Adoption of model-based development for machine controller software

Johan Van Noten*, Davy Maes

Flanders' Mechatronics Technology Centre
Heverlee, Belgium
johan.vannoten@fmtc.be

Abstract— **The complexity of mechatronic systems in general and their control software in particular keeps increasing. For years already, model-based development promises to ease a lot of the challenges in the development of mechatronic machine controllers. Still, its adoption by machine builder companies often remains limited to the algorithmic part of the controller, typically using tools such as Mathworks® Simulink®.**

This paper explains how model-based development can support other aspects of a mechatronic controller (such as its componentization) and how machine builders can adopt the required techniques. We present some trends observed in industrial use cases, the trade-offs the software architects are facing and the issues they have experienced. Although no silver bullet exists, some valuable guidelines are provided for developing a model-based software development strategy.

I. INTRODUCTION

The power and flexibility of software enables the development of countless machine features. While software's flexibility is a significant strength, it is also a fearsome enemy. Software becomes complex, difficult to manage and error prone. For some aspects of machine controller software, model-based development has come to the rescue. For example the control algorithms contained in controllers are well covered by tools such as Mathworks® Simulink® [1]. Two major qualities of such a tool are key to its success. First, the tool offers concepts on a higher abstraction level than the traditional software concepts for implementing them. This way, the control designer can reason at that higher level, less disturbed by the details of lower abstraction levels. Second, the created models become part of the mechatronic product by generating code from them. This distinguishes them from other model kinds used for analysis (e.g. UML use cases [2]) or simulation (e.g. Modelica physical modeling [3]), which add good value during other phases or branches of the system engineering life cycle.

Unfortunately, considering the investigated industrial cases, the advantages of model-based development didn't penetrate far beyond the control algorithms. Numerous other aspects of the software of a modern machine controller, such as configuration, safety, variability, error handling and security are still satisfied by manual coding, although model-based methodologies exist [4], [5], [6] and have proven valuable [7].

The paper's purpose is to explain the lessons learned during the introduction of model-based techniques in the controller software development of small-sized machine builders. For the sake of clarity, the aspect of componentization of machine controller software is used as an example throughout the paper. It is shown that the automotive standard Autosar [10] is a valuable source of inspiration for machine controller components (II) and that customization benefits from formal metamodeling (III). Next, the paper discusses machine builders' challenges to create and maintain custom tooling (IV) in particular model editors (V) and model exploitation (VI). Several drivers for cost and value specific to the machine builders were observed and are listed in (VII). The paper concludes that textual model editors or UML profiles based on formal metamodels are the only viable alternatives for small-sized machine builders (VIII). It finally suggests future work (IX) in the direction of AutoSAR-alike standard architectures and tooling for machine controllers.

II. AUTOSAR AS A COMPONENT FRAMEWORK FOR MACHINE CONTROLLERS?

To support the aspect of reuse, software developers often use the technique of modularization or more specifically, componentization [8]. Components can be individually designed, developed and tested. Later, they can be reused in different configurations, each with a different selection of components or with different connections between them. As such, they are key building blocks for the reliable production of machine variants or full product lines.

Contrary to the availability of component frameworks for desktop software [9] or in the automotive industry with AutoSAR [10], there is no real standard component framework for the machine builder industry. Since machine building is

*Corresponding author

technologically close to the automotive industry, it could seem a safe bet to go for a mature automotive industry standard.

Unfortunately, machine builders – typically having way smaller production volumes and development teams than the automotive industry – usually consider this standard too broad for them. Adopting AutoSAR would force them into a steep learning curve. The standard covers many more features than they need and requires AutoSAR compliant tools and hardware. For example, machine builders often only use one controller per machine. In that case, AutoSAR's features for inter ECU communication are superfluous. AutoSAR compliant tools enforce rules that enable the easy switching of suppliers of subparts. Such rules limit the machine builder's freedom without the benefits for which they are created.

Yet, these observations do not invalidate AutoSAR's component framework. Being mature and industry proven it can be treated as a valuable source of inspiration for the component frameworks of a machine controller, next to other more generic frameworks like Orocos [11] and parts of CORBA [12].

III. FORMAL METAMODELING

Since no component standard exists for machine builders, the architect needs to thoroughly analyze the company's needs and constraints. It is good practice to adhere to well proven frameworks and carefully determine where deviations are required. Those important analysis and scoping decisions need to be taken and documented in a clear and unambiguous way. This could be achieved in some textual document, but formal metamodeling invaluably contributes to the efficiency and effectiveness of this process.

