Automated Construction of Data Integration Solutions for Tool Chains

Matthias Biehl, Jiarui Hong, Frederic Loiret
Embedded Control Systems
Royal Institute of Technology
Stockholm, Sweden
{*biehl,hong,floiret*}*@md.kth.se*

***Abstract*—Modern software development relies increasingly on the orchestrated use of development tools in the form of seamless, automated tool chains. Tool chains are becoming complex software systems themselves, however, the efficient development of tool chains is a largely unsupported, manual engineering task. We propose both a domain specific modeling language for systematically specifying tool chains and generators for efficiently realizing the tool chain as software. Tool chain software consists of diverse components, such as service-oriented applications, models and model transformations, which we produce by different generative techniques. We study both the separate generative techniques and the dependencies between the generated artifacts to ensure that they can be integrated. We evaluate the approach both quantitatively and qualitatively, and show in a case study that the approach is practically applicable when building a tool chain for industrially relevant tools.**

Keywords-Domain Specific Modeling; Tool Integration; Prototyping; Higher-Order Model Transformation; Code Generation.

I. INTRODUCTION

Since modern development relies more and more on sophisticated development tools, the integration of these tools becomes an important issue. The development tools may be modeling tools, simulation tools, verification tools etc., which are typically not designed with ease of integration in mind [7]. The integration of development tools thus requires a sizable engineering effort, including the extraction of data from the integrated tools, adherence to integration standards and mapping of the data between the formats of different tools. Manually implementing the tool chain is time-consuming and error-prone.

Realizing a model-based tool chain as software involves writing source code for two distinct parts. The first part deals with setting up the infrastructure for model-based tool integration, such as transformation engines or tracing tools. The second part realizes the actual exchange of tool data, such as the extraction of data from the tool and its transformation into a different format or representation.

Several approaches for tool integration are mentioned in the literature, for example model-based tool integration [1], weaving-based tool integration [6] or ontology-based tool integration [12]. Model-based tool integration assumes that data of different tools is available in the form of models, which adhere to metamodels, and model transformations, which describe the data conversion [1]. The focus of model-based tool integration is thus on describing tool data and its relations (expressed as models, metamodels and transformations).

The entire tool chain, however, is only an implicit concept. As a result of this lack of an overall picture of the tool chain, existing approaches do not uncover the potential for supporting the development of complete tool chains. Existing approaches typically assume that the source code for providing tool data and functionality is implemented manually. If the implementation needs to follow integration standards, such as OSLC (Open Services for Lifecycle Collaboration) [24], it can be tedious to implement this code. Existing approaches also assume that the data conversion rules, which are necessary for data exchange between tools, are implemented manually, e.g., in the form of weaving models or model transformations. Support to synthesize the conversion rules is missing.

The goal of this paper is to systematize and partly automate the development of tool chains. The central question we address is thus: *To what extent can the development of tool chains be automated through generative techniques?* Our approach is model-based, but differs from previous model based approaches, as not only the tool data is modeled, but also the architecture of the complete tool chain. For this purpose we use a domain specific modeling language for tool chains, which structures the tool chain into ToolAdapter components and connectors. A generative approach uses the structured information to synthesize implementations for the components, the connectors and the infrastructure of the tool chain. In previous work, we have described the modeling language [4], code generation of the components [5] and infrastructure [4]. The contribution of this paper is the automated synthesis of the data conversion rules in the connectors and the integration of the various generated parts into a complete, cohesive tool chain.

II. APPROACH

From our experience of developing tool chains in an industrial context, tool chains are often developed with an iterative prototyping approach. While the general goal for the tool chain might be clear to all stakeholders, the exact

 ISBN: 978-1-61208-230-1

details of the execution and conversion in the tool chain might not be clear and different design options are explored with prototype implementations. The challenge lies in the large effort of creating one or potentially many prototype implementations of the tool chain. It involves creating a software adaptation layer – known as ToolAdapter – for each integrated tool and model transformations that convert the data between the proprietary data format of the tools. The method and automated techniques of the proposed approach aim to reduce the necessary development effort for tool chains.

In our approach, the user models the tool chain using abstractions from the domain of tool integration. A prototype implementation of a tool chain is produced by generating both source code and transformations from the model. In the following sections, we describe how we achieve the specification and the automated synthesis of the executable prototype. We divide the approach into several steps, as illustrated in Figure 1:

- Step 1 - Specification of a Tailored Tool Chain: The essential design decisions are described by modeling the tool chain in the Tool Integration Language (TIL) [4]. More details on modeling with TIL are presented in Section III.
- Step 2 - Synthesis of ToolAdapters: The synthesis automatically generates code based on the specification of the ToolAdapters. It is presented in Section IV.
- Step 3 - Synthesis of Channels: The synthesis automatically generates code and transformation rules, based on the specification of the Channels. It is presented in Section V.
- Step 4 - Integration of Generator Results: The generated parts need to be integrated into a tool chain, as presented in Section VI.
- Step 5 - From Prototype to Production Software: The prototype tool chain can be refined into a production tool chain, as presented in Section VII.

Steps 1 - 4 are typically iterated multiple times, until a satisfactory prototype is identified. Since only step 1 is manual and steps 2, 3 and 4 are automated, this is a viable iterative development approach for tool chains. We evaluate the approach qualitatively by a running example, which is embedded in Sections III-VII. We evaluate the approach quantitatively in Section VIII. In Section IX, we show the relation of this approach to other work in the field; in Section X, we mention future work and conclude.

A. Introduction to the Running Example

We illustrate all steps of the approach by stepwise constructing a tool chain, which serves as a running example. The intended use of the tool chain presented as running example is in the early design phase of automotive embedded system development [2]. An engineer creates a UML-conform model with behavioral and fault propagation

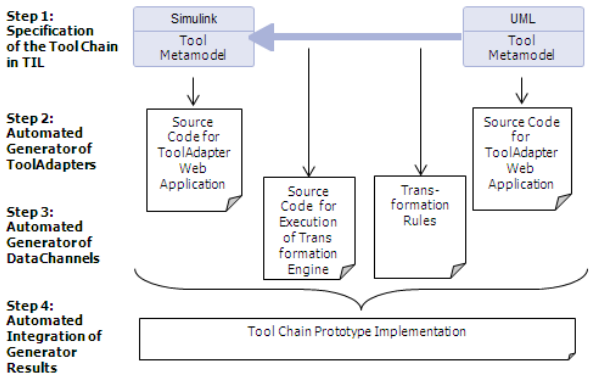

Figure 1. Overview of steps 1-4 of this approach

models using the GUI of the development tool and commits the model to the repository. Every time a new version of the UML model is committed, the tool chain executes a transformation of the UML model to the input format of the fault tree analysis tool HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [25] and executes HiP-HOPS. Another model transformation creates a MATLAB/Simulink model that mirrors the structure of the UML model. When the results of the fault tree analysis are satisfactory, i.e., there are no single points of failure, the engineer manually extends this Simulink model to perform simulations.

III. Step 1 - Specification of a Tailored Tool Chain

The prototyping process starts by specifying the big picture of the tool chain. We use TIL, a declarative, domain specific modeling language for tool chains. TIL models are concise, and expresses domain concepts, so users can relate to it. TIL allows us not only to describe a tool chain graphically and with well-defined semantics, but also to analyze it and generate code from it.

Here, we can only give a short overview of the language, for more detailed description of syntax and semantics, we refer to [4]. In the following, we introduce the language concepts and their concrete graphical syntax (compare ① .. ⑦ in Figure 2). TIL is a component-based language and consists of Components (ToolAdapter, Repository, Sequencer, User) and Connectors (ControlChannel, TraceChannel, DataChannel).

- A ToolAdapter ① exposes the data and functionality of a tool in a common technical format, making the data and functionality accessible for other ToolAdapters within the tool chain. A ToolAdapter is specified by means of a ToolAdapter metamodel. It describes the selection of data and functionality of the tool, which is exposed to the tool chain (see Figure 4, for an example). Creating the metamodel requires some engineering

 ISBN: 978-1-61208-230-1

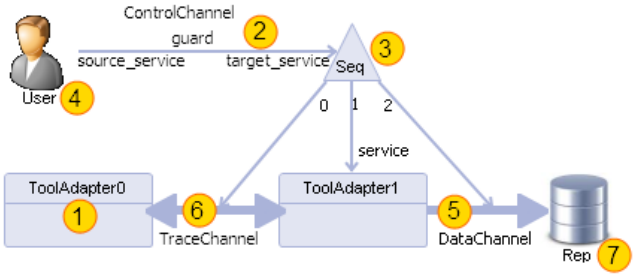

Figure 2. A simple TIL model illustrating the graphical syntax of the language concepts

effort, since each tool has its own metamodel, there is no common metamodel.

- A ControlChannel ② describes the control-flow between two Components by specifying a triggering event in the source, a called service in the target and a guard for conditional execution.
- A TraceChannels ⑥ connects two ToolAdapters and describes the possibility of creating traces between elements of certain data types of the ToolAdapter metamodels.
- A DataChannels ⑤ connects two ToolAdapters and describes the data-flow between them. The DataChannel preserves the semantics of the data. Since the ToolAdapter delivers all exposed data in a common technical format, only the data structure needs to be adapted. The data of the source ToolAdapter needs to be transformed into the structure expected by the target ToolAdapter. A model transformation can either be manually specified or it can be automatically synthesized, as described in Section V.
- A Sequencer ③ describes sequential control-flow; it executes a sequence of services in a specified order. The sequencer is used in combination with ControlChannels: it is activated by a ControlChannel and each of the sequentially called services is connected via a ControlChannel.
- A User ④ is a representative for a real tool chain user. This concept is used to describe the possible interactions of the real users with the tool chain. Outgoing ControlChannels from the User denote services invoked by the user, incoming ControlChannels to a User denote a notification sent to the user.
- A Repository ⑦ is a specific type of ToolAdapter that provides storage and version management of tool data.

A. Running Example: Step 1

We specify the previously introduced tool chain in TIL, resulting in the model displayed in Figure 3. An engineer, depicted by the user symbol, develops a new function of an embedded system as a UML component model. The engineer checks the model into the Subversion Repository, depicted by a ControlChannel, which activates the DataChannel uml2repository. Automatically, the model will be analyzed by a safety analysis tool to detect single points of failure in the embedded system. This is depicted by the triangular shape for the Sequencer `Seq0`, which is activated by a ControlChannel, whenever new UML models are checked into the repository. The Sequencer `Seq0` first triggers the DataChannel `uml2safety` to transfer the UML model to the safety analysis tool involving a model transformation. The Sequencer `Seq0` then calls the function to analyze single points of failure in the safety analysis tool. If no single points of failure have been found, which is expressed as a guard condition on the ControlChannel, a simulation of the behavior of the new model is started in Simulink. This is realized by another set of ControlChannels and the Sequencer `Seq1`. Finally, the engineer receives an email notification about the simulation results.

The TIL model presented in Figure 3 is linked to several ToolAdapter metamodels, which are illustrated in Figure 4. In addition, model instances of these metamodels are linked to each ToolAdapter. They serve as test data for the prototype implementation. Each of the metamodels describes the subset of the data of the tool that is exposed by the ToolAdapter towards the tool chain. The metamodel for the MATLAB/Simulink tool (A) describes a basic block diagram. The metamodel for the UML tool (B) comprises the elements of a basic UML component diagram. The metamodel for the safety analysis tool HiP-HOPS (C) is organized into systems and subsystems.

IV. Step 2 - Synthesis of ToolAdapters

ToolAdapters are realized as software components that have a web-based, RESTful architecture. This provides platform independence and allows for a distributed tool chain, where tools may reside on different network nodes. The input of the generator is the ToolAdapter metamodel and a model with test data, which conforms to the metamodel. The output is a Java source code and configuration files for the service infrastructure, such as a web server listening for requests. The generated Java source code provides a skeleton of the tool adapter implementation, including an implementation that operates on static test data and serves it conform to the format and protocols of the industrial initiative OSLC. The generated ToolAdapter is thus functional, but does not yet connect to the tool instance via APIs. The connection to the tools has to be added manually in step 5.

Since the details of generating service-oriented tool adapters are not the focus of this work, we refer to paper [5] for more details and examples.

V. Step 3 - Synthesis of Channels

The Channels between ToolAdapters can describe control-flow or data-flow. Control-flow is expressed by ControlChan-

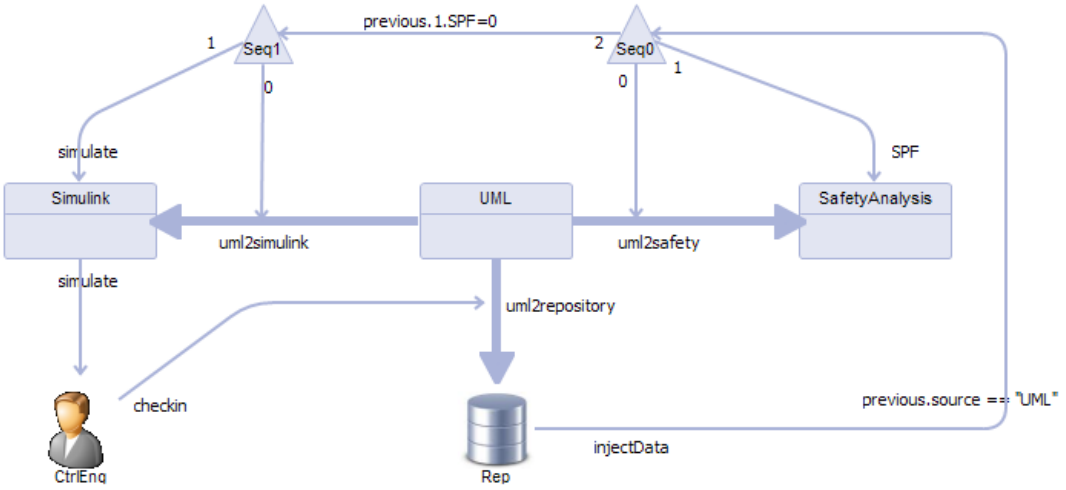

Figure 3. Specification of the tool chain as a TIL model

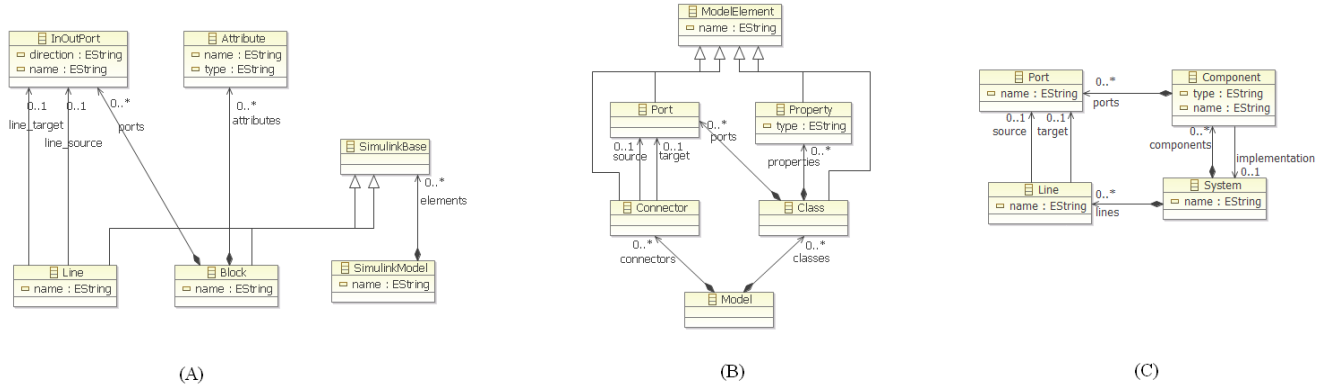

Figure 4. ToolAdapter metamodels for the Simulink tool (A), UML tool (B) and safety analysis tool (C)

nels, which are straightforward to synthesize as remote service calls to other ToolAdapters. Data-flow can be expressed by TraceChannels or DataChannels. TraceChannels provide the infrastructure for creating traces at runtime of the tool chain and this infrastructure is quite straightforward to generate.

DataChannels denote the transfer of data from a source ToolAdapter to a target ToolAdapter. The tool data is served by the ToolAdapter in the form of a model that conforms to the ToolAdapter metamodel. If the metamodels of source and target ToolAdapters are the same, the data can be simply copied between the ToolAdapters. In the more common case that the metamodels are different, the data needs to be transformed before it can be accepted by the target ToolAdapter. For this purpose, TIL offers the possibility to link a model transformation to each DataChannel. This transformation is a part of the implementation of the DataChannel and can be either manually specified or synthesized. If a transformation is required (due to different source and target ToolAdapter metamodels), but none is specified, a prototype transformation can be automatically synthesized. Another part of the implementation of DataChannels is the infrastructure for executing the transformation and transferring the data to another ToolAdapter.

The synthesis algorithm generates source code for the ControlChannels as well as source code for the infrastructure of DataChannels. This infrastructure accomplishes the following tasks at runtime of the tool chain: it gets the source model from the source ToolAdapter and provides it together with the transformation to the transformation engine. The target model, which is produced by the transformation engine, is sent to the target ToolAdapter.

In the following, we take a closer look at the automated generation of an appropriate prototype transformation. A transformation is appropriate if its source and target metamodel is identical to the metamodels of source and target ToolAdapters of the DataChannel and if it is semantics preserving. The TIL model contains some information for synthesizing the transformation, such as its execution direction and both its source and target metamodels. This

 ISBN: 978-1-61208-230-1

information is not sufficient for an algorithmic approach, but a heuristic approach for prototyping model transformations can be realized. The intention is to use an automated approach to quickly create a first model transformation. In the following sections we explain each step of the generator for prototype transformations in more detail.

A. Step 3.1 - Finding Correspondences

We assume a certain level of similarity between the source and target metamodels; so we can find correspondences between them based on structural and naming similarities. To find correspondences, we use a matching algorithm that is based on the similarity flooding algorithm [20] and the Levensthein distance [16]. The similarity flooding algorithm is used to detect structural similarities, the Levensthein distance is used to identify naming similarities. The matching algorithm produces a matching table, consisting of a number of correspondences between metaclasses, metaattributes and metareferences of source and target metamodels. In more formal terms, the matching algorithm $\sigma(m_s, m_t) = \mu$ produces a matching table μ for a given tuple of source metamodel m_s and target metamodel m_t.

B. Step 3.2 - Refining the Matching Table into a Matching Model

To ensure that a valid target model can be produced by the synthesized transformation, we automatically refine the matching table μ into a matching model ν by adding information about the containment hierarchy of the target metamodel with the refinement function ρ, which is defined as $\rho(\mu, m_t) = \nu$. This refinement is necessary, so the synthesized transformation can produce target models with an adequate containment structure, which is specified in the target metamodel m_t. The containment hierarchy of a metamodel is a partial order over all metaclasses in the metamodel that have a direct parent-child relationship.

The metamodel of the matching model is depicted in Figure 5. It consists of a number of ordered matchings. A matching describes a correspondence and consists of a description of the source and target elements, a number of related matchings and a type. The type of the matching is based on the role that the target element of the matching takes in the target metamodel. We differentiate five types of matchings:

- Top: A top matching has a target element that is the root element of the containment hierarchy of the target metamodel, i.e., the element that is not contained anywhere else. A top matching specifies the names of classes and usually has a number of containment matchings.
- Containment: A containment matching represents a reference between two metaclasses in the target metamodel, where one class is the parent and the contained class is the child. A containment matching specifies the names of the reference and metaclasses.
- Reference: A reference matching represents a link between two arbitrary metaclasses in the target metamodel. A reference matching specifies the names of the reference and metaclasses.
- Class: In a class matching, the target element is a metaclass. A class matching specifies the names of the metaclasses and usually has a number of related matchings, which are of type containment, reference or attribute.
- Attribute: In an attribute matching, the target element is a metaattribute of a metaclass. An attribute matching specifies the names of the metaattribute and metaclass.

The automated refinement ρ adds a containment matching for a target metaclass if necessary and ensures that all target metaclasses are properly contained. It also checks that no target element is produced by more than one rule. Note, that this classification depends on the target element only, since the target element needs to have proper containment hierarchy to be produced.

C. Step 3.3 - Synthesis of Transformation Rules from the Matching Model

The model transformation τ_1 is automatically synthesized based on the matching model ν, which is produced in the previous step and is executed at runtime of the tool chain for the exchange of tool data. The synthesis of τ_1 is performed by a second model transformation τ_2, which is a higher-order model transformation, defined as $\tau_2(\nu) = \tau_1$. The transformation τ_2 produces τ_1 and is executed at design time of the tool chain. The transformation $\tau_1(n_s) = n_t$ maps the model n_s to the model n_t, where n_s corresponds to the source metamodel m_s, and n_t corresponds to the target metamodel m_t. The synthesized model transformation τ_1 is a model-to-model transformation, implemented with OMG QVT-R [23]. The synthesizing model transformation τ_2 is a model-to-text transformation, implemented with OMG MTL [22].

For each matching in the matching model ν, the transformation τ_2 produces one or several QVT relations. For each type of matching a different template for the relations is used. The template is instantiated with the values from the current matching. All customized QVT relations for all matchings together form the synthesized transformation τ_1. See listing 1 for an example.

D. Running Example: Step 3

For the DataChannels simulink2uml and simulink2hiphops, this transformation is synthesized by the generator. The metamodels for UML, Simulink and HiP-HOPS show a certain degree of structural and naming similarity, as they represent a hierarchical composition of

 ISBN: 978-1-61208-230-1

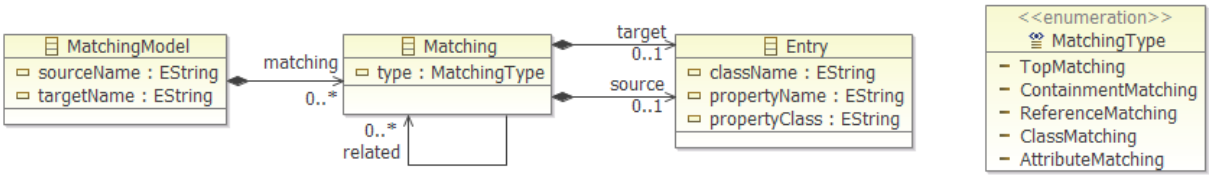

Figure 5. Metamodel of the matching model

components and components are linked by connectors via ports.

Table I
MATCHING TABLE FOR UML AND SIMULINK

UML Metaclass	Simulink Metaclass
Port	InOutPort
EString	EString
Class	Block
Property	Attribute
Connector	Line
Class.properties	Block.attributes
Property.type	Attribute.type
Class.ports	Block.ports
Connector.source	Line.line_source
Connector.target	Line.line_target

Table II
MATCHING TABLE FOR UML AND HIP-HOPS

Simulink Metaclass	HiP-HOPS Metaclass
Port	Port
Connector	Line
EString	EString
Class	Component
UMLModel	System
Property.type	Port.name
UMLModel.connectors	System.lines
Connector.target	Line.target
Connector.source	Line.source

As a first step in the automated synthesis of the transformation, a matching table is created. The automated metamodel matching algorithm is applied on the UML and the Simulink metamodels, yielding the matching table I. Applying the algorithm on the UML and the HiP-HOPS metamodels yields matching table II. Since the matching algorithm in step 3.1 is a heuristics, the automatically created matching table needs to be checked manually. All the mappings identified between Simulink and UML are correct, between UML and HiP-HOPS the matching algorithm correctly identified many mappings, however the heuristics introduced one error by mapping Property.type to Port.name. The subsequent refinement step 3.2 automatically corrects this error by replacing the matching with (Property.type,Component.type) through analysis of the containment hierarchy of the target metamodel, which is the Simulink metamodel.

Afterwards, the higher-order model transformation τ_2 of step 3.3 converts the matching model into a QVT-R transformation. A part of this synthesized transformation for the mapping between UML and Simulink is depicted in Listing 1. It shows the transformation code for different types of matchings, namely top, containment and class matchings. As the root element, the UML model is mapped to a Simulink model, contained UML Classes to Simulink Blocks, and the attributes of Classes to attributes of Blocks. In a similar manner – but not shown here due to space constraints – the value of each attribute is mapped, as well as Connectors and their attributes.

The computed mapping is 100% correct, but not complete, as not all elements are mapped. The missing mappings describe attributes, e.g., mapping the name attribute of the UML Class to the name attribute of Component or Block. Such missing attribute mappings concern only values and are relatively easy to add, since no other mappings depend on them. We evaluate the generated transformations with precision/recall metrics in Section VIII-B.

Listing 1. Synthesized QVT-R transformation from UML to Simulink

```
0  transformation uml22simulink2(source: uml2, target: simulink2) {

       --Top Matching
       top relation r_Model {
         checkonly domain source p : uml2::Model {
         };
5        enforce domain target s : simulink2::SimulinkModel {
         };
         where{
               r_Model_classes(p,s);
10             r_Model_connectors(p,s);
         }
       }

       --Containment Matching
15     relation r_Model_classes {
         checkonly domain source p : uml2::Model {
               classes = co: uml2::Class{
               }
         };
20       enforce domain target s : simulink2::SimulinkModel {
               elements = sb: simulink2::Block{
               }
         };
         where{
25             r_Class(co,sb);
         }
       }

       --Class Matching
30     relation r_Class {
         checkonly domain source co: uml2::Class{
         };
         enforce domain target sb: simulink2::Block{
         };
35       where{
               r_Class_name(co,sb);
               r_Class_ports(co,sb);
               r_Class_properties(co,sb);
         }
40     }
   [..]
}
```

VI. Step 4 - Integration of Generator Results

The generators in steps 2 and 3 produce ToolAdapters and DataChannels using different generative techniques. The generator for ToolAdapters uses code generation; the generator for DataChannels applies a matching algorithm to produce model transformations, which realize the conversion of tool data. Both generators use the ToolAdapter metamodels that are linked to the TIL model, but they use the metamodels in a different way.

The generator for ToolAdapters uses the tool metamodel as specification of the data managed by the ToolAdapter. The tool data is accessible as a model through the generated ToolAdapter and conforms to the tool metamodel.

The generator for transformations uses the metamodels of both ToolAdapters it connects to. Data-flow connections between ToolAdapters need to translate the tool data. The rules for the translation can be determined at designtime, since it is independent of the actual data, and only depends on the tool metamodels. As we have generated the ToolAdapters we know that they provide tool data that conforms to the tool metamodels.

The ToolAdapter metamodels are used as interface between the different generators to ensure compatibility between the generated artifacts.

VII. Step 5 - From Prototype to Production Software

The proposed approach promotes the iterative development of tool chains, where steps 1 - 4 can be repeated frequently to explore different what-if scenarios. This is supported by the generative approach, which produces executable source code and transformations automatically with only a small effort from the tool chain designer. The completely automatically generated source code works on test data and the automatically generated transformation might not be complete. While this level of accuracy might be sufficient for prototyping different what-if scenarios for tool chains, it needs to be improved for production software.

To create production software, the generated source code and transformation rules need to be manually adapted. The generated source code of the ToolAdapter needs to be extended to interact with the API of the integrated development tool, to extract and inject the data from the tool and to forward service calls to it. The generated model transformation needs to be refined, mainly by adding new transformation rules and less frequently by changing the generated transformation rules.

VIII. Evaluation

In this section, we intend to quantify to what extent a tool chain can be generated with the proposed approach. We separately evaluate the generator for tool adapters and for transformations.

A. Generator for ToolAdapters

The generator for ToolAdapters produces code skeletons and a complete prototype implementation, which serves test data. This prototype implementation needs to be manually replaced with code that serves the actual tool data using the API of the tool. The size of the code that needs to be manually added depends on the tool.

To show the effectiveness of the generator, it is not sufficient to compare the LOC of generated code with the manually added code. Instead, we create a baseline implementation completely manually. We now have two code bases realizing the same functionality. To quantify the generated and manually added code, we measure lines of code. This measurement has been criticized as a general measurement of software size for complete software, but here we apply it to fragments of code. We measure the lines of generated vs. manually implemented code for all ToolAdapters in the case study and present the measurements in Table III.

Table III
LOC of the ToolAdapters: generated and manually implemented code with the TIL approach vs. completely manually implemented baseline

ToolAