, it should be clear that our catalogue could be used, under suitable constraints, as an effective implementation of the ISO/IEC 11179 standard.

This applies also to the processes of registration, versioning, and publication. Every object stored in the catalogue is managed as an administered item, in the language of the standard. The notion of linking in the catalogue implementation allows us to exploit this administrative information in the automatic creation and maintenance of semantic links, and the administrative processes are generalised to provide support for collaborative development.

The existing model catalogue is built using the Groovy/Grails framework, which takes a model-view-controller approach to data management and presentation. The key advantage of this platform has been the ability to revisit the underlying data representation—the domain model—without needing to re-implement the presentation layer, and vice versa. As the software was developed in the course of application, this was particularly important.

A 'discourse' plugin provides support for collaborative development of models and data definitions, with users able to contribute to a comment history for each administered item, prompting responses from other users as necessary. This proved particularly important given that many of the clinical scientists were contributing to the dataset development in their spare time.

3.4 Generation pipelines

The existing implementation has been used to generate several different types of artefact, including:

- **Case report form models** for consumption by the OpenClinica clinical trials management system. These take the form of Excel spreadsheets with columns specifying form structure, question text, response types, logical constraints (including skip logic), and presentation controls. These models are generated from form models in the data modelling language by way of a complex transformation: the hierarchical structure of the data model is flattened to produce lists of sections, repeating groups, and questions. Default values and implementations are included as part of the transformation: for example, we provide custom validation for textual fields that are tagged with constraints in the form of regular expressions.
Database triggers To support the automatic processing of data received from the clinical trials system, we require a collection of triggers for the underlying database. These ensure that the combination of existing and newly-received data is properly normalised. This is particularly important where data is being collected against different versions of the same form.

XML schemas for electronic document submission This is a more straightforward transformation, as the structure of the XML schemas is closer to that of the data models. However, additional processing is required to produce normalised, readable schemas. For example, if there are several data elements sharing the same datatype, we would wish to include that datatype only once within the schema.

Tools for creating and validating .csv files For some of the systems that we are working with, the easiest way to import or export information is in comma-separated value format. In this case, we are not generating a specification of the data format in some implementation language; we are instead generating tools that will ensure that the values presented in a file comply with the model constraints.

Data manuals Datasets and data standards in health informatics are communicated through documents in which each data point is listed along with its intended interpretation. These manuals are automatically generated, ensuring consistency between the information that they present and the tools used for data acquisition and processing.

4. Experience

4.1 Re-use of data from clinical information systems

The UK National Institute of Health Research (NIHR) is funding an £11m programme of work across five large university-hospital partnerships: at Oxford, Cambridge, Imperial College London, University College London, and Guy’s and St Thomas’. The aim of the programme is to create the infrastructure needed to support data re-use and translational research across these five institutions.

The programme, the NIHR Health Informatics Collaborative (HIC), was initiated in 2013, with a focus upon five therapeutic areas: acute coronary syndromes, renal transplantation, ovarian cancer, hepatitis, and intensive care. The scope was increased in 2015 to include other cancers—breast, colorectal, lung, and prostate—and other infectious diseases, including tuberculosis.

The key component of the infrastructure consists in repositories of patient data within each of the five institutions. The intention is that these repositories should hold a core set of data for each therapeutic area, populated automatically from clinical systems, together with detailed documentation on the provenance and interpretation of each data point.

Researchers can use the documentation to determine the availability and suitability of data for a particular study. They can use it also to determine comparability across institutions: whether there are any local differences in processes or equipment that would have a bearing upon the combination and re-use of the corresponding data. Once a study is approved, the repositories act as a single source of data, avoiding the need for data flows from individual clinical systems.

The development of the infrastructure required the development of a 'candidate data set' for each therapeutic area, as a core list of data points collected in the course of routine care that would have value also in translational research. Each institution then set out to determine which information systems, within their organisation, could be used to populate each of the candidate data sets: this was termed the 'data exploration exercise'.

The results of the exercise informed further development of the data sets, and data flows were established. To demonstrate and evaluate the new capability, 'exemplar research studies' were initiated in each therapeutic area, using data from all five institutions.

Each institution had a different combination of existing systems, a different approach to data integration, and a different strategy for informatics development. It was not feasible or appropriate to develop a common 'data repository' product for installation. Instead, a set of data models were distributed, and each institution worked to implement these using their own messaging, business intelligence, or data warehousing technologies.

None of the institutions had the capability to provide documentation on the provenance and interpretation of their data in any standard, computable format; the model or metadata aspect of the infrastructure was entirely new. It was this that drove—and continues to drive—the development of a comprehensive model catalogue application.

At the start of the project, teams of clinical researchers and leading scientists were given the responsibility of creating the candidate data sets for each therapeutic area. They did this by exchanging spreadsheets of data definitions in email. This proved to be a slow process, and face to face meetings were needed before any real progress could be made.

It proved difficult to properly represent repeating sections of the dataset—corresponding to investigations or interventions that may happen more than once for the same patient. Researchers resorted to Visio diagrams to try to explain how observations fitted into clinical pathways or workflows—and discovered that there were significant differences between pathways for the same disease at different institutions.

In one therapeutic area, these differences had a profound effect upon the interpretation of certain observations, and the candidate dataset was extended to include additional information on the pathway. Due to the complexity of the pathways involved, this was a time-consuming and error-prone process. Furthermore, the spreadsheets quickly became inconsistent with the Visio diagrams.

The candidate datasets were distributed to the informatics teams at the five institutions in the form of XML schemas. At first, these were created from scratch, rather than being generated. There were...
many requests for changes to the schemas; these proved difficult to track and coordinate.

The exploration exercise was reported by adding columns to the distributed versions of the candidate dataset spreadsheets, listing the information systems containing the data points in question, or suggested alternatives where there were significant differences due to local systems and processes.

This was despite the availability of an initial version of the model catalogue. Researchers and local informatics teams preferred to work with spreadsheets, having little or no knowledge of modelling languages such as UML and no automatic support for model creation and maintenance. It fell to the software engineering team at the coordinating centre to record the datasets and variations in the catalogue.

While it was disappointing to have the researchers still working in spreadsheets, the ability to generate XML schemas from models, and to manage relationships between data items in different models and different versions, proved invaluable. In the second phase of the project, researchers are starting to abandon the spreadsheet mode of working, and are instead maintaining the datasets as data models, in the catalogue.

4.2 Coordination of clinical data acquisition

The UK Department of Health, through the NIHR and the National Health Service (NHS), is providing funding for the whole genome sequencing of blood and tissue samples from patients with cancer, rare disorders, and infectious disease. A network of regional centres is being established to collect samples and data, and to provide access to genomic medicine across the whole of the country. The funding committed to date is approximately £300m.

The results of the whole genome sequencing will be linked to detailed information on each participant: clinical and laboratory information drawn from health records, ontological statements regarding abnormal features or conditions, and additional information obtained from the participant or their representatives. The information required will depend upon the nature of the disease that the patient is suffering from. For example, information on breast density is required in the case of breast cancer, but not for other diseases.

131 different diseases have been included in the sequencing programme thus far. Each disease corresponds to a different combination of clinical and laboratory data points, a different set of ontological statements, and a different set of questions for the participant. There are, however, significant overlaps between diseases: for example, many different rare diseases will require the same information on kidney or heart function.

The modelling task is at least an order of magnitude greater than that required for the NIHR HIC, and yet candidate datasets have already been created for more than half of the diseases included. This is due partly to the availability of the model catalogue application from the start of the project, and partly to the availability, within the catalogue, of the full complement of HIC-defined data models and related data sets—including the national NHS data dictionary and the national cancer reporting datasets.

Two routes are available for the provision of data from the network of contributing centres: direct data entry into electronic case report forms, in a on-line clinical trials management system; and electronic submission of data in XML format. The intended interpretation of the data required is explained in a regularly-updated set of data manuals.

It is important that the forms used for direct data entry, the schemas used for XML submission, and the data manuals are properly synchronised. An initial approach to this, in which a single model was used as the basis for the generation of all three kinds of artefact, proved inconvenient in practice. Although the same data points were to be collected in each case, the distribution of these data points across classes and sections was different.

Accordingly, the model catalogue is used to store three different data models for each dataset: one for the generation of the forms, another for the generation of the XML schemas, and one for the generation of the data manual. These models are semantically-linked. If one is updated, then the fact that the others may now be inconsistent will be flagged to the user.

The same linkage is made with regard to existing reporting datasets and clinical audits. To avoid duplication of effort, the reporting datasets for the genomics medicine programme have been aligned with these activities. The existing datasets have been modelled, and updates to them will be tracked in the catalogue: again, potential inconsistencies can be flagged.

5. Related Work

The work described in this paper has evolved from the CancerGrid project [8], where an ISO/IEC 11179-compliant metadata registry was developed for curation of semantic metadata and model-driven generation of trial-specific software [5, 7]. The approach to generating forms in the CancerGrid project has been generalised significantly with the introduction of a data modelling language and a broader notion of semantic linking.

Another effort to develop an implementation of ISO/IEC 11179 is found in the US caBIG initiative [12]; however, their caCORE software development kit [11] applies model-driven development only to generate web service stubs, requiring developers to create application logic by hand, whereas our technique integrates with existing clinical Electronic Data Capture tools and workflows, such as OpenClinica [4].

Several efforts have addressed ontological representations for enabling data integration across metadata registries (MDRs). Sinaci and Erturkmen [16] describe a federated semantic metadata registry framework where Common Data Elements (CDEs) are exposed as Linked Open Data resources. Jeong et al. [10] present the Clinical Data Element Ontology (CDEO) for unified indexing and retrieval of elements across MDRs; they organise and represent CDEO concepts using SKOS. Tao et al. [17] present case studies in representing HL7 Detailed Clinical Models (DCMs) and the ISO/IEC 11179 model in the Web Ontology Language (OWL), but do not present any systematic metamodeling or language definition framework.

Ontology repositories can be considered closely analogous to model catalogues, they provide the infrastructure for storing, interlinking, querying, versioning, and visualising ontologies. Relationships capturing the alignments and mappings between ontologies are also captured, allowing easy navigability. Linked Open Vocabularies [2] provides a service for discovering vocabularies and ontologies published following the principles of linked data.

In the Model Driven Health Tools (MDHT) [3] project, the HL7 Clinical Document Architecture (CDA) standard [9] for managing patient records is implemented using Eclipse UML tools [1]. In principle, this is similar to our Model Catalogue approach, where the CDA metadata can be represented and implementations derived. However, MDHT supports only the CDA standard, whereas the Model Catalogue can interoperate with any metadata standard. The CDA standards are large and complex; Scott and Worden [15] advocate a model-driven approach to simplify the HL7 CDA.

6. Conclusion

The experience of applying the data model language, the model catalogue, and the associated generation tools, in the context of clinical research informatics has led to the following suggestions. A data dictionary is not enough. A simple, flat list of data definitions does not support re-use at scale: it requires the user
to place all of the contextual information into the definition of each data item, and mitigates against the automatic generation and application of definitions. Instead, a compositional approach is required, in which data elements are defined in explicit context.

A catalogue is not enough. The models in the catalogue must be linked to implementations, and to each other, with a considerable degree of automatic support. If the models are out of sync with the implementations, and with the data, then their value is sharply diminished. If you are going to manage data at scale, you need a data model-driven approach.

The tools must be usable by domain experts. To have the processes of model creation and maintenance mediated by software engineers is problematic: there may be misunderstandings regarding interpretation, but—not more importantly—there are not enough software engineers to go around. An appropriate user interface, that closely matches the intuition and expectations of domain experts, is essential.

There will be more models than you think. Different models will be required for different types of implementation, and—in any research domain, at least—data models will be constantly evolving, with data being collected against different versions.

Intelligent, automatic support is essential. The information content of precise data models is considerable, and there may be complex dependencies between data concepts and constraints. A considerable degree of automation is required if users are to cope with this complexity.

The model catalogue and the associated toolset should, as far as possible, automatically; create or propose links, including classifications; manage model versioning, and the consequences for linked data concepts; manage dependencies, including those between different models for same dataset, targeted at different implementation platforms.

This should come as no surprise. If, as Warner and Kleepe [18] suggest, the model-driven approach is about “using modelling languages as programming languages rather than merely as design languages” then we should aim to provide models with the same kind of support that programmers have come to expect from modern integrated development environments.

Acknowledgments

We would like to acknowledge the support of the National Institute for Health Research (NIHR), and in particular that of the NIHR Health Informatics Collaborative, and the financial support of the Medical Research Council. We would like to acknowledge also the support of the European Union Horizon 2020 Project ALIGNED. Project Number 645055. We are very grateful for the contributions made by colleagues at the University of Oxford: in particular, Steve Harris and Charles Crichton.

References

CHARIOT: A Domain Specific Language for Extensible Cyber-Physical Systems

Subhav M. Pradhan, Abhishek Dubey, Aniruddha Gokhale
Institute of Software Integrated Systems
Department of EECS, Vanderbilt University
Nashville, TN, USA
<firstname.lastname>@vanderbilt.edu

Martin Lehofer
Siemens Corporate Technology
Princeton, NJ, USA
<firstname.lastname>@siemens.com

Abstract
Wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices is facilitating a paradigm shift by allowing us to transition from traditional statically configured vertical silos of Cyber-Physical Systems (CPS) to next generation CPS that are more open, dynamic and extensible. Fractionated spacecraft, smart cities computing architectures, Unmanned Aerial Vehicle (UAV) clusters, platoon of vehicles on highways are all examples of extensible CPS wherein extensibility is implied by the dynamic aggregation of physical resources, affect of physical dynamics on availability of computing resources, and various multi-domain applications hosted on these systems. However, realization of extensible CPS requires resolving design-time and run-time challenges emanating from properties specific to these systems. In this paper, we first describe different properties of extensible CPS - dynamism, extensibility, remote deployment, security, heterogeneity and resilience. Then we identify different design-time challenges stemming from heterogeneity and resilience requirements. We particularly focus on software heterogeneity arising from availability of various communication middleware. We then present appropriate solutions in the context of a novel domain specific language, which can be used to design resilient systems while remaining agnostic to middleware heterogeneities. We also describe how this language and its features have evolved from our past work. We use a platform of fractionated spacecraft to describe our solution.

Categories and Subject Descriptors D.2.2 [Software Engineering]: Design Tools and Techniques

Keywords System description language, Model-driven development, Extensible Cyber-Physical Systems

1. Introduction
Cyber-physical Systems (CPS) consists of numerous sensors, actuators, network resources and computation resources that form cyber components used to monitor and control the surrounding physical environment by working closely with human operators at times. Traditionally, CPS have been designed as a single point solution with a focus on a specific domain. These systems are designed as composition of sensors, actuators, computation resource and networking technologies designed for specific purposes with self-contained resources and mostly closed architectures. This approach results in vertical silos of capabilities that do not support next generation CPS [13] – such as Smart Cities, fractionated spacecraft [5] – that require open, dynamic, extensible and interoperable solutions.

However, wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices such as wearables, smart phones, tablets provides us with a great opportunity to move away from traditional CPS towards next generation, extensible CPS. The key idea behind extensible CPS is the notion of open and extensible CPS platforms. These platforms are built not as a single function system but rather as potentially loosely connected networked platforms that “virtualize” and share their resources to host multi-domain cyber-physical applications that provide a variety of objectives to satisfy system goals. Extensible CPS takes inspiration from existing fields, such as cloud computing that provide elastic and multi-tenant computing resources. However, interaction with physical devices is rarely an issue in cloud computing where everything is virtualized without consideration for management of resources that are not part of the computation platform.

Dynamism, extensibility, remote deployment, security, heterogeneity and resilience are some of the key properties of extensible CPS. These systems are dynamic because physical entities that form different platforms can join or leave a group at any time. Similarly, these systems are extensible as physical or software resource can be added to existing platform. Security is also an important property as these platforms are open and therefore host applications belonging to different organizations with varying security requirements. Extensible CPS are heterogeneous since physical nodes that form a platform, as well as devices hosted on these nodes can be of different kinds. In addition to these hardware related heterogeneities, there can also be software related heterogeneities such as communication middleware hosted on each node. Finally, resilience is required because anything can go wrong at any time. As such, the platform must be able to handle both internal faults as well as environmental changes while ensuring that system properties and requirements are met.

The aforementioned properties of extensible CPS result in design-time and run-time challenges that need to be resolved in order to realize these systems. In this paper, we present our initial work, which focuses on design-time challenges arising specifically due to heterogeneity and resilience requirements. Below we describe each challenge addressed in this paper:

Challenge 1: The first challenge we address in this paper emanates from heterogeneous property of extensible CPS. Even though heterogeneity can be related to physical resources, we focus on software heterogeneity. To be more precise, we particularly focus on communication middleware because they serve as existing solutions to overcome hardware and operating system heterogeneity since they provide required abstractions to facilitate platform independent interaction between applications hosted on heteroge-
neous physical resources. However, there currently exists many different middleware solutions, such as RTI DDS [12], AllJoyn [2], MQTT [11], AMQP [18], etc. Some of these middleware, such as RTI DDS are implementation of a standard, while others are vendor-specific technologies; they all have their own advantages and disadvantages. Therefore arriving at common solutions is almost impossible and maybe undesirable. More importantly, these middleware do not provide a clean separation between the computation and communication aspects. As such, applications written using one middleware are completely tied to that particular middleware and have to be completely re-written in order to work with any other middleware.

**Challenge 2:** The second challenge we address in this paper emanates from resilience requirements of extensible CPS. An extensible CPS can host applications providing different objectives to satisfy different system goals. Some of these objectives are critical and it is of utmost importance to make sure that these objectives are not affected by failures and anomalies such that requirements for associated system goals are always met. This requirement necessitates a resilient system that supports self-reconfiguration mechanisms to facilitate autonomous fault tolerance. Autonomy in the resilience mechanism is importance since extensible CPS can be remotely deployed systems with limited opportunity for human interference. To support any run-time solution that performs self-reconfiguration, we require the design-time solution to allow modeling of design-time related resilience logic - such as configuration space of a system - that can be used at run-time.

In order to resolve the above-mentioned challenges, in this paper we present a Domain Specific Language (DSL) that is part of our application architecture CHARIOT (Cyber-Physical Application Architecure with Objective-based reconfiguration). CHARIOT DSL is a textual DSL developed using Xtext [6]. It incorporates lessons learned from previous experiences with DSLs for CPS. As such, we show how our approach to implementing DSLs has evolved and how our current solution can be used to design resilient systems while remaining agnostic to middleware heterogeneities. Following are the key contributions of this paper:

- **Contribution 1:** We address Challenge 1 by presenting a solution that enforces clean separation-of-concerns between computation and communication aspects and therefore allows users to design cyber physical applications while remaining completely agnostic to the underlying middleware that can be used by these applications to interact with each other. We also briefly describe how our design manifests in the run-time system.

- **Contribution 2:** We address Challenge 2 by presenting a solution that allows application designers to explicitly model systems goals, objectives and associated functionalities. These become part of a configuration space that will be used at run-time to support self-reconfiguration mechanisms.

The rest of this paper is organized as follows: Section 2 presents related work and compares them to our work presented in this paper; Section 3 uses fractionated spacecraft as a motivating scenario; Section 4 presents the evolution of our DSL and describes in detail how it facilitates design-time heterogeneity and resilience requirements; finally, Section 5 provides concluding remarks, and describes our on-going and future work.

### 2. Related Work

Extensible CPS are Distributed Real-time and Embedded (DRE) systems and these classes of systems are commonly constructed using component-based approaches. Architecture Analysis and Design Language (AADL) [7] is one such standardized architecture description language, which has notion of software components and hardware components that can be used to architect different systems. An AADL model is essentially a tree of components that can interact with each other via connections that are modeled as features. Our work presented in this paper is similar to AADL since we are also trying to achieve a generic system description language. However, our solution includes a well-defined (software) component model that captures interactions as part of components but with clear distinction from the computation aspect. In addition, AADL does not provide support for modeling resilience specific entities.

Currently there exist numerous component models targeted towards specific embedded system domains. For example, the Persvasive Component Systems (PECOS) [9] project describes a component model for embedded systems that is specifically tailored to field devices. Field devices are reactive, embedded devices fitted with sensors and actuators that are developed using the most inexpensive of hardware. AUTomotive Open System ARchitecture (AUTOSAR) [8] is an open and standardized automotive software architecture that supports a component model and is specifically targeted towards supporting vehicular design. These component models are specific to certain domain and they are, more often than not, tied to a specific middleware.

### 3. Motivating Scenario

Consider a platform of fractionated spacecraft, which is a cluster of independent satellite modules flying in formation and communicating with each other via ad-hoc wireless networks. Each independent satellite that is part of a fractionated spacecraft cluster, can come from different organization. Together these independent satellite modules provide an extensible CPS platform that facilitates sharing sensors and other computing and communication resources across multiple applications. This architecture can realize the functions of monolithic satellites at a reduced cost and with improved adaptability and robustness [4]. Several existing and future missions use this type of architecture including NASA's Edison Demonstration of SmallSat Networks, TANDEM-X, PROBA-3, and PRISMA from Europe. In each of these missions, the cooperating fractionated satellites are expected to provide the foundations for applications, used by many, possibly concurrent missions.

Individual satellite modules of a fractionated spacecraft cluster are present in the Low Earth Orbit (LEO), where one of the basic requirements is to be able to maintain orbital flight so that they can overcome the atmospheric drag and orbit the Earth while remaining in the LEO. Each individual satellite achieves this functionality by periodically using their thrusters to adjust their position. In addition to this flight control objective, which is required to be satisfied all the time by each individual satellite, the platform of fractionated spacecraft can be used to host different applications and depending on which application is hosted, more objectives are added and new system goals are created.

Figure 1 presents a schematic overview of an application that can be hosted on a platform of fractionated spacecraft to achieve a system whose goal is ImageSatelliteCluster i.e. to capture images of various resolutions and process them. As mentioned before, regardless of what applications are hosted on a fractionated spacecraft platform, it always needs to satisfy objectives required to maintain orbital flight of individual satellites. These are represented by objectives ClusterFlightPlanning and SatelliteFlight in Figure 1. The ClusterFlightPlanning objective is associated with the tasks of receiving ground commands, processing each command and calculating flight plan (new target location) for each satellite that is part of the cluster. The SatelliteFlight objective is associated with the tasks of receiving the aforementioned target locations and controlling satellite thrusters to move towards that position. The SatelliteFlight objective is a local objective. A lo-
A local objective implies that at least one of its functionality should be present in each node belonging to its associated node category. Imaging specific requirements are fulfilled by the Imaging objective.

Each objective is a collection of one or more functionalities provided by different components. For example, as shown in Figure 1, the Imaging objective is a collection of three functionalities provided by three different components (a) LowResolutionImageCapture to capture low resolution images, (b) HighResolutionImageCapture to capture high resolution images, and (c) ImageProcessor to process images with different resolutions. The components that capture low and high resolution images, publish those images for the image processing component to consume and process.

Generally, the components are based on a component model, which determines how components are designed, composed, managed as well as how they interact using their ports. Traditional component models are tightly coupled with specific middleware solutions, for example, Component Integrated ACE ORB (CIAO) [19] uses the ACE ORB (TAO) [17]. Furthermore, these component models do not support clean separation-of-concerns between their communication and computation aspects. This becomes problematic when we consider the above described fractionated spacecraft scenario where individual or a group of satellite modules can come from different organization and therefore support different middleware. This results in communication heterogeneity and thus necessitates resolution of Challenge 1 described in Section 1.

All three objectives of the imaging satellite cluster presented above are critical to achieving the associated system goal. As such, when there are failures or anomalies in the system, it is of utmost importance that the system adapts itself to recover and maintain all objectives for as long as possible. This necessitates resolution of Challenge 2 described in Section 1.

4. CHARIOT DSL

This section presents detailed description of our solution and show how it resolves challenges listed in Section 1. We first present a brief description explaining how our work has evolved to meet the requirements of next generation CPS. Then we present detailed description of relevant aspects of CHARIOT DSL that addresses the different challenges.

4.1 Evolution of our DSL from prior efforts

Our initial approach [3] towards achieving a DSL for next generation CPS focused on platforms of fractionated spacecraft, which had static communication channels declared at design-time in order to guarantee secure interaction between applications at run-time. Furthermore, we had static deployment and configuration plans generated from systems modeled at design-time. These plans contained information about artifacts, parameters, and communication flows of different application components. They did not capture information about other aspects of the system such as resource availabilities, constraints, system objectives and goals; these are critical configuration information required to achieve a resilient system. Finally, since our initial approach was designed for a specific CPS domain – fractionated spacecraft – capturing and supporting heterogeneity was not a consideration at that time.

Second phase of our effort focused on advancing our initial approach by supporting resilience. We supported modeling of configuration space instead of a specific deployment plan [15]. A configuration space represents the state of an entire platform. It includes information about different resources available, well known faults, system goals, objectives, application components, where these components are deployed and how they are configured. A configuration space can contain multiple configuration points, where each configuration point represents the state of the entire platform at any given time. Resilience in the run-time was facilitated by self-reconfiguration mechanisms that used configuration space to compute and transition to a new configuration point.

Third phase of our effort is our current solution presented in this paper. In this phase, similar to the last, we allow users to model the configuration space for different systems. In addition, we also provide mechanisms to model communication heterogeneity by enforcing a strict separation-of-concerns between communication and computation logic thereby allowing users to model their applications using generic interaction patterns that can be implemented on top of any communication middleware.

Figure 2 presents different first class modeling concepts and their interdependencies in CHARIOT DSL. The responsibility of modeling these concepts are assigned to three different roles (a) application developers, who are responsible for modeling Data Types, Functionalities, Compositions, and Components belonging to their applications, (b) SDK developers, who are responsible for modeling Platform Interactions corresponding to platform interaction libraries they develop, and (c) systems architects, who are responsible for modeling Node Categories, Nodes, and Systems. Table 1 presents a brief summary of these modeling concepts.
Table 1. Table summarizing different modeling concepts supported by CHARIOT DSM-L.

