Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt

Justin Cooper, Alfonso de la Vega,
Richard Paige, Dimitris Kolovos
Department of Computer Science
University of York, York, UK
{justin.cooper, alfonso.delavega,
richard.paige, dimitris.kolovos}@york.ac.uk

Michael Bennett, Caroline Brown,
Beatriz Sanchez Piña, Horacio Hoyos Rodriguez
Rolls-Royce, Birmingham/Derby, UK
{mike.bennett, caroline.brown2,
beatriz.angelica.sanchez.pina, horacio.hoyos}
@rolls-royce.com

***Abstract*—Rolls-Royce Control Systems supplies engine control and monitoring systems for aviation applications, and is required to design, certify, and deliver these to the highest level of safety assurance. To allow Rolls-Royce to develop safe and robust systems, which continue to increase in complexity, model-based techniques are now a critical part of the software development process. In this paper, we discuss the experiences, challenges and lessons learnt when developing a bespoke domain-specific modelling workbench based on open-source modelling technologies including the Eclipse Modelling Framework (EMF), Xtext, Sirius and Epsilon. This modelling workbench will be used to architect and integrate the software for all future Rolls-Royce engine control and monitoring systems.**

***Index Terms*—Domain Specific Language, Component Oriented Architecture, Graphical Modelling Workbench, Xtext, Sirius, EMF**

I. Introduction

CaMCOA (Controls and Monitoring Component Oriented Architecture) is a new software architecture designed to support future generations of Rolls-Royce's Controls and Monitoring systems.

Component-Oriented Architectures (COAs) allow functionality to be encapsulated within a component that has a clearly-defined interface and conforms to a prescribed behaviour (e.g., scheduling, communication) common to all components within the architecture. This highly standardised approach is intended to improve productivity and quality. The resultant system is intended to be highly cohesive with well-defined system behaviour, yet be loosely coupled, allowing components to be upgraded, replaced or moved.

Examples of the use of COAs in related industries include:

- AUTOSAR [1] – The AUTomotive Open Software ARchitecture is a multi-partner standard intended to provide a standardised platform (supporting specification of basic software, middleware for engine control unit information interchange, and application software) for automotive software.
- ECOA [2] – the European Component Oriented Architecture is intended to create a market for standard defence mission system software components.
- SAVOIR [3] - the Space AVionics Open Interface aRchitecture has similar goals to ECOA but is targeted at space applications. This has resulted in the definition of a standardised On-Board Software Reference Architecture (OSRA) with supporting toolset.

The work in this paper has been partially funded through an InnovateUK co-funded Knowledge Transfer Partnership between the University of York and Rolls-Royce plc (contract no. KTP011043), the HICLASS InnovateUK project (contract no.113213) and the Engineering and Physical Sciences Research Council (EPSRC) under Grant No.: EP/R512230/1

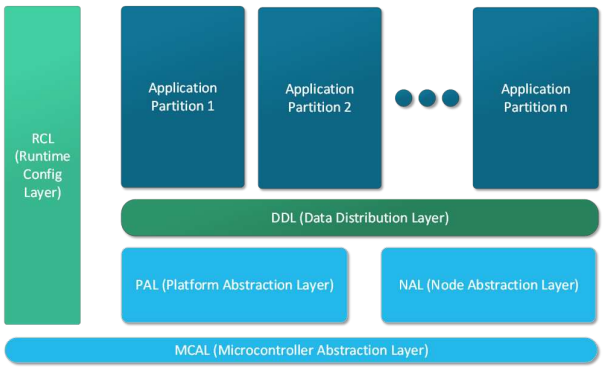

Fig. 1. CaMCOA Architecture

One of the main motivational points for CaMCOA is support for Model-Based Application Development. Application development for Controls and Monitoring systems is increasingly undertaken using model-based approaches. Tools such as Simulink [4] allow algorithms to be designed and simulated graphically and then the implementation to be produced automatically using code generation technologies. These environments are increasingly supporting qualifiable toolchains that provide evidence to support the required certification objectives, according to relevant standards such as DO-178C, automatically.

The CaMCOA architecture, whose main components are shown in Figure 1, is defined as follows:

- Application Partitions provide a container to execute Application Services. Application Services can be implemented in multiple programming languages, including Simulink Models.

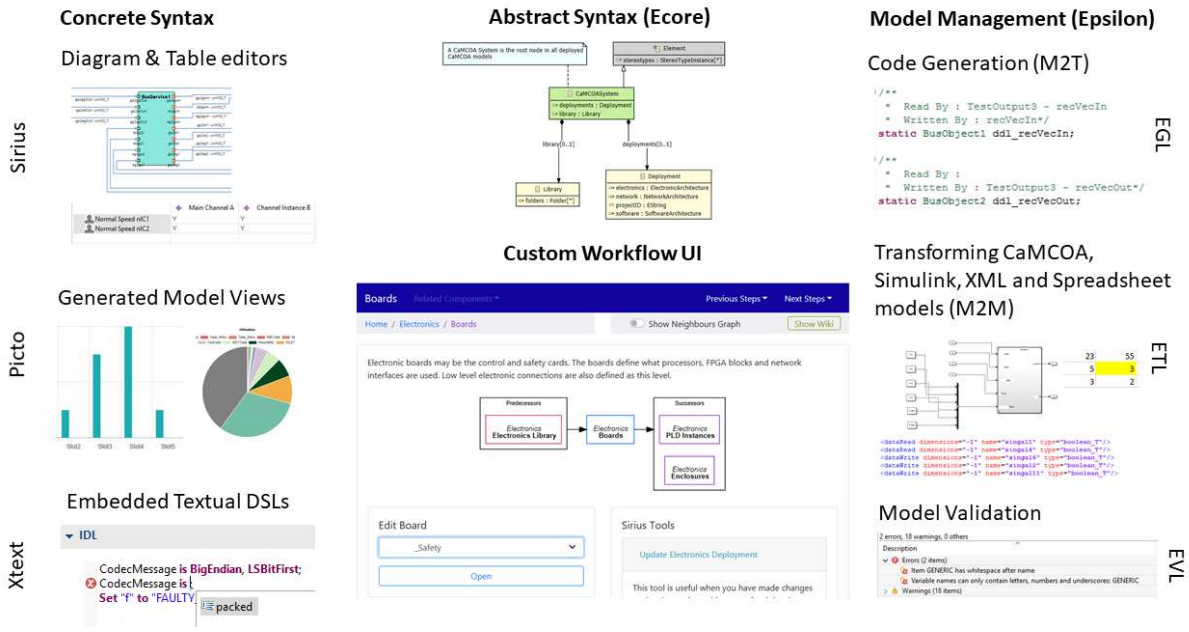

Fig. 2. Components of CaMCOA Studio

- The Runtime Configuration Layer (RCL) contains all the specific configuration data for activities such as scheduling, initialisation, and network messages.
- The Platform Abstraction Layer (PAL) contains all the OS services that are required for the system (e.g. a Real-Time Operating System (RTOS), Debug and Network).
- The Node Abstraction Layer (NAL) contains hardware specific services (e.g. device drivers) which are defined in Simulink and auto-generated to C code.
- The Microcontroller Abstraction Layer (MCAL) abstracts low-level microcontroller functionality from the higher layers.

In this paper, we present the experiences of creating a domain-specific modelling workbench, CaMCOA Studio, whose main components are shown in Figure 2. CaMCOA Studio is being used on all new engine control and monitoring projects at Rolls-Royce to deploy instances of the CaMCOA architecture.

CaMCOA Studio has been under development since 2017, and from 2018 its development has been partially supported by a Knowledge Transfer Partnership (KTP) between Rolls-Royce and the University of York. Motivation for building a new domain-specific workbench came from many sources:

- The desire to support a bespoke DSML acceptable to Rolls-Royce engineers, instead of a general-purpose modelling language. The use of a DSML is expected to have substantial adoption benefits, e.g., easier validation [5];
- The need to support integration with MATLAB/Simulink;
- The need to support traceability between CaMCOA models and MATLAB Simulink models, including specification, querying and maintenance of tracebility links, e.g., to enable impact analysis;
- The desire to support model management (e.g., validation, transformation, comparison, querying), including both known model management scenarios, and future innovative scenarios;
- The desire to exploit advanced open-source technologies while reducing the chance of vendor lock-in;
- The necessity to eventually support qualification of the toolchain, e.g., against DO-330 [6].

The selection of the individual technologies atop which CaMCOA Studio has been built has been broadly driven by the combined team's expertise and prior knowledge, and by common open-source software health indicators such as the size and activity of the community around each technology, its development and maintenance activity, and the availability of up-to-date documentation. With this in mind, it is entirely possible that other technologies (e.g. Graphiti [7] instead of Sirius [8] for graphical model editing, or Acceleo [9] instead of EGL [10] for model-to-text transformation) would have been as – or even more – effective for the respective tasks. On a similar note, we make no direct or indirect claims on the relative fitness of the Eclipse Modelling ecosystem compared to alternatives such as JetBrains MPS [11] or MetaEdit+ [12], which we have not explored for CaMCOA Studio.

The paper is structured as follows. Section II discusses the approach and challenges that we encountered when defining the abstract syntax of the core domain-specific language of CaMCOA Studio using the Eclipse Modelling Framework and Ecore. Section III discusses the concrete syntax of the lan-

guage, noting our experiences with tools such as Sirius, Picto and Xtext. Section IV introduces our approach to automated model management using the Epsilon family of languages to validate and migrate models and to generate code, Simulink models, Excel spreadsheets and XML documents. Section V discusses the tooling facilitating collaborative model development. Section VI discusses how we are testing CaMCOA Studio. Section VII introduces and motivates a custom workflow UI tooling that can guide engineers through the supported modelling and model management activities. Section VIII discusses the feedback received from Rolls-Royce engineers. Section IX details the open challenges we face with CaMCOA Studio. Section X summarises related work, and Section XI provides an overall conclusion of our experiences.

II. Abstract syntax

Defining the abstract syntax of the CaMCOA DSL[1] was a challenging task, as there is no easy metric or method to determine whether the abstract syntax is correct. During the initial development of the abstract syntax, we needed several attempts as we attempted to create the metamodel in a "big-bang" style approach (i.e. creating the complete metamodel in absence of creating the concrete syntax or any model transformations). We then found when subsequently developing the concrete syntax or model transformations, the metamodel was not fit for purpose. After several iterations of the metamodel, we established the following guidance.

Firstly, it is crucial that the modelling team work closely with the domain experts in "pair programming" style sessions. When we did not follow this approach, we found that the domain expert attempted to model in Ecore directly, causing integration issues where the domain expert did not understand the current metamodel structure, or the metamodeling team were modelling the domain incorrectly.

Secondly, we found it important to model the domain incrementally, ensuring there was a continuous thread from abstract syntax to concrete syntax to generating artefacts. This ensured we had continuous feedback that the abstract syntax could be represented appropriately in the concrete syntax and that the abstract syntax captured all the necessary detail to be able to generate artefacts (e.g. code and models) from the language.

Thirdly, we found it useful to have a simple extension mechanism in the DSL. We decided to follow the pattern of lightweight UML 1.x-style stereotypes, allowing new parts of the domain to be modelled without changing the underlying metamodel. If required, these features could then be promoted to a first-class citizen in later versions of the abstract syntax.

Finally, we decided to split the domain into separate metamodels where appropriate to improve the maintainability of the DSL. For example, rather than modelling different parts of the architecture (such as electronics and software) in the same metamodel, we split the metamodel into smaller metamodels relating to specific viewpoints of the DSL.

[1]This section does not include or describe the complete abstract syntax of the CaMCOA DSL due to it being the intellectual property of Rolls-Royce.

When defining the abstract syntax of the CaMCOA DSL, we used the Ecore language, part of the Eclipse Modelling Framework (EMF) [13]. In general, we found that Ecore allowed us to express the domain accurately, however we encountered challenges when attempting to instantiate and reuse modelled components.

In initial modelling attempts, we created a shared library model containing re-usable components (such as processors for example). A processor may have attributes which will remain the same for all its instances (such as a name and endianness), but also attributes that differ per instance (such as serial number and clock speed). When an engineer wanted to use an instance of this processor, they would copy the processor into the relevant part of the model and change serial number attribute as desired. This "clone-and-own" approach had multiple issues. For example, if there was a mistake on the clock speed attribute the engineer would have to find all instances of this processor and update the clock speed in them manually or by writing a script. Ideally, we wanted all changes to be propagated to all instances, without the need to synchronise them manually.

One approach we considered was to define two classes in the metamodel. One "Processor" class and a "ProcessorInstance" class. In this approach, static attributes of processors (such as the name) can remain attributes of a "Processor", however attributes specific to the instance (such as serial number and clock speed) are now attributes of the ProcessorInstance. Finally, a "ProcessorInstance" has a non-containment reference allowing the instance to reference the processor (shown in Figure 3).

This pattern worked in simple cases of instantiation; however, we still had a problem when trying to instantiate elements inside of other instances. Consider the slightly more complex example where we needed to model and create instances of electronic boards. A board contains a processor instance, and for each instance of the board, the processor must have a unique serial number. In this case, we need to introduce a "ProcessorConfiguration" class which captures the serial number as an attribute and a (non-containment) reference to the "ProcessorInstance" class (shown in Figure 4). Although this approach provided the required functionality, the verbosity negatively impacts the readability of the metamodel. A multi-level modelling approach could have helped reduce this accidental complexity in principle, however none of the multi-level modelling frameworks we are aware of (e.g. Melanee [14], MetaDepth [15]) are compatible with mainstream frameworks for graphical modelling, model comparison/merging and transformation in the Eclipse modelling ecosystem.

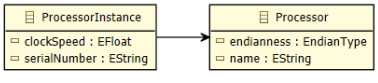

Fig. 3. Processor with Processor Instance

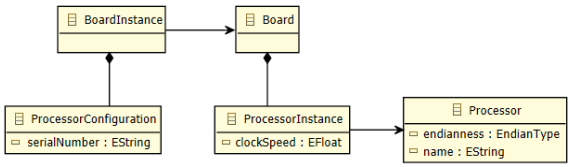

Fig. 4. Processor with Instance and Configuration Classes

Another issue we encountered was preserving and reviewing the hand-written Java implementations of Ecore operations and derived attributes. Ecore allows for derived attributes and operations to be implemented in Java, but by default, the user is required to modify the generated Java implementation with handwritten code using "@Generated Not" annotations. Although this does preserve the hand-written code, it can be very difficult to review and maintain this handwritten code when surrounded by generated code. We decided to split our handwritten and generated code into separate source folders (src and src-gen) and then use the "factory_override" extension point provided by EMF. This meant that, when reviewing code changes, the generated code directories did not need to be reviewed in detail. Alternatively, OclInEcore [16] and Xcore [17] provide support for specifying the body of derived attributes and operations within the metamodel itself.

We initially used Ecore Tools [18] to define the metamodel using a graphical class diagram-like notation as many domain experts were already familiar with UML. As the DSL grew, we found it increasingly difficult to use Ecore Tools to explore and edit the metamodel. For example, Ecore Tools provides a layer to show related EClasses, but activating this layer on an existing diagram which already contains many elements can result in very complex diagrams showing all the related elements of all classes in a diagram. Also, the layout tooling did not produce readable layouts reliably, meaning time was spent having to manually layout diagrams.

As such, we eventually decided to use Emfatic [19] (i.e. a textual syntax for Ecore) and then generate graphical views using Picto [20], an Eclipse view for visualising models via model-to-text transformations. This approach meant that we could benefit from the advantages of textual modelling (fewer clicks when defining model elements, easier to review in version control tooling), but also easily communicate with stakeholders as they were more familiar with a graphical syntax. An additional feature of Picto meant that we could visualise constraints when reviewing the metamodel. This meant that when combined with the Epsilon Validation Language (EVL) we could verify that appropriate constraints had been defined without polluting the metamodel validation logic.

III. Concrete Syntax

When creating the concrete syntax for CaMCOA, it was critical to consider the engineers using the DSL, as this would be their primary way they interact with the model. It is generally good practice to follow a user-centered design process before creating any parts of the concrete syntax and avoid rushing into the implementation. Our design process is as follows:

- Persona - Define a typical user, noting their background (e.g. familiar with specific tools such as Simulink) and goals (e.g. model a processor)
- Scenario - Detail a situation of how the user will interact with the model editor to achieve their goals
- Prototype - Construct a low fidelity sketch of the concrete syntax
- Review - Can the persona use the prototype in the scenario to achieve their goals? Does the prototype consider the persona's background?
- Iterate - iterate over each of the previous stages, adding extra details and refining if necessary
- Implement - implement the final prototype

A. Graphical syntax

The majority of engineers in the company have strong skills in Simulink and Microsoft Excel, therefore an early decision was made for the concrete syntax to be based on node-edge diagrams and tables. We briefly evaluated Sirius, EuGENIA [21] and Graphiti [7] to build our graphical syntax. Sirius was preferred over the other two options due to the native support it provides for both node-edge style diagrams and tables, due to being interpreted (allowing rapid prototyping without having to start run-time instances of Eclipse) and under active development.

Sirius allows for a graphical concrete syntax (diagrams, tables and trees) to be defined by a viewpoint specification model (VSM), building atop lower-level Eclipse graphical frameworks such as GMF Runtime and GEF [22]. This VSM is typically modified via a tree-based editor, allowing the user to build the graphical concrete syntax without having to write code. However, for more complex diagrams, it is necessary to write model queries in the provided Acceleo Query Language (AQL) [9] and Sirius also provides the ability to write custom logic in Java where appropriate.

Sirius also provides many user-oriented features. For example, dialogs, filters, layers and wizards can be created to help the user construct and navigate around the model. To support our instancing solution (presented in II), Sirius provides support for building custom Properties views via the Eclipse Extended Editing Framework (EEF) [23]. All of these features significantly improved the usability of the tool for the end user.

Although Sirius is interpreted, we did not notice any major performance issues when rendering diagrams and tables, although typically our largest single diagram contains 300 elements and models have fewer than 5000 model elements. The diagram containing 300 elements opens in less that 10 seconds on a mainstream laptop, there is no noticeable lag whilst browsing the model and running auto-layout (ELK layered algorithm) typically takes less than 10 seconds.

One of the challenges we found when using Sirius was that it is tedious to create certain aspects of the viewpoint

specification model, such as dialog boxes, edge re-connection tools and palette tools. In order to reduce this effort, we created model-to-model transformations which automatically generated parts of the viewpoint specification model from an annotated Ecore metamodel.

We have faced challenges dealing with changes to the abstract syntax (model migration). All information relating to the concrete syntax of a Sirius model is stored in a representations (.aird) model. This model contains details such as the layout of diagrams, what layers and filters are active on diagrams and what representations (tables/trees/diagrams) exist in the model. This representations model references semantic model elements, however as the underlying abstract syntax of the language evolves, semantic model elements may not exist or might have changed in later versions of the language. To the best of our knowledge, there is no current best practice when dealing with changes to the abstract and concrete syntax, and we have been relying on bespoke automated migration scripts, as described in Section IV-D, to migrate and repair broken references, or even have to ask engineers to delete and re-create the diagrams where possible.

Another challenge relating to diagram editing is layouting. The default strategies for diagram layouting in Sirius provided unsatisfactory results, and in most instances caused a worse layout than the one created by the users themselves. In recent versions of Sirius, however, there is experimental support for including layouting from the ELK [24] (Eclipse Layout Kernal) project. This has provided us with significant improvements in the layouting of diagrams.

We have also found it challenging when creating a strategy to best manage the .aird representations model. By default, all representations are stored in a single model at the root of the project. This can however cause conflicts when multiple engineers are working collaboratively on the model. Sirius does provide tooling to fragment and extract parts of the main representations.aird to separate aird files, reducing the risk of creating merge conflicts, however we then need additional logic to manage the model fragments (for example, if the semantic model element representing the root of a diagram had been deleted, the diagram should be deleted too).

In some Sirius diagrams, it was necessary to provide additional read-only views to complement the information provided by the diagrams. An example of this is when a user was scheduling the system using CaMCOA Studio. When an engineer is scheduling a system, they typically need to analyse different parts of the system to see details of what tasks are consuming the most resources, how much slack there is available on the system etc. To render these read-only views, we again relied on Picto [20] to create visualisations via model-to-text transformations. Engineers would still edit the model via Sirius diagrams and these extra views are rendered alongside the editor.

Picto has been used with ChartJS [25] to render charts and PlantUML [26] to render timing diagrams, and provides many useful features such as navigation between elements in the rendered view and the Sirius-based editor, support for layers and lazy evaluation, meaning only the details needed to be shown are computed.

B. Textual syntax

In some cases, a tree, table, or diagram does not provide the most intuitive concrete syntax for editing the model, and a textual syntax allows a user to more efficiently create and modify model elements. One requirement within the CaMCOA DSL was to be able to precisely define the format of messages being sent and received on the internal and external data buses, and to associate specific semantic attributes with elements of the message so they can be processed on transmission/reception. These messages require an Interface Description Languages (IDL) that declares how the message should be formatted and treated. IDLs are used to precisely define the required format of interfaces between computing systems, particularly where processor architecture and language differences could lead to a mismatch in the representation of the expected data. There are several standardised IDLs, for example ASN.1 [27] and the OMG IDL [28].

In a white paper [29] Obeo and Typefox have shown how it is possible to integrate Xtext [30] based textual DSLs into the Sirius properties view. This approach stores textual DSL snippets in textual attributes of the DSL. Sirius then provides the ability to embed custom widgets into the properties view and Xtext provides an embedded editor widget as part of the framework. The implementation in the white paper showed a basic implementation of the glue-code required to embed an editor providing content-assist, syntax highlighting, code completion, references between the Xtext and Sirius models and navigation. We did encounter several challenges when trying to embed a textual DSL into the Sirius properties view which are detailed in [31].

IV. Model management

CaMCOA Studio needs to generate many different artefacts from a CaMCOA model to create and test software builds, including source code, Simulink models, tests and XML documents. Although the generation of these artefacts can be performed in general-purpose programming languages such as Java, we decided to use the Epsilon family of tools [32] for model management tasks. The architecture of Epsilon contains a model connectivity layer that abstracts any of the specific low-level modelling technology details such as querying the CaMCOA model (EMF model), parsing XML or modifying a Simulink model, allowing access to the models in a uniform way. Epsilon then provides a set of languages built atop the Epsilon Object Language (EOL) [33] to write and maintain model management operations following a declarative style. This means time is not wasted having to write, maintain and review boilerplate Java code for any of our model transformations. Epsilon programs can also be called "headlessly" (i.e. invoked by command-line interfaces). This allows the model management programs to be executed outside of an Eclipse environment, ensuring that any issues with the models

or model management tasks can be identified as part of a continuous integration build.

One of the powerful features of Epsilon is its dynamic typing support. However, Epsilon's Eclipse-based editors do not provide context-aware code completion or static type-checking facilities, meaning some errors are not always identified until the script has been executed. One way to improve the execution time error reporting is to make sure types are assigned to variables (unless it is intended to hold values of different types) and to operations, parameters and return types. We have also found it very useful when developing Epsilon programs to use the Epsilon interpreter[2] to check queries whilst writing transformations. Our current model management solution is split across 73 model-to-text transformation (EGL) templates, 28 model-to-model transformation (ETL) scripts and approximately 150 validation (EVL) constraints.

A. Model-to-text transformation

CaMCOA Studio is required to generate source and build files. All model-to-text transformations are implemented using the template-based Epsilon Generation Language (EGL) [10]. EGL supports many powerful features such as traceability from the source model to the generated code. EGL transformations can be orchestrated via a dedicated rule-based sub-language (EGX). EGX ensures that all the logic for running the EGL templates is captured in a declarative way, without having to maintain boilerplate code for coordinating model-to-text transformations. Although EGL supports features such as protected regions (i.e. allowing handwritten code and generated code to co-exist in the same file), this has not been necessary in CaMCOA.

The main challenges we have encountered whilst writing EGL templates is ensuing the generated code conforms to the Rolls-Royce coding standard. In an attempt to automate this process, we run both static analysis and code formatting compliance tooling during our continuous integration builds. This means that any errors on the templates can be identified automatically.

B. Model-to-model transformation

1) CaMCOA-to-Simulink transformations (EMF to Simulink): As discussed in Section I, support for model-based application development in tools such as Simulink is a major motivational factor for CaMCOA Studio. This is because the logic for any application or platform services can be defined in the Simulink environment where an engineer can easily simulate and test behaviour, auto-generate the implementation and then provide evidence to support certification objectives.

The CaMCOA model captures deployment-specific details relating to the Node Abstraction Layer (NAL) in the CaMCOA Architecture. With these details, Simulink models can be instantiated for the deployment of an engine project and then the Simulink code generator can generate all the code related to the behavioural part of the engine deployment. Note that CaMCOA Studio does not generate complete behavioural models for the Node Abstraction Layer; it creates Simulink models that instantiate Simulink library components created internally within Rolls-Royce. CaMCOA allows for the deployment-specific data to be captured and when combined with the library models, a complete implementation for the NAL layer in the CaMCOA architecture can be generated.

CaMCOA Studio uses the Epsilon Transformation Language (ETL), combined with Epsilon's EMF and Simulink Epsilon drivers [34] to perform the EMF to Simulink transformations. In general this has been a successful approach to generating Simulink models from CaMCOA Studio. One challenge we have encountered is performance. The Simulink models generated from CaMCOA Studio are typically in the region of a few hundred model elements and take approximately 5 minutes to generate. After initial investigations, we found that each command sent via the official MATLAB Java engine takes approximately 10ms. When using it through Epsilon to generate a Simulink model from a CaMCOA model with approximately 300 elements, over 40,000 commands need to be sent to the engine to create, configure and link the elements.

2) Simulink-to-CaMCOA transformations (Simulink to EMF): CaMCOA needs to be aware of the interfaces of the application services (defined in the Application Partitions in the CaMCOA architecture). The behaviour of the services is defined in Simulink models and then the implementation is generated via the MATLAB code generator. For creating an instance of the CaMCOA architecture, CaMCOA Studio must import the relevant data from the application service models into the CaMCOA model.

Although the Epsilon Simulink driver can query the Simulink model directly, to improve performance and to allow for other tools to consume the application service interfaces, a model-to-model transformation happens in Simulink to transform the Simulink model to an intermediate XML model (this is out-of-scope of CaMCOA Studio and therefore not performed using Epsilon). The intermediate XML model is then is imported into CaMCOA Studio using the Epsilon XML driver [35].

One issue we have faced during this approach is keeping the application services (Simulink models) and CaMCOA models synchronised. For example, if the interface of the application service component changes, this change needs to be updated in the CaMCOA model. To detect such changes, we have utilised the Epsilon Comparison Language (ECL) to identify mismatches between the application services and the CaMCOA model, and the Epsilon Validation Language (EVL) to suggest quick fixes to perform the synchronisation. Bidirectional transformation languages such as JTL [36] or eMoflon::IBeX [37] may have been better suited to this task, however the constraints of the project did not allow for a comprehensive evaluation of all available options and direct model transformations may suffer due to the performance of the MATLAB Java engine.

[2]https://www.eclipse.org/epsilon/doc/articles/eol-interpreter-view/

3) CaMCOA-to-XML and CaMCOA-to-Excel transformations: In order to test software deployments, internal Rolls-Royce testing tools need setting up with engine specific data. The internal testing tools consume XML models and Microsoft Excel spreadsheets to configure the test environment. By using the Epsilon XML driver [35], the Epsilon Excel driver [38] and ETL, the test model can be generated automatically from the data in the model. Before CaMCOA Studio this was a manual process where an engineer would have to construct the models or spreadsheets by hand from Microsoft Excel and Word documents.

C. Model Validation

Model validation is an essential part of CaMCOA Studio to alert engineers of potential problems in the model. The Epsilon Validation Language (EVL) [39] provides a declarative syntax for defining constraints on model elements. Violations of these constraints can be presented to the user as either a warning or an error depending on the severity and the ability to write "quick-fixes" to help users solve any issues in their model. EVL provides extension points allowing it to integrate with many different frameworks such as EMF, Xtext and Sirius editors without having to write any additional code other than the constraints themselves.

To ensure any regressions or errors can be identified in the model as early as possible, we have found it very useful to run model validation scripts as part of any continuous integration builds involving the CaMCOA model. Another useful feature of Epsilon is that it provides good profiling tooling to identify constraints which were taking a long time to execute. Before running any profiling, model validation was taking approximately 1 minute with a model size of approximately 2,000 elements and 150 EVL constraints. However, by using the profiling we reduced this down to approximately 15 seconds.

D. Model migration

As the abstract and concrete syntax of the DSL are continuously evolving as the tooling becomes more stable, it has been important to migrate older models conforming to earlier versions of the DSL. In order to do this, we have needed to use model migration frameworks.

To define and perform model migrations, we investigated two frameworks: Edapt [40] and Epsilon's Flock [41]. Edapt is a migration framework for Ecore based models that can automatically generate any migrations by tracking the changes to the abstract syntax. This is very powerful as it means a user does not need to manually define migration rules for simple changes. One of the main disadvantages of this approach, however, is all metamodel edits must occur in the Ecore tree editor. As described in Section II, we use Emfatic as an editor for our Ecore based metamodel, therefore this would require us to prototype the changes in Emfatic and then re-implement the changes based in the Ecore tree-based editor for Edapt to capture all the of the changes.

Flock on the other hand allows for migration scripts to be written in a similar fashion to ETL (model-to-model) transformations. An advantage of using Flock rather than a plain model-to-model transformation is that Flock will automatically copy elements unaffected by the transformation, avoiding unnecessary logic to be defined by the user compared to writing the equivalent ETL transformation. Flock requires the source and target Ecore metamodels to be present for the transformation to work correctly. This means that the metamodel needs to be versioned correctly, for example by using the namespace URI attribute in Ecore to capture the version. All versions of the metamodel must then be shipped with the tool. Edapt, differs slightly as it captures all revisions in a single "history" model with releases being defined after certain set of changes have been made.

E. Epsilon in CaMCOA Studio

At this point it is worth mentioning that Epsilon was the technology of choice for automating model management tasks such as model validation and transformation in CaMCOA Studio before the involvement of the authors from the University of York (who are also contributors to Epsilon) in the project. However, we also recognise that the adoption of further Epsilon-based technologies such as Flock for model migration and Picto for model visualisation (discussed in Sections II and IV-D) may have been influenced substantially by the availability of expertise and direct support from the York collaborators.

V. Collaborative model development

A. Comparison and merging

Defining a CaMCOA deployment takes expertise spread across multiple teams, and different users are required to work on different parts of the model in parallel. At Rolls-Royce, all artefacts are stored as files using the Git version control system, including CaMCOA models and Sirius representation models. Without good support for comparison and merging, it can be time consuming and error prone to review model changes, identify merge conflicts and automatically merge non-conflicting changes. Both the CaMCOA model and the Sirius representation models are stored using the XMI [42] format. This means if a user were to use traditional comparison and merging tools such as WinMerge [43], users would see many low-level details of the model which are not easily understandable to humans (such as XMI IDs). Two frameworks which allow for model comparison and merging to be carried out at the model level are EMF Compare [44] and EMF Diff/Merge [45].

1) EMF Compare: By default, EMF Compare allows performing model comparison and merging via a tree interface. EMF Compare has good performance, allowing a three-way merge across a model with over 2k elements being loaded in less than 10 seconds. Being able to compare and merge models as trees is very powerful, however it is not the syntax that the users are typically editing the model in. For example, in CaMCOA, users are typically editing the model within Sirius diagrams rather than a tree editor.

EMF Compare has support for comparing models as diagrams and provides integration with Sirius. However, users can become overwhelmed with the low-level details of changes to the Sirius representations model. For example, some of the diagrams in CaMCOA Studio typically contain more than 100 elements. If two users had run the auto-layout feature on the same diagram on different branches, there may be 100 differences or conflicts in the x,y positioning of the diagram elements. This means users typically prefer to merge the semantic model (CaMCOA model) using the tree syntax provided by EMF Compare, leading to the user having to take either "Theirs" our "Ours" during a Git merge of the Sirius representations model. This can have side-effects, such as the diagram model referring to semantic elements that no longer exist.

2) EMF Diff/Merge: EMF Diff/Merge was also reviewed and was found to provide similar functionality to EMF Compare. Comparison and merging were only supported via a tree interface, however we found EMF Diff/Merge did not properly support references to model elements stored in separate files, resulting in it being unusable for CaMCOA Studio.

3) Xtext: Serialising the model using Xtext, rather than XMI has also been considered. By storing the CaMCOA model as text, all the standard code reviewing tooling (such as WinMerge) can be used as elements, e.g., references, can be stored as a more meaningful name. We rejected this approach however, as an Xtext grammar would need to be implemented and maintained for the DSL, on top of the Ecore metamodel and Sirius viewpoint specification model. Language specific details such as custom scoping rules may need to be implemented too, to avoid multiple elements containing the same name being identified as conflicting references (as Xtext references elements by name rather than ID).

B. Model Reviewing

Any code or artefacts at Rolls-Royce are required to be reviewed on a pull-request before the changes can be merged onto the main development branch. This review typically happens within a web-browser, using the reviewing tooling provided by Microsoft Team Foundation Server (TFS) [46]. This interface allows users to attach review comments on specific parts of artefacts (e.g. locations in text files), and allow the author to respond or raise further work items (tickets) to record any required follow-up work.

As the reviewing of artefacts typically happens in a web browser, users have to annotate any model review comments to the XMI file directly (as TFS cannot render CaMCOA or Sirius models from within the browser). This means that to properly review the model, a user must check-out the branch, use EMF Compare to review the changes and then attach any review comments in the web-browser based on what was displayed in EMF Compare. This is a very tedious and error-prone task for reviewers and there have been occasions where regressions have been introduced due to the difficulty of reviewing of models from within a browser. There is a strong desire for users for better browser-based reviewing tooling; we discuss this further in Section IX.

VI. Testing and Release Engineering

In order to test CaMCOA Studio, we have used a range of testing frameworks. All of the tests run as part of a continuous integration build to ensure any regressions or errors in the tool can be identified as early as possible.

A. Testing

a) JUnit: Custom code relating the metamodel, such as the implementation of Ecore operations or derived attributes are tested using the JUnit [47] framework. JUnit supports coverage frameworks such as JaCoCo [48] to gain an insight into the quality of testing.

b) EUnit: Model management tasks such as EVL constraints and model-to-model transformations are tested using the Epsilon Unit Test framework (EUnit) [49]. EUnit provides extra support for loading model management operations and models compared to JUnit when testing model management tasks. Currently, however, there is no support for gaining coverage metrics for EUnit tests.

c) SWTBot: To ensure no regressions have occurred in the Sirius viewpoint specification model (which defines the editors for the graphical concrete syntax), we use the SWTBot [50] framework and support libraries provided by the Sirius framework.

We had initially investigated RCPTT [51] to test the Sirius editors as RCPTT provides features such as the Eclipse Command Language DSL [52] and a recording application to reduce the amount of effort when writing tests. In our experiences however, the test recorder generated lots of noise that needed to be manually removed.

Although SWTBot does have a test recorder, the test recorder does not support Sirius based editors meaning the tests must be handwritten. We did, however, successfully generate approximately 40% of our tests by writing custom model-to-text transformations from the CaMCOA metamodel and Sirius viewpoint specification model.

B. Release Engineering

The Eclipse Tycho project provides good support for building Eclipse plug-ins and tooling based on an Eclipse target platform. On each successful merge back to our main development branch, we generate an Eclipse P2 update site [53] for CaMCOA Studio. We also create Oomph [54] setup models to allow users to install CaMCOA Studio.

C. Managing External Dependencies

Eclipse follows a 13 week release cycle and many of our dependencies (Sirius, Xtext, EMF) obey this. To minimise disruption, we follow 1 release behind (e.g. in June 2020 we adopted the March 2020 release). We detect breaking changes by running our extensive suite of automated tests, and any such changes can be identified one release ahead which allows for major issues to be dealt with prior to adoption.

VII. Custom parts

A. CaMCOA Workflow UI

In early CaMCOA Studio versions, users relied heavily on the tree-based Model Explorer view provided by Sirius to navigate around the model and perform model management tasks. As the CaMCOA DSL grew in complexity, it became difficult for novice users to know how to interact with the model as it required a good understanding of the underlying metamodel structure. The main challenges we faced were: 1) a growing number of types: What elements must be defined?, 2) a growing number of relations: Which elements must be defined first?, and 3) identifying transitive relations: How can changes in a part of the model affect other parts?.

Our initial solution was to define a wizard which guided engineers through the construction of a CaMCOA model, known as the CaMOCA Workflow. This addressed the first two challenges and partially addressed the third one. 1) Each step required the user to instantiate a set of elements: As new classes were added to the CaMCOA metamodel, we could add them to a step's set or create a new step. 2) The step order implicitly captured the before-after relations and ensured that elements required at later steps had been created. 3) Modifying elements as part of a step implied that all subsequent steps should be revisited as they might have been impacted.

After deploying this solution, we discovered flaws with our approach. Firstly, a guided process works well for creating a model from scratch, however when editing a particular part of the model, the workflow did not provide any information to find the step to start that change process. Secondly, the guided process works well for engineers that are unfamiliar with CaMCOA Studio, but but can become obtrusive for experienced users that knew where to go but where forced by the workflow to follow a predefined path to get there. Thirdly, according to the user role (e.g. Electronic Engineers, System Architects, etc.), users need to work on different parts of the model, revealing multiple entry points. Finally, pre- and post-dependencies that are not in the immediate step were easy to miss. For example if a user added a new component using the wizard and skipped to the code generation step, the user may miss the step to schedule the component. Overall, this made it harder for engineers to visualise and understand the impact of a change.

After taking these issues into consideration, the workflow was redesigned as a graph, where nodes represent the steps that can be carried out to complete a CaMCOA model and edges capture the dependencies between these steps. In early development stages the relations can be used as a guide of what needs to be completed to populate a CaMCOA model from scratch, as in the old workflow process. For later development stages, the relations can be used to inform users on how steps can affect each other and which parts of the model are impacted. The graph also provides a less restrictive view, which allows users to quickly navigate to different steps of the development.

To easily maintain the workflow, we have implemented a separate workflow DSL to capture the steps of the workflow along with their activities, tools and dependencies. The steps may be linked to Sirius views (tables or diagrams) where the user can modify the CaMCOA model. We then transform the workflow model into the HTML, CSS and JavaScript for rendering in an embedded browser within Eclipse.

We have also introduced a context-sensitive documentation view which, depending on the active editor, will navigate to the correct CaMCOA Wiki page (stored in TFS) allowing the user to see all relevant documentation without having to manually navigate to it in a web browser.

VIII. CaMCOA Studio Reception

CaMCOA Studio is now being used on all new engine control and monitoring projects and has received very positive feedback from users. Engineers appreciate and are exploiting the higher level of abstraction and automation that CaMCOA Studio provides. Previously, engineers would need to work across many sources of information (including Microsoft Word and Excel documents) and manually construct engine software deployments. During the development of CaMCOA Studio, four trial events were undertaken. In each trial event, when engineers used CaMCOA Studio, it took a team of five engineers approximately one week per trial event to deploy components. Informal feedback from experienced engineers at the company has indicated that these activities would have taken several months with a much larger team.

Users have also highlighted areas for improvement. Firstly, some users have found it challenging to understand and navigate around the model. To address this, we have implemented the workflow as described in Section VII. We are continuously working with engineers to improve the workflow to help them better navigate and use CaMCOA Studio. Secondly, users find collaborative working difficult, even when using tooling such as EMF Compare. Some users have commented that they are spending a large percentage of their time trying to review changes and resolve conflicts as they find EMF Compare confusing. We are hoping to customise EMF Compare and investigate diagram comparison and merging in the future based on user feedback. Finally, as integration engineers were previously manually constructing software deployments, they were more easily able to modify and debug code. Now, as the level of abstraction has increased and code is generated, engineers have to seek advice from the CaMCOA Studio development team to help with debugging and modification of transformations. As CaMCOA Studio further matures, we expect to create distinct roles for the development and maintenance of the model transformations to avoid integration engineers attempting to debug issues themselves. To support Rolls-Royce engineers, we have adopted the Scrum framework with three week sprint cycles. As part of each sprint, our users are involved in planning and review sessions to provide feedback and requirements. We run quarterly trial events for more formal evaluation.

IX. Open challenges

In this section we summarise the main open challenges we are facing in CaMCOA Studio for which we have not found publicly available solutions that work satisfactorily out of the box.

A. Diagram comparison and merging

As discussed in Section V-A1, the only tool that provides some support for comparing and merging Sirius diagrams at present is EMF Compare. While EMF Compare can display the two versions of the diagram in a graphical form side by side, comparison and merging needs to be carried out through EMF Compare's standard tree-based representation. This representation exposes so much low-level detail to users that in practice we have found it to be unusable. As a result, engineers tend to fall back to either comparing and merging diagrams at the XMI level or accepting the incoming version of the diagram and repeating their changes manually through the Sirius diagram editor. We are hoping to implement custom filters in EMF Compare to filter irrelevant changes.

B. Model reviewing

As discussed in Section V-B, our procedures for collaborative development include a standard review process via pull-requests managed in a web browser. The main issues come from the review system presenting model changes in their serialised XMI textual format. As it is not possible to reliably visualise changes in XMI, reviewers need to perform the comparison locally by using EMF Compare. XMI also makes it very laborious to leave comments in specific parts of models. For instance, reviewers wanting to put a comment in a model element would first need to know the line in which that element was serialised into XMI. We are looking at ways to improve this review process from inside the Eclipse IDE, so that comments can be left on individual model elements, and the back-and forth process of relating those comments with the serialised XMI is done automatically. Our investigation of existing tools suggest that it could be feasible to extend Mylyn [55] to support this functionality.

C. Comparing models conforming to evolving metamodels

As the CaMCOA metamodel evolves at a fast pace, it is not uncommon for engineers to have to compare and merge models that conform to different versions of the metamodel. Existing model comparison frameworks such as EMF Compare and DiffMerge are based on the assumption that the models compared conform to the same metamodel and therefore they cannot be of assistance in such scenarios. As a result, models conforming to different versions of the metamodel end up being compared and merged in their XMI representation, which is far from ideal.

X. Related Work

Multiple experience reports on the application of model-based software development in real-world projects can be found in the literature, which usually come in one of two types.

The first type of reports consists of surveys carried out by researchers [56]–[59], where MBE industry practitioners are interviewed to identify, among other things, where and how modelling is used (or not [60]), or what are the social or technical factors limiting its adoption on each concrete context. A result often provided by these surveys is a research agenda for the academic community in the form of a list of open challenges in the field, which have evolved over time [61].

The second type includes those works where experiences, success stories and pain points are shared by the modelling practitioners themselves [62]–[64], as we do in this paper. For instance, authors of [62] presented their experience and problems when doing collaborative work over software models within teams of hundreds of developers. In [63], cultural and institutional issues resulting from the adoption of MBE are discussed. One of these issues refers to the reception of new tools and methodologies (that we discussed in Section VIII), which were easier to adapt to by junior workers than by those with decades of experience doing work in a concrete manner. In [5], experiences of building and applying a DSML-based workbench for safety critical systems engineering is described. Of particular importance is the highlighting of a number of risks associated with using DSML-based workbenches in a safety domain, and mitigations for these risks. The approach is also applied in some detail to a healthcare system.

XI. Conclusions

In this paper we have presented the experiences and lessons learnt through the implementation of CaMCOA Studio, a domain-specific model-based development workbench, used to architect and integrate engine control and monitoring systems at Rolls-Royce. We have detailed our approach to developing the abstract and concrete syntaxes of its underpinning domain-specific language, and we have discussed the model management operations, testing and release engineering aspects of the workbench, noting any best practices we have developed and challenges we have encountered.

Feedback from engineers using the tool in practice has been very positive, as CaMCOA Studio has been shown to automate several previously manual processes, allowing software deployment activities that previously took months worth of effort with larger teams, to be carried out by smaller teams in days. The main open challenges we see at the moment are support for diagram comparison and merging, model reviewing, and comparison and merging of models conforming to evolving metamodels.

Going forward, and to eliminate issues with deploying and installing Eclipse instances on user PCs and aid in collaborative working, moving CaMCOA Studio to the web (or at least providing a web-based counterpart for the Eclipse-based workbench) is very appealing to the business. In this direction, we plan to evaluate Sirius Web [65] and GLSP [66] as both frameworks offer strategies for migrating existing Eclipse modelling based tooling to web-based tooling.

REFERENCES

[1] AUTOSAR. AUTomotive Open System ARchitecture. https://www.autosar.org/.

[2] T. E. Consortium. European Component Oriented Architecture (ECOA). http://www.ecoa.technology/.

[3] European Space Agency. Space Avionics Open Interface Architecture (SAVOIR). https://savoir.estec.esa.int/.

[4] MathWorks. Simulink. https://mathworks.com/products/simulink.html.

[5] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann, "Using language workbenches and domain-specific languages for safety-critical software development," *Softw. Syst. Model.*, vol. 18, no. 4, pp. 2507–2530, 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-0679-0

[6] Radio Technical Commission for Aeronautics (RTCA), "DO-330 Software Tool Qualification Consideration," 2011.

[7] Eclipse Foundation. Eclipse Graphiti. https://www.eclipse.org/graphiti/.

[8] ——. Sirius. https://www.eclipse.org/sirius/.

[9] ——. Eclipse Acceleo. https://www.eclipse.org/acceleo/.

[10] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack, "The epsilon generation language," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, ser. Lecture Notes in Computer Science, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-3-540-69100-6_1

[11] Jetbrains. Meta Programming System (MPS). https://www.jetbrains.com/mps/.

[12] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+ A fully configurable multi-user and multi-tool CASE and CAME environment," in *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, J. A. B. Jr., J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. Sølvberg, Eds. Springer, 2013, pp. 109–129. [Online]. Available: https://doi.org/10.1007/978-3-642-36926-1_9

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2009.

[14] C. Atkinson and R. Gerbig, "Flexible deep modeling with melanee," in *Modellierung 2016 - Workshopband*, S. Betz and U. Reimer, Eds. Bonn: Gesellschaft für Informatik e.V., 2016, pp. 117–121.

[15] J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Objects, Models, Components, Patterns*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–20.

[16] Eclipse Foundation. OCLinEcore. https://wiki.eclipse.org/OCL/OCLinEcore.

[17] ——. Xcore. https://wiki.eclipse.org/Xcore.

[18] ——. Ecore Tools. https://www.eclipse.org/ecoretools/.

[19] ——. Eclipse Emfatic. https://www.eclipse.org/emfatic/.

[20] D. Kolovos, A. de la Vega, and J. Cooper, "Efficient generation of graphical model views via lazy model-to-text transformation," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 12–23. [Online]. Available: https://doi.org/10.1145/3365438.3410943

[21] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck, "Taming EMF and GMF Using Model Transformation," in *Model Driven Engineering Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6394, pp. 211–225, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16145-2_15

[22] Eclipse Foundation. Eclipse Graphical Modelling Project. https://www.eclipse.org/modeling/gmp/.

[23] ——. Eclipse EEF. https://www.eclipse.org/eef/.

[24] ——. Eclipse Layout Kernel. https://www.eclipse.org/elk/.

[25] ChartJS. ChartJS. https://www.chartjs.org/.

[26] PlantUML Team. PlantUML. https://plantuml.com.

[27] International Telecommunication Union. Abstract Syntax Notation One (ASN.1). https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx.

[28] Object Management Group. Interface Definition Language. https://www.omg.org/spec/IDL/.

[29] Obeo/TypeFox. Xtext Sirius Integration - The Main Use-Cases. https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf.

[30] Eclipse Foundation. Eclipse Xtext. https://www.eclipse.org/Xtext/.

[31] J. Cooper and D. Kolovos, "Engineering Hybrid Graphical-Textual Languages with Sirius and Xtext: Requirements and Challenges," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany: IEEE, 2019, pp. 322–325. [Online]. Available: https://ieeexplore.ieee.org/document/8904580/

[32] Eclipse Foundation. Eclipse Epsilon. https://www.eclipse.org/epsilon/.

[33] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon Object Language (EOL)," in *Model Driven Architecture – Foundations and Applications*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. Rensink, and J. Warmer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4066, pp. 128–142, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/11787044_11

[34] B. Sanchez, A. Zolotas, H. Hoyos Rodriguez, D. Kolovos, and R. Paige, "On-the-fly translation and execution of ocl-like queries on simulink models," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2019, pp. 205–215.

[35] D. Kolovos, L. M. Rose, J. Williams, N. Matragkas, and R. F. Paige, "A lightweight approach for managing xml documents with mde languages," in *Modelling Foundations and Applications*, A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 118–132.

[36] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Jtl: a bidirectional and change propagating transformation language," in *International Conference on Software Language Engineering*. Springer, 2010, pp. 183–202.

[37] E. Leblebici, A. Anjorin, and A. Schürr, "Developing emoflon with emoflon," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2014, pp. 138–145.

[38] M. Francis, D. S. Kolovos, N. Matragkas, and R. F. Paige, "Adding spreadsheets to the mde toolkit," in *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–51.

[39] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 204–218. [Online]. Available: https://doi.org/10.1007/978-3-642-11447-2_13

[40] Eclipse Foundation. Eclipse Edapt. https://www.eclipse.org/edapt/.

[41] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack, and S. Poulding, "Epsilon Flock: a model migration language," *Software & Systems Modeling*, vol. 13, no. 2, pp. 735–755, May 2014. [Online]. Available: http://link.springer.com/10.1007/s10270-012-0296-2

[42] Object Management Group. XML Metadata Interchange Specification. https://www.omg.org/spec/XMI/.

[43] WinMerge. WinMerge. https://winmerge.org/.

[44] Eclipse Foundation. EMF Compare. https://www.eclipse.org/emf/compare/.

[45] ——. EMF DiffMerge. https://www.eclipse.org/diffmerge/.

[46] Microsoft. Microsoft Team Foundation Server. https://docs.microsoft.com/en-us/azure/devops/server/tfs-is-now-azure-devops-server?view=azure-devops-2020.

[47] JUnit. JUnit. https://junit.org/.

[48] EclEmma. JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco/.

[49] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "EUnit: A Unit Testing Framework for Model Management Tasks," in *Model Driven Engineering Languages and Systems*, J. Whittle, T. Clark, and T. Kühne, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6981, pp. 395–409, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-24485-8_29

[50] Eclipse Foundation. SWTBot. http://www.eclipse.org/swtbot/.

[51] ——. RCP Testing Tool. https://www.eclipse.org/rcptt/.

[52] ——. Eclipse Command Language (ECL). https://www.eclipse.org/rcptt/documentation/userguide/ecl/.

[53] ——. Eclipse Equinox p2. https://www.eclipse.org/equinox/p2/.

[54] ——. Eclipse Oomph. https://projects.eclipse.org/projects/tools.oomph.

[55] ——. Eclipse Mylyn. http://www.eclipse.org/swtbot/.

[56] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. Waikiki, Honolulu, HI, USA: ACM Press, 2011, p. 633. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1985793.1985882

[57] M. Petre, "UML in practice," in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 722–731. [Online]. Available: http://ieeexplore.ieee.org/document/6606618/

[58] H. Burden, R. Heldal, and J. Whittle, "Comparing and contrasting model-driven engineering at three large companies," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*. Torino, Italy: ACM Press, 2014, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2652524.2652527

[59] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, vol. 19, no. 1, pp. 5–13, Jan. 2020. [Online]. Available: http://link.springer.com/10.1007/s10270-019-00773-6

[60] T. Gorschek, E. Tempero, and L. Angelis, "On the use of software design models in software development practice: An empirical investigation," *Journal of Systems and Software*, vol. 95, pp. 176–193, Sep. 2014. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121214001022

[61] R. France and B. Rumpe, "The evolution of modeling research challenges," *Software & Systems Modeling*, vol. 12, no. 2, pp. 223–225, May 2013. [Online]. Available: http://link.springer.com/10.1007/s10270-013-0346-4

[62] L. Bendix and P. Emanuelsson, "Collaborative work with Software Models - Industrial experience and requirements," in *2009 International Conference on Model-Based Systems Engineering*. Herzeliya and Haifa, Israel: IEEE, Mar. 2009, pp. 60–68. [Online]. Available: http://ieeexplore.ieee.org/document/5031721/

[63] J. Aranda, D. Damian, and A. Borici, "Transition to Model-Driven Engineering," in *Model Driven Engineering Languages and Systems*, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 692–708.

[64] A. Nordmann and P. Munk, "Lessons Learned from Model-Based Safety Assessment with SysML and Component Fault Trees," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Copenhagen Denmark: ACM, Oct. 2018, pp. 134–143. [Online]. Available: https://dl.acm.org/doi/10.1145/3239372.3239373

[65] Eclipse Foundation. Sirius Web. https://www.eclipse.org/sirius/sirius-web.html.

[66] ——. Graphical Language Server Protocol. https://www.eclipse.org/glsp/.