Although the need for metamodeling is not specific to the machine builder domain, the complexity of this domain is an additional motivation for a formal approach. Unfortunately, the metamodeling technique and the corresponding languages such as MOF (Meta Object Facility) [13], Eclipse Ecore [14] and some proprietary ones all originate from the software world, which is not the background of most mechatronic engineers.

A fragment of a simplified metamodel for components is shown in Figure 1. Such a metamodel clearly describes all the concepts that occur in a domain. For each concept, it describes attributes and relationships. Additional constraints, not directly expressed in the metamodel, can be captured in notes or in formal constraints (e.g. using OCL [15]).

The creation of a formal metamodel helps in multiple ways.

- Clarity – During discussions between team members, it leaves limited room for interpretation.
- Verification – Because of its strict semantics, a metamodel can be verified. Creating instances quickly proofs whether the metamodel matches the needs.
- Solid base – Next steps build on the formal metamodel. If well integrated with the metamodel, the next steps can automatically respect the rules defined in the metamodel. Skipping the formal metamodel altogether, causes a vague distribution of domain knowledge over the next steps, as will be indicated later.

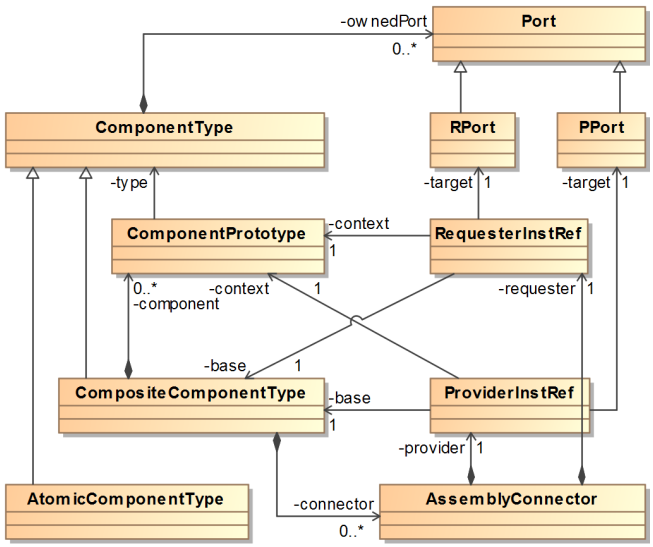

Figure 1. *A fragment of a simplified metamodel for components. It shows ComponentTypes that can own requester ports (RPort) and provider ports (PPort). Additionally it supports component nesting and connectors.*

When machine builders enter the world of modeling, the apparent ease of widely used UML diagrams and the flexibility of profiling cause modelers to immediately think of a UML profile as a way to express their formal metamodel. Unfortunately, the following UML profiling pitfalls then seem commonly neglected:

1) A profile is not a metamodel

There is a gap between the expressiveness of a metamodel and of a profile. OMG states "A profile is a restricted form of metamodel that can be used to extend UML." (from the chapter on Profiles in [2]). An example of such a restriction concerns composition. While a metamodel knows the frequently used concept of composition, profiles can't handle composition between stereotypes. If both Car and Wheel are stereotypes extending the MetaClass Class, a profile has no native way to state that a Car is composed of four Wheels. OCL or tool specific configuration would be required.

In fact, a profile could be better considered as a mapping of a real metamodel to UML concepts. In that sense it would be advantageous if profiled elements would be automatically consistent with the underlying metamodel. In most cases, a profile does not have such an automatically maintained link, but is manually crafted to match metamodel concepts. Creating this mapping and primarily maintaining it, causes additional effort and risk for inconsistencies. Some decent approaches exist [23], [24], [25] but have no broad adoption.

2) The desired metamodel is not UML's metamodel

The desired domain is not necessarily close to UML. Since UML knows components, the example about components introduced above seems close to how UML defines components, but even then some care should be taken. A profile can add some concepts to UML, but cannot conflict with UML. Once a domain defines some extension that is not allowed according to UML, a profiled UML editor will flag it as an

error. The editor should not allow any violation of the UML rules and most of them enforce this.