<table>
<thead>
<tr>
<th>Modeling concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data types</td>
<td>Most basic modeling construct. It facilitates modeling of data types that will be used for interaction as well as computation.</td>
</tr>
<tr>
<td>Functionalities</td>
<td>Logical concepts used to compose objects. Functionalities are associated with components, a component can provide one or more functionalities.</td>
</tr>
<tr>
<td>Compositions</td>
<td>Logical groups of functionalities, where each functionality can be part of multiple compositions. Dependencies between functionalities of a composition are also modeled as part of the composition. Objectives are instantiations of compositions.</td>
</tr>
<tr>
<td>Components</td>
<td>Applications in CHARIOT are composed of components that communicate with each other. Components have well defined ports for interaction and use workflows and tasklets to describe computational behavior.</td>
</tr>
<tr>
<td>Platform interactions</td>
<td>Artifacts that can be used to interact with platform specific resources.</td>
</tr>
<tr>
<td>Node categories</td>
<td>Categories to which different nodes belong.</td>
</tr>
<tr>
<td>Nodes</td>
<td>Different nodes that are part of a platform.</td>
</tr>
<tr>
<td>Systems</td>
<td>A system consists of a goal that is satisfied by one or more objectives. An objective depends on functionalities provided by components.</td>
</tr>
</tbody>
</table>

4.2 Addressing Challenge 1: Supporting communication heterogeneity

In order to facilitate communication heterogeneity, we enforce strict separation-of-concerns between the communication and computation logic of application components. Having the computation logic clearly separated from the communication logic results in more predictability, which is an important real-time property. Similarly, having a communication logic clearly separated from computation logic results in highly configurable communication and therefore support for heterogeneous communication middleware. Using software components to design distributed application is not a new concept as significant amount of prior work has been done in the field of Component-Based Software Engineering (CBSE) [10]. However, as mentioned in Section 3, existing component models are generally tightly coupled to a specific middleware solution. This is why our approach is different; in essence, we are providing a universal component model, which does not depend on any specific communication middleware. This aspect of our work aligns well with current efforts of the Object Management Group (OMG) to achieve a Unified Component Model (UCM) [1].

CHARIOT supports different kinds of ports that can be used by application developers to model common interaction patterns – such as point-to-point (client/server) interaction and group publish/subscribe interaction – that are supported by most middleware solution. Therefore, this allows application developers to focus on modeling application interaction using different ports without having to worry about what middleware will be used to support those interactions at run-time. Different kinds of ports supported by CHARIOT are - (a) client port, which is used to send request to one or more server ports, (b) server port, which is responsible for receiving requests from one or more client ports, (c) buffered receiver port, which receives messages sent from one or more sender ports and stores them in a buffer of predefined size, (d) sampling receiver port, which also receives messages sent from sender ports but unlike buffered receiver port, it does not use a buffer to store multiple messages, and (e) sender port, which is used to send messages to one or more buffered or sampling receiver ports.

Figure 3. Snippet of OrbitController component declaration.

In order to show how interactions are modeled at design-time, Figures 3 and 4 present snippets of declaration of OrbitController and SatelliteBus components that are part of the imaging satellite cluster, previously presented in Section 3. As shown in Figure 3, the OrbitController component uses thruster_control_sender_port (line 8) as sender port to send thruster_control_message, and state_client_port (line 5-7) as client port to send sat_state_request and receive sat_state_message. Similarly, the SatelliteBus component, shown in Figure 4, uses state_server_port (line 5-6) as server port to receive sat_state_request and send sat_state_message, and thruster_control_receiver_port (line 7-8) as buffered receiver port in order to receive thruster_control_message.

Figure 4. Snippet of SatelliteBus component declaration.

At the very core of supporting communication middleware heterogeneity is the fact that we support generic data types that are supported, in one form or other, by most existing middleware. This is what ensures interoperability between different middleware solutions, as component ports are associated with these generic data types. For example, the thruster_control_sender_port in Figure 3 send data to type thruster_control_message, whose declaration using CHARIOT DSL is shown in Figure 5. Table 2 shows how our data types map to that of Java, as well as data types supported by different middleware. Table 2 does not include MQTT [11] since it is totally data-agnostic and virtually every data can be sent in its binary format; serialization and deserialization must be handled by application developers.

Figure 5. Snippet of data types declaration.
Table 2. Table showing how CHARIOT DSL data types map to that of Java and different existing middleware.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
<td>DDS_Float</td>
<td>N/A</td>
<td>float32</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>DDS_Double</td>
<td>DOUBLE</td>
<td>float64</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>DDS_Short</td>
<td>INT16</td>
<td>int16</td>
<td>int16_t</td>
<td>short</td>
</tr>
<tr>
<td>long</td>
<td>int</td>
<td>DDS_Long</td>
<td>INT32</td>
<td>int32</td>
<td>int32_t</td>
<td>int</td>
</tr>
<tr>
<td>long long</td>
<td>long</td>
<td>DDS_LongLong</td>
<td>INT64</td>
<td>int64</td>
<td>int64_t</td>
<td>long</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>DDS_Char</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>char</td>
</tr>
<tr>
<td>wchar</td>
<td>wchar</td>
<td>DDS_WChar</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>wchar</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>DDS_Boolean</td>
<td>BOOLEAN</td>
<td>bool</td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>octet</td>
<td>byte</td>
<td>DDS_Octet</td>
<td>BYTE</td>
<td>N/A</td>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>String</td>
<td>STRING</td>
<td>string</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>struct</td>
<td>struct</td>
<td>STRUCT</td>
<td>STRUCT</td>
<td>structure</td>
<td>structure</td>
<td>composite type</td>
</tr>
<tr>
<td>sequence</td>
<td>array</td>
<td>sequence</td>
<td>ARRAY</td>
<td>array</td>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
<td>DDS_Enum</td>
<td>BYTE, INT16, INT32, INT64</td>
<td>constants</td>
<td>constants</td>
<td>restricted type</td>
</tr>
</tbody>
</table>

Figure 6. Run-time mapping of component communication logic.

Figure 6 shows how the CHARIOT component communication logic maps to the run-time. As shown in the figure, an application container hosts transports, transport proxies, and components. Transports allow interaction with specific middleware, whereas transport proxies facilitate interaction between transports and component ports. These entities constitute the communication logic and are strictly separated from the computation logic. Depending upon availability of appropriate transports and transport proxies, a component can use any middleware.

4.3 Addressing Challenge 2: Modeling resilient systems

Maintaining system goals is of utmost importance for extensible CPS. Therefore, resilience is a very important property of extensible CPS. In order to be resilient, a system must support run-time mechanisms to monitor, detect, diagnose, and mitigate failures and anomalies. In addition, resilience requirements of extensible CPS also necessitate design-time solutions that allow modeling of appropriate information at design-time such that they can be used to facilitate run-time resilience mechanisms.

Since our ongoing work on run-time resilience mechanism is based on self-reconfiguration capabilities, our design-time tool allows an application developer to model (a) a complete configuration space of a system, and (b) an initial configuration point, as a complete or partial deployment specification. A configuration space represents the state of an entire platform. It includes information about different resources available, well known faults, system goals, objectives and corresponding functionalities, components that provide different functionalities, where these components are deployed, and how they are configured. A configuration space can contain multiple configuration points and a configuration point represents state of the associated platform at any given time; change in state of the platform is represented by transition from one configuration point to another.

Figure 7. Snippet of imaging satellite cluster system declaration.

Figure 7 presents a declaration of the imaging satellite cluster system previously presented in Section 3. In order to model a system in CHARIOT DSL, we need to declare its goal (line 3), different objectives (line 4-11), deployment constraints (not shown in Figure 7), and initial deployment specification (line 12-25). As shown in the figure, the system comprises three objectives where the SatelliteFlight objective is a local objective that applies to all nodes of Satellite category. This local objective implies that OrbitController and SatelliteBus functionalities should be present in each node of Satellite category.

Above described system declaration is part of the overall configuration space of the imaging satellite cluster. In the run-time resilience infrastructure, which is part of our ongoing work, we store configuration space and points in a distributed database such as MongoDB. As such, a configuration space becomes a list of collection, where each collection can contain multiple documents. Due to space restriction we do not show the entire configuration space, but Figure 8 shows part of the configuration space of imaging satellite cluster by presenting snippet of the system declaration presented in Figure 7. Each system declaration stored has - (a) an associated id, (b) constraints, (c) name of the system, which also represents system goal, (d) a list of objectives, which themselves

1 http://www.mongodb.org
contain constraints, list of required functionalities and their dependencies, name, and a node category to represent objective locality.

5. Conclusions

Traditionally CPS have been designed as vertical silos of capabilities. With wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices, we are moving towards the paradigm of extensible CPS wherein the sensors, actuators, and computing resources of one or more CPS form an open platform whose resources can be “virtualized” and shared among different applications. However, practical realization of extensible CPS requires us to resolve design-time and run-time challenges emanating from system specific properties such as heterogeneity and resilience.

This paper focuses on resolving design-time challenges related to communication heterogeneity and resilience. We present our solution for these challenges in the context of CHARIOT DSL, which is a novel domain specific language that supports - (a) a component model with clean separation-of-concerns between the communication and computation aspects to handle communication heterogeneity, and (b) configuration space modeling in order to capture information required to facilitate run-time self-reconfiguration mechanisms. Our ongoing work focuses on realization of a complete end-to-end system, which includes a model interpreter, a complete runtime solution that includes our universal component model, and a comprehensive resilience infrastructure. In future we plan to extend our work by using model checkers to perform formal verification of system models at design-time, and supporting redundancy patterns to enhance run-time resilience mechanism.

Acknowledgments

This work is sponsored in part by a research grant from Siemens Corporate Technology. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Siemens Corporate Technology.

References


Figure 8. Snippet of system description that is part of the overall configuration space of imaging satellite cluster.
Experience Report: Constraint-Based Modeling of Autonomous Vehicle Trajectories

Kennon McKeever
University of Arizona
kennonmckeever@email.arizona.edu

Yegeta Zeleke
UC Santa Cruz
yzeleke@ucsc.edu

Matt Bunting
University of Arizona
mosfet@email.arizona.edu

Jonathan Sprinkle
University of Arizona
sprinkjm@email.arizona.edu

Abstract

Autonomous vehicles and other robotics systems are frequently implemented using a general-purpose programming language such as C++, and prototyped using domain-specific tools such as MATLAB/Simulink, and LabVIEW. Such an approach is not efficient when programming primitive motions of autonomous vehicles when considering important safety constraints, and when promoting the broad access to robotic systems through involvement of students and aspiring students who do not know conventional low-level programming languages. Aside from general-purpose programming languages, there are languages that are specifically designed to model autonomous vehicles, such as SHIFT [3], but these languages are typically for simulation purposes only. This experience report discusses the creation of a domain-specific language that allows for faster programming of autonomous vehicles while ensuring valid constraints will be met. This language generates code for multiple controllers that will operate alternatively to allow for fast and effective programming of vehicle trajectories using primitive motions. In addition to improving coding efficiency and reducing the number of programming errors, the language adds a level of abstraction so that autonomous vehicle behaviors may be generated by people with little knowledge of low-level details of the car’s operation. Furthermore, this language ensures safe operation of the vehicle by enforcing a set of user-definable constraints on the output path. A main set of constraints that are applied to every generated path have been specifically chosen to enforce safe switching between controllers and prevent the planning of unsafe actions. A novel application of the language is its ability to permit users to add specific constraints for a particular path; these constraints are checked for validity after the main constraint check is performed.

Keywords Autonomous Systems, Metamodelling, Cyber-Physical Systems

1. Introduction

Self-driving cars, also known as autonomous vehicles (AVs), are vehicles capable of operating without human control. Autonomous vehicles have multiple sensors that assess the environment and provide feedback in order to support safe vehicular function. The development of advanced sensors, communication media, outstanding planning algorithms and control technology has empowered the advancement of AV technology. Although this achievement provides a great opportunity to enhance autonomous vehicle safety, ensuring safe trajectories while considering vehicle dynamics in dynamic environments is still an open research problem. The degree of freedom in velocity and position provide infinite possible outcomes of trajectories for a given time horizon. While it is not difficult to implement flawless controllers that are safe under certain assumptions and restrictions, it is much harder to develop an extensible controller that continues operating safely when outside of the conditions it was designed for. One solution to this issue in terms of autonomous vehicles is to add constraints on the allowable trajectories to ensure that the controllers operate within the safe range that they were designed for. This can be achieved in a clean and effective manner with the help of a domain-specific language.

A domain-specific language (DSL) is a programming language that offers appropriate notations and abstractions for a particular domain application. Designing a DSL requires a careful consideration of the underlying application domain. A well designed DSL yields an increase in reliability, optimizability, portability, testability and consistency [13]. Furthermore, a DSL allows domain experts to remain as experts in their domain, without the need to become experts in other domains required in a compositional system.
In this paper, we explain how we defined a domain-specific modeling language using a Generic Modeling Environment (GME), MetaGME. We begin by summarizing our approach and presenting relevant background information. Next, we describe our approach in-depth with examples of our solution. Then, we describe our approach to designing controllers used within our project and how we can guarantee safety with these controllers. Finally, we discuss the education applications and extensibility of our research.

2. Background

The popularity and usage of domain-specific languages has been growing over the years; in the past few years, they have attracted attention from researchers due to their various applications. DSLs are different from general-purpose programming languages in the sense that they are specialized for a specific domain or problem, such as HTML for the creation of web pages [14], SQL for developing queries on relational database [8], data storage and exchange languages such as XML and many others. DSL programs are relatively synced, self-documenting, and have various use cases [5].

A well-formed DSL is capable of optimizing a system at a domain level. Despite their many benefits, the limited availability of DSLs and the difficulty of obtaining valid scope for a DSL are major challenges when considering DSLs [5, 14].

Programming autonomous vehicles is generally done with low-level general-purpose languages and middleware. Although there are standards to the interface of the autonomous vehicle, such as the Joint Architecture for Unmanned Systems (JAUS), and there are methods to send JAUS command using Robot Operating System (ROS) [11], there is no standard language used to program these vehicles. Safety is a major concern in autonomous vehicles, and simple software mistakes may be catastrophic when implementing new control software. A single system may be developed and be provably safe for a certain behavioral model, however later changes to the model may require further tests to be performed. High scalability in the development of safe autonomous vehicles is not achievable when using low-level languages. Instead, small primitive behaviors may be designed to be provably safe. Domain-specific modeling languages may further be used to combine such primitive behaviors to form a more complex model and quickly generate code for high level changes. Some behaviors may not be combinable depending on the state of the vehicle, so primitive motions may have constraints that must be met before being performed. The domain-specific modeling language may also perform constraint checking to ensure that constraints are not violated for any primitive motion.

3. Method

3.1 Primitive Motions

In order to effectively allow for the programming of an autonomous vehicle at a high level with a modeling language, we need a way to represent a set of primitive motions of the car. For this language, we divide the car’s motions into straight and turn types. These were defined as primitive motions within our study because they are very basic operations of the car that are easily understandable by many people.

In order for more complex trajectories to be designed, these primitive motion types were given attributes to modify their behavior. All path pieces have a velocity, which is the desired final velocity of the path piece. Turns have a radius of curvature and final turn angle (as shown in Figure 1) to describe the arc taken by the vehicle. Straight path pieces, on the other hand, contain a distance attribute. These attributes come with default values to allow for rapid generation of very simple paths; however, it is more useful to combine multiple primitive motions with set attributes to form more complex paths.

3.2 GME

In order to implement our own domain-specific modeling language, we used a generic modeling environment called MetaGME [7]. This is an environment that essentially allows the user to create a modeling environment using predefined components. Then, the created modeling environment can be registered and used within GME. This new environment now forces the user to follow a specific design pattern and syntax. Additionally, the modeling environment can call interpreters created in C++ that can be used to perform virtually any desired task.

Our implementation of the DSML in MetaGME is shown in Figure 2. From the metamodel, it is evident that paths may contain the primitive motions discussed above in section 3.1. Additionally, path models can contain other path models. This is important because it allows for paths to become new building blocks that can be used in future paths. (See Section 3.3)

3.3 Functional Programming

In order to ensure that larger projects are as manageable as smaller projects, paths are able to contain other paths, as shown in Figure 2. This is analogous to using functions in a functional programming language such as C. Although some
paths need to be defined purely out of primitive motions, path models can be built using other existing path models. Thus, the user can continue to work efficiently as a project grows larger. As seen in Figure 3, both primitive motions and a defined path are used to define a vehicle trajectory.

3.4 Metaprogramming

In programming, there are often repetitive tasks that come up and must be done repeatedly throughout the span of the project. In order to speed up development times, it is common to resort to metaprogramming. Metaprogramming allows for programs to be generated from within another program. For instance, our constraint language is able to take short high-level program statements as inputs and generate MATLAB software to check the programmed path’s constraints. That way, the programmer is saved from having to write the setup code, and the overall program is produced more quickly and with a much less chance for errors.

For this particular work, we used metaprogramming in the form of code generation. When the interpreter for our DSML is run, it uses a template and sub-templates to generate a MATLAB script that checks all of the constraints defined for the described path. This approach was chosen because template-based code generation is verifiably syntactically correct [1] and it is easy to update the templates to change or add functionality of the code generator.

3.5 Intermediate Path Format

In addition to the domain-specific modeling language created in MetaGME, we chose to generate a small file to represent the path in what we called the Intermediate Path Format. Although we could have generated the controller code directly from our MetaGME interpreter, we decided that direct creation of the controller code would be less flexible. By using an intermediate format, the controllers and modeling environment can be operated independently. Just as how intermediate languages have been created to reduce the amount of effort needed in developing new programming languages and supporting new architectures [9], our prototype Intermediate Path Format will allow easy re-use of the controllers and modeling environment we’ve created in future projects. (See Section 5.2)

3.6 Checking Path Safety

In addition to the path paradigm in GME, we have created a constraint language that allows the user to add restrictions to each of the defined paths. The constraints defined in this language can be applied to a single primitive motion type, multiple primitive motion types, or transitions from one primitive motion type to another, as in Figure 4. As an example, a constraint could be used to limit the maximum velocity of the AV in any given primitive motion type, as shown in Figure 5.

After the constraints are defined, an interpreter in GME is called that checks the states of each primitive motion and makes sure that none of the defined rules are violated. If the interpreter encounters a rule violation, it prints the error and refuses to generate code to run on the car, as shown in Figure 6 where the maximum velocity constraint from above was violated. By treating a failed constraint condition as a compile error, the user can be certain that any code generated is verified as safe with respect to both our pre-defined constraints and user-defined constraints.

These constraints are useful because planning paths using primitive motions can lead to unsafe behavior. For example, the user may use primitive motions to define a path where the vehicle is commanded to accelerate to a high velocity and take a sharp turn. With the constraint language, it is possible to prevent the user from describing unsafe actions for the car by simply preventing primitive motions from being combined in an unsafe sequence. Moreover, the constraint language is also useful because it can apply restrictions on primitive motions that will aid in ensuring controller safety, such as preventing states that would be unsafe in switching control.

Figure 2: Metamodel of our DSML

Figure 3: An example path defined in our DSML
We decided that switching control was an effective way to implement controllers for our modeling language. We designed a controller for each primitive motion described in section 3.1. The transitions between these controllers then corresponded with the natural transitions defined in the user’s sequence model of the vehicle trajectory, yielding a simple way to define the rule used to govern the individual controllers.

In addition to these natural transition points, it is important to note that the implementation of certain controllers may require additional transitions. For example, one might choose to add more controllers that handle changing speeds more effectively than the basic controllers. In this scenario, one would have to identify what conditions justify switching to these new controllers and when these conditions occur. In our implementation however, there was no need for these intermediary controllers because we prevented such conditions from occurring due to the constraint language.

### 4.2 Switching Control Safety

Our system is guaranteed to be safe in the sense that it is impossible for the system to transition into an invalid state. An example of an invalid state in this sense would be taking a sharp turn immediately following a piece of the path with a high velocity. As described in [2], there is an algorithm to find safe switching surfaces within our state space. This can be done in future work but our current implementation involves very basic controllers, so we guaranteed that the controllers remain in safe states by imposing restrictions on generated paths with the constraint language described in section 3.6.

### 5. Applications

#### 5.1 Use In Education

As mentioned previously, our modeling environment is extremely powerful in the sense that paths can be planned out and generated in a matter of minutes. Like most DSLs, the high-level nature of the language means that it can be programmed more easily by a larger group of people [10], including people that are not experts in control systems. Thus, high school students will be able to use our path-planning language to design vehicle trajectories with minimal knowledge of how the car works. Even without understanding the mathematics used to calculate the primitive motion attributes, they will still be able to design very simple paths using the primitive motions’ default attribute values discussed in Section 3.1.

#### 5.2 Project Extensibility

Due to the simple nature of the intermediate path format described previously, our environment can easily be used in the development and testing of other programs. For example, one would be able to test a path-finding algorithm on an autonomous car as long as the algorithm can output the
desired primitive motions. This would prevent the user from having to write their own controllers or learn how the vehicle operates. Additionally, the global constraints will make sure that the algorithm comes up with a feasible path.

The use of the intermediate path format also allows for faster development of new controllers for our vehicle. Although we have developed basic controllers to work with our DSML, one might choose to update these controllers to better suit their needs, such as improving primitive motion accuracy or smoothing transitions between primitive motions. As long as these primitive motion controllers are designed to accept their inputs from the intermediate path format, our DSML and global safety constraints can be used to quickly design safe new paths used to test these controllers. Most importantly, the constraint language can be used to impose further limitations on path design that will guarantee safe conditions for each individual controller and valid transitional states in switching control.

Finally, the intermediate path format can be used to allow our path-planning modeling environment to be used by other vehicles. Since the controllers we have created are not necessarily compatible with all autonomous ground vehicles, new controllers that operate on the intermediate path format that are made for a different vehicle can be used in place of our basic controllers. Then, the general primitive motion restrictions defined in our constraint language could be updated to reflect the attributes of this new AV. The end result is that the DSML that we’ve created will essentially be extended to work with this new vehicle as well.

6. Demonstration and Results
On August 11, 2015, the modeling language was used to demonstrate the CAT Vehicle in front of an audience of graduate students, faculty, visitors to the University of Arizona, and various broadcast media. In the live demonstration, two different trajectories were demonstrated:

1. a trajectory that performed a figure-8 style maneuver around several predefined obstacles; and
2. a trajectory of straight and turning primitives, which was designed to end in exactly the same place where it began.

In Figure 7 images of the live demonstration are featured. In these images the vehicle is under autonomous control of the synthesized software from the modeling language, and the output code is verified to obey each of the runtime physics-based constraints using the approach defined in previous sections.

The results were robust enough to provide “rides” in the vehicle to visitors, and some of the rides were featured in stories covered by the various broadcast media outlets [4, 12].

7. Conclusion
In this article, we have introduced a domain-specific modeling language for programming autonomous vehicles. This is a much more effective than current methods for programming AVs because the DSL is much more expressive than a general-purpose language. Additionally, larger projects are easy to manage because the DSML allows for metaprogramming.

The constraint language included is also a crucial part of the DSML because it allows the programmer to guarantee the safety of a trajectory using the predefined set of constraints. It also allows for the creation of user-defined constraints that can guarantee custom controller safety, guarantee safe operation of a different AV, and offer adaptability to the needs of the testing environment.
8. Future Work

Due to rain in the live demonstration, the approach to utilize dead reckoning localization resulted in several behaviors that strayed from their intended trajectories. Although the vehicle never violated any of its safety constraints, these constraints did not include some obstacles that are not accounted for a priori, so integration of various additional sensors is an important safety feature to add to the system setup.

Further, the trajectory synthesis depends on knowledge of local sensor values and integration of those values to achieve distance estimates; future efforts may utilize global sensors (such as GPS) to transform local estimate goals into projections into the global sphere, in which case global sensors can be utilized to ensure that heading and distance traveled can be accurately computed.

Acknowledgments

This work was supported in part by the National Science Foundation and the Air Force Office of Scientific Research, under awards IIS-1262960 and CNS-1253334. Thanks to other members of the CAT Vehicle 2015 REU Site at the University of Arizona for their help in testing the vehicle, and providing safety supervision for the test area on the date of the demonstration.

References

Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures

Katrin Hölldobler * Pedram Mir Seyed Nazari Bernhard Rumpe
Software Engineering, RWTH Aachen University
{hoelldobler,nazari,rumpe}@se.rwth-aachen.de

Abstract
Many textual software languages share common concepts such as defining and referencing elements, hierarchical structures constraining the visibility of names, and allowing for identical names for different element kinds. Symbol tables are useful to handle these reference and visibility concepts. However, developing a symbol table can be a tedious task that leads to an additional effort for the language engineer. This paper presents a symbol table meta model usable to define language-specific symbol tables. Furthermore, we integrate this symbol table meta model with a meta model of a grammar-based language definition. This enables the language engineer to switch between the model structure and the symbol table as needed. Finally, based on a grammar annotation mechanism, our approach is able to generate a symbol table infrastructure that can be used as is or serve as a basis for custom symbol tables.

Keywords meta model, symbol table, code generation

1. Introduction
Developing a (domain-specific) modeling language or general purpose language involves a multitude of design decisions including the concrete concepts it should be capable of. Textual languages are usually defined by a grammar which results in a tree like structure of the models internal representation, the abstract syntax tree (AST).

Most textual languages share some common concepts. Typically, a language allows the user to define model elements (resp. program entities [15]) and refer to those from the same model or from different models. For instance, a type of a field in a Java class can refer to another Java class. Furthermore, in many languages some kind of import mechanism provides access to elements of other models. Moreover, some languages provide hierarchical structured elements which enable shadowing names or using identical names for different kinds of elements. For example, fields and methods in Java may have the same name even within the same class.

Handling references and visitibilities requires some kind of resolving mechanism that can either be realized by the underlying AST or handled by an additional structure such as a symbol table [1]. A symbol table can be a simple name-information mapping or even a more elaborate data structure that includes the semantic model [3] and can even be used for black box integration of models [12]. However, developing an additional structure can be a tedious task that leads to an additional effort for the language engineer.

Thus, this paper presents an approach to ease the creation of language-specific symbol tables. Therefor, we defined a meta model for symbol tables containing first-level classes for, among others, reference and visibility concepts. By designing this at the M3 meta level, on the subjacent meta level a concrete instance of this symbol table can be created and is fully typed.

Furthermore, whether the AST or the symbol table is better suitable depends on the task to be done, such as generating code, validation, and model integration. In order to enable switching between these different data structures, we integrate the meta models of the symbol table and grammar at the M2 level.

Finally, we adapted the existing annotation mechanism of the MontiCore grammar format [11] to be able to generate a completely systematic symbol table as an instance (i.e., M3 model) of the symbol table M3 model. This can either be used directly or serve as a basis for the creation of a custom symbol table.

In sum, this paper’s contribution is (1) a language-independent meta model for symbol tables that allows defining language-specific symbol tables, (2) an integration of this meta model and the grammar meta model, (3) a naming convention for annotations of grammar elements using the example of a MontiCore grammar and (4) based on it the generation of a symbol table that can be used as is or serve as a basis for custom symbol tables.

In the following we first explain the different meta levels involved in this approach in Sect. 2 and examine the grammar meta model in Sect. 3. Thereafter, the symbol table meta model is described in Sect. 4, while in Sect. 5 the integration of both meta models as well as the symbol table generation is explained. Finally, related work is discussed in Sect. 6 and the paper is concluded in Sect. 7.

2. Meta Modeling
Our approach of a meta model for symbol tables in conjunction with the generation of a completely systematic symbol table acts on the different meta modeling levels, thus, in this section we give a brief overview of meta modeling levels and clarify which meta level is meant by M0, M1, M2 and M3.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3 meta level</td>
<td>grammar describing grammars</td>
</tr>
<tr>
<td>M2 meta level</td>
<td>instances</td>
</tr>
<tr>
<td>M1 model level</td>
<td>grammar</td>
</tr>
<tr>
<td>M0 system level</td>
<td>model</td>
</tr>
</tbody>
</table>

* K. Hölldobler is supported by the DFG GK/1298 AlgoSyn.

Figure 1. Overview of the Different Meta Levels

In meta modeling a distinction is made between different levels/layers of meta modeling referred to as M0, M1, M2, M3, etc., where every level Mn is considered as an instance of the level Mn+1 [2, 14]. The lowest meta level is M0 which is the real world system (cf. Fig. 1). As our approach targets language design the lowest instance level (M1, model level) considered here is a concrete model,
3. Grammar Meta-Model

Using MontiCore [11] a modeling language is defined by an EBNF-like grammar. A simplified meta model\footnote{Please note that the $M_2$ model corresponds to the AST.} of a MontiCore grammar in form of a class diagram is shown in Fig. 2. Thus, a grammar consists of a set of productions. Each of which consists of an arbitrary number of terminals and nonterminals. Similar to EBNF, every nonterminal is defined by one production. Additionally, each grammar element (not shown for Terminal) can be annotated with further information, e.g., for documentation purposes.

An excerpt of a simplified meta model for the Java programming language is shown in Fig. 3. This meta model is an instance of the meta model of a grammar shown in Fig. 2. However, for readability reasons we omitted all terminals, chose different names for nonterminals and their defining productions (I-prefix for nonterminals is omitted) and denoted the instance relation in form of stereotypes, e.g., <<nonterminal>>. The JClass production in Fig. 2 represents a Java class. It consists of several Method and Field nonterminals. Field is defined by the production JField that represents a Java field or variable declaration, while Method is defined by JMethod that represents a Java method. A JField has a Type nonterminal defined by the Name production. Furthermore, a JMethod consists of several While nonterminals defined by JWhile. JWhile represents a while loop which, among others, consists of fields.

4. Symbol Table Meta-Model

Many software languages share same or similar concepts, such as: a) The possibility to define and reference model elements. b) References to elements of the same model as well as of another model are allowed. The latter includes the (re-)loading of models. c) The ability to shadow names that are already defined. d) Import statements to enable the use of simple names.

\footnote{We deliberately omitted different production kinds and classes representing alternatives etc. to narrow the meta model to parts relevant for the presented approach.}

Often, these tasks are conducted by so-called symbol tables (see Sect. 4.1). In the following, we will present the symbol model $M_3$ of MontiCore. For this, we introduce some core concepts that are common in many languages by taking the example of Java and present the corresponding Java symbol table $M_3$ model.

Many of the presented concepts and mechanisms are complex and must usually be fully understood by the language engineer in order to apply them. Therefore, the aim of the $M_3$ model is to encapsulate the complexity within the framework and enable the generation of reasonable defaults for a concrete language on the $M_2$ level (see Sect. 5).

4.1 Symbols and Symbol Tables

In general, languages have different kinds of model elements, e.g., classes, methods, and fields in Java, each of which has its specific information. Java classes, for example, can be abstract or final and may sub-class other classes. A method can define, among others, a parameter list and a return type. The model elements are represented by a symbol. A symbol contains all essential information about a named model element and has a specific kind depending on the model element it denotes. Additionally, a symbol can provide information that is not directly part of the model element, but useful for the language engineer (resp. generator engineer). For example, a symbol representing a Java class could provide easy access to all non-private fields and methods of all its direct and indirect super types. A symbol table (ST) is a data structure that maps names to symbols. It allows to effectively organize and find declarations, types, signatures, implementation details etc. associated with a symbol (resp. model element). A ST consists of a scope-tree (see 4.2) with an associated collection of symbols at each scope.

The $M_3$ model for symbols and symbol kinds is shown in the top part of Fig. 4. The $M_3$ class Symbol has exactly one SymbolKind, whereas a SymbolKind can belong to several Symbols.\footnote{Representing a symbol kind by its own class simplifies the integration of heterogeneous models [5]} The corresponding (shortened) $M_2$ model for Java is presented in the bottom part of Fig. 4. The $M_2$ class JClassSymbol is an instance of the $M_3$ class Symbol. Its associated kind JClassSymbolKind is an instance of the $M_3$ class SymbolKind. Analogously, JFieldSymbol and JFieldSymbolKind are instances of Symbol and SymbolKind, respectively. Same applies to methods (not shown).

4.2 Scopes

A scope holds a collection of symbol definitions. In Java, for example, methods and fields are defined in a class scope. Scopes are structured hierarchically, i.e., they can have a direct enclosing scope and several sub-scopes. The resulting structure is a scope-tree (resp. scope-graph) modeled by the enclosing-maps association of the $M_3$ interface Scope in Fig. 5.

Symbol Visibility Scopes limit the visibility of symbols, i.e., the logical region where the symbol is accessible by its name. For instance, a local variable in Java is only visible within the method scope it is defined in. Outside the method, the local variable is "out-of-scope".
4.3 Scope Spanning Symbols

Symbols that span (i.e., define) a scope themselves are called scope spanning symbols. A Java class symbol, for instance, spans a scope to enable field and method definitions within that scope. Fig. 7 shows that the JClassSymbol is not just an instance of Symbol (as shown in Fig. 4) but strictly speaking an instance of ScopeSpanningSymbol. It spans a shadowing scope, namely a JClassScope. Analogously, JMethodSymbol is a scope spanning symbol spanning a JMethodScope.

Please note that on the M3 level the relation between ScopeSpanningSymbol and Scope has the cardinality 0..1-to-1, which means a Scope may optionally be spanned by a symbol. However, on the M3 level a scope is either always spanned by a symbol or never. For example, a JClassScope is always spanned by a JClassSymbol since the cardinality is 1-to-1. In contrast, a JWhileScope is never spanned by a symbol, as no association to a symbol exists (i.e., 0-to-0).

4.4 Symbol References

Symbol references refer to symbols that are defined elsewhere, e.g., in other scopes. A symbol definition exists exactly once and is stored in a scope. In contrast, several symbol references may exist, which are managed in the referencing symbol. In Java, for example, a class C refers to its super class S, since S is defined elsewhere, e.g., in another file. Fig. 8 shows the corresponding M3 classes. The definition association has the cardinality 0..1 instead of 1, since a SymbolReference could refer to a non-existing symbol. Also, it should be possible to load the corresponding symbol definition lazily. The bottom part of Fig. 8 shows an example of a symbol reference for a Java class.

JClassSymbol represents the definition and JClassSymbolReference its reference. By referring to JClassSymbol, JClassSymbolReference can delegate every request to the symbol definition. For that, the symbol reference contains all information needed to resolve the corresponding definition, usually, the name and kind of the symbol definition. A field in Java always has a type, hence, JFieldSymbol has a JClassSymbolReference. It is important to separate symbol definitions and references into different classes since the references can contain additional information that is specific to the reference. For example, the field List<String> has a reference to List with the type argument String. Since other type arguments are possible, such as List<Integer>, List<Boolean>, etc., it is important to store the information about the type arguments in the symbol reference.

4.5 Symbol Resolution Mechanism

Finding the definition of a symbol and the information associated with it is called symbol resolution (cf. name resolution [13]). To resolve a symbol usually its name and kind are needed [17]. The symbol kind is needed in the resolving process since many languages allow to use the same name for different elements, e.g., in many
object-oriented languages fields and methods may yield the same name.

Resolution algorithms often are very complex and rely on several (language-specific) aspects, such as shadowing, visibility and accessibility rules. Furthermore, these rules can differ depending on the scope level, e.g., Java has different shadowing rules for method and while-blocks. However, many languages share some common resolving mechanisms: a) The starting point is the innermost scope [1]. The resolution continues with the enclosing scope until the symbol definition is found. b) Name occurrences in \textit{shadowing scopes} shadow the same symbol kinds in enclosing scopes. c) Same names may be used for different symbol kinds, e.g., field and method. d) Some import mechanisms are used —usually in the artifact scope—to resolve elements from outside the model.

As described previously, we introduced those concepts on the $M_3$ level, such as \textit{ShadowingScope}, \textit{VisibilityScope}, \textit{ArtifactScope}, \textit{Symbol}, \textit{SymbolKind}, and the corresponding enclosing-subs relations. This enables us to define a resolution algorithm once on the $M_3$ level and apply it for every language on the $M_3$ level by using language-specific elements, e.g., \textit{JClassScope}, \textit{MethodNameScope}, \textit{ArtifactScope}, \textit{JClassSymbol}, \textit{JMethodNameSymbol}, \textit{JClassSymbolKind}, etc. In order to match language specific requirements that are not covered by the default behavior, MontiCore provides specific extension points (see Sect. 5).

5. Integrating and Generating Symbol Tables

The language engineer usually needs both the $M_2$ model of the specific grammar and its corresponding symbol table $M_3$ model. For this reason, we connect the respective $M_3$ models, and hence, enable the composition of the $M_2$ models [7]. Moreover, the language engineer obtains all necessary information about a model element. For example, $\textit{JClass}$ contains syntactical information about a class production. Since it is related to $\textit{JClassSymbol}$, all further information such as the super class and the members are available in a moderate way. Furthermore, the language engineer need not deal with (re-)loading of referenced models (e.g., the super class). The whole mechanism is encapsulated in the underlying symbol table infrastructure. Thus, on $M_1$ level every processed model provides information about its AST nodes and symbol table elements.

In the following we describe (1) how the two $M_3$ models and the corresponding $M_2$ models are composed, (2) how the composition is conducted syntactically, and (3) how parts of the symbol table infrastructure can be generated by using grammar annotations.

Composing the Grammar and Symbol Table Meta Models

As described in Sect. 4.1, a symbol represents an essential model element. Since those model elements syntactically are defined by grammar productions, we connect the $M_3$ classes \textit{Production} of the grammar and the \textit{Symbol} of the symbol table (see Fig. 9). Note that a symbol always "knows" its kind and its spanned scope (if it is a scope spanning symbol). Hence, it is sufficient to relate a production to the symbol only and obtain the other dependencies transitively. This simplifies the integrated meta models and reduces potential inconsistencies, e.g., if a scope spanning symbol is related to a production, but its spanned scope is not. For the $M_2$ models of the Java example this implies that the productions \textit{JClass}, \textit{MethodName}, and \textit{JField} are related to \textit{JClassSymbol}, \textit{MethodNameSymbol}, and \textit{JFieldSymbol} respectively. We use a * cardinality for the relation between \textit{Production} and \textit{Symbol} for two reasons. First, not every production has an associated symbol and vice versa. The \textit{Name} production, for example, is not represented by a symbol. Secondly, a production can define several model elements each represented by a dedicated symbol. For example, a production \textit{JClassMember} could define both, a field and a method that have the corresponding symbols \textit{JFieldSymbol} and \textit{JMethodNameSymbol}, respectively. Analogously, a symbol \textit{JClassMemberSymbol} can represent the production \textit{JField} as well as \textit{MethodName}.

A \textit{Production} can also be related to a \textit{Scope} without a corresponding symbol. The \textit{JWhile} production, for example, is associated with \textit{JWhileScope}. Again, a * cardinality between \textit{Production} and \textit{Scope} is needed, since, for example, a production for an if-else block might be mapped to two scopes.

\textit{Nonterminals} are associated with \textit{SymbolReferences}. For example, the nonterminal \textit{Type} is contained in the \textit{JField} production (see Fig. 3) which itself is associated with the symbol \textit{JFieldSymbol}. Consequently, relating \textit{Type} and \textit{JClassSymbolReference} entails that a \textit{JFieldSymbol} refers to a \textit{JClassSymbol} using the class \textit{JClassSymbolReference}.

Generating the Symbol Table Meta Model

The composition of the two $M_3$ models is affected by the grammar design as well as the symbol table design, which are both determined by the language engineer. The grammar can be designed with just one production describing the whole model right up to many small productions for every model aspect. Similar, the symbol table can consist of only one symbol or several symbols for each model element. As a consequence, the composition of the grammar and the symbol table at the $M_3$ level must be conducted manually.

In our experience, there often exists a dedicated production for each essential model element. Hence, a \textit{Production} is related to at most one \textit{Symbol} and vice versa. The same holds for \textit{Production} and \textit{Scope} as well as \textit{Nonterminal} and \textit{SymbolReference}.

In such cases, we can assist the language engineer not only in composing the two $M_3$ models, but also in developing the symbol table using a generative approach.

As described in Sect. 3, the meta model for the grammar is the AST class diagram. MontiCore provides an extended grammar that enables to describe and systematical derive both the concrete syntax and the abstract syntax of a language. A comprehensive description of the MontiCore grammar is given in [11].

We now go one step further and automatically derive besides the concrete and the abstract syntax, the symbol table from the grammar. As stated before, this is only possible to a certain extent, since the language engineer determines the abstraction level of the symbol table. Furthermore, the symbol table might contain information that is not directly contained in the grammar (see Sect. 4.1), and hence, it cannot be derived automatically from it. However, in many cases at least the infrastructure of the symbol table can be derived automatically. In the following, we describe this approach and show how the language engineer can make use of it.

We use the existing annotation mechanism of the MontiCore grammar in order to (1) automatically derive the language specific (i.e., $M_2$) symbol table infrastructure from it and (2) simultaneously integrate it with the language-specific grammar $M_2$ model. Annotation a production with @ specifies that this production is related to a symbol. The mapping is conducted by a naming convention: a production $Prod$ is mapped to a symbol $ProdSymbol$. In List 1 both productions $\textit{JClass}$ and $\textit{JField}$ are annotated with
of, hence, they are related to the symbols JClassSymbol and JFieldSymbol, respectively.

By solely marking the productions with an annotation, the two points mentioned above are fulfilled for $M_3$ symbol classes. First, the symbols JClassSymbol and JFieldSymbol can be generated along with their kinds JClassKind and JFieldKind, respectively. Second, the grammar elements and the symbol table elements are related to each other, e.g., JClass and JClassSymbol.

Furthermore, the following aspects can be derived from the grammar without being explicitly defined. JClass contains the nonterminal JField which is defined by the same named production that is used in the JField production. Similarly put, the JClassSymbol contains JFieldSymbol. Thus, JClassSymbol is a scope spanning symbol. So, its scope JClassScope is generated, too.

A Java field has a type that refers to a class defined elsewhere. Syntactically, the type reference is just a name as stated by the nonterminal type:Name is used in the JField production (l. 2, lst. 1). The nonterminal's annotation JClass specifies that a JClass production is referenced. Again, we choose a naming convention: if a nonterminal is annotated with Attr and Attr is the name of a production, then the nonterminal will be related to a symbol reference AttrSymbolReference.

From this information we can infer that JFieldSymbol refers to a JClassSymbol as its type. The $M_3$ class Type (see Fig. 3) is related to JClassSymbolReference, hence, a JFieldSymbol has a JClassSymbolReference.

Fig. 9 shows a (highly simplified) production of a Java while-block. The JWhile production contains, among others, the nonterminal JField, meaning that Java fields (or rather variables) can be defined in it. Since the JField production is related to a symbol (see lst. 1, 1, 2), we can derive — following the convention-over-configuration approach — that JWhile spans a scope. Consequently, a corresponding JWhileScope class is generated that (only) contains JFieldSymbols. By default, a scope not spanned by a symbol is considered to be a VisibilityScope unless the corresponding production contains the name (resp. names:Name) nonterminal. For example, if no symbol was created for JClass, it would be considered as a shadowing scope, since it has a name and contains JFields. Finally, JArtifactScope is generated, too, since — as stated in Sect. 4 — models of textual languages typically are stored in an artifact (resp. file). The symbols and scopes that may be contained in JArtifactScope are determined from the grammar as follows. Beginning from the start production, find the first productions that are related to a corresponding symbol or scope. Those symbols and scopes may be defined in the artifact scope. In the simple example of Lst. 1, JClass is the start production and is also related to a symbol. Consequently, JArtifactScope may only contain JClassSymbols and their spanned scope. As an artifact scope corresponds to the file instead of a model element, there is no class resp. production related to a generated artifact scope.

Using the above mentioned conventions for annotations in the grammar and the derivation rules, a lot of the symbol table's language-specific infrastructure (i.e., $M_2$ model) is generated with the default behavior described in Sect. 4.1. The language engine has the following options: 1) Customize and extend parts of the generated infrastructure using the different integration mechanisms as described in [4] without changing the code directly (see next point). 2) Use it or parts of it as one-shot generation, i.e., change the code directly. Consequently, changes in the grammar do not affect the code anymore. 3) Use it unchanged, if it fits all the requirements. 4) Ignore it and develop the symbol table manually instead.

6. Related Work

Classical symbol tables typically are simple hash tables where a key, the identifier, is mapped to the associated information. Using those symbol tables, some possible implementations of nested blocks are the use of (unique) qualified identifiers or nesting symbol tables per block [1]. Furthermore, if two different kinds of model elements may have the same name, e.g., a field and a method in Java, often one symbol table is created per kind. In our approach, the symbol table is rather conceptually a table. The underlying infrastructure is a scope-tree containing a collection of symbols (similar to [15]). Each symbol encapsulates the identifier and the associated information. Also, we use explicit symbol kinds to distinguish different kinds of model elements. This way, same-named symbols with different kinds may be defined in the same scope.

The purpose of our symbol table approach goes further than in classic compiler construction. It is rather a combination of a simple hash table and a meta model for the semantic model as described in [3]: "a semantic model is a representation [... of the same subject that the DSL describes]." Furthermore, it is "based on what will be done with the information from a DSL script." Since the purpose of a DSL is determined by the language engine, the semantic (meta) model cannot be created (resp. generated) automatically. For this

```java
JClass1 = "class" Name "(" [JField | JMethod]=" ");
JField1 = type:Named JClass Name ";";
```

Listing 1. Simplified Java Grammar Excerpt with Annotations

```java
JWhile = "while" "(" ... ")" "(" [JField | JMethod]=" ");
```

Listing 2. Simplified Production for a while-Block
reason, we support the language engineer by generating parts of the infrastructure and provide mechanisms for customization.

The meta-DSL name binding language (NaBL) [10] allows to specify name bindings (resp. name resolution) and scoping rules declaratively. It provides concepts, such as scoping, definition of imports and names, and referencing rules (cf. [13]). The language workbench Spoofax [6] combines NaBL with the syntax definition formalism (SDF) [16]. Since NaBL models are separated from the syntax definition, they can be easily exchanged and adjusted for different compositions. Unlike our approach, Spoofax's symbol table is a global index with qualified identifiers. Also, we do not provide an own DSL, but follow the convention-over-configuration approach by deriving as much information from the grammar as possible and useful.

Model transformation approaches, e.g., as in [8, 9] conduct transformations between a source and a target $M_2$ models in order to make the corresponding $M_3$ models interchangeable. The transformation is processed in three steps. Firstly, the concepts of the $M_3$ models are mapped to each other. This mapping then enables the transformation of the source $M_3$ model to the target $M_2$ model. Finally, with the $M_3$ mapping and the $M_2$ level transformations the $M_2$ level transformations are derived automatically. Same as in our approach, the mapping is conducted on the $M_3$ level. However, since we need to use information of both $M_3$ models on the $M_2$ and $M_1$ levels, we furthermore compose $M_3$ models. Similarly, our grammar $M_3$ model is the source model from which the target symbol table $M_2$ model is generated. In contrast to the model transformation approach, we also integrate these two $M_2$ models.

7. Conclusion

Textual software languages often share common concepts such as defining and referencing model elements of the same model or other models, shadowing already defined names, and limiting variable visibility using some kind of scoping. Those concepts can be realized by symbol tables which enable easy and efficient access to useful information associated with a model and its elements.

In this paper, we presented a $M_3$ model for symbol tables containing first-level classes for the above mentioned concepts. Based on this meta model the language engineer can develop a language-specific symbol table model at the $M_2$ level. As the symbol table information and the grammar information are related and often required in conjunction, we compose the $M_3$ models, and hence, enable a corresponding composition at the $M_2$ and $M_1$ levels.

Typically, the symbol table is handcrafted, since it highly depends on the purpose of the DSL. Also, composing it with the grammar is a manual task conducted by the language engineer. However, in cases where the grammar matches some criteria—such as containing a dedicated production for each essential model element—it can serve as source model for (1) generating a default symbol table $M_2$ model or parts of it and (2) for directly composing the grammar and symbol table $M_2$ models. For this, we use an existing annotation mechanism for grammar elements and follow the convention-over-configuration approach when generating the symbol table. Different extension mechanisms enable extending and customizing the generated symbol tables.

As a next step, we plan to extend the symbol table $M_3$ model in order to match the requirements of a broader range of software languages. Furthermore, we plan to run experiments to determine whether the suggested defaults and their configuration mechanisms are well understandable and helpful or can be optimized. Currently, the grammar design widely influences the default generation of symbol tables. Therefore, we will examine whether more complex deriving patterns, e.g., the (transitive) dependency between productions, can improve the symbol table generation.

References


Automating Engineering with a Domain-Specific Language and a Code Generator *

Al Niessner  Oh-Ig Kwoun  Belinda Randolph  Honghanh Nguyen
Jet Propulsion Laboratory, California Institute of Technology.
4800 Oak Grove
Pasadena, CA 91109
Al.Niessner@jpl.nasa.gov
Oh-Ig.Kwoun@jpl.nasa.gov
Belinda.Randolph@jpl.nasa.gov
honghanh@andrew.cmu.edu

Abstract

The following is an industry experience on using well established concepts, domain-specific language and code generation, to automate the engineering process of sub-system interactions. A domain-specific language was used to improve the communication efficiency among several teams of engineers. A code generator transforms the models formed for communicating into executables that process data. While, the savings in terms of man-effort and schedule-time were high, the single point of failure is end-user adoption. Adoption failure was attributed to two factors: One, the distance between the end-user and the realized benefit. Two, the amount the end-user had to change (become proficient in skills outside of their discipline). In our experience, the lack of an appropriate editor was the dominate cause to both factors in adoption failure.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Specialized application languages

General Terms domain-specific modeling language, code generator

Keywords code, domain, generator, language, model, specific

1. Introduction

For more than four decades, JPL has been transforming its raw instrument data into science data through some amount of processing. The implementation of the data processing has changed over the decades, but the procedure for data processing has remained relatively constant and is shown in Figure 1 on the following page. Once the raw instrument data has been received at the processing site, it enters what is called the Science Data Systems (SDS) where it passes through various pieces of software, typically called a Product Generation Engine (PGE), to generate the various levels of Science Data Products1 (SDP).2 The new acronyms are samples from the language that evolved over the decades to improve the exchange of information within the SDS domain.

While the language was evolving so was the culture. Without diving into unnecessary details, the culture subdivided SDS into four primary groups of people, system engineers (SE), data engineers (DE), instrument-domain scientists (ADT), and software engineers (DST). The four groups, also known as the end-users, communicate desires, needs, and requirements through a library of word documents, presentations, and pseudo code that are peppered with inconsistent uses of the SDS-specific language that ushers in misinterpretation.

A Domain-Specific Model (DSM) (see Figure 2 on the next page) was organically developed to improve the communication efficiency and reduce effort and time of software development[1, 2]. Each of the SDP’s has its own XML document describing the translation of data. Figure 1 shows each PGE generating a corresponding SDP. An XML document (DOC of Figure 2) defines the corresponding data flow within the PGE and data contained in the SDP. A Python code generator, shown in Figure 2, uses the DOC to produce C++ code that is compiled, along with a framework, to a PGE to do the actual work of transforming the data as shown in Figure 1.

* Funding for this work provided by SMAP.

1 For Earth Science Data (ESD), the data processing levels are defined here http://science.nasa.gov/earth-science/earth-science-data/data-processing-levels-for-oesdis-data-products

2 Many of the names and acronyms have changed over the decades but their roles and responsibilities have remained largely the same.
Simplified view of data flow from the spacecraft to the end products (SDP 1..N). The spacecraft down-links instrument data to a receiving station that then forwards it to an SDS facility. Once at the facility, the raw data is processed to level 1, then level 1 to level 2, etc to level N.

Figure 1. SDS

The DSM has multiple instances of the XML documents (DOC 1..N) that are validated with an XML schema. The XML schema is also used to generate a data binding for the code generator. The code generator produces a PGE for each XML instance. Hence DOC 1..N directly corresponds to PGE 1..N and SDP 1..N of Figure 1.

2. Language

2.1 Definition

Numerous publications provide solid advice and engineering trades for designing a DSL metalanguage[3]. However, the SDS-specific language itself was designed over 40+ years of evolution and the organic development nature of this DSM simply extracted the key parts of the language into XML tags and constraints. While misinterpretation was still a problem, with the aid of the code generator to C++ and g++, all of the misinterpretations and multiple uses of terms were quickly resolved.

2.2 Implementation

The important lesson learned about using XML for the DSL is that editing XML is the real problem. Editing was a significant enough impediment to adoption that it has an entire Section 2.3 devoted to it.

A design choice was to make the key terms XML tags and constraints and then use existing XML tools to do much of the heavy lifting for the syntax and semantic checks. Because XML schema was used as the meta-metalanguage, XML tools that supported validation, like xmllint, were used for syntax checking. Much of the semantics could have been encoded in the XML schema as well as constraints, but the first edition of the language required extensive help from Python for the semantic checks mostly because of the organic nature of the development. While the semantic checking with Python was less than optimal, using PyXB to validate while reading an XML document made the left over semantic checking effort low but not insignificant.

There were three lessons learned with respect to the syntax and semantic checking: One, syntax and semantic checks saved significant time during code generation. The first code generator performed poorly, to the point of being unusable, because of simple semantic errors in the XML documents. Python was used to apply semantic checks improving the code generator’s performance and broadening its adoption.

Two, make adding semantic checks easy because it is hard to know checks the full set of checks a priori. There were 7 semantic checks at the start, and the code generator became robust at 17 semantic checks. Three, put as many semantic checks into the XML schema as possible, but use other languages to help if necessary and move the semantic checks into the schema when mature and appropriate. The second edition of XML schema currently being developed incorporates as many of the semantic constraints into the schema as possible. Resulting XML documents demonstrate improved usability and readability from the schema constraint checks, but having semantic checks in any language or tool is more important than delaying their existence and use while they wait to be put in the XML schema.

2.3 Editing

Editing the XML documents (a complete example is contained in Appendix A) is the complete downfall to the DSL using XML because the end user wants to edit the content of the XML and not the XML itself. The XML technology was a good choice from the perspectives of the off-the-shelf tools, the size of user community, the size, variety, and depth of the support community, and the shallow, long learning curve. However, the tag nature of XML raises the noise floor for the end-user that only cares about the content and that is formatted to express internal relationships. In this specific instance, the average XML document is 4000 tags and the end user wants to view subsets of information within these tags in various formats like tables, ordered graphs, etc.

The first attempt to improve editing was to add XSL transforms for viewing the XML documents within a browser in the format the end-user wanted but using a text editor to change the XML document as shown in Figure 3 on page 5.

3While maybe not the best engineering choice, using XML schema with XML is not necessary.

4In other words, one does not have to digest the entire technology all at once in order to use it, but, rather, can learn small portions of it over long periods of time while using it.
Quick cycles between random edits and what they could view in a browser was sufficient at the beginning. As more people joined the team and the project progressed requiring more information be contained within the XML documents, the XSL transforms fell apart and the end-users became more vocal about the difficulty of using a text editor.

The second attempt was to customize an XML editor to combine the views developed with XSL and direct editing. Light effort at customization with just a few XML editors quickly led to the conclusion that we wanted to edit the content not the XML. The customization of the XML editors was more about displaying various hierarchies of the tags and not building an editable ordered graph using the tags and its attributes to define the nodes and edges and their content to become the labels\textsuperscript{a}. The end-users found no value in this approach.

The third attempt was to hire visualization specialists to fill in the knowledge gap of the DSL development team to help produce an editor with end-user desired views. The visualization team wanted to use a browser-cloud approach because it was how they did all of their work. With hindsight, it would have been better to have them play a bit with customizing the XML editors, but the budget was tight and an editor was desperately needed. The budgetary pressure aided in the requirement degeneration\textsuperscript{b} to an XML editor.

The fourth attempt, and the one currently slated for deployment while this paper is being published, was also done by visualization experts. However, the focus was kept on content editing\textsuperscript{c} (see Figures 4 through 6 on page 5) rather than allowing the requirements devolve, despite budgetary and schedule pressures, to another XML editor. The results of this editor are unknown, but preliminary demonstrations to the end-users were very positive unlike the previous attempts.

The ideal editor would have a DSM, like graphuml, that allows a team to generate a content editor within the metalanguage. The fourth attempt at an editor is specific to this DSL, but it is easy to see how to generalize it and use the <xs:annotation> in the DSL's XML schema to pass information to the editor for layout and relationships of content. A DSM tool aimed at generating content editors would be the biggest boon to the quick development of a DSM\textsuperscript{d}.

\textsuperscript{a}If they are capable of such customization, then learning about it was beyond the given budgetary constraints.

\textsuperscript{b}Degeneration because every software engineer that looked at the XML document saw XML. In turn, the requirement to edit the content degenerated to mean the XML tag in an XML hierarchy and not the content within the tags in a more context-intuitive composition and layout.

\textsuperscript{c}LyX is the best example of a content editor.

\textsuperscript{d}Attempts to use UML[4, 5] or Eclipse[6] did not succeed for the same reasons as using XML.

3. Code Generator

3.1 Definition

Since the DST had already defined most of the external tools and libraries they required and wrapped them with a framework, the code generator was very straightforward. Reading through the APIs and perhaps experimenting with some oddities or corner cases was all that was needed. The generated C++, FORTRAN 2003, and XML code connects the ADT developed science algorithm code to the DST supplied C++ framework.

3.2 Implementation

The code generator was implemented in Python using PyXB library for parsing, validating, and data binding. The code generator first validated the XML documents with syntax and semantic checks as described in the DSL in Section 2.2 on the preceding page. Conversion of objects from the PyXB to C++ code resulted in several thousand lines of C++ for each PGE. While it is very difficult to precisely measure the manpower saved, a conservative first-order estimate indicates 5 man-years of savings for a one man-year effort for both the DSL and code generator.

3.3 Testing

Testing of the generated code added some rigor to the code generator itself. The first level of testing were the compilers. The compilers highlighted misinterpretations of the DSL from context sensitive definitions\textsuperscript{e} or word misuse, which resulted in improvements in the XML schema, semantic checks, and code generator. The second level was static analysis tools like Coverity. With a false positive ratio below 5%, the static analysis tools highlighted nearly 100 poor code generator choices that were quickly remedied. The third level of testing was a stub from the code generator that allowed the software team to run the generated code without the addition of the ADT science algorithms exercising all of the generated code. Since the generated code was exercised in isolation, run-time problems found in the PGE after ADT science algorithm integration had to be the purview of ADT and not the code generator.

While it would be nice to say that the DSM experiment presented here was done formally and with a control, it is wholly misleading to do so. However, there was one team that shunned the DSM approach resulting in an experimental control group. In order to call them a control group, they use the same framework and external libraries as the DSM group and the code developed versus generated did the same task. Both the code by the control group and that by the code generator used the same compiler and compiler flags achieving the same level of code quality or defect density. Coverity was used to statically analyze both the control group's code and the generated code, but both groups did

\textsuperscript{e}Some words had different meanings depending on which team was using them.
not use the results the same. In the control group, only one

team member of six used Coverity result to fix problems

that had been identified. In contrast, the code generator was
modified to drive the problems to zero. The control group
was responsible for 6 PGEs while the code generator was 17
PGEs. Lastly, running the external libraries, framework, and
generator stub allowed the generated code to pass another
level of testing that the control group did not have. The
end result was fewer delivered bugs, faster repairs, and less
frantic deliveries as compared to the control group.

4. Conclusion

There are two reasons for the high return on investment:
One, the language was predefined from four decades or more
of repetition reduced the investment cost. Two, the code
quality cycle saves effort that scales with N because any
problem detected in one PGE would be fixed in the code
generated resulting in all PGEs being fixed.

The primary obstacle to adoption was the ability of the
end-user to easily design, edit, and view their models. Gen-
eral modeling languages, editors, and tools, such as UML
and SysML, are outside the domain of expertise for most
of the SDS end-user obscuring their benefits and relevance.
Presenting the model in an end-user specified layout in-
creased adoption. Find or build a content editor as soon as
possible even though it significantly increases the investment
because, unless the end-user realizes immediate benefit from
the DSL, they will resist any changes from the status quo.

Acknowledgments

The research was carried out at the Jet Propulsion Labo-
rary, California Institute of Technology, under a contract
with the National Aeronautics and Space Administration.

We would like to thank all those that helped us improve
the tools for wider adoption with special thanks to those that
suffered the tools in their infancy.

References

Domain-Specific Language Architecture for Automation Sys-
tems: An Industrial Case Study European Conference on Mod-
eling Foundations and Applications Graphical Modeling Lan-

[2] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-
Specific Modeling. IEEE Software, vol26, no. 4, pp 22-29,

Domain-Specific Modeling Languages: Collected Experiences.

[4] Juha-Pekka Tolvanen. Industrial Experiences on Using DSLs in
Embedded Software Development. Proceedings of Embedded
Software Engineering Kongress (Tagungsband), Dec 2011.

[5] Mark Dalgamo and Matthew Fowler.UML vs. Domain-Specific


A. Example XML Document

See Algorithm on page 6.
**Product Name: LevelOne**

**Table of Contents:**
1. Comments
2. Interface
3. Product

**Comments**

The code interface is quite simple since it is a demo of what the code generator can do and how the ppe.py works.

First, we load the data with `Engine.load_data()`. The load reads from several sources and collates them for later use. It stores them as state in the derivative of `Engine` for the other methods to use. While `PIX` allows parameters to be passed, sometimes hiding the information in the state of the class seems more appropriate with C++.

Second, we do something just to illustrate that all methods do not need to be bound to the `Engine` class.

Third, we are back to operating on the data by computing the 2-D magnitude of the 1-D samples by multiplying them. It should make a fun surface, but is not overly complex since doing math is not the item of interest.

Lastly, we take the results back out of the `Engine` class and put them in a container for the rest of the world.

**Interface**

```
begin
  demo::1::Engine.load_data
  demo::1::do_something
  demo::1::Engine.compute_magnitude
  demo::1::Engine.emit_results
end
```

**Interface Tree**

**Product**

**Groups:**
- Results

**/results**

<table>
<thead>
<tr>
<th>Element</th>
<th>Shape</th>
<th>Concept</th>
<th>Bytes</th>
<th>Signed</th>
<th>Unit</th>
<th>Min</th>
<th>Max</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_axis</td>
<td>SpatialArray</td>
<td>real</td>
<td>8</td>
<td>NA</td>
<td>mm</td>
<td>0.0</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>y_axis</td>
<td>SpatialArray</td>
<td>real</td>
<td>8</td>
<td>NA</td>
<td>mm</td>
<td>0.0</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>magnitude</td>
<td>x_axis</td>
<td>XAxisArray</td>
<td>real</td>
<td>8</td>
<td>NA</td>
<td>ev</td>
<td>-1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Back to Table list**

**Shapes:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Dimensions</th>
<th>Index Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpatialArray</td>
<td>Spatial=0,</td>
<td>Irrelevant</td>
</tr>
<tr>
<td>XAxis</td>
<td>XAxis=0, YAxis=0,</td>
<td>Slowest...Fastest</td>
</tr>
</tbody>
</table>

![Figure 4. Content Editor Front-Page](image1)

The directed graph is based on the `<method>` content (see Appendix A) of the XML documents.

![Figure 5. Content Editor Interface-Page](image2)

Contains a directed graph based on `<method>` content and then encapsulates larger `<product>` concepts (see Appendix A) in the DSL into active boxes that expand to different views when selected.

![Figure 6. Content Editor Advanced-Page](image3)

Represents the `<nodes>` content (see Appendix A) as table.
Algorithm 1 Complete Example

```xml
<?xml version="1.0" ?>
<algorithm name="LevelOne"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://smap-sds-web.jpl.nasa.gov/schema/pix.xsd">
  <interface language="C++">
    <annotation app="code">
      The code interface is quite simple since it is a demo of what the code generator can do and how the pge.py works.
      First, we load the data with Engine.load_data(). The load reads from several sources and collates them for later use. It stores them as state in the derivative of Engine for the other methods to use. While PIX allows parameters to be passed, sometimes hiding the information in the state of the class seems more appropriate with C++.
      Second, we do something just to illustrate that all methods do not need to be bound to the Engine class.
      Third, we are back to operating on the data by computing the 2-D magnitude of the 1-D samples by multiplying them. It should make a fun surface, but is not overly complex since doing math is not the item of interest.
      Lastly, we take the results back out of the Engine class and put them in a container for the rest of the world.
    </annotation>
    <flow>
      <sequence>
        <call>demo::li::Engine.load_data()</call>
        <call>demo::li::do_something()</call>
        <call>demo::li::Engine.compute_magnitude()</call>
        <call>demo::li::Engine.emit_results()</call>
      </sequence>
    </flow>
    <method name="demo::li::Engine.compute_magnitude"/>
    <method name="demo::li::do_something"/>
    <method name="demo::li::Engine.emit_results">
      <output name="mag" node="/results/magnitude"/>
      <output name="x" node="/results/x_axis"/>
      <output name="y" node="/results/y_axis"/>
    </method>
    <method name="demo::li::Engine.load_data">
      <annotation app="code">
        Load the data from the level 0 product and the ancillary file.
        The ancillary file is a simple binary that is simply two float values that both conversions from V to eV and scale to the sensor.
        The data in level 0 is defined by the container.
      </annotation>
      <input name="ifn" node="/ancillary_file_name" prod="LevelOneRuntimeArguments"/>
      <input name="x" node="/time_series/x" prod="LevelZero"/>
      <input name="y" node="/time_series/y" prod="LevelZero"/>
    </method>
  </interface>
</algorithm>
```

```python
<xsd:complexType name="SpatialArray">
  <xsd:sequence>
    <xsd:element name="max" type="xsd:double"/>
    <xsd:element name="min" type="xsd:double"/>
    <xsd:element name="units" type="xsd:string"/>
    <xsd:element name="width" type="xsd:integer"/>
  </xsd:sequence>
  <xsd:attribute name="shape" type="xsd:string"/>
  <xsd:attribute name="order" type="xsd:string"/>
  <xsd:attribute name="extent" type="xsd:string"/>
</xsd:complexType>
```
Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table

Pedram Mir Seyed Nazari  Alexander Roth  Bernhard Rumpe
Software Engineering  RWTH Aachen University  {nazari,roth,rumpe}@se-rwth.de

Abstract
An essential part of model-driven development to systematically generate concrete source code from abstract input models are code generators. Regardless of their importance, abstract input models are not always suited to describe the output in a concise and precise way. Hence, customizations and adaptations of the code generator and the generated products are needed. Existing approaches mainly regard the code generation process as their primary concern and require the setup of an additional infrastructure in order to manage the customizations and adaptations. Thus, the goal of this paper is to propose an extension for template-based code generators to enable customizations and adaptations within a code generator that also respects referential integrity and reuses existing data structures for efficient management. First, we present a classification of common code generator customization and adaptation approaches (guided and unguided approaches) to identify the main concepts and elements of the approaches. Then, using the derived information relevant to manage guided and unguided approaches, we reuse the existing data structure (symbol table) to manage the customizations and adaptations. We achieve this by associating all relevant information directly with a template. This approach enables dynamic management of customizations and adaptations at runtime of the code generator and allows for statically checking before code generation. Our main contribution is an approach to combine guided and unguided customization approaches with a symbol table for efficient management.

Keywords  Code Generator Adaptation, Symbol Table, Template-based Code Generation

1. Introduction
In order to systematically generate source code in a model-driven development (MDD) environment, code generators are essential. In general, a generator is a software system that generates a concrete implementation from an abstract input model and consists of a front-end and a back-end [12]. While the front-end is concerned with language processing to create an abstract syntax tree and a symbol table, the back-end is responsible to systematically generate code from these abstract representations of the input model.

Even though a code generator is of high importance and full code generation is a goal of MDD, which should be achieved by providing multiple abstract models for each part of the generated system, abstract models are not always suited to describe the output in a concise and precise way. For instance, algorithms usually cannot be described in a more abstract and easier form than the implementation of those algorithms itself. Hence, it is an intrinsic property of a good code generator to be able adapt either the code generator itself or the generated product. An overview of existing approaches for managing variability and customizations of code generators [3, 12, 13] shows that their primary focus is on the whole code generation process. Customizations and adaptations are either handled before or after generation-time (i.e., run-time of the code generator), e.g. aspect-oriented code generation [8]. Within a code generator such concerns are barely addressed by e.g. Preprocessors [3] and aspect-oriented programming [14]. Hence, referential integrity, i.e., checking if the referenced customization exists, is not ensured.

Thus, in this paper we propose an approach to manage customizations and adaptations of code generators explicitly, statically, and dynamically at generation-time by employing a symbol table. Note that we explicitly neglect customizations of the input model and only regard customizations and adaptations within the code generator, because further research is required to address related challenges, e.g. How can modular languages be managed in code generators? How can language customizations be handled by the code generator? We first restrict existing guided and unguided approaches to their basic elements in order provide required extensions to template languages. Guided customization approaches are approaches that explicitly define hook points that can be extended, but they do not allow for further customizations except for the hook points. In contrast, unguided approaches do not restrict customizations and adaptations. However, with this freedom of customization a high probability of failure is introduced. Having such an understanding of customizations, we use the symbol table as a data structure to explicitly and efficiently manage related information. For guided approaches, each hook point is stored in the symbol table. For unguided approaches, template replacements are stored. During generation-time the code generator can dynamically access this information. Whenever a hook point or template changes the changes directly affect the customizations. Moreover, after language processing, referential integrity can be ensured. In other words, the defined and configured customizations can be statically checked.

Hence, we first provide an understanding of guided and unguided customization approaches (Sect. 2) in order to extract their basic elements. Then, we identify static and dynamic checking as a major requirement in Sect. 3. Afterwards, we elaborate on how to use the symbol table for efficient management of guided and unguided customizations in Sect. 4. Finally, we concluded our paper in Sect. 5.

2. Guided and Unguided Customization of Code Generators
In a model-driven environment, where code generators generate source code artifacts, adapting a generated output means that the changes only affect one single artifact. Instead, adaptations of the code generator may affect all generated artifacts. For template-based code generators, adaptations mean changes to the templates.
Adding functionality by template adaptations can, however, be challenging because the developer requires knowledge of the template, the languages (target and the generator language), and the architectures (generator and generated artifact architecture).

We classify customization and adaptation approaches of a code generator into two categories based on the overview of common approaches [5]. This classification aims at giving an understanding of the main characteristics of common customization and adaptation approaches. Completeness of categorizing all available approaches and their characteristics is not targeted. Guided approaches focus on explicit declaration of spots that can be extended. At the same time guided approaches explicitly forbid all other ways of customizations. For instance, variation points [2, 11] are a guided way that are defined at the design time of a code generator and later used for customizations. In the remainder of this paper, we refer to template languages that provide these extensions as extended template languages.

A major disadvantage is that during the evolution of a code generator variation points may change. As a consequence, the overall code generator needs to be adapted. This challenge can be seen as a variant of the “fragile base class problem” [9]. In contrast, unguided approaches are less restrictive but are more error prone, as they allow to change every piece of the code generator and the generated code without explicitly denoting the elements that can be customized.

Subsequently, we elaborate on each type of customizations and present an abstract description of their realization.

2.1 Guided Customization

To avoid the complex and tedious task of adapting templates directly, hook points can be defined in templates. A hook point provides an approach to extend a template at a predefined spot. It is a place in a template that is planned for adaptation. Typically, such hook points are set during design time of a code generator and may be changed during the evolution of the code generator. Existing template languages provide such concepts, e.g. Xpand [4].

The basic elements of guided approaches are a way to define a hook point, and a way to bind values to a hook point. For template-based code generators each template may define multiple hook points, which are identified by a unique qualified name that consists of the path to the template, i.e., package name, the template name, and the hook point name. During configuration of the code generator hook points are bound to either one or multiple values. Each value can either be a simple string or another template. The values - in the case of a template it is the result of evaluating the template - are inserted at the spot the hook point is defined.

The major benefits of this approach are the guided way of customization, which shrinks the probability of errors, and the need to only regard a hook point not the overall code generator architecture. However, the lack of customizability is its main disadvantage, because each hook point needs to be carefully preplanned. Moreover, the result of a hook point may be syntactically invalid, since each template typically produces a string that conforms to a non-terminal of the target language.

2.2 Unguided Customization

Unguided approaches provide more freedom in adapting a code generator. The simplest approach is to directly adapt a code generator by e.g. adapting a template. Since templates build a complex structure, modifying them is hard. It requires a solid understanding of the template architecture and the template itself. This may also be challenging since the template sources may not be at hand.

Based on the knowledge of only the architecture, the basic concept of the approaches are the overriding of elements of the architecture. In the case of a template-based code generator, these elements are templates. The overridden template will not be used in the entire generation process anymore. A similar approach is provided by object-oriented programming languages in the form of overriding classes in generated code. An extension to template overriding is to add a new template before or after an existing template. This is similar to aspect-oriented programming [6]. Each time the existing template is executed, the added templates are either executed before or after.

Another extension that can be found for template-based code generators are global variables. Once a global variable is defined, it can be accessed from any template of the code generator. Such a value is a string and is used to exchange information between templates, e.g. names.

Certainly guided and unguided approaches do have commonalities. For instance, extending a template at the beginning of its execution can be seen as a generic hook point at each template start. The same holds true for extensions at the end of a template execution. Overriding a template is similar to point at the call of a template. It can also mimic extension by defining a new template, add an explicit hook point at the beginning and the end, and call the original template.

The flexibility introduced by this approach is dangerous as no type check ensures correctness of the resulting code. Only compilers will detect syntax errors. Clearly, code generators cannot prevent or detect such errors, but such customizations increase the potential of syntax errors drastically, as every adaptation is possible. Additionally, when overriding templates, a template may be in use elsewhere. This may cause side effects.

3. Requirements for a Data Structure to Manage Customizations and Challenges

Even though guided and unguided customization approaches for template-based code generators introduce adaptability, they also come with several challenges, which we subsequently explain in greater detail with the help of an example.

Assuming the template architecture as shown in Fig. 1 to be given. When a code generator realizing this template architecture is started, the root template T1 is called. This template then calls two other templates (T2 and T3). Finally, T2 calls T4, which then calls T3 again. Furthermore, we assume that T2 adds some content to a hook point defined in T3 but T3 is replaced by T5 at generation-time, i.e., run-time of the code generator. In this situation two challenges are present.

The first challenge when realizing guided and unguided customizations in template-based code generators is static checking. For example, when a hook point is defined within a template and a value is bound, the hook point’s name may change. In order to prevent resulting compilation errors of the generated code and make this error visible as soon as possible static checking of hook points and template replacements needs to be addressed. With existing

![Figure 1. Example of a template architecture with template replacements and hook points.](image-url)
mechanisms of current template languages, this is not feasible. To realize static checking an additional infrastructure is required.

The second challenge is to manage the template replacements and hook points dynamically at generation-time. In particular, while this information is defined statically, the values are accessed dynamically and may change during execution of the code generator. As most template languages provide global variables, they can be used to realize such behavior. A major disadvantage of this approach is that global variables are managed locally by one data structure and, thus, the information is not stored where it belongs to, i.e., at the template. Consequently, a mapping of template to dynamic information (template replacement and hook point values) is required.

In the remainder of this section, we identify further requirements for a data structure to efficiently manage customizations.

### 3.1 Data Structure Requirements

In order to efficiently manage and manipulate information related to customizations, an adequate structure is needed. In the following, we specify the requirements for such a structure by the example of two simplified templates.

```c
#define HP("HP1")
#define HP2("HP2")
#define aGV("aGV", "aVal")
... 
```

Listing 1. Template p.T1 contains two hook point definitions and a global variable.

```c
#define HP("HP1")
#define pT1HPSString(\"p.T1. HP1\", \"value of hp\")
#define replace("q.T3", "p.T1")
... 
```

Listing 2. Template k.T2 defines a hook point, binds a string value, and replaces a template.

### Requirements

Lst. 1 and Lst. 2 demonstrate some syntactical usages of the concepts introduced in Sect. 2. The first two statements of template T1 in Lst. 1 define the hook points HP1 and HP2. These two hook points are related to T1, but are bound by other templates. Thus, the first requirement is to manage template specific information that is defined within that template and make it accessible (from outside) (RQ1).

Furthermore, information specific to a certain template can also be defined within other templates. The replace statement in Lst. 2 (L3), for example, replaces the template q.T3 (not shown here) by p.T1. Although, syntactically contained in k.T2, it is important to associate this information with q.T3 instead of k.T2. The reason is that a template that uses (e.g., includes) q.T3 might not know about k.T2. Hence, it must obtain the information from q.T3. Following from this, the second requirement is to manage template specific information that is defined outside that template and make it accessible (RQ2).

The last statement in Lst. 1 defines the global variable aGV with the string value "aVal". This variable can be accessed and manipulated by any template. Other than the previous cases, it is not associated with a specific template. Hence, the third requirement is to manage global values, i.e., template unspecific information, that can be defined from within any template and make them accessible (RQ3).

The requirements RQ1, RQ2, and RQ3 concern the definition of information, i.e., definition of hook points, replacements, and global variables. However, these information must also be retrieved. The bindHPSString statement in Lst. 2 (L2), for instance, refers to the hook point HP1 that is defined in p.T1. Analogously, the replace statement (L.2) refers to the templates q.T3 and q.T1 in order to define a new replacement. From this, the fourth requirement follows: a given reference, the corresponding definition must be obtained in order to access its associated information (RQ4). The definition can be in the same (intra-model) or in another template (inter-model) as the referencing template (RQ4.1).

### 4. Symbol Table for Templates

A natural choice to tackle the challenges and requirements described in Sect. 3 is a symbol table which is a data structure that maps names to essential model elements [1]. A symbol table entry is called symbol (resp. symbol definition), which contains all essential information about a model (element) and has a specific kind depending on the model (element) it denotes. The symbol table is built up during the analysis phases [1].

In the extended template language as described in Sect. 2, essential model elements are, among others, templates and hook points. Hence, the symbol table can store corresponding template and hook point symbols including their associated information (fulfills RQ1 and RQ2). A symbol table can also manage built-in types that are accessible by all elements [10]. This concept can be used for global variables described in Sect. 2.2 (fulfills RQ3). Furthermore, given a name and a (symbol) kind, the symbol table’s resolving mechanism finds the corresponding symbol definition with all its associated information (fulfills RQ4). The resolving begins within the referencing model and—if not found—continues with other models (fulfills RQ4.1). Furthermore, symbol tables usually provide some more concepts, such as visibility and import mechanisms [7].

![Figure 2. Symbol table structure for templates.](image)

Fig. 2 demonstrates a symbol table infrastructure for the extended template language. TemplateST is the root of the symbol table and, thus, contains directly or indirectly all other elements (resp. symbols). A template is represented by TemplateSymbol. Essential elements of a template are hook points and replacements. Consequently, dedicated symbols for each of them exist, namely HPSymbol and ReplacementSymbol, which are stored in the TemplateSymbol. A HPSymbol can be bound to a Value, which is either a plain string (represented by StringValue) or a template (represented by TemplateValue). Note Value and its sub-classes are no symbols themselves, but data associated with HPSymbols and GVSymbols. A TemplateSymbol optionally has a ReplacementSymbol stating by which other template T is replaced. GVSymbols represent global variables. Since template-unspecific, they are directly stored in the TemplateST instead of in a TemplateSymbol. A global variable has a StringValue.
4.1 Example

Fig. 3 shows an instance of the symbol table for the templates specified in Lst. 1 and Lst. 2. The root element of the symbol table is an instance of :TemplateST. T1:TemplateSymbol represents the p.T1 template (see Lst. 1). The two hook points HP1 and HP2 (l. 1-2) are represented by instances of HPSymbol. Since defined within p.T1, T1:TemplateSymbol contains these two.

If the global variable gV is defined in p.T1, but directly stored in :TemplateST, to be visible for every symbol.

Similarly, for the template k.T2 (see Lst. 2) a corresponding TemplateSymbol exists and is stored in :TemplateST. Also, HP1:HPSymbol is defined in T2:TemplateSymbol (l. 1). In line 2 of Lst. 2, the HP1 hook point of T1 is bound to the string "value of hp". This information concerns HP1, hence, the corresponding symbol in T1:TemplateSymbol stores this information, highlighted by a StringValue instance linked with HP1:HPSymbol.

That way, HP1:HPSymbol manages an information that is associated with it, but defined outside its own template. Similarly, the replacement in line 3 is associated with the template q.T3, and thus, a ReplacementSymbol is stored in the T3:TemplateSymbol, not in T2:TemplateSymbol. Additionally, the replacement symbol has a reference to T1:TemplateSymbol to indicate that q.T3 is replaced by p.T1. The link is resolved by the symbol table’s resolving mechanism.

4.2 Static vs. Dynamic Information in the Symbol Table

The symbol table in Fig. 3 contains two different types of information: static and dynamic information. Static information can be obtained without executing the templates, e.g., during design-time, whereas dynamic information can only be determined at generation-time. Hence, the static information is extended during the runtime. In Fig. 2 and Fig. 3 dynamic elements are highlighted with dashed lines.

Static Information The templates as a whole are stored in corresponding files. From this files the name of the template can be obtained, e.g., the template p.T1 is stored in p/T1.tmp. Furthermore, all template-specific information defined within that template are static. This applies to hook points. Consequently, by solely analyzing the template files, we can build up a symbol table containing all TemplateSymbols and their HPSymbols. This enables conducting static validations in order to check the referential integrity of the templates. Template k.T2, for example, refers to the hook point p.T1:HP1 (l. 2, Lst. 2) and the templates q.T3 and p.T1 (l. 3). The static information in the symbol table enables to check whether the hook point and the two templates exist. If, for instance, q.T3 does not exist, it will be detected before code generation.

Dynamic Information A global variable, is considered to be a dynamic information. The reason is that the template it is defined in might not be processed during runtime. Hence, storing the global variable in the symbol table during design-time might be wrong. For example, p.T1 defines the global variable aV (l. 3, Lst. 1). However, this has no affect if p.T1 is not executed itself. Also, aV’s value can be changed from within any template. Similarly, the boundTo information of a hook point can be set from any template. In Lst. 2 (l. 2), the template k.T2 binds the hook point HP1 of p.T1 to the string value "value of hp" and overwrites the previous binding, if one existed. Finally, the replacement information can be (re-)set from within any template, and therefore, is a dynamic information, too.

5. Conclusion

While code generators are an integral part of model-driven development, customization approaches for the code generator itself that allow for dynamic and static checking for template-based code generators are currently lacking.

Hence, in this paper we extracted the basic elements of existing customization approaches and provide a classification and the basic elements of guided and unguided customization approaches. For guided approaches hook point definition and binding a value to a hook point are essential. For unguided approaches template replacements are the basic concepts. With this understanding, we derive requirements that have to be fulfilled for a data structure that should manage the customizations. A data structure that naturally fulfills these requirements is the symbol table. We have employed the symbol table to manage guided and unguided customization. The approach shows that static and dynamic checking of customizations becomes feasible. In consequence, errors can be detected before execution of the code generation rather than afterwards.

References


Mixed Generative and Handcoded Development of Adaptable data-centric Business Applications

Pedram Mir Seyed Nazari Alexander Roth Bernhard Rumpe
Software Engineering RWTH Aachen University {nazari,roth.rumpe}@se-rwth.de

Abstract

Consistent management of structured information is the goal of data-centric business applications. Model-driven development helps to automatically generate such applications. However, current approaches target full or one-shot generation of business applications and often neglect simplicity and adaptability of the code generator and the generated code. Moreover, it is necessary to inspect the generated code in order to add functionality. Thus, here we discuss mechanisms for a code generator to generate a lightweight and highly customizable data-centric business application that is targeted for a variety of users including generated application users, tool developers, and product developers. We achieve simplicity by reducing the mapping of the input model to the generated code to a minimal core of easily understandable concepts. As a consequence, the generated code does not need to be read or understood, since the input model clearly describes what is generated. High customizability is achieved by providing a variety of mechanisms to extend the generator and the generated code. These include template overriding and hook points to extend the code generator. Moreover, to extend the generated code we use hot spots and additional manual extension approach. It is even possible to fully control the code generator and the entire generation process via a scripting language.

Keywords Data-centric Business Application, Generative Development

1. Introduction

Data-centric business applications provide management functionality for structured and consistent information. They offer CRUD (create, read, update, and delete), search, and persistence functionality [16, 17]. Existing model-driven development approaches allow nearly full code generation [14]. Such generators can be powerful tools when used by experienced users. However, developers not familiar with such approaches hardly accept them, because of their complexity and the loss of control [12, 15]. Consequently, adapting and customizing the code generator or the generated output becomes a labor-intensive and time-consuming task.

Even if nearly full code generation is achieved, simplicity (the amount of languages needed to describe the business application and the amount of approaches to integrate hand-coded extensions), ease-of-use, and adaptability is not much addressed by current research [2, 3, 9, 18]. Previous work has proposed an infrastructure for generating enterprise applications [11, 13]. This infrastructure consists of multiple code generators and languages, which describe the enterprise applications in a generative way. Nevertheless, the provided code generators and infrastructures employ a variety of modeling languages and may require to develop entire code generators when changes to the generated software system are required.

In this paper, we present a generator that aims at demonstrating the power of the generative software development methodology using the generator framework MontiCore [10]. Our main contribution is a demonstration of easy-to-use generation of almost ready-to-use business applications from abstract models as shown in Fig. 1. This approach is different to existing work as it only requires one input language to describe the data to be managed, provides clear customization approaches for the code generator and the generated systems, and presents a code generator that is designed to automatically integrate hand-written and generated code. In particular, we use a variant of UML class diagrams and produce running Java applications. The generated applications provide a graphical user interface to manage instances of the modeled system. Furthermore, they allow to persist instances in the cloud and share them among users, which may have different roles and rights.

Figure 1. Overview of generation process.

The proposed generator provides an approach to design highly customizable and adaptable code generators by offering a variety of extension mechanisms to even allow to fully control the entire generation process.

The rest of this paper is structured as follows. We first give a brief description of the input language and the generated output in
Sect. 2. Then, we describe how we achieve high customizability by using hook points, hot spots, template overridings, and a control script in Sect. 3. Finally, we conclude our paper in Sect. 4.

2. Generated Applications from UML Class Diagrams

The input language of our generator is a variant of UML class diagrams that allows to focus on data modeling (without addressing methods). From this input the generator produces (parts of) business applications. The generated product is an executable application for managing data that conforms to the UML class diagram description. It offers CRUD (create, read, update, and delete) management functionality to manage objects and associations. Additional support is provided by the graphical user interface that allows browsing, searching, and filtering. On invalid input the generated interface offers instant feedback. In addition, database and multi-user support with role-based access control is generated to allow specification of users, roles, and CRUD operations. Both functionality is provided without employing additional modeling languages.

To simplify the usage of the code generator, we identified three roles with different requirements. First, end users of the generated product are unaware of the technical details but simply want to use the generated application. Their main focus is on a user interface that is systematically structured and easy to use. In contrast, product developers need to handle and manage the generated code to provide extensions and customizations. However, they are usually not interested in the implementation details but mainly in the interfaces and provided APIs. Finally, tool developers are highly interested in the implementation and how the overall architecture of the generated application is to be able to adapt and extend functionality of the generated code. As a consequence, we provide a highly customizable code generator and additional concepts to adapt the generated code without the need for a detailed inspection.

2.1 Input Models

The input language is a reduced variant of UML class diagrams and provided in textual form designed using current understanding of semantics and domain-specific design guidelines [7, 8]. Certainly, it does not provide much application-specific functionality. Therefore, various extension and adaptation mechanisms are introduced to extend the functionality of generated products. Nevertheless, the input language is sufficient to describe the managed data and generate a working application.

```java
class SocialNetwork {
    abstract class Profile {...}
    class Person extends Profile {...}
    association Person -> (friend) Profile [1];
}
```

Listing 1. Input model example

The input language focuses on the most important concepts of UML class diagrams especially suited for documenting analysis results. An example is given in Lst. 1. It contains classes, interfaces, and abstract classes. Classes may extend other classes and implement interfaces. We use associations with navigation directions and cardinalities. An association as well as each of its role ends may have names. Associations can be ordered or qualified. Ordered associations are marked using the ordered stereotype and qualified associations require a qualifier. Classes have attributes with associated types. Compositions are omitted in the model, but they are supported as a special form of associations.

2.2 Generated Applications

The generated application is a typical 3-layered architecture composed of the graphical user interface, the application core, and the persistence management to structure its products. The application core realizes only business functionality. As illustrated in Fig. 2, the layers are independent and can easily be exchanged by different implementations. Each layer has its own runtime environment and standard components for accessing predefined not generated functionality.

![Figure 2. Overview of the generated application's architecture.](image)

Since not every functionality can and needs to be generated from the input model, already existing code is reused. There are many sources for this kind of reuse. In order to make use of this kind of reuse, the generated architecture relies on a run-time, which is deployed with the generated application and provides access to external libraries. An example for the need of external libraries is role-based access control. The generated applications allow for create users, roles, and to define the CRUD operations of each role. Since generating role-based access control from abstract models is hard, we generate code that is compatible with Apache Shiro [1]. Hence, no need for introducing a new modeling language is given and developers can rely on existing infrastructure. A major benefit of employing Shiro is the possibility to be very fine grained and for example to define rights on attribute or association level.

3. An Intelligent and Customizable Generator

A code generator becomes helpful, when it effectively assists developers to speed up their work. This is only possible, when the generator actually takes some burden from the developer. For example, by making certain decisions and generating corresponding functionality. Our generator for example targets desktop applications with a layered architecture. Based on that choice, it embodies a variety of additional functionality that can be generated automatically.

Besides taking some burden from the developer, it is an intrinsic property of a good generator to be able to adapt either the generation process or the generated code. In particular, for algorithms that usually cannot be described in a more abstract form than the implementation of the algorithm itself, manual implementation is necessary. Due to this, we provide a variety of extension mechanisms to allow for high customizability of the code generator and the generated code.

For the code generator, we provide explicit hook points, which are dedicated spots in templates that are intended to be customized and extended. Additionally, a more detailed level of customization is provided by allowing to replace every templates of the code generator with a custom template. Finally, in order to give developers full control of the generation process, which includes parsing models, checking context conditions, and generating code, we employ Groovy [6] as a scripting language to control the generator. Hence,
the generator becomes an active library [4], where only parts of the code generator can be executed and the generated code can be adapted.

For the generated applications, we offer hot spots as a dedicated spot in the generated code, which is usually known from frameworks as provided methods that have to be overridden, and concepts to extend the generated classes [5]. We strictly separate hand-coded artifacts from generated artifacts to allow complete regeneration without loss of the customizations and adaptations. This provides from overriding hand-coded extensions by generated code and requires developers to only version the input model and the hand-coded artifacts. The code generator detects handwritten extensions and adapts the generated code accordingly to regard it. As Fig. 3 shows this kind of extensions are supported on each layer of the generated architecture.

![Figure 3. Overview of the generated application's architecture with handwritten extensions.](image)

4. Conclusion

Generating data-centric business applications is a complex task and currently requires deep knowledge of multiple modeling languages and the underlying code generators. To tackle the low acceptance of model-driven development approaches to generate business applications, a simplified approach to generated business applications, which requires only one UML class diagram and provides clear customization concepts, is helpful.

In this demonstration, we present a code generator that uses a variant of UML class diagrams as input to generate lightweight but feature rich business applications. The code generator’s main focus is on simplicity and adaptability. This is achieved by reducing the input language to one simplified language, adapting the code generator to take some decisions from the developer, and simplifying the mapping of input language concepts to output language concepts. However, since no application specific logic as well as behavior can be expressed, we provide a variety of adaptation mechanisms to adapt the code generator and the generated applications. It is even possible to customize the complete code generation process.

References

Reusing Legacy DSLs with Melange

Thomas Degueule  
Benoit Combarele  
INRIA, France  
first name.last name@inria.fr

Arnaud Blouin  
INSA Rennes, France  
arnd.blouin@irisa.fr

Olivier Barais  
University of Rennes 1, France  
olivier.barais@irisa.fr

Abstract
The proliferation of independently-developed and constantly-evolving domain-specific languages (DSLs) in many domains raises new challenges for the software language engineering community. Instead of starting the definition of new DSLs from scratch, language designers would benefit from the reuse of previously defined DSLs. While the support for engineering isolated DSLs is getting more and more mature, there is still little support in language workbenches for importing, assembling, and customizing legacy languages to form new ones. Melange is a new language workbench where new DSLs are built by assembling pieces of syntax and semantics. These pieces can be imported and subsequently extended, restricted, or customized to fit specific requirements. The demonstration will introduce the audience to the main features of Melange through the definition of an executable DSL for the design and execution of Internet of Things systems. Specifically, we will show how such a language can be obtained from the assembly of other popular languages while maintaining the compatibility with their tools and transformations.

Categories and Subject Descriptors D3.2 [Language Classifications]: Specialized application languages

Keywords Domain-specific languages, language workbench, language reuse, model typing, melange

1. Introduction
Domain-specific languages (DSLs) are increasingly used to handle specific concerns in the development of complex software systems [1]. However, developing a DSL is still a costly and time-consuming task that requires advanced skills in language design: language designers have to specify the abstract syntax of their languages, their concrete syntax, semantics, and tooling (e.g. editors, checkers, code generators, etc.). Language workbenches assist language designers by providing the right tools and methods to tame the complexity of language design and reduce the development costs [2]. While current workbenches (e.g. MetaEdit+, Spoox, MPS, Xtext – to cite just a few) propose a diffuse way to reuse language modules, there is currently little support for assembling languages with customization facilities. Yet, it is likely that the creation of new DSLs could benefit from the efforts spent on the development of other ones, especially when their domains overlap. A mere example is the family of statecharts languages which, despite their specificities, share many similarities in their syntax and semantics [3]. The expected outcomes are twofold: one would like to import the definition of legacy language artifacts to engineer a new one while ensuring the compatibility with the tools and transformations defined on its ancestors. Of course, imported artifacts may not fit exactly the designer’s expectations. It follows that support for language extension, restriction, and customization is required to tune them finely.

This paper is organized as follows. In Section 2, we introduce Melange, a new language workbench attempting to address each of these challenges. In Section 3, we present an outline of the proposed demonstration.

2. The Melange Workbench
Melange [4] is an open-source language workbench built on top of the Eclipse Modeling Framework (EMF) and tightly integrated with its ecosystem. Thanks to the success of EMF in both academia and industry, this enables Melange’s users to import and manipulate a wide spectrum of existing DSLs. In Melange, the abstract syntax of DSLs is defined in the form of a metamodel with the Ecore formalism. Their operational semantics is specified using the Xtend programming language. More precisely, Melange supports the definition of aspects which allows to define the operational semantics of the concepts contained in a metamodel in a non-intrusive manner, based on static introduction [5]. In Melange, pieces of syntax and semantics can be imported and assembled to form new languages. The resulting languages follow the same design principles: they consist of a metamodel and a set of aspects that can be directly bundled and deployed as is, or reused in other assemblies. In order to fit unforeseen requirements or new environments, Melange also provides customization operators: languages can be merged together, inherited, or sliced. Each of these operators takes both syntax and semantics into account. The merge operator serves as a language unification mechanism and is inspired by the UML Package-
Merge relation [6]. The slice operator is inspired by model slicing [7] and consists in extracting a subset of an existing language to be imported in a new one. Finally, the inherits operator allows to reuse the definition of one or more super-languages into a new language. In addition to the merge operator, the inherits operator ensures that the resulting language remains compatible with its super-languages, i.e. the tools defined on a super-language can always be applied on its sub-language. Additionally, every language in Melange is associated with a structural interface captured in a model type [8]. This interface exposes the features that are publicly accessible on a language, e.g. its meta-classes, their structural features, and the methods defining its operational semantics. Most importantly, model types are linked one another with subtyping relations [9]. Intuitively, a model type is a subtype of another one if it exposes the exact same features, and possibly others, i.e. it is not any less capable. These relations and the associated type system provide model polymorphism, i.e. the possibility to manipulate a model through tools defined for different languages, providing that their interfaces match. Concretely, this means that when a language is built from another one (e.g. through inheritance or slicing), tools and transformations can be reused if the resulting interface remains substitutable with the one of its ancestor. The model polymorphism facility is not only available within Melange, but also contributed as a specialized resource management system directly in EMF. This means that any project relying on the EMF framework for model loading and serialization can benefit from the model polymorphism facilities provided by Melange. The overall approach is depicted in Figure 1.

3. Demonstration Outline

The demonstration will highlight the main features of Melange, illustrated through the creation of an executable DSL for the design and execution of Internet of Things (IoT) systems. The resulting IoT language is inspired from both general-purpose executable modeling languages such as fUML [10] and modeling languages dedicated to IoTs such as ThingML [11]. We will show how the assembly operators of Melange foster the reuse of pre-existing languages. Specifically, the IoT language will be designed as an assembly of publicly-available DSLs: (i) an IDL language for specifying the structural interface of sensors (ii) Lua for expressing their behavior and (iii) an activity diagram to express concrete scenarios involving different sensors. Taken independently, each of these languages has been defined by different groups of people for specific purposes, unrelated to IoT systems. Combining them in a consistent way, however, leads to a new DSL particularly suited to a new context, i.e. the IoT domain. Because most of their syntax and semantics can be reused as is, this drastically reduce the development costs compared to a top-down approach. Of course, the syntax and semantics of two independent languages may not fit together perfectly when composed. Therefore, the demonstration will also focus on the customization of these reused languages. Finally, the demonstration will illustrate how the resulting DSL remains compatible with the tools and transformations (e.g. checkers, editors) previously defined on the imported languages.

References

Figure 2: Assembling several variants of a finite-state machine language in Melange. The outline presents the abstract syntax of each language, the methods and runtime data inserted using aspects (e.g. a `fire()` method on transitions and the current state), and the subtyping relations linking their types. Language designers can thus build new languages on the one side and observe the results on the other side at the same time.
Supporting Users to Manage Breaking and Unresolvable Changes in Coupled Evolution

Juri Di Rocco  Davide Di Ruscio  Alfonso Pierantonio
Department of Information Engineering Computer Science and Mathematics
University of L’Aquila
name.lastname@univaq.it

Ludovico Iovino
Gran Sasso Science Institute, L’Aquila, Italy
ludovico.iovino@gssi.infn.it

Abstract
In Model-Driven Engineering (MDE) metamodels play a key role since they underpin the specification of different kinds of modeling artifacts, and the development of a wide range of model management tools. Consequently, when a metamodel is changed modelers and developers have to deal with the induced coupled evolutions i.e., adapting all those artifacts that might have been affected by the operated metamodel changes. Over the last years, several approaches have been proposed to deal with the coupled evolution problem, even though the treatment of changes is still a time consuming and error-prone activity. In this paper we propose an approach supporting users during the adaptation steps that cannot be fully automated. The approach has been implemented by extending the EMF Migrate language and by exploiting the user input facility of the Epsilon Object Language. The approach has been applied to cope with the coupled evolution of metamodels and model-to-text transformations.

1. Introduction
Metamodels play a key role in any metamodeling ecosystem [4] since they underpin the development of a wide range of modeling artifacts and tools including models, model transformations, textual and graphical editors, and code generators. Similarly to any software component metamodels are expected to evolve during their life-cycle [5] and as such it is necessary to deal with the coupled evolution problem i.e., managing the ripple effects that metamodel evolutions might have on all the existing related artifacts. Depending on the corrupting or not-corrupting effects, metamodel changes can be classified as i) non-breaking, if they do not break the relations between the evolving metamodels and existing artifacts defined on them, ii) breaking and resolvable if they break such relations even though the affected artifacts can be automatically co-adapted, and iii) breaking and unresolvable if they break the existing relations which cannot be automatically recovered because of lack of information [1]. In the last case, user intervention is required [5], e.g., to set the value of new obligatory attributes or references.

Over the last years, the coupled evolution problem has been largely investigated in MDE and different approaches have been proposed. Some authors have investigated the metamodel/model coupled evolution problem, and propose techniques and tools able to adapt models that are no longer conforming to the initial version of the changed metamodel and thus that have to be adapted with respect to its new version [1, 9, 10, 16, 17, 20]. Similar approaches have been proposed to deal with other coupled evolution problems, including model transformations [2, 6, 15], graphical and textual concrete syntax editors [3, 7], and OCL queries [13]. All these approaches aim at automating the management of non-breaking, and breaking and resolvable changes, whereas the management of breaking and unresolvable changes is still a challenging and error-prone activity.

In this paper we propose an approach enabling users to give input during the adaptation phases when needed. This permits to overcome the limitations of existing coupled evolution techniques that currently provide modelers with limited support for solving the lack of information induced by breaking and unresolvable changes. In fact, these are usually managed by applying default migration actions according to predefined heuristics [5]. In some cases partially migrated artifacts are produced and they have to be finalized by modelers to fix those parts invalidated by the unresolvable metamodel changes. The proposed approach is supported by an extension of the EMF Migrate language [5] that is implemented atop of the Epsilon platform. Even though the approach and the supporting tools are general and can be applied to adapt any kind of modeling artifacts [4], in this paper we consider an explanatory scenario requiring the adaptation of Acceleo-based model transformations.

1 This research was supported by the EU through the Model-Based Social Learning for Public Administrations (Learn Pad) FP7 project (619583)
2 http://www.eclipse.org/epsilon/
The paper is organized as follows. In Section 2 the coupled evolution problem is presented and we discuss the impact that breaking and unresolvable metamodel changes have on Accceleo-based transformations. In Section 3 we propose an adaptation process that makes explicit the activities involving users during the migration of affected artifacts. The supporting tool consisting of an extension of EMF/MIgrator, and its application on a concrete scenario are also presented. Related work is described in Section 4, and conclusions and research perspectives are given in Section 5.

2. Motivating scenario

The problem of restoring the consistency among evolving metamodels and existing artifacts is intrinsically difficult. Figure 1 shows a simple coupled evolution scenario consisting of the WebApp metamodel and two different artifacts depending on it, i.e., the WebApp Model and the WebApp2HTML transformation. The WebApp metamodel consists of modeling constructs specifically conceived to develop Web applications which can be generated by means of the provided WebApp2HTML transformation out of source WebApp models. As shown in the upper side of Fig. 1 the WebApp metamodel might require changes in order to address unforeseen requirements or to better represent the considered application domain. Such changes might invalidate existing artifacts, as for instance the WebApp Model and the WebApp2HTML transformation on the left-hand side of Fig. 1 that might have to be adapted to recover their relations (i.e., conformance and domain conformance [19]), respectively with the new version of the WebApp metamodel.

Figure 2 shows the initial version of the WebApp metamodel consisting of modeling constructs for developing Web applications implemented by following the Model-View-Controller pattern (MVC)[14]. The WebApp metamodel permits to specify models like the one shown in Fig. 3.a.

Figure 3.b shows different JavaScript (JS) and HTML files automatically generated by means of an Accceleo-based WebApp2HTML transformation applied on the model shown in Fig. 3.a. The generated source code consists of the index.html file used as main entry-point of the application and different JS files, one for each main building block of the application i.e., models, collections, templates, views, the controller and the router. Accceleo transformations are based on templates that identify repetitive and static parts of the applications, and embody specific queries on the source models to fill the dynamic parts.

As previously mentioned, metamodels are living entities that can be changed during their life-cycles. Thus the metamodel shown in Figure 2 can be subject to a number of changes that affect all the artifacts defined on it. In the remaining of the paper, we focus on the effects that breaking and unresolvable changes might have on existing Accceleo transformations. In [2] we already proposed a tool-supported approach able to adapt Accceleo-based transformations with respect to the performed changes on the source metamodels. In particular, the approach already automates the management of non-breaking changes and breaking and resolvable ones. Concerning breaking and unresolvable changes the approach applies default heuristics that necessarily have to be checked by developers once the whole adaptation process has been executed.

Even though the availability of a tool which is able to support non-breaking and breaking and resolvable changes represents an important achievement, managing breaking and unresolvable changes by hand still remains an error-prone process.

---

1 https://eclipse.org/acceleo/
process, which necessarily demands a dedicated support. In the remaining of the section we discuss the extract metaclass change as a representative metamodel modification that supports such a claim. Because of space limitation, we limit the explanatory discussion to this metamodel change only. A treatment of all the changes included in the catalog that we initially described in [1] and that we are iteratively extending [18] is beyond the scope of this paper.

Figure 4.a shows the new Style metaclass with two attributes (src, href both of type String) that have been added in the evolved version of the WebApp metamodel. Such a new metaclass has been added to represent an external style sheet document that can be applied on a given template. The attribute style in the old metaclass Template has been removed and a new reference with the same name has been added, by giving place to a replacement of an attribute with a reference. Such a change affects the Accelere template devoted to the management of Template elements (see Fig. 4.b). In particular, the [aTemplate.style] expression at line 57 cannot be evaluated since the attribute style is not existing in the new version of the metaclass Template. In order to make the template again applicable, it is possible to adapt the expression in two alternative ways, i.e., [aTemplate.style.src] or [aTemplate.style.href], each corresponding to the EString attributes in Style.

3. Proposed adaptation approach

In this section we propose a human-in-the-loop adaptation process (Section 3.1), which is an improvement of the work in [4] and that relies on an extension of the EMFMove language [5] presented in Section 3.2. The proposed extension permits users (i.e., humans that have developed the modeling artifacts to be migrated) to provide feedback during the execution of migration programs as discussed in Section 3.3.

3.1 Process overview

Figure 5 is an overview of the proposed adaptation process, which enables user feedback during the execution of migration phases. Adapter is the entry-point of the overall process and given two subsequent versions of the same metamodel, it invokes the operation getMetamodelChanges of the MetamodelComparator component which gives back the calculated metamodel differences. For each metamodel change properly represented in a difference model, Adapter invokes the Migrator component, which in turn involves different components depending on the kinds of changes to be managed. If the change is not breaking, then all the related elements of the artifact to be co-evolved are copied by means of the ConservativeCopier object similarly to what the Flock [20] approach does. If the change is breaking and resolvable then the BRCMigrator is involved in order to migrate the affected artifacts. Finally, if the change is breaking and unresolvable then the BUCMigrator is involved, which in turn properly asks user inputs that are necessary to resolve the change.

In the next section we describe the tool we have implemented to support the development of the Migrator and the UserDialog components shown in Fig. 5. Since the management of non-breaking, and breaking and resolvable changes has already been discussed elsewhere [1, 5–7], in the next section we focus on the management of breaking and unresolvable changes that represents the main novelty of this paper.

3.2 Supporting tool

The specification of migration programs able to manage breaking and unresolvable changes is performed by means of an extended version of EMFMove [23] which is presented in the remaining of the section. Its implementation relying on the Epsilon platform is also discussed.

3.2.1 EMFMove in a nutshell

EMFMove is a domain specific language for specifying migration programs consisting of migration rules applied on a given artifact A conforming to a metamodel MM. Migration rules are applied if the corresponding guards evaluated on an input delta model hold. Such delta model represents the changes operated on the initial metamodel MM. The body of a migration rule consists of a sequence of rewriting rules like the following:

\[ s[guard] \rightarrow t_1[assign_1]; t_2[assign_2]; \ldots; t_n[assign_n] \]

where \( s, t_1, \ldots, t_n \) refer to metaclasses of the considered artifact metamodel (e.g., the metamodel of Accelere templates to be migrated), and guard is a boolean expression which has to be true in order to rewrite \( s \) with \( t_1, t_2, \) and \( t_n \). It is possible to specify the values of the target term properties by means of assignment operations (see assign, above). The guard of each rule is evaluated on the delta model in order to apply the corresponding migration rule. The delta model can be manually specified, or automatically generated by means of existing model differencing approaches (e.g., EMFCompare\(^4\)). EMFMove, originally implemented by a semantic anchoring towards EMFTVM [23], has been suc-

\(^4\)http://www.eclipse.org/emf/compare
cessfully applied to migrate different kinds of artifacts [4],
even though before the extension proposed in the following it was not effective in managing breaking and unresolvable changes. In fact, in such cases the language permitted to specify at development time default adaptations by implementing some predefined heuristics.

Listing 1 is a fragment of the AcceleoMigration program, which has been developed for adapting Acceleo-based templates like the one discussed in the previous section. In lines 6-15 the rule specifies the metamodel changes that have to match with the content of the delta model in order to trigger the subsequent migration rules. The specified changes represent the extract metaclass modification discussed in Section 2. In particular, according to the specified guard, the extractMetaClass rule matches when a metaclass ccl is changed and a new metaclass ac2 is added. The changes on the ccl consist of the addition of a new reference ar1 and the deletion of an existing attribute dal. According to the where expression at line 27, the metaclass ac2 should be the type of the added reference ar1 in ccl, and the name of ar1 should be the same of the deleted attribute dal. By considering the running example, the guard specified in lines 6-15 matches with the extract metaclass change shown in Fig. 4. In particular, ccl matches with the metaclass Template of the WebApp metamodel, whereas ac2 matches with the added style metaclass. The reference style of the changed Template metaclass matches with the ar1 reference of the guard.

3.2.2 Extension of the EMFMigrate language
As discussed in Sect. 2, when the extract metaclass change occurs there might be different ways to migrate affected Acceleo templates. Instead of choosing one of the possible ways and fixing such ambiguities at development time, we have introduced in EMFMigrate the new constant prompt which permits i) at development-time to specify alternative adaptations for each metamodel change, and ii) users to be noticed about them and fix the ambiguities at run-time. In the migration program shown in Listing 1 the prompt construct (see line 20) is used to fix all the PropertyCallExp elements of existing Acceleo templates that refer to metamodel features that have been affected by the extract metaclass change (e.g., the expression aTemplate.style in Fig. 4b).

In the shown migration program, the ambiguity is solved by asking users to choose the added attribute aal or aa2 that will be instantiated at run-time. In the running example, such attributes would match with the attributes src and href, respectively of the metaclass Template.

Listing 1: Extract metaclass migration for Acceleo templates in extended EMFMigrate

```
1: migration AcceleoMigration;
2: metamodel MTL;
3: migrate AcceleoTransformation: MTL with delta;
4: ...
5: rule extractMetaClass [ ...
6: package cpl = changePackage;
7: class ccl = changeClass;
8: attribute dal = deleteAttribute ()
9: reference ar1 = addReference ()
10: ]
11: class ccl = addClass {
12: attribute aal = addAttribute ()
13: attribute aa2 = addAttribute ()
14: }
15: }
16: }
17: PropertyCallExpOld : MTL!PropertyCallExp -> propCallExpNew : MTL!PropertyCallExp { ...
18: name <- propCallExpNew.name,
19: ...
20: var attrChosen <- System.prompt("Choice the migration operation", Sequence(aal, aa2)),
21: eType <- attrChosen.eType,
22: source <- attrChosen.eContainingClass,
23: source.eType <- ar1,
24: referreProperty <- attrChosen
25: };
26: };
27: where [ar1.eType = aal and ar1.name = dal.name]...
```

3.2.3 Implementation
As shown in Fig. 6 the implementation of the extended EMFMigrate has been done by means of the EMFMigrate2-Epsilon transformation targeting the Epsilon Pattern Language (EPL) and the Epsilon Transformation Language (ETL). EPL implements a pattern matching language, which

---

Footnote: 1 https://github.com/MDEGroup/EMFMigrate
Figure 6. Generation of Epsilon artifacts from EMF\texttt{Migrate} provides support for specifying patterns that involve elements of models conforming to different modelling technologies. ETL is a hybrid, rule-based model-to-model transformation language built on top of BOL\textsuperscript{6}. ETL provides all the standard features of a transformation language, and has the ability to stop the execution of the transformation flow and then ask for user intervention.

Given an EMF\texttt{Migrate} specification like the one shown in Listing 1, the developed EMF\texttt{Migrate2Epsilon} transformation generates EPL specifications (e.g., see Listing 2) and ETL transformations consisting of operations and transformation rules (e.g., see Listing 3). More specifically, the guard of each migration rule in EMF\texttt{Migrate} gives place to a corresponding EPL pattern which represents how the specified guard of the migration rule can be matched with elements in the source delta model. The generated pattern plays a key role during the application of the generated ETL transformation, which consists of several transformation rules and operations each devoted to the management of a specific metamodel change. For instance, Listing 3 shows the ETL operation and transformation rule, which have been generated to manage the extract\texttt{MetaClass} metamodel change. In particular, the extract\texttt{MetaClass} rule makes use of the generated operation is\texttt{extractMetaClass} in order to check if the considered source element has to be migrated because of an operated extract metaclass change. The implementation of the is\texttt{extractMetaClass} operation makes use of the pattern in Listing 2.

\textbf{Listing 2.} Generated EPL pattern specifying the extract\texttt{MetaClass} change

\begin{verbatim}
1 pattern extractMetaClass
2    cpl : Delta\texttt{ChangedPackage},
3    col : Delta\texttt{ChangedClass},
4    da1 : Delta\texttt{DeletedAttribute},
5    arl : Delta\texttt{AddedReference},
6    acl : Delta\texttt{AddedClass},
7    sa1 : Delta\texttt{AddedAttribute},
8    sa2 : Delta\texttt{AddedAttribute},
9    match : cpl.\texttt{Classifiers}.\texttt{includes}(col)
10    and arl.\texttt{Classifiers}.\texttt{includes}(ar1)
11    and acl.\texttt{Classifiers}.\texttt{includes}(ar1)
12    and acl.\texttt{Classifiers}.\texttt{includes}(sa1)
13    and acl.\texttt{Classifiers}.\texttt{includes}(sa2)
14
15...\end{verbatim}

\textbf{Listing 3.} Generated ETL code related to the extract\texttt{MetaClass} migration program

\begin{verbatim}
2 var cpl = Delta\texttt{ChangedPackage}.\texttt{allInstances}()->select(c
3    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{cpl}));
4 var col = Delta\texttt{ChangedClass}.\texttt{allInstances}()->select(c |
5    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{col}));
6 var da1 = Delta\texttt{DeletedAttribute}.\texttt{allInstances}()->select(c |
7    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{da1}));
8 var ar1 = Delta\texttt{AddedReference}.\texttt{allInstances}()->select(c |
9    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{ar1}));
10 var acl = Delta\texttt{AddedClass}.\texttt{allInstances}()->select(c |
11    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{acl}));
12 var sa1 = Delta\texttt{AddedAttribute}.\texttt{allInstances}()->select(c |
13    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{sa1}));
14 var sa2 = Delta\texttt{AddedAttribute}.\texttt{allInstances}()->select(c |
15    c.instanceOf(Pattern\texttt{extractMetaClass}\texttt{sa2}));
16
17 return (not Pattern\texttt{extractMetaClass}.\texttt{isUndefined}())
18 and cpl.\texttt{applicationElement} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
19 and cpl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
20 and arl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
21 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
22 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
23 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
24 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
25 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
26 and acl.\texttt{propertyName} = propCall\texttt{ExpOld}.\texttt{source.\texttt{eType}}
27...\end{verbatim}

It is important to remark that the generated ETL transformations consist also of rules (not shown in Listing 3) that perform a conservative copy of all those elements that have not been affected by the occurred metamodel changes, and of rules managing breaking and resolvable changes. As discussed in the next section the execution of the generated Epsilon artifacts is performed by means of ANT documents, which are also generated by means of the EMF\texttt{Migrate2ANT} transformation shown in Fig. 6.

3.3 Execution of migration programs

The execution of EMF\texttt{Migrate} migration programs is performed according to the workflow shown in Fig. 7.a. In particular, the generated Epsilon artifacts are executed by means of the Epsilon execution environment by taking as input the artifact to be migrated and the difference model representing the operated metamodel changes. The required user feedback is also asked when needed during the migration that once completed produces the migrated artifacts.

Technically the workflow shown in Fig. 7.a is realized by means of generated ANT documents like the one show in Fig. 7.b, which is related to the WebApp running example. The ANT document consists of a number of tasks. In particular, by means of the \texttt{epsilone.emf.loadModel} task all the required artifacts are loaded i.e., the Delta model (see lines 3-6) representing the differences of the subsequent versions of the WebApp metamodels loaded in lines 15-22,

\textsuperscript{6} http://www.eclipse.org/epsilon/doc/eol/
4. Related work

Over the last years, the coupled evolution problem has been carefully investigated. In [1, 20] authors proposed an approach to deal with the coupled evolution of metamodels and models. In [11] Garcia et al. defined the correspondences between changes applied on metamodels and how they affect existing model transformations. In [22] authors propose a transformational approach to assist metamodel evolution by stepwise adaptation. They defined several relations between metamodels to characterize metamodel evolution. In [2] we have presented a similar approach for dealing with metamodel changes and Accelelo template-based code generators, without considering breaking and unresolvable changes.

In [17] authors use graph transformations to support model co-evolution. In particular, co-evolutions are specified as related graph transformations ensuring well-formed model migration results. The model migration approach has been extended by allowing transformation rules that have less restrictions so that graph manipulations such as merging of types and retyping of graph elements are allowed [16]. In [15] authors presented an approach devoted to an evolution method for model transformations in the context of GREAT [12]. Based on the evolution, the approach is able to automatically migrate certain parts of the transformations. When automation is not possible, the algorithms automatically warn the user if some semantic information is missing. The lack of information is manually fixed by users after the execution of the automatic part of the interpreter evolution. All the previously mentioned approaches aim at automatically migrate artifacts because of operated metamodel changes and when this is not possible they provide some support to the user when the automated process has completed and manual intervention is still required. Such approaches differ to what has been presented in this paper since they do not provide the means to guide users at run-time during the execution of the migration processes.

In [21] the authors present an interactive and iterative approach to meta-model construction enabling the specification of model fragments where they say that if a change is unresolvable, the user is asked to provide additional information or to discard the no longer conformant fragment. Even though this can appear similar to what has been presented in this paper, the approach proposed in [21] is intended to be used during metamodeling phases and to deal with the metamodel and model coupled evolution problem only. Our approach aims at dealing with the coupled evolution problem occurring when metamodels are already in production and several related artifacts have been developed from them. Edapt [8] is a promising tool providing mechanisms to record the changes that are executed on Ecore models. The approach permits to attach migration instructions written in Java to metamodel changes in order to restore the affected models. Differently to Edapt, EMPMigrate is a rule-
based DSL enabling users to specify migration actions that are translated to a target transformation language.

5. Conclusions and future work

In this paper we addressed the problem of supporting users to manage the adaptation of artifacts which have been affected by breaking and unresolvable metamodel changes. A general adaptation process has been proposed by making explicit those activities that are related to the management of user feedback. So far breaking and unresolvable changes have been solved by deciding at development time how to solve possible ambiguous situations, or by partially adapting affected artifacts. Thus after the migration steps, usually users have to perform error-prone and time-consuming activities to completely adapt the artifacts that have been only partially adapted by the employed migration approach. With the techniques proposed in this paper, we permit to involve users during the execution of the migration phases and to fix ambiguities at run-time. The proposed process is supported by an extension of the EMFReSolve language, which is implemented atop of the Epsilon framework. In the future, we plan to perform an extensive evaluation of the approach by undertaking different activities. In particular, we plan to consider all the metamodel changes reported in the catalog we are maintaining [18]. In turn, we will apply the approach to breaking and resolvable changes related to other kinds of coupled evolution problems e.g. those involving models, model-to-model transformations, and editors. We will also investigate how to extend the approach to migrate multiple artifacts defined on the same metamodel.

References

Towards Improving Software Security using Language Engineering and mbeddr C

Markus Voelter
independent/itemis
voelter@acm.org

Zaur Molotnikov
fortiss
molotnikov@fortiss.org

Bernd Kolb
itemis AG
kolb@itemis.de

Abstract
This paper explores the use of domain-specific languages for improving software security, which deals with developing software in a way that is not maliciously exploitable. Specifically, we demonstrate how modular extension of the C programming language can help with technical and process-related aspects of software security. Some of these examples are already implemented, some are analytical extrapolations from related work we have done in the past; a detailed empirical evaluation has not yet been done. We rely on mbeddr, an extensible version of C developed with the JetBrains MPS language workbench. We conclude the paper with a discussion of the potential drawbacks of the approach and how these can be addressed in the future.

1. Introduction and Contribution
Software security refers to the security properties of a software system’s implementation [1]. Various programming techniques as well as process practices can help with building a secure implementation. Many of the software security weaknesses originate from careless or wrong use of programming languages [2]. C is widely used in embedded software, cyber-physical systems and the Internet of Things as well as network infrastructure. The software in these domains is also often critical in the sense that security (and/or safety) flaws can cost a lot of money, expose networks, damage physical systems or endanger lives (examples of attacking an aircraft can be found in [3]). Hence, addressing potential security problems in C-based software is of paramount importance for these systems. We claim in this paper that it is possible to improve over the general purpose languages (and in particular, C) by transitioning to new, domain-specific tools and languages. Language engineering [4], the notion of building, extending and composing languages, makes developing such languages and tools feasible. Language engineering relies on language workbenches [5, 6], which are a class of tools that makes language implementation efficient.

Contribution This paper demonstrates security-enhancing language extensions for C. Some are implemented, others are analytical extrapolations of previous work. Based on these extensions, we proceed to point out the directions for future research.

2. Language Engineering, MPS and mbeddr

Language Engineering and MPS Language engineering refers to building, extending and composing languages. The field encompasses general-purpose programming languages and domain-specific languages (DSLs) [4]. Language workbenches [5, 6] are tools for efficiently designing and implementing languages. JetBrains Meta Programming System (MPS) [7] is an open-source language workbench that provides comprehensive support for many aspects of language definition, including structure, syntax, type systems, transformation and generation, debugging and integrated development environment (IDE) support. MPS relies on a projectional editor which avoids parsing the concrete syntax of a language to construct the abstract syntax tree (AST); instead, editing gestures directly change the AST, and the concrete syntax is rendered ("projected") from the changing AST. This means that MPS can work with a wide variety of (unparseable) notations such as mathematical symbols, tables and diagrams [8]. Since a projectional editor never encounters grammar ambiguities, they can support language composition [9]. Traditionally, projectional editors were hard to use and were not adopted much in practice. MPS, in contrast, makes editing in a projectional editor as close to “normal text editing” as possible and also supports diff/merge on the level of the projected concrete syntax; the study in [10] shows that users are mostly agreeable with the editor after a short while of getting used to it.

Embedded Software, C and mbeddr The benefits of projectional editors relative to notational flexibility and language composition have been explored in the context of embedded software engineering in the mbeddr project [11]. It provides a user-extensible version of C and ships with a set of predefined extensions such as physical units, interfaces and components, state machines and unit testing. The benefits of these extensions in terms of developer productivity, maintainability and robustness are discussed in [12]. mbeddr also supports product line variability, requirements traces and documentation. Finally, mbeddr explores the syn-
ergies between language engineering and formal verification by providing domain-specific verifications [13]. mbeddr is an open-source project licensed under the Eclipse Public License. It is currently being used in several commercial development projects and forms the basis for a future controls engineering product by Siemens PLM Software.

**Modular Language Extension** The extensions provided by mbeddr are modular, in the sense that the base language (C in our case) is extended with additional language concepts *without invariably changing* the base language. We call such extensions modular language extensions (MLEs); they include concrete syntax, type system, execution semantics as well as IDE support. They can also be seen as little embedded DSLs. We rely on MLEs in this paper to add security-relevant language abstractions to mbeddr. [4, 14] and [15] provide details on building MLEs in MPS.

### 3. Language Engineering and Security

Developing secure software relies on techniques and processes. Techniques (Section 3.1) refers to the languages, architectures and tools used to implement a software system. Different choices may make it more or less easy to build secure systems. C is a problematic language for secure systems because programs written in C are prone to low-level mistakes that can be exploited maliciously. Low abstraction level also makes it hard to analyze. Process (Section 3.2) refers to the practices employed to build the system [16]: reviews, education of the developers and a strong test and verification culture are ingredients of a process that can lead to more secure software. We now explore the potential benefits of language engineering, MPS and mbeddr for both aspects.

#### 3.1 Techniques

In this subsection we describe how extensible languages and language workbenches can be utilized to enhance software security by, for example, adding additional markup and checks to the source code, using higher level abstractions that prevent users from getting low-level details wrong or by changing the semantics of existing constructs in a way that makes the binary more secure.

**Code Markup and Checking** Code markup refers to annotations that are added to the code to express additional semantics. Checks associated with these annotations verify the semantics. One example is the support for physical units in mbeddr. Types and literals can be annotated with units, and the type system then checks for the correct use of units in expressions and assignments:

```c
int16/m/ dAlt = cur->alt - prev->alt;
int8/s/ dTime = cur->time - prev->time;
```

mbeddr's units do not directly focus on security, instead they address correctness and robustness (the Mars Climate Or-

biter crashed in 1999 due to a unit mismatch [17]). However, a similar approach can be used for security. Consider a system that deals with sensitive data. The data can exist in encrypted or unencrypted forms ($d_e$ and $d_u$). The software system is correspondingly structured into a non-secure and a secure part ($P_n$ and $P_s$). For security, it is crucial that no unencrypted data is in the non-secure part ($d_u \notin P_n$) and that data is encrypted as it moves from $P_n$ to $P_s$. A set of annotations on types, variables and modules similar to physical units can be used as the basis for data flow checks that verify these properties.

**Straightforward Language Extension** MLEs extend existing languages with additional, first-class language constructs in a modular way; they include syntax, type system, semantics and IDE support. An example for a security-relevant MLE is the *trysequentially* statement (which is part of the mbeddr tutorial). It can be used to address the `goto fail` bug found in Apple SSL implementation in 2014 ([https://gotofail.com/](https://gotofail.com/)).

```c
trysequentially {
  validateStep1(data, ...);
  validateStep2(data, ...);
  validateStep3(data, ...);
} on fail (errorcode) {
  handleFailedValidation(data, errorcode, ...);
}
```

trysequentially invokes a sequence of functions, each returning an error code. If a function returns a non-zero value (i.e., reports an error), the trysequentially branches to the error handler. This is a higher-level version of the following C-level idiom:

```c
if (validateStep1(data, ...) != 0) goto fail;
if (validateStep2(data, ...) != 0) goto fail;
if (validateStep3(data, ...) != 0) goto fail;
performValidation(data, errorcode, ...);
```

Apple's `goto fail` had a superfluous, unconditional `goto` statement; this prevented the correct validation of SSL certificates. Since the idiomatic C code is automatically generated from the more intentional and less error-prone `trysequentially`, fewer coding mistakes can happen, thereby improving security. All of mbeddr's existing MLEs (for unit tests, physical units, interfaces and components and state machines) represent direct language support for lower-level C idioms, thereby improving robustness and security by reducing the risk of mistakes in the lower level details.

We find another example to apply MLEs in Amazon's s2n TLS library ([https://github.com/avslabs/s2n](https://github.com/avslabs/s2n)) They use macros like the following one extensively:

```c
#define GUARD(x) if ((x)<0) return -1
```

The problems here are typical for macros. For example, the definition can be changed externally and it will not be prevented by an IDE: `GUARD (do_something ) + 1;` will return 0 in the case of failure. Using a language extension instead of a macro, an IDE can restrict the way the extension is used and prevent such problems.
The second problem of this macro is typical for macros that contain a \texttt{return} statement. Before returning from a function one should deallocate all the resources allocated in the function. However, the \texttt{return} statement in the macro can lead to dangling resources. Using language engineering, this problem can be solved in various ways. One could prevent the use of \texttt{return} statements in macros, or one could require to put resource deallocation code into a separate block which is then generated to be executed with every return statement (similar semantics to smart pointers).

\textbf{Adapting Semantics}\hspace{1em} The semantics of existing language are changed to make them more secure. Consider a system that works with secret keys. Likely, the key is itself encrypted (as $k_{enc}$), but to work with the key, it has to be available in the clear as $k_{clr}$. For the software to be secure, it is important that $k_{clr}$ is kept in memory only when absolutely necessary. Consider the following code:

\begin{verbatim}
char* encryptData(char* k_enc, char* data) {
    char k_clr[256];
    decryptKey(k_enc, k_clr);
    char* encryptedData = // encrypt with k_clr
    return encryptedData;
}
\end{verbatim}

At the end of this function, the $k_{clr}$ local variable becomes invalid by moving the stack pointer, but the memory allocated on the stack still contains the actual data and can potentially be exploited. To avoid this, the semantics of C should be changed in the following way: memory used by local variables that leave their block should automatically be zeroed. This can be achieved easily by a transformation that inserts zeroing code for each local variable at the end of a block. Optionally, these semantic changes can be combined with code markup (discussed above). For example, instead of performing the zeroing globally, it can be limited to functions that are annotated as \texttt{secure} or to local variables that are marked as \texttt{secure}. In embedded systems, this may be important to avoid unacceptable performance overhead.

A related feature is the prevention of writing this variable to disk as part of paging. Operating systems provide APIs to mark memory areas so as to prevent them from being paged. The code generator can call these APIs for all variables marked as \texttt{secure}.

\textbf{Exploiting the Generation Step}\hspace{1em} Most language workbenches are generative. For example, in mbeddd/MPS the AST of a program is translated to C text for compilation. MLEs are transformed to C in one or several steps. Beyond adapting language semantics, code generation can also be used for other security-related purposes.

A common attack vector are side-channel attacks [18] which exploit non-functional properties of a system to reverse-engineer details about the program’s implementation or about key material. A timing side-channel attack exploits timing properties. To prevent this, the timing behavior of a system must not deterministically relate to the operation of the system. To make this dependency harder to observe, random instructions (essentially noise that does not affect the program’s behavior) can be scattered throughout the code. Because the final to-be-compiled source code is generated, it is easy to automatically inject the noise with one cross-cutting transformation, without mixing the side-channel attack prevention concern with the business logic of the software. Code markup can be used to select critical areas where noise should be added.

An alternative way of decoupling execution time from input is to ensure that the respective part of the program always runs for the same time, for every possible valid input. This requirement is expressed through naming conventions in Amazon's s2n library as shown below, and developers ensure the requirement manually.

\begin{verbatim}
/* Returns 1 if a and b are equal, in constant time */
int s2n_constant_time_equals(string a, string b, len);
\end{verbatim}

Encoding this information in the name prevents the IDE from ensuring that the function actually runs in constant time. Using MLEs, an annotation can mark functions as \texttt{constant time}. Static analysis can ensure that every path through the program has the same execution time. A less sophisticated solution measures the elapsed time for any particular execution and then busy-waits to extend the time to the required constant time as necessary.

Another example of exploiting the generation step is the introduction of additional runtime checks. For example, a language extension can be defined that provides length-aware arrays or strings, and generates length/buffer checking code. Similarly, NULL-checks can be inserted before each pointer access to avoid segmentation faults and the subsequent crash of the application.

\textbf{Additional Constraints}\hspace{1em} An MLE can also contain constraints that prevent the use of insecure language constructs or library functions. For example, pointer arithmetic can be prevented or limited, and the use of insecure functions (such as \texttt{strcpy}) can be flagged as an error. Alternatively, the constraints can report as an error all uses of functions that are not explicitly marked as a secure API.

\textbf{Verification and MLE}\hspace{1em} Software verification refers to proving specific properties of a program. In contrast to testing, the program is not executed; instead, a verifier analyzes the program, often performing the semantic equivalent of an exhaustive search of possible execution paths. Verifying security-relevant low-level C details (such as division by zero, pointer or array access safety) is supported directly by tools such as CBMC [19] or Java Pathfinder [20]. However, verifying application-level properties is much harder. From a user’s perspective, the challenge is the specification of the expected properties (often done through code annotations or label reachability checks), configuration of the verifier (when configured wrongly it may not find existing property violations) and the interpretation of the results (which
are often too low-level and detailed. In [13] we introduce
an approach called domain-specific C verification that ad-
dresses these usability challenges. It relies on the follow-
ing steps: (1) define MLEs that imply or explicitly specify
application-level semantics (2) generate the correspon-
ding low-level C code including verification-specific code
notations or labels (3) automatically invoke the verifier (4)
that the low-level results back to the application level. We
have used this approach to verify component contracts in
Figure 1. Running the verifier to find the Heartbleed prob-
mbeddr C.
lembeddr, and in [13] we describe how to verify the func-
tional safety of a pacemaker implementation. An example
Figure 1 shows the mbeddr user interface after running a
CBMC-based robustness analysis. The top-right table shows
2 of the 40+ checked properties, one of which failed. The
dereference failure happens in the selected line contain-
ing a memcpy call. The bottom-right part shows a trace
that leads to the error. The nondeterministic assignment
in prepareUntrustedMessage() results in 1 byte allo-
cated in the payload, and the length set to 25. The ef-
fort to implement this verification is low, assuming the
MLEs for CBMC-based verification are available; only the
prepareUntrustedMessage() and a call to the verified
dispatching function are required.

MLEs could also be used to avoid such problems in the
first place: a native message type could be defined that
enforces consistency between a declared size and the buffer.
Associated serialization and deserialization functions can be
generated and can enforce this consistency. Such a data type
recently been added to mbeddr.

Finally, another area where first-class extensions can be
combined with verification is the specification of communi-
cation protocols, which are a major vector for attacks [21, 22]. If they are expressed as tables, state machines or se-
quence diagrams, this helps users to visually detect invalid
states. In addition, model checking techniques, as discussed
above, have also been used successfully to verify the correct-
ness of communication protocols [23, 24].

3.2 Process

In this subsection we discuss how better abstraction, domain-
specific notations and review support enable a more secure
development process. The stakeholders for the process as-
pects below are mainly the developers themselves; only for
the audits discussed at the end of this subsection do we ex-
pect non-developers to become involved.

Better Abstraction, Simplified Review Good abstrac-
tions can simplify the code review process. For a review
to be productive, it is important that the code can be ex-
plained and understood easily. The more directly the code
represents relevant domain abstractions, the more produc-
tive the review process becomes. For example, reviewing the
message extension can be more effective than the
review on the level the corresponding C code.

Better Notation, Simplified Review Beyond just suitable
abstractions, suitable notations are also essential because
they can more directly resemble established notations in
the domain, or because a particular notation reveal certain
problems in the code. Consider mbeddr’s state machines.
While the abstraction “state machine” is already a significant
improvement over its encoding as switch statements, the
textual notation can still be improved to make review even
easier. Fig. 2 shows a state machine represented as text and
as a table; a graphical notation is also available in mbeddr.
Another example for an easily-readable notation is given in
Fig. 3.

Tracing Code reviews are done to ensure the correctness of
the code (verification), but also to establish the code’s corre-

3This assumes that all involved parties know the semantics of the
tsequential extension. However, this is a reasonable assumption
in a team that develops software together.
Figure 2. A state machine edited as text and as a table.

response to the original requirements (validation). For this to be effective, the relationship between a piece of code and its associated requirements must be clear. Requirement tracing [25] addresses this problem by establishing explicit links between (parts of) implementation artifacts and particular requirements. In mbeddr, a requirements trace can be attached to any program node [26] (supporting tracing on any level of granularity) expressed in any language. Fig. 4 shows an example. If the code review should be driven by the requirements, navigation from a requirement to the traced program nodes is possible via MPS’ Find Usages support as well as dedicated trace reports (see below).

Expressing Security Requirements mbeddr ships with a requirements language [26]. Each requirement is specified with an ID, a short summary, tags, and a prose description. However, just like mbeddr C, the requirements language is extensible. For example, a classification scheme can be added that classifies requirements according to their security impact. Alternatively, requirements themselves can be traced to a set of overall security guidelines. Assessments in mbeddr are customizable reports over a model. They can be used to verify that every section of code is traced to a requirement (code for which there is no requirement is a potential attack vector), or that every security requirement has at least one trace. An example of an assessment is shown in Fig. 6.

Code Review and Security Audit mbeddr supports tracking the review state of code. This can be done at a customizable granularity, and for code expressed in any language. Code starts out as not reviewed. It can then be marked as ready for review (yellow; see Fig. 5). Once reviewed, the state changes to reviewed (green; the color scheme is based on [27]). Upon the change to yellow or green, a hash of the code structure is created and stored with the code itself (in an annotation that is optionally visible to the user). An assessment can be used to get an overview of the review state of the different parts of a system (an example is shown in Fig. 6). When the assessment is updated, the hashes are recalculated to determine which parts have changed and must be reviewed again. Code that has been modified since the last review is marked as raw (red).

This facility can be used for regular, team-internal code reviews that aim at detecting bad practices, potential for reuse, convoluted algorithms or bad naming. However, the same approach can also be used for security audits. Compared to code reviews, these are typically performed by different people and have a different goal: finding security vulnerabilities. They often go deeper, and should ideally be performed after every change to the code base, and only on the parts that changed (plus the locations affected by this change, which can be found through data flow and other analyses). The facilities discussed in the above paragraph can detect such changed pieces of code. To ensure that the code has been audited by the external team, the hash used for detecting the changes can be signed with the private key of the auditors. This way it can be cryptographically ensured

![Figure 4. The green-shaded labels are requirements traces. They can be attached to any program node, here they are attached to C constants and a state machine exit action.](image)

![Figure 3. Mathematical symbols used in C code simplify review of algorithmic code.](image)

![Figure 5. A piece of code can be annotated with a review state. The colors have the following meaning: yellow is ready for review, green is reviewed, red is recently created, raw. The review state is persistent and survives diff/merge operations.](image)
that the review has been performed by those authorized to do the review.

4. Discussion

mbeddr’s approach to improving security relies on domain-specific extensions to C programs. We have demonstrated the potential advantages and opportunities of this approach in the previous section. In this section we critically discuss the approach.

Evaluating the MLEs. Whether MLEs actually improve security can only be shown by experience, systematic attempts at exploiting the systems, or systematic code review. None of this has been done. In this paragraph we make two arguments why MLEs are a promising direction nonetheless. First, the experience gathered with mbeddr’s extensions have shown to improve modularity, testability and robustness of embedded software [12, 14]. A completely verified pacemaker implementation is discussed in [13]. We argue that improved robustness is an important building block of software security, since robust software has a reduced attack surface.

Second, we argue that the MLEs make C generally a better language according to Green’s Cognitive Dimensions of Notations [28], a set of established language evaluation criteria. The table in Fig. 7 contains the dimensions most relevant to this paper (the other dimensions are largely unaffected by the MLEs). Incrementally adding MLEs to C is a direct implementation of the Abstraction Gradient: the abstraction level can be increased incrementally if and when it makes sense. The user is not forced to encode everything in

Figure 7. Relevant Cognitive Dimensions of Notations.

<table>
<thead>
<tr>
<th>Abstraction Gradient</th>
<th>What are the minimum and maximum levels of abstraction exposed by the notation? Encapsularion?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closeness of Mapping</td>
<td>How closely does the notation correspond to the problem world?</td>
</tr>
<tr>
<td>Diffuseness/Terseness</td>
<td>How many symbols or how much space does the notation require to produce a certain result or express a meaning?</td>
</tr>
<tr>
<td>Error-proneness</td>
<td>To what extent does the notation influence the likelihood of mistakes?</td>
</tr>
</tbody>
</table>

either a (too) low- or a (too) high-level language. A suitable MLE can be used (or developed) for each particular case. Adding domain-specific abstractions and notations increases the Closeness of Mapping between the program and the domain. The traces also help bring the prose requirements closer to the implementation code. The additional abstractions and notations are also a way of adjusting the Diffuseness/Terseness of a language (or a specific program). Generally, a more terse program is better, since it exhibits lower complexity [29], assuming the language constructs used to achieve the terseness are known to all involved parties. Finally, as we have discussed above, using the right abstractions reduces the Error-proneness of programs because programmers do not have to deal with low-level details irrelevant for the problem at hand. These are all the reasons why we believe that MLEs have a good potential in secure software development.

Learning the MLEs. In order to use the MLEs effectively, users have to learn them. This cannot be avoided. However, as a consequence of the ubiquitous IDE support available in MPS, learning the MLEs is relatively simple. We also feel that learning the MLEs is a worthwhile price to pay for the security benefits. As discussed in [30], the usability and learnability of the projectional editor are appropriate for most end users. Of course, to make this practical, training material on the MLEs and the concepts behind them must be provided.

Developing the MLEs. The effort of developing the MLEs obviously depends on the level of sophistication of the MLE, but it is generally moderate. For example, the trysequentially MLE (including syntax, type system, transformation and IDE support) can be developed in one hour by an experienced MPS language engineer. Developing the tabular notation for an existing state machine language takes less than a day. The reason for the low efforts is that language workbenches such as MPS are optimized for rapid development of languages (this is discussed in [14]). The modular nature of the MLEs makes the overall complexity manageable. Modularity also allows growing the language [31] over time, developing extensions only as the need arises, avoiding costly up-front investments into MLEs that might not in fact be needed.

Trusting the MLEs. When we use higher-level extensions of a language in order to abstract over “irrelevant” details we implicitly trust the extension in two ways. First, we trust that we understand the MLE well enough for us to use it correctly. A well-defined extension should be relatively obvious to the users, so the risk of “using it wrong” is low (but not zero). Second, we trust the transformation that maps the MLE to its equivalent base language implementation. This is an example of tool qualification [32] in the sense through some mechanism we have to build trust that the semantics of the MLE is correct. In practice, this is done via a suf-fi-
ciently large) set of test cases as well as based on experience in practice ("proven in use" in ISO 26262). Although this may be sufficient in some use cases, others require the correctness of the transformation to be proven by analyzing the transformations. Eelco Visser and his group are working on using more formal, more analyzable languages for defining languages [33]. A related issue is known as feature interaction [34]: currently there is no way of predicting what happens if several independent MLEs are combined in the same program. Structurally and syntactically it is never a problem (thanks to projectional editing). But semantic interactions cannot be predicted because there is no formal description of the semantics. However, in practice this problem has not occurred with the 50+ C extensions developed in mbeddr.

In addition, the implementation of MLEs itself now becomes a critical part of the secure software. If an attacker or a careless programmer changes the language or the transformations he could break the security of the software implemented in the language without changing the software code itself. Thus MLEs should potentially be a part of the software project. They should be a subject for audits and reviews, changes to them should be tracked.

Working with Legacy Code Existing C code has not been written using the facilities discussed in this paper, but it may make sense to add the extensions retroactively. It is possible to import existing code into mbeddr. Once the code is in mbeddr, it can be refactored towards the MLEs. Our future work includes an investigation into whether such refactorings can be (partially) automated.

Tool Lock-in mbeddr, as well as the MLEs developed and suggested for improving software security require the use of the MPS language workbench (for developing the MLEs and also for writing code). At this point there is no way this can be avoided; there are no interoperability standards for language workbenches. However, both MPS and mbeddr are open-source software.

Other Languages In this paper we focus on C because a lot of secure software (in embedded and cyber-physical systems, the Internet of Things, as well as in operating systems and web servers) is written in C. However, the approach can also be used with other languages and tools. For example, MPS ships with an extensible version of Java; similar MLEs can be developed.

Other Tools The approach can also be used with other language workbenches: Spoofax [35] and Rascal [36] support some of the same language extension facilities as MPS (they are not projectional editors and hence are not as flexible regarding the notations).

5. Related Work

Modular C extensions have been developed for mbeddr [37] and in Cox [38] (without IDE support in the latter). Similarly, language extensions for improving security are an established idea [39], and so are dedicated DSLs for specifying aspects of software security [40]. Using static analyses to verify security-relevant properties of C has also been done before [41]. What our contribution adds is the modular language and IDE extensibility for security-relevant extensions, the use of integrated non-textual notations, as well as the combination of language extension and static analysis.

6. Summary

We have shown how modular language extension, in combination with the infrastructure provided by mbeddr and MPS, can be used to improve the security of embedded software. While empirical evaluation is still pending, we have argued why we consider the approach promising. Future work includes the development of specific security-relevant MLEs, as well as their systematic evaluation. We are convinced that MPS and mbeddr are useful platforms for research on improving software security through language engineering and we encourage other research groups to experiment with it.

References


Extensible Visual Constraint Language

Brian Broll
Vanderbilt University
brian.broll@vanderbilt.edu

Ákos Lédeczi
Vanderbilt University
akos.ledeczi@vanderbilt.edu

Abstract
The paper presents a visual, imperative language for the specification of constraints corresponding to domain-specific modeling languages (DSML) in the new WebGME modeling environment. The language is based on the visual notation introduced by Scratch. The novel feature of the approach is that the constraint language is just another DSML defined through UML class diagram-based metamodels. As such, it is easily extensible via metamodel inheritance. The visual constraint programs are automatically translated into asynchronous JavaScript code that utilizes the native WebGME APIs for evaluation.

Keywords: DSML, constraints, visual programming

1. Introduction
Model-Integrated Computing (MIC) is one approach that advocates the definition of domain specific modeling languages (DSML) via metamodels [11]. The DSML, in turn, enables domain engineers to model their system and use various analysis, simulation and code generation tools to solve their problems.

Constraints are used to specify well-formedness rules for a DSML. They allow the user to enforce rules that are difficult or impossible to capture by the metamodel. For example, a constraint may state that all models in a project have a unique name. Although this example is very simple, constraints can become rather complex as the complexity of the DSML and the models increase.

Typically, constraints are written in a declarative manner. Some current constraint languages include the Object Constraint Language (OCL) [12] and Microsoft FORMULA [5]. Although these languages provide a concise and unambiguous representation of constraints, they tend to be unnatural for domain engineers, increasing both development time and the likelihood of errors. On the other hand, using a traditional imperative programming language for constraint specification would not necessarily overcome these problems. Such constraints would need to utilize the general API of the given modeling environment with its associated—typically steep—learning curve.

Recently, many visual programming languages emerged with the explicit goal of making programming more approachable to novices, typically children. Of particular importance is Scratch [10] because it has gathered millions of users and many other environments adopted its intuitive visual notation. This paper presents our experience in creating a visual language based on Scratch for constraint specification for DSMLs.

1.1 WebGME
The Generic Modeling Environment (GME) [7] is a well-known tool supporting MIC. Recently the newest generation MIC toolsuite called WebGME has been introduced [8]. Designed with a strong focus on scalability and extensibility, WebGME provides many features improving the viability of modeling in large, real-world contexts including its web browser-based user interface, version controlled cloud-based backend and collaborative, real-time editing support similar to Google Docs.

Probably the single most important distinguishing feature of GME has been its support for prototypical inheritance. Any model in GME can be used as a prototype for the creation of a derived model. Any subsequent changes in the prototype (or base model) automatically propagate to the derived model. In turn, the derived model can be used as a prototype for further specialization. This way an entire inheritance tree can be constructed. This feature promotes model reuse and can be used, for example, to model product lines. WebGME takes prototypical inheritance one step further and uses it to fuse the metamodel and the domain models. The metamodel and domain models reside in the same inheritance tree and any change to the former immediately propagate to the latter. This makes DSML evolution seamless and fits nicely with the inherently iterative nature of DSML design.

As the WebGME client needs to run in the browser, it is implemented in JavaScript. WebGME supports the creation of custom plugins that, when executed by the user, can access the model through a set of JavaScript APIs and perform meaningful operations including modifying the models and/or generating output such as code. Similarly, constraints can be written directly in JavaScript to allow them to be evaluated on the client side. Note that the models are edited in a distributed environment in which the browser only loads the immediately required data from the server; hence, not all model elements required for evaluating a given constraint may be immediately accessible. Therefore, the client application may need to request additional data from the server during the evaluation of a constraint. To accommodate this, the constraint code often must contain asynchronous network requests to retrieve the desired data. Writing asynchronous JavaScript code is beyond the expertise of most domain engineers who would otherwise be inclined to use WebGME for their needs.

Therefore, the paper presents a visual alternative to both JavaScript code and declarative constraint languages such as OCL and FORMULA. This visual constraint language is designed for generating constraint code for use in the distributed environment of WebGME. The visual constraint language not only supports asynchronous code generation from synchronous visual programming blocks, but due to its WebGME-based implementation, it also provides scalability features not present in other modern visual programming languages such as version control and real-time collaboration. The visual constraint language also supports library functionality as well as extensibility using a visual interface. The extensibility not only allows the user to add new blocks to the language, but also to easily modify the base language. That is, if desired, the user can simply import the structural patterns of the base language and, using the modeling interface of WebGME, she can customize the language to support an alternative paradigm.
2. Related Work

In recent years, there have been approaches to visualizing constraints in object oriented models such as Visual OCL and Constraint Diagrams [6][11][4]. Visual OCL is a logical, typed, object oriented language which provides a graphical representation of the Object Constraint Language to make OCL easier to use and integrate into diagrams [3]. Visual OCL is designed to adhere to the UML standard to minimize the requirements of learning a new language [2][11][4].

Constraint Diagrams is another visual representation of constraints in object oriented models [6]. Like Visual OCL Constraint Diagrams, the visual constraint language uses an imperative programming paradigm. This provides the user a significant amount of flexibility over the constraints and allows the user a more natural way to perform any necessary operations on data prior to evaluating any given value. That is, in a case where the value to be evaluated must be calculated through the traversal of a large model, an imperative language is typically a better fit. However, as the visual constraint language is imperative, this requires the user to not only consider the constraint to be enforced for the given model but also how to check if the constraint is violated. This can provide increased complexity to creating the constraints and a less natural way to considering model validation.

Our visual constraint language is created with the ability to be easily extended and customized for specialized domains. Rather than simply using the generic visual constraint language, this allows users the ability to create domain specific visual constraint languages for their respective domain by simply extending or restricting the generic language. As the language is defined using a metamodel within WebGME, the modification of the generic visual constraint language to a domain specific visual constraint language can be performed without requiring the user to learn new languages or tools.

3. Visual Constraint Language

3.1 Architecture

Our visual constraint language is developed within WebGME, hence, it provides a number of advantages including a generic data model and a framework for custom data visualization. Also, using WebGME provides additional features such as version control and import/export functionality to the visual constraint language. These advantages stem from both the flexibility of the component-based nature of WebGME as well as the advanced functionality provided natively by WebGME.

The visual constraint language is composed of four main components: metamodel, model, visualizer and the compiler (implemented as a plugin in WebGME). The metamodel defines the syntax of the programming language. The model represents constraints created using the visual language. The visualizer provides the concrete syntax for the data by visualizing the model in an intuitive way for the user. The visualizer also allows for editing the model with respect to the rules defined in the metamodel. Finally, the compiler (using the provided language specification for the constraints) will parse the model and generate the constraint code with respect to the provided specified language. The dataflow among these relationships is illustrated in Figure 1.

![Figure 1: Visual Constraint Language Components](image)

We use visual blocks to represent the syntactic code elements of the visual constraint language. Blocks can have two different types of relationships between one another: either one precedes another or one block supplements the meaning of the other. If we consider the example given by Figure 2, we can see that the `begin` and `let` blocks are visually connected, representing the relative ordering of the two blocks. If we consider the orange `children` block and the `let` block, we can see that the `children` block provides meaning to what is being assigned by the `let` block. It is apparent that the `children` block supplements the meaning of the `let` block.

![Figure 2: Basic Block Relationships](image)

Using the WebGME data model, we utilize two concepts to represent the relationship between our language blocks: hierarchical containment and pointers from one node to another. That is, when a block in the language precedes another, we will simply create a pointer from the former to the latter (e.g., from `begin` to `let` in Figure 2), creating a singly linked list of block ordering. When a block supplements another, like `children` and `let` in Figure 2, the supplementing block will both be contained by the other in the WebGME data model and be the target of a pointer from the given block. Consider the containment tree visualization of Figure 2 below.

![Figure 3: Hierarchical Structure of the Constraint in Figure 2](image)

Along with relationships between blocks, a block may need to also include a user specified value in the block. In the WebGME data model, these fields are represented simply adding an attribute to the given block. However, if the block contains a value that can be specified with either another block or a user specified
value, then the block will contain both an attribute and a pointer with the same name.

Although the default WebGME data visualizer (using boxes and lines) is not appropriate for the visualization of a visual programming language, the component-based nature of WebGME allows for the creation of a custom visualizer to fit the needs of the visual programming language. This visualizer must be able to intuitively represent both the precedence and supplementary relationships as described above. The precedence relationship will be represented by visually connecting the subsequent block to its predecessor. As in the data model, the supplementary relationship will be represented with containment. Both relationships can be illustrated in Figure 2.

As WebGME allows for the creation of custom plugins to interpret models, the compiler can be implemented as a JavaScript plugin. This plugin will then have access to the WebGME data model and will be able to traverse the model as needed to generate the JavaScript constraint code. The WebGME plugin also contains an output language specification which provides the necessary information about the relationship between the blocks and the desired output language. Using this output language specification and the access to the WebGME model, the compiler can then generate the necessary asynchronous JavaScript constraint code.

3.2 Language Syntax

The syntax of the visual language is based on Snap! (a Scratch derivative), the educational visual programming language developed at Berkeley[9]. Like Snap!, the most fundamental level of the visual constraint language contains the following abstract concepts: base, hat, command, and predicate. As the name suggests, the base concept is the most fundamental element of the language and simply represents a syntactic element of the language. The hat is an element that can only have subsequent elements and the command can have both predecessors and successors in a code block.

In the metamodel, relationships between blocks are represented with pointers. User specified values (such as text entered into a block) are captured as an attribute of the node. If the block accepts either a block or user specified values, the block will contain both a pointer and an attribute of the same name.

Figure 4 shows the abstract syntax of these core concepts. The constraint block represents an element containing constraint code and the blue arrow represents a pointer, named “next”, from the hat block definition to the command block definition. The “next” pointer represents the next block to be executed in the interpretation of the code.

As the command block inherits from the hat block, the command also can have a “next” pointer to other command blocks (as can the hat block). The predicate represents a code element that is dependent on either a hat or command block; that is, a predicate can be used only to supplement the meaning of another block. This is shown in Figure 4 as the “next” pointer points only from a hat block to a command block and a predicate can be neither the source or destination of this pointer.

3.2.1 Data Types and Coercion

Using these core concepts, we extend the metamodel to add data types and functions to the programming language. The base language has 5 default data types: boolean, string, number, map, collection.

These specific data types were chosen as they provide intuitive, natural data types; boolean, string, number, and collection types are certainly natural concepts to a person without a programming background. The map data type, although it may be less natural, should allow the user to write more natural and concise code.

The creation of the language as a model allows us to imply data coercion from the structure of the metamodel; data coercion can be represented with inheritance in the metamodel. As a prototypical child in the metamodel will inherit its parent’s relationships, it follows that any child of a data type allows the child to be used in place of the given data type in any block connection. In the context of the visual constraint language, this results in an implicit casting of any child data type to the parent data type. Given this relationship of metamodel inheritance between data types and data coercions, we will now look at the specific implicit casting allowed by the metamodel of the base language in Figure 5.

![Figure 5: Base Data Types](image)

The two most important coercion relationships in the metamodel can be seen between the collections and maps data types and the string, boolean, and number data types. As the map data type inherits from the collection data type, it follows that the hashmap can be used in place of a collection data type. This implies that a map is treated as simply a specific type of collection.

3.2.2 Functions

In the metamodel, functions inherit their return type. This allows a function with a given return type to be used in place of any blocks of this data type (similarly to coercive data types).

Figure 6 provides an example of a function definition in the metamodel, the concat block. As the concat block returns a string type, it inherits from the string data block in the metamodel. As it contains 2 pointers to the string block, we know that it accepts 2 string arguments. Also, as the concat block has attributes with the
same name as the pointers, the `concat` block supports user entered text as arguments.

![String Functions Diagram](image)

Figure 6: String Functions

### 3.2.3 Additional Concepts

Along with the given data types and a number of functions, the programming language utilizes a number of additional concepts to facilitate the creation of constraints. This includes standard elements of imperative programming languages such as if and if-else statements, while loops, repeat loops and for-each loops as well as concepts specific to constraint generation.

The constraint concepts include a number of different types of concepts from model traversal to model querying. For facilitating model traversal, the language contains concepts for loading the hierarchical children or parent of a given node as well as loading entire subtrees of the containment tree. Concepts for querying the names and values of a node's attributes and pointers are also provided. Additionally, concepts for filtering collections of nodes by type and marking constraint violations in the model are also supported by the language.

### 3.3 Constraint Generation

As the generated constraints are to be executed in the distributed context of WebOM, the constraint generator must support creating asynchronous code from the synchronous visual programming blocks. This generation of asynchronous code from the synchronous visual programming blocks is perhaps the most novel aspect of the constraint generation. Using this asynchronous code generation, the compiler enables the user to create more manageable, synchronous code using the blocks and hides the complexity introduced by the asynchronous nature of network programming.

Supporting asynchronous code generation in the template requires support for moving subsequent code into a callback while preserving any variable assignment or similar function that requires the return value of the asynchronous function. Consider the following example of loading a node in JavaScript and assigning it to the variable `myNode` (where `nodeId` is the id of the desired node).

Using naïve template based code generation:

```javascript
myNode = getNode(nodeId, function(node) {
  
});
```

However, as `getNode` is an asynchronous function, the resulting node is actually the input to the callback function (rather than the return value of the function). Therefore, in our asynchronous code generation, we handle this by allowing the block's code to move its parent's code (the `assignment` block in our example) inside of the given code's callback.

To prevent undesirable behavior in the case of nested asynchronous functions, we only allow this movement of the parent code inside of the callback to occur once. As the parent code snippet contains the placeholder for the subsequent commands to be executed, this single movement will account for movement of all following generated code. Using this code generator, the previous example correctly moves the parent code inside of the child code as shown below.

```javascript
getNode(nodeId, function(node) {
  myNode = node;
});
```

Nesting the parent code within the child block’s code effectively allows the parent block to use the result of the asynchronous function. However, as subsequent code may also depend on the result of the the callback, all subsequent code is also executed within the scope of the callback. Just as every visual block contains a connection area to connect to subsequent code blocks, every block's code snippet contains a placeholder for the following block's code snippet. Maintaining the location of the following block's code snippet allows the subsequent code to be lifted into the asynchronous callback with the appropriate parent block’s code snippet. Allowing the current insertion point of subsequent code to be held with a placeholder facilitates the generation of more complex asynchronous code.

Supporting asynchronous functions also requires some modification to loops in the code blocks as they cannot necessarily be mapped to a synchronous “for” or “while” loop. This mapping could cause unexpected behavior as, if the loop contains asynchronous calls, the loop may enter subsequent iterations before the asynchronous call returns. In order to ensure the appropriate behavior, we map loops to recursive function calls where the recursive call is moved into the callback of the asynchronous function. As a loop with many iterations could result in a stack overflow, the recursive call is made asynchronously. Performing the recursive call asynchronously prevents the call stack from growing during subsequent iterations.

In JavaScript, this is implemented using the `setTimeout` function. JavaScript is implemented with an event queue which contains functions to be executed by the global object. The `setTimeout` function allows functions to be placed on this event queue. In the generated constraint code, loops are converted to recursive calls where the subsequent iterations of the loop (recursive function) placed on the event queue using `setTimeout`. This utilization of the event queue effective shrinks the call stack as desired and prevents any stack overflow errors as a result of any large loops in the constraint code.

Along with the support for asynchronous code generation, the visual constraint language code generation also contains a framework for testing of new constraint code blocks, an intelligent variable name mapping, basic name collision avoidance, lazy loading of nodes and a decoupled output language specification.

### 3.4 Example Constraints

#### 3.4.1 Unique Name Constraint

Figure 7 shows an example of validating that all containment descendants of a node have a unique name. As with all constraints, this constraint starts with a `Begin` node which marks the entry point of the constraint. The constraint then retrieves all descendents of the current node and assigns them to the “name” variable. The “name” node set is then iterated through using the `forEach` loop using “node” as the iterator. For each node, the name attribute is retrieved from the node and assigned to the “name” string variable. If the name has already been visited (and added to the “names” collection variable), then the constraint marks the current node as violating this constraint. Otherwise, the code will simply add “name” to the list of seen names (recorded in the collection, “names”) and continue.
Given the visual constraint defined in Figure 7, the following Javascript code is generated.

```javascript
function (core, currentNode, callback) {
  "use strict";
  var names = [];
  var name = null;
  var node = null;
  var queue = [];

  getNode(currentNode, function(arg0_7) {
    getDescendants(arg0_7, function(arg1_6) {
      queue = arg1_6;
      var fn_1 = function(){
        var arg1 = Object.keys(queue);
        var arg2 = arg1[0];
        while(arg0_2[arg2] &amp; arg1.length){
          arg2 = arg1.pop();
        }
        if ((arg0_2[arg2]){
          arg0_2[arg2] = true;
          node = queue[arg2];
        }
        function getDimension(names) {
          getDimension(name)
          names = names.concat(name);
        }
        setTimeout(fn_1, 0);
      } else {
        callback(err, violationInfo);
      }
    });
  });
}
```

3.4.2 Equal Incoming/Outgoing Connections

Figure 8 presents a constraint which will verify that the given node has an equal number of incoming and outgoing connections. The constraint first retrieves the incoming connections of the current node and stores the count in the "incoming count" variable. Similarly, the number of outgoing connections is stored in the "outgoing count" variable. Next, the two counts are compared and the node is marked as violating the constraint if the two values are unequal.

```
if (incomingCount !== outgoingCount) {
  // Mark violation
  // Define message
  // Set violation message
  this.violationMessage = Connections are unbalanced!
}
```

Figure 9 shows the three boxes added to the metamodel to create the given extension of the visual constraint language. As the connections are a type of node recognized by the visualizer, the connection set block inherits from the node set block. GetIncomingConnections and getOutgoingConnections both return a block of type connection set. As functions in our visual constraint language inherit from their return data type, these blocks inherit from the connection set data type. As both functions also are performed on a given node, they both have a pointer to a block of type node. Along with the modifications to the metamodel, these concepts will need to be added to the target language specification provided to the compiler. In this language specification, the names of the new visual blocks need to be provided with the corresponding code snippet. The code snippet for this example is provided below.

1 In the generated code, there are some convenience functions defined prior to the variables, such as "getDimension". For brevity, these convenience functions have been omitted from this code snippet.
getIncomingConnections:  
  'getConnectionsWith(\'src\', {{ node }},  
    function(\'placeholders.ARG[0]+\') {\n      ('' + async.START + '') + placeholders.ARG(0)  
      + ('' + async.END + '');  
    }\n  );

getOutgoingConnections:  
  'getConnectionsWith(\'dst\', {{ node }},  
    function(\'placeholders.ARG[0]+\') {\n      ('' + async.START + '') + placeholders.ARG(0)  
      + ('' + async.END + '');  
    }\n  );

In this code snippet, "async.START" and "async.END" mark the beginning and end of the desired content in the asynchronous function and allow the compiler to hoist the earlier code (i.e., length of block) into the callback of the "getConnectionsWith" function. The argument placeholders mark the insertion of an anonymous argument to be created by the function callback. The number specifies that the given arguments are the same value; argument placeholders with different numbers will be given unique names to prevent name collisions. As with "getDimension" in the code generated in the previous example, "getConnectionsWith" is a convenience method for retrieving all connection nodes with the given source or destination.

Although this provides a simple example of the extensibility of the language, it also illustrates using new data types to refer to types of nodes which could be difficult to define otherwise. Given a domain specific application, this concept can be further utilized to create very precise data types and functions that are unique to the given domain. At the visual constraint language is easy to edit as well as extend, this would allow the user to then remove unwanted generic types to create a smaller, more precise language to define their constraints. As this new language is domain specific, this can also create constraints that are more natural and familiar to a domain engineer.

4. Conclusion

The paper introduced a constraint language for using visual programming blocks to represent model constraints to be evaluated in a distributed environment. The visual language is based on a proven notation that works well for novice programmers. Building our visual constraint language as a WebGME DSMI provided a number of advantages over other visual programming languages including version control support and a collaborative development environment. These enable our visual constraint language, as well as any future versions or variations, to be useful for large scale projects.

Despite these benefits, the language is not always ideal for constraint creation. Complex constraints can be cumbersome to specify and working with visual language blocks is not always as flexible as writing textual constraint code. Also, the language does not currently support first class functions that can be called from within a constraint definition as in some other languages [9][10]. Unfortunately, this can make the definition of complex constraints quite cumbersome.

The presented constraint language not only provides a useful tool for constraint definition within WebGME, but it also constitutes the basis for future research. First class functions would ease the effort required for the creation of more complex constraints. Also, modifying the base language to a logic programming paradigm could potentially create a more concise constraint representation for some domains.

References

Systematic Evaluation of Three Data Marshalling Approaches for Distributed Software Systems

Hugo Andrade  Federico Giaino  Christian Berger  Ivica Crnkovic
Chalmers | University of Gothenburg, Sweden
{sica, giaimo, crnkovic}@chalmers.se, christian.berger@gu.se

Abstract
Cyber-physical systems like robots and self-driving vehicles comprise complex software systems. Their software is typically realized as distributed agents that are responsible for dedicated tasks like sensor data handling, sensor data fusion, or action planning. The modular design allows a flexible deployment as well as algorithm encapsulation to exchange software modules where needed. Such distributed software exchanges data using a data marshalling layer to serialize and deserialize data structures between a sending and receiving entity. In this article, we are systematically evaluating Google Protobuf, LCM, and our self-adaptive delta marshalling approach by using a generic description language, of which instances can be composed at runtime. Our results show that Google Protobuf performs well for small messages composed mainly by integral field types; the self-adaptive data marshalling approach is efficient if four or more fields of type double are present, and LCM outperforms both when a mix of many integral and double fields is used.

Categories and Subject Descriptors D.2.12 [Software Engineering]: Interoperability—Data mapping; E.4 [Coding and Information Theory]: Data compaction and compression; I.2.9 [Robotics]: Autonomous vehicles, Sensors

General Terms distributed software systems, data marshalling, self-adaptive data marshalling

Keywords distributed software systems, data marshalling, self-adaptive data marshalling

1. Introduction
Distributed software systems are powering complex cyber-physical systems like self-driving vehicles [3]. Also in the area of Internet-of-the-Things (IoT), where remote entities collecting data are connecting back to a cloud based data processing infrastructure, a distributed software system is present. Realizing such a distributed software allows system designers to flexibly deploy software components or to exchange the behavior of one component while preserving its public interfaces. This flexibility is achieved by standardizing the data modeling and exchange among them. Popular examples therefore are Google’s Protobuf [1] or Lightweight Communication and Marshalling (LCM) [6] that is also being used in experimental vehicles.

1.1 Problem Domain & Motivation
Both aforementioned approaches offer a message modeling language and code generation engine with bindings to different languages. Thereby, a distributed software system can even be realized on different platforms using different languages. However, once a message’s design is specified during development time, it cannot be modified during runtime anymore. In our previous study Giaino et al. [5] on the example of a cyber-physical system, we showed that the data that is exchanged in practice allows for saving bandwidth consumption as the difference between two consecutively sent messages is typically small in such application contexts. Thus, this delta can be encoded more effectively resulting in a faster data exchange as fewer bytes needed to be sent.

1.2 Research Goal & Research Questions
The goal for this work is to systematically evaluate the performance of the three different data marshalling approaches: Google Protobuf, LCM, and our self-adaptive data marshalling approach. The following questions were of particular interest:

RQ-1: How can different data marshalling approaches be systematically evaluated?

RQ-2: What is the performance of the respective data marshalling under various, application-independent conditions?

1.3 Contributions of the Article
We present a generic message description language (DSL) that serves as a super-set to common language features of Google Protobuf and LCM. This DSL is used to systematically create different message structures at runtime by our C++ middleware OpenDaVINCI\(^1\), where the native implementation of the respective data marshalling approaches was evaluated.

\(^1\)http://opendavinci.cse.chalmers.se
1.4 Structure of the Article

The rest of the article is structured as follows: In Sec. 2, we are outlining related work. In Sec. 3, the design of the DSL as well as implementation details are presented, followed by the evaluation in Sec. 4. The article concludes in Sec. 5.

2. Related Work

Huang et al. present in their work [6] Lightweight Communication and Marshalling, which is a spin-off of their 2007 DARPA Urban Challenge competition vehicle. They describe the language features alongside with a performance comparison towards the Robot Operating System (ROS) [7]. LCM encodes a message starting with a 4 byte magic number that also encodes the version of the protocol. Next, a 4 byte sequence number is part of a message header to describe fragmented messages. The third field in the header is a null-terminated string describing the channel number under which a message is transmitted. The last field of the header is an 8 byte hash value, which is iteratively computed for all <field name, field type> tuples constituting a message. This hash value allows the validation at receiver side whether the deserialization could successfully read back all data as the fields of a message are not separated by delimiters or identifiers.

Schwitzer and Popa [9] present an implementation of Google Protobuf [1] for resource constrained devices. Their C-based implementation of a self-contained serialization approach enables the use of the protocol in the domain of Internet-of-the-Things (IoT). The fundamental idea behind Protobuf is to encode integral data types not on their type as defined during design time but on their concrete value at runtime; thus, even fields with an int32_t field that would consume 4 bytes, could only occupy 1 byte if the value of that field is smaller than 128. This approach is called variable-length quantity (VLQ).

The general structure of a Protobuf message can start with a magic number followed by the length of the message; both attributes are encoded as VLQ. Next, the fields are encoded as tuples in sequence as they are specified. First, the field identifier in combination with the field’s data type, which is consuming 3 bits, is stored in the first byte; the key/type combination is encoded using VLQ as well. Next, the value is written to the byte sequence. For any integral type, VLQ is used. Floating point types are encoded either as 4-byte floats or 8-byte double fields. Strings or raw byte fields are encoded including their respective data length.

For processes running on the same computation node, low-level inter-process communication (IPC) as defined in POSIX like message queues or shared memory can be used. As these means would allow a higher performance between communicating processes, truly distributed software entities running on different nodes would not be supported. Thus, IPC is not considered in this study.

```java
message automotive.VehicleData {  double heading;  double absoluteTravelledPath;  double relativeTravelledPath;  double speed;  double temp; }

message automotive.VehicleControl {  double speed;  double acceleration;  double steeringWheelAngle;  bool brakeLights;  bool flashingLightsLeft;  bool flashingLightsRight; }
```

Figure 1. Example of a message description file from OpenDaVinci.

3. Generic Message Description and Self-Adaptive Marshalling

Our framework OpenDaVinci is used on different cyber-physical experimentation platforms such as scaled self-driving vehicles [2] and for distributed simulations [4]. It allows the realization of distributed software systems by providing different communication patterns like publish/subscribe or centralized hub-based communication, as well as centralized scheduling for algorithms running on distributed nodes.

Messages to be exchanged among the interacting software agents can be modeled at design time by a data description language specified with Eclipse Xtext. An accompanying code generator to our C++ environment was realized with Xtend. The data description language exhibits the following language features:

- Scalar types like uint8, float, double;
- Definition of initialization values;
- Nested types;
- Enumeration types and constants;
- Lists, fixed size arrays, and maps;
- Specialization via message inheritance.

An example of a design time artifact is depicted in Fig. 1. The language itself has similarities with Google Protobuf and LCM but also provides further concepts, like inheritance to describe relations of an application domain.

At design time, concrete message classes are derived from the given specification file providing methods to access the data fields and to serialize and deserialize the message. In addition, every class also implements the interface Visitable allowing a Visitor to query the data fields of a message without knowing the concrete type of given object at runtime.

The concept of visiting any message in an abstract way was also used to realize a generic message representation by transforming a given data structure into a list representation of its attributes. In this case, an attribute comprises an identifier,
its type, and the current value. This generic representation of any message while preserving its properties like nested types allows the definition of model transformations at runtime by defining appropriate visitors.

This concept was used twofold: Firstly, it was used to realize the actual mapping of a given data structure from the OpenDaVINCI environment to Google Protobuf and LCM, respectively. Therefore, the visitor for the target language was instantiated at runtime to visit a data structure to serialize the containing data into the corresponding byte representation. For this purpose, Google Protobuf as well as LCM were natively implemented in OpenDaVINCI allowing a transparent data exchange between the three environments.

Furthermore, the generic visitor approach also allows the realization of an adaptive data serialization approach. As we have pointed out in [5], the differences between two consecutively sent messages in cyber-physical systems, which are interacting with their environment based on data perceived by sensors, is rather small. Moreover, sensors for control tasks are typically sampled with higher frequencies resulting in only small delta increments between two consecutive sampling time points.

The aforementioned domain properties can be used to exchange data more effectively between interacting software agents. Therefore, the difference between the current message to be sent and its preceding message is calculated and only its difference values are communicated to avoid consuming bandwidth for redundant information that can be safely reconstructed at receiver side.

Fig. 2 depicts the delta-based deserialization process using the generic runtime message description. An application realized with OpenDaVINCI receives a new message encapsulated in a Container. The container contains the actual serialized message from the sender as payload, and meta-information as time stamp when the container left the sender, and time stamp when it arrived at the receiver. By using the container identifier, the application starts to access the serialized content in the container with the given type of the expected message.

During the first communication cycle, a complete message is exchanged between the sender and receiver and thus, the contained data, in our case for instance a VehicleData message, is deserialized. The content of that message is stored to serve as basis in the case that the sender would send a delta message only in the next communication cycle. In that case, a DeltaDeserializerVisitor is instantiated to read the difference information from the received container. Next, the corresponding previous message is restored so that DeltaReconstructor can calculate the new complete VehicleData resulting from the differences applied to this message’s predecessor.

The process described above is realized in the lower communication layer of OpenDaVINCI to encapsulate it from the user. Thus, the adaptive data marshalling is fully transparent to the user. Furthermore, this adaptation layer can also apply different delta strategies by not only considering just the previously received message as basis for reconstruction. Here, further properties of the application domain could be considered in the design of data reconstruction algorithms.

4. Evaluation

The evaluation of a given data marshalling approach typically depends on the intended application context. To circumvent this issue and properly evaluate the aforementioned marshalling approaches, we have designed the data collection step in a systematic way, as described in the remainder of this section. Further, the experimentation procedure is intended to be reproducible, i.e., the evaluation could be repeated with both the same setting or with different parameters, so other scenarios can also be considered in the future.

4.1 Experimentation Procedure

Instead of choosing a specific scenario in which the different approaches are evaluated, we decided to use the generic message description feature to dynamically create different message types at runtime. Thereby, we could systematically vary a message’s parameters influencing the performance of the respective approaches.

Due to the nature of the exchanged messages, the integer values had generally orders of magnitude of at most $10^5$, while the floating point values were usually in the $10^0$ magnitude and increasing or decreasing by the centesimal digit. For these reasons, in the study we have selected the following parameters:

- Varying the number of integral data fields (16 bit integer) in a message from 0 to 10;
- Varying the number of double data fields in a message from 0 to 10;
- Increasing the value for the integral data fields from an order of magnitude from $10^0$ to $10^5$;

Figure 3. Structure of the evaluation message, which is dynamically composed runtime.
Figure 2. Sequence chart of the visitor-based delta deserialization.

- Increasing the value for the double data fields from 1.2 to 1.3 using the steps 0 (no increment), 0.01, 0.02, 0.05, and 0.1.

The steps in the floating point numbers were defined based on our domain experience focusing on high frequency data exchange. In total, 100 different message attribute fields times four different order of magnitudes for the integral types times five different double differences, resulting in 2,000 different message configurations, were evaluated with all three data marshalling approaches. The overall structure of the evaluation message generated at runtime is depicted in Fig. 3.

4.2 Results

After systematically iterating all parameters using the generic message representation, we obtained the following results. The smallest message could be created with Google Protobuf consuming 6 bytes only for just one integral data field with values up to an order of magnitude of $10^2$. The largest message created with Protobuf occupied 139 bytes for 10 integral data fields for data up to an order of magnitude of $10^3$ and 10 double fields.

The smallest message with LCM consumed 21 bytes for 1 integral data field up to an order of magnitude of $10^3$; the largest message with LCM covering 10 fields of both types consumes 119 bytes and thus, approximately 15% less than Protobuf. The reason therefor is that the internal structure of Protobuf uses a key field while LCM simply writes the data in sequence without further control data.

The smallest delta message with 22 bytes - and thus, approximately 3.5 times larger than the smallest Protobuf message - was obtained for a message carrying 1 integral field only. The largest delta message with 156 bytes was created for a message having all 10 integral data fields and all 10 double data fields.

A chart depicting the increasing amount of bytes required to store a message of the respective type is depicted in Fig. 4. Defining a message with just 4 data fields of type double results in a serialized message of the same length of 44 bytes for Google Protobuf and the self-adaptive data marshalling. Having already 2 additional data fields of an integral type resulted in the self-adaptive approach to fall behind LCM and Google Protobuf. And finally, having 7 fields of an integral
Figure 4. Increasing amount of bytes to store a message with 4 double data fields and increasing amount of integral data types: Google Protobuf and the self-adaptive delta approach are identical with 44 bytes for just 4 double fields. Having already 2 additional integral type fields, LCM and Protobuf consume less than the delta approach, and with 7 integral data types, LCM outperforms all other approaches due to its design.

Fig. 5 summarizes the result for all message configurations where the integral data field can hold values up to an order of magnitude of $10^3$. The circles represent configurations where Google Protobuf is the best choice in terms of the shortest resulting serialization byte sequence; squares show configurations, where LCM would trump over the other approaches; and the stars show message configurations where the self-adaptive data serialization would be the best choice.

4.3 Discussion

In order to obtain a systematic evaluation of the examined marshalling approaches, auto-generated messages were used, and they were populated with a varying number of integer and floating point variables, containing values with fixed ranges and increments. From the results of this evaluation, as shown in the charts, it is apparent that in case of messages with only integral types, Google Protobuf is the best choice; however, the more data fields a message contains, the more compact data format of LCM is paying off resulting in a growing number of squares in Fig. 5. By design, the self-adaptive delta approach is paying off for messages with a higher number of non-integral data fields as its concept aims for reducing the amount of data by assuming that the difference between two consecutively sent messages with only a short time gap in between is rather small.

Another observation from Fig. 5 is that once a specific design decision is made regarding the use of a specific serialization approach, the message design during system development should obey the identified boundaries for data fields to avoid unwanted side-effects at runtime affecting the performance of the data marshalling.

4.4 Threats to validity

In this section, we discuss the validity of this study according to four perspectives [8].

Considering construct validity, the design of the study by using a generic message representation that can be systematically defined allows for a scenario-independent analysis of the performance of the three approaches. Thus, no specific application domain is favored during the experimentation design.
Regarding internal validity, four different aspects influencing a serialized message’s length were identified and systematically varied. All identified factors directly contribute to the message size; however, we did not study a random ordering of the attribute fields.

Concerning external validity, the comparison of our self-adaptive data marshalling approach with Google Protobuf and LCM can be considered as relevant as both other approaches are widely used and have shown the applicability in the domain of cyber-physical systems. Thus, the findings presented in this study have an impact on the design of such systems.

Regarding reliability, the range for the floats used in the evaluation was inspired by applications in the self-driving vehicles domain. As the goal for the delta approach was to address high frequency data exchanges, the motivation for the increment values was due to the small numeric difference between values in consecutive packets.

5. Conclusion and Future Work

Distributed software systems with interacting agents base on communication protocols to exchange information to act properly or to synchronize tasks. At a system’s development time, domain specific languages assist the developers to quickly specify messages to be exchanged between the interacting system entities. While such DSLs facilitates faster system development and modularization, our study shows that the composition of a message at design time can negatively influence the performance of a distributed software system.

One way to systematically evaluate different marshalling approaches is to have the components of the distributed system to automatically generate a number of messages that will be exchanged using one of the marshalling techniques that are analyzed. In this way, the total number of bytes exchanged in the system will act as one of the main parameters to compare the different marshalling approaches and their efficiency. The messages were generated containing an incrementing number of integer and floating point variables, and their values were incremented by predefined steps.

We experimentally showed that Google Protobuf is well suited for compact messages with few data fields focusing primarily on integral types. LCM in contrast is paying off as the number of message fields increases. Our self-adaptive data marshalling approach, which is making use of the practical fact that the difference between two consecutively sent messages is rather small, is beneficial in the case of messages that are heavy on non-integral data fields.

Future work would include extending the comparison to more marshalling approaches in order to get an even broader view of their performances and how these are affected by the nature of the processed messages in terms of number and type of contained variables. More efforts would also be required to further improve the self-adaptive data marshalling approach, especially for messages where the floating point types are not the predominant ones, since the result of the evaluation clearly shows the boundaries in which this technique has to become more efficient. Studying the way other approaches successfully process messages with integer variables exchanging smaller amounts of bytes will prove beneficial in this sense. Furthermore, alternative algorithms to model the hysteresis of previously received messages considering more messages than only the previously received one need to be explored to make better use of the delta marshalling approach in different application contexts.

References