Additionally, a profile has only limited abilities in reducing the UML complexity. Sure some elements can be left out, but the definition depth usually remains. A profile cannot allow the user to skip the construction of intermediate model elements that do not match any domain concepts. It forces the user to create them anyway, adding unnecessary complexity. Some proprietary tool customization allow to work around this to some extent.

So as a conclusion, in all executed industrial cases it seemed optimal to define the domain formally in an independent metamodel, not limited to just a UML profile.

IV. TOOLING FOR COMPONENT MODELING

To support the model-based development, the product developer needs tools that allow him to create, update and use the models.

For machine builders, this poses a next important challenge. Most of them do not want to produce custom tools and they have valid reasons for this attitude. Machine builders produce machines, not software development tools. Because of their typically small software development teams, it can be challenging to acquire the skills to create such tools, maintaining those skills and maintaining the tools themselves. A custom tool is by definition a significant cost.

On the other hand, a dedicated tool also returns value, typically by reducing development time and bugs and increasing clarity and adherence to company standards. The question therefore is not whether to create a tool. The question is rather how to optimize the balance between a tool's value and its cost (VII).

Different kinds of tools are required to support the consecutive steps of the model-based development chain. The following sections highlight some of them and their usability for machine builders.

V. MODEL EDITORS

As mentioned above, a component metamodel indicates that there exists such a thing as a "Component" and that it can have "Ports", a product developer can use the model editor to make any number of these "Components" and give them the "Ports" he needs.

A big part of the value of such a model editor comes from the fact that it enforces the metamodel. The metamodel defines what is a valid model and what not. While the metamodel defines the concepts (their semantics), it doesn't define how these concepts should be represented (their syntax). Components could be represented textually, somewhat like classes in C++. Alternatively, they could be drawn graphically, similar to UML diagrams. The choice between textual and graphical syntax needs to be made.

Whatever syntax is chosen, creating a model editor is quite some work, but some differences exist. Below, a quick overview of today's options followed by model editor conclusions and guidelines observed in the industrial cases.

A. *Custom textual syntax*

Good tools to develop textual model editors exist. Xtext [17] for example has a perfect integration with Ecore metamodels and a reasonable learning curve, also for quite advanced features. Additionally, a textual syntax has the advantage of being quite manageable in the face of metamodel evolution, model comparison and versioning. Since all stored models are text files, generic text processing tools or light-weight scripts can be used to apply small changes to the models.

Textual editors support the user with features such as syntax highlighting, code completion, context sensitive help, and fast model navigation to referenced elements.

```
CompositeComponent ForkLiftHeader {
      components {
            tiltAngle : TiltSensor,
            load : WeightSensor,
            tiltValve : ValveActuator,
            upDownValve : ValveActuator,
            headerControl : ForkLiftHeaderControl
      }
      connectors {
            tiltAngle.angle -> headerControl.angle,
            load.weight -> headerControl.load,
            headerControl.tiltOut -> tiltValve.flow,
            headerControl.upDownOut -> upDownValve.flow
      }
}
```

Figure 2. *Example of composite component defined with a textual syntax. (Figure 3. shows the same concepts in a graphical syntax.)*

B. *Custom graphical syntax*

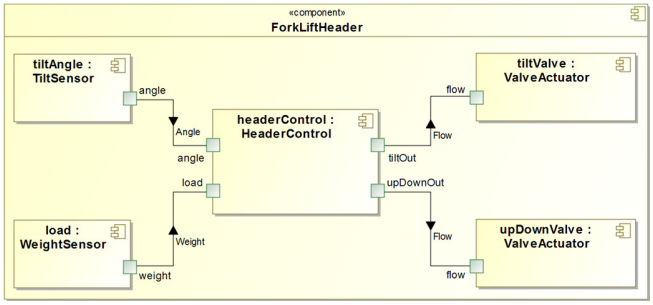

Figure 3. *Example of a composite component defined with a graphical syntax. (Figure 2. shows its textual counterpart.)*

For representing relations between parts (e.g. associations between the created components), diagrams often win from textual descriptions. Unfortunately, graphics are inherently more complex in their representation, hence developing a graphical model editor is more complex. Tools like Sirius [18] aspire to make the creation of a graphical model editor easier, but currently available tools like GEF [19] or GMF [20] have a steep learning curve.

C. *UML profile as graphical syntax*

Because of the ubiquity of UML as a modeling language for software systems and because of the availability of decent graphical editors for UML, it is often considered as a first choice to graphically edit domain specific models. As stated above, the UML standard defines how profiles can be used to

customize UML models and most UML editors provide a way to define and use such profiles [2].

Although profiling is a mature and valuable technique [21], there are some risks. Without much of a thought, companies seem to be choosing for UML (profiled or not) as a graphical editor, directly representing their domain knowledge, skipping the formal metamodeling step. When it becomes apparent that UML cannot capture an aspect of the domain, the domain knowledge is captured in naming conventions or by using specific UML features for other than their intended purpose or even by delaying corresponding aspects to the code generation phase. Such inconsiderate use of UML defeats clarity, increases the cost of the next steps in the model-based workflow, and causes maintenance issues.

D. Model editor guidelines

Given all those possibilities, choosing the right model editor strategy is a difficult choice in the development of a model-based strategy. Based on the elements discussed above, the following guidelines proved useful in the industrial use cases.

1) Metamodel and model editor should be strongly linked

When creating a model editor, one could consider the metamodel as a kind of requirements document and manually code the editor to comply with that metamodel. Unfortunately, this causes an important amount of additional development and maintenance effort. By preference, the conformance of the model editor to its metamodel is automatic. While this is possible with custom textual and graphical editors, UML profiles typically do not support a hard link with a metamodel.

2) Textual editors have a good cost-benefit ratio

Most software development teams of mechatronics companies are small and tool development is not their core business. They hesitate to acquire and maintain significant knowledge that is not directly related to product development. This is one of the reasons why only textual editors are realistically feasible, given their relatively low complexity.

Graphical tool development can be initiated by specialized external consultants, but the future maintenance effort of a graphical tool still makes this option uncomfortable. The higher cost of this development and maintenance is only acceptable in case bigger groups of product developers benefit from the tooling. Since the focused machine builders have small development teams, the leverage is often too limited.

3) Textual editors are good for prototyping

Because of their relative ease of development, textual editors prove valuable during the analysis and validation of the metamodel. Creating instances early in the process allows team members to better understand the consequence of the decisions taken in the metamodel. This approach is also followed in Example Driven Modeling [17], [18].

4) A mix of editors can be optimal

As indicated above, several kinds of model editors exist and none of them is a perfect fit for all cases. Sometimes the choice is even different for different parts of one metamodel. Referring back to the example of a component metamodel, one could prefer a textual syntax for the definition of the component interface and a graphical syntax to define relations between components.

As long as the model editors produce instances of the same metamodel, this approach is feasible. Once different metamodels come into play, cross editor referencing becomes more difficult. This is quite a common scenario for machine builders. Traditionally, machine controller software is often coded in C/C++ and frequently Simulink® is used for the control algorithms. So, whatever model-based strategy is adopted, the introduced metamodel will need to integrate with the metamodel of C++ and Simulink®. Similarly, if a UML profile is used as a graphical editor covering a part of the metamodel, UML's metamodel comes into play, causing the need for potentially challenging transformations or synchronizations.

VI. EXPLOITING MODELS

Once the product developer created models, he needs to exploit them. One obvious exploitation is code generation. Other exploitations can be thought of, such as additional model validations and transformations to or from other tools for the sake of interoperability.

The focus of this paper is on models that do not merely describe the product, but that become part of a product. For this reason, the models need to be integrated with other parts of the product. This is commonly achieved by converting the models into the same programming language as the one in which the other parts of the product are developed. For machine controllers this is usually C or C++. For years already, the complexity and flexibility of C and C++ are reasons to adopt coding guidelines such as MISRA C/C++ [22] in addition to company coding standards. Code generation helps enforcing these, since for the code generated by the code generator, only the code generator needs to respect the guidelines, thereby automatically emitting conforming code.

In case of machine component models, the model typically does not cover the component's internal behavior. The component's functionality could be implemented in C or C++ or generated from models created by Simulink®. Figure 4. shows the component case where the generated code covers aspects such as the wiring between the components, interfacing towards the component framework, framework configuration and interfacing towards the component's internals.

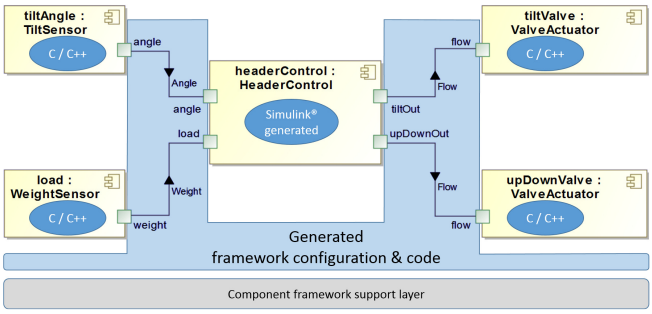

Figure 4. *In the component modeling example, the generated code covers the framework, the wiring between the components and the component interfaces. It glues the components' internal behaviour to the framework.*

Several guidelines allow for lowering the code generator development and maintenance cost.

A. *Accidental complexity of the metamodel*

The code generator converts instances of the metamodel to executable code. Ideally, the metamodel is well-defined and pure. Unfortunately, a metamodel usually is not completely pure [16]. Moreover, sometimes UML is used because of its convenience as a graphical editor. In such a case, the inherent complexity of UML does in no way contribute to the value of the metamodel and therefore adds quite some accidental complexity [28]. The code generator now needs to browse the full UML structure, while in fact it is only interested in those pieces that relate to the underlying pure metamodel. Profiled UML makes this task even harder.

As an example of this added complexity, compare two representations of a Car with four Wheels. Figure 5. shows a representation in UML, while Figure 6. shows a pure instance of the Ecore metamodel drawn in Figure 7. The first instance clearly carries a lot more accidental complexity because of the underlying UML metamodel.

```
(...)
<uml:Model xmi:type='uml:Model' xmi:id='id1' name='Car'>
  <ownedComment xmi:type='uml:Comment' xmi:id='id2'>
    <annotatedElement xmi:idref='id1'/>
  </ownedComment>
  <packagedElement xmi:type='uml:Package' xmi:id='id3' name='Instances'>
    <packagedElement xmi:type='uml:InstanceSpecification' xmi:id='id4'
                     name='myCar'>
      <classifier xmi:idref='id5'/>
      <slot xmi:type='uml:Slot' xmi:id='id6' definingFeature='id7'>
        <value xmi:type='uml:InstanceValue' xmi:id='id8' instance='id9'/>
        <value xmi:type='uml:InstanceValue' xmi:id='id10' instance='id11'/>
        <value xmi:type='uml:InstanceValue' xmi:id='id12' instance='id13'/>
        <value xmi:type='uml:InstanceValue' xmi:id='id14' instance='id15'/>
      </slot>
    </packagedElement>
    <packagedElement xmi:type='uml:InstanceSpecification' xmi:id='id9'
                     name='frontLeft'>
      <classifier xmi:idref='id16'/>
      <slot xmi:type='uml:Slot' xmi:id='id17' definingFeature='id18'>
        <value xmi:type='uml:LiteralInteger' xmi:id='id19' value='16'/>
      </slot>
    </packagedElement>
    <packagedElement xmi:type='uml:InstanceSpecification' xmi:id='id11'
                     name='frontRight'>
      <classifier xmi:idref='id19'/>
      <slot xmi:type='uml:Slot' xmi:id='id20' definingFeature='id18'>
        <value xmi:type='uml:LiteralInteger' xmi:id='id19' value='16'/>
      </slot>
    </packagedElement>
    <packagedElement xmi:type='uml:InstanceSpecification' xmi:id='id13'
                     name='rearLeft'>
      <classifier xmi:idref='id21'/>
      <slot xmi:type='uml:Slot' xmi:id='id22' definingFeature='id18'>
        <value xmi:type='uml:LiteralInteger' xmi:id='id19' value='16'/>
      </slot>
    </packagedElement>
    <packagedElement xmi:type='uml:InstanceSpecification' xmi:id='id15'
                     name='rearRight'>
      <classifier xmi:idref='id16'/>
      <slot xmi:type='uml:Slot' xmi:id='id23' definingFeature='id18'>
        <value xmi:type='uml:LiteralInteger' xmi:id='id24' value='16'/>
      </slot>
    </packagedElement>
  </packagedElement>
</uml:Model>
(...)
```

Figure 5. *A Car with four Wheels, represented as instance specifications of UML classes. This demonstrates the significant amount of accidental complexity for UML as compared to Figure 6. The ellipsis indicates minor parts left out for clarity.*

```
<carmeta:Car name="myCar" (...)>
  <wheel name="frontLeft" diameter="16"/>
  <wheel name="frontRight" diameter="16"/>
  <wheel name="rearLeft" diameter="16"/>
  <wheel name="rearRight" diameter="16"/>
</carmeta:Car>
```

Figure 6. *A Car with four Wheels, represented as a pure Eclipse Ecore instance. To be compared with the complexity of Figure 5.*

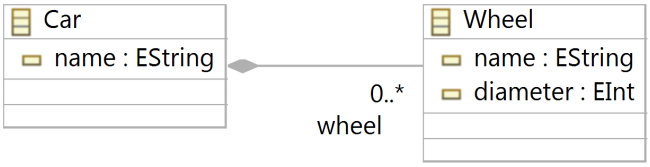

Figure 7. *The simple Ecore metamodel backing the instances in Figure 6.*

Additionally, it is undesirable that models depend significantly on the choice of model editor. A code generator based on a profiled UML model needs to be rewritten in case a company later shifts to a textual model editor. By preference, the code generator should therefore always work on pure instances of the metamodel.

In case profiled UML is chosen for the models, it is even worthwhile to invest on a transformation from the profiled UML model towards a pure instance of the metamodel and generate code from that pure instance. This return to the central pure metamodel achieves code generator immunity for accidental complexity introduced by model editor choices and therefore lowers development and maintenance cost.

B. *The code generation language*

Code generation is a very specific task. Some languages are dedicated to this task. A language such as Xtend [30] closely integrates with EMF metamodels. The language understands the metamodel and some of its constructs are specifically intended for code generation, which considerably reduces the effort. Moreover, Xtend is nowadays positioned as a widely usable language, broader than code generation only.

Most companies in the industrial use case are reluctant to adopt yet another language and prefer sticking to general purpose languages that they already master. Typical choices are scripting languages such as Python. These could already be used in build systems or for other automation tasks. Their knowledge is spread amongst the development team and therefore they are considered as first candidates for code generation tasks as well. While this is a valid argument for smaller code generation efforts, for any more advanced use of code generation the power of a dedicated code generation language quickly outweighs the effort of learning it.

In this context, it was surprising to see that the reluctance against adopting dedicated languages seems to be present even with junior engineers. It became clear that mechatronics courses, even on university level, only sparingly discuss the power of model-based techniques and the corresponding tools. This is clearly a missed chance to enhance the adoption of model-based in the mechatronics industry.

As far as UML editors are concerned, most of them don't allow a free choice of code generator language. Their models are often stored in a proprietary format. Except for an export to semi-standardized XMI, they are usually only accessible through the product's internal API. This forces the adoption of their custom code generation language. While their close integration in the tool makes these code generator languages very effective, it is unfortunate that the skills cannot be used anywhere outside this tool. If any other code generation task occurs on metamodels outside this tool, yet another language needs to be adopted.

VII. THE COST VALUE BALANCE

The bottom-line question for every machine builder is: What can be gained by going model-based? While there are clear advantages, there is also a cost. This section highlights the main cost and value drivers perceived in the industrial use cases.

A. Drivers of cost

1) Graphical tooling

With the current state of the art, the creation of a custom graphical tool remains a significant investment. Certainly during the metamodel analysis, the use of textual languages is way more cost effective. Also in production, textual languages are a useful choice, sometimes accompanied by generated graphical visualizations. A typical developer in a machine builder company is used to languages like C, C++ or scripting languages like Python. For such a developer, a textual syntax feels quite natural. This is quite different if the models would need to be created by potentially less code-oriented people, like system engineers.

If graphical editing is really required, the usage of a profile in a UML tool is the only viable alternative as long as it is backed by a formal metamodel. Some dedicated tools exist for the development of graphical syntaxes (e.g. MetaEdit [31]).

2) Learning curve

Converting a part of a development workflow from traditional hand coding to model-based development consists of several parts, each of them coming with a learning curve. It is a design goal to align the model-based tool as close as possible with concepts known by the product developer. The learning curve on that side is therefore limited to getting accustomed with the new way of working. The biggest learning curve is to be expected on the tool development side. A traditional product developer does not necessarily master the techniques to create a domain model, a model editor, code generators and other transformations. Building this knowledge and experience is a major cost driver. Although it can be eased by attracting external experienced consultants, companies don't want to make their product development dependent on custom tools that they do not master themselves.

The key objective is therefore to adopt as few new technologies as possible. Since a full model-based chain has to be constructed, the choice of well integrated, flexible tooling is a must. The choice for Eclipse, Ecore, Xtext, Xtend and other Ecore related tools has proven to provide one such ecosystem.

3) Maintenance

Closely linked to the learning curve, is keeping the knowledge and spending the time to support evolution of the model-based tooling. Like any part of product development, the model-based development tools are subject to change. Looking back at the component modeling example, one could want to introduce a new kind of port or relationship. This involves adjusting the metamodel, the model editor, the code generator and any other transformations. Moreover, one has to be able to still recreate old products, based on old versions of the tooling, so the tooling has to be versioned alongside the product versions. Changes in the metamodel could also require conversion of the already created models. Such conversions require time and the product potentially requires some retesting afterwards.

Textual syntaxes clearly offer the lowest possible maintenance cost, but even for these textual tools the potential maintenance cost remains considerable.

B. Drivers of value

1) High abstraction levels

Not much can be gained if the complexity of the models comes close to that of the corresponding generated code. The higher the abstractions in the models, the more value they will render. This value shows in the leverage of the code generator (write a bit, generate a lot), but also in communication. Although the communication benefit is difficult to quantify, every industrial case perceived it as significant. People with different backgrounds can discuss better on models than on the underlying code. Coaching time for new hires is reduced, since the relevant knowledge is not anymore in the experienced developers' heads only, but could be found in a formal metamodel and corresponding tools and code generators.

2) Many developers, big scope

The effort for building a model-based tooling doesn't scale linearly with the amount of developers nor with the scope of the tooling. In fact, it almost doesn't scale with the amount of developers at all. The required knowledge and experience can be used for any scope. Of course it is more work to support a bigger scope, but that is only a part of the full effort.

3) Traceability

In contrast to traditional hand coded development, models are much easier to reference or to contain references. Given the increasing importance of functional safety in machine building applications and the fact that functional safety standards require traceability, the questions is not whether to switch to model-based development for all parts of the controller, but rather when machine builders will have to switch.

4) Multi-platform

To a certain extent, the model can be seen as platform independent. The code generator closes the gap between the model and a given platform. So, if the platform architecture is improved, updating the code generator fixes the problem for all models, without the tedious effort of lots of manual changes. Even more, if several platforms have to be supported in parallel: models remain unique and switching the code generator reliably produces output for the other platform.

VIII. CONCLUSIONS

Successful application of model-based techniques in machine controllers is a challenge. This paper presented the experiences from industrial use cases. Working with these cases, no single solution was discovered that fits them all. Several trade-offs remain to be made. The following can be concluded:

A. Formal metamodeling is a must

All next steps benefit from a formal metamodel. Even without the next steps, it already forces the architect to explicitly think about the solution he's creating and it enhances the communication between all stakeholders.

B. *Model editors are by preference textual*

At the time of writing, the value for money of textual editors is high compared to graphical ones. In case graphical is desired and the domain does not conflict with UML, profile definition for a UML editor can be a viable alternative. In that case, all transformations should return to the formal metamodel before building further on the models. Custom graphical editors are possible if the rendered value outweighs their significant cost.

C. *Integrated tooling reduces cost*

The choice for a multi-purpose modeling ecosystem avoids the cost of learning many new tools. The comfort of one open environment reduces the cost for the tool developer. Eclipse currently comes closest to this ideal.

D. *Iterative prototyping builds confidence*

During development of a metamodel, a textual editor and a corresponding code generator allow to quickly gain feedback from all stakeholders. Iteratively increasing the scope and quality of each of the parts, finally leads to a solid and verified choice of domain concepts and corresponding code generation strategies. If finally a graphical editor would be desired, it can be developed now with a reduced risk of expensive iterations.

E. *Education plays an important role*

A wide variety of machinery exists, so custom tooling will always be a part of the machine builder's world. Although existing domain specific tooling already provides significant abilities, current engineers and management hesitate to adopt it. An important contribution can come from educational institutions, who could in their mechatronic curricula focus more on teaching model-based engineering concepts than focusing on yet another programming language.

IX. FUTURE WORK

The paper summarizes findings from industrial use cases and clearly some important improvements could still be made.

For custom tooling, products like Xtext have already dramatically reduced the effort for textual editors. A similar breakthrough on the graphical side could enable the adoption of custom graphical tooling by smaller development teams.

Tools serving the domain of machine builders, should not try to cover the full development chain, but will gain adoption rates in case their tools allow for easy, open integration. Such open tool integrations are challenging if no standard is available for the machine builder world. Machine builders have language and tool integration challenges (SysML, C/C++, Simulink®) similar to those of the automotive world. An effort such as AutoSAR has been realized through the commitment of many automotive players and tool venders. The establishment of an equivalent standard focused on machine building would allow tooling players to provide for flexible tool integrations.

REFERENCES

[1] Mathworks® and Simulink® are registered trademarks of "The Mathworks Inc", http://www.mathworks.com/products/simulink/

[2] Object Management Group – Unified Modeling Language 2.5 beta 2, http://www.omg.org/spec/UML/2.5/Beta2/PDF

[3] The Modelica Association, https://www.modelica.org/

[4] Anjali Joshi, Mats P.E. Heimdahl, Steven P. Miller, Mike W. Whalen, "Model-based Safety Analysis", NASA/CR-2006-213953, 2006-02-01

[5] Mathieu Acher, Patrick Heymans, and Raphaël Michel. "Next-generation model-based variability management: languages and tools." Proceedings of the 16th International Software Product Line Conference - Volume 2 (SPLC '12), Vol. 2. ACM, New York, NY, USA, 276-277.

[6] J.F. Broenink, M.A. Groothuis, P.M. Visser, B. Orlic, "A Model-Driven Approach to Embedded Control System Implementation", Proceedings of the 2007 Western Multiconference on Computer Simulation, pp. 137-144, ISBN 1-56555-311-X, 2007

[7] Kärnä J., Tolvanen JP, Kelly S., Evaluating the Use of Domain-Specific Modeling in Practice, DSM Workshop, 2007

[8] J.Sametinger, "Software engineering with reusable components", Springer-Verlag New York, Inc. New York, NY, USA 1997

[9] ISBN:3-540-62695-6 OSGi Open Services Gateway initiative, http://www.osgi.org

[10] Automotive Open System Architecture, http://www.autosar.org/

[11] H. Bruyninckx, "Open robot control software: the OROCOS project," in Proceedings of the IEEE International Conference on Robotics and Automation, pp. 2523–2528, May 2001.

[12] Object Management Group – Common Object Request Broker Architecture, http://www.omg.org/spec/CORBA/3.3

[13] Object Management Group – Meta Object Facility Core v2.4.1, http://www.omg.org/spec/MOF/2.4.1

[14] Eclipse Modeling Framework – http://www.eclipse.org/modeling/emf/

[15] Object Management Group – Object Constraint Language 2.3.1, http://www.omg.org/spec/OCL/2.3.1/PDF

[16] Colin Atkinson, Thomas Kühne, "Reducing accidental complexity in domain models", Software & Systems Modeling, July 2008, Volume 7, Issue 3, pp 345-359

[17] Eclipse Xtext – https://www.eclipse.org/Xtext/

[18] Eclipse Sirius – http://eclipse.org/sirius/

[19] Eclipse GEF – http://www.eclipse.org/gef/

[20] Eclipse GMF – http://www.eclipse.org/modeling/gmp/

[21] Customizing UML: Which Technique is Right For You? http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html

[22] Motor Industry Software Reliability Association, http://misra.co.org.uk

[23] Florian Noyrit, Sébastien Gérard, Bran Selic, "FacadeMetamodel: Masking UML", Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science Volume 7590, 2012, pp 20-35

[24] Ana Petricic, Luka Lednicki, Ivica Crnkovic, Using UML for Domain-Specific Component Models, Fourteenth International Workshop on Component-Oriented Programming, 2009

[25] Giovanni Giachetti, Beatriz Marín, Oscar Pastor, Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles, 2009

[26] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara, "Example-driven meta-model development", Software & Systems Modeling, 2013-12

[27] Bąk, K., D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wąsowski, and D. Rayside, "Example-Driven Modeling. Model = Abstractions + Examples", New Ideas and Emerging Results (NIER) track of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 2013

[28] Brooks, Fred P., "No Silver Bullet – Essence and Accident in Software Engineering", Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-1076.

[29] Keith Cooper, Linda Torczon, "Engineering a compiler", 2003-11-10, ISBN-13: 978-1558606982 Edition: 1

[30] Eclipse Xtend, https://www.eclipse.org/xtend/

[31] MetaCase MetaEdit+, http://www.metacase.com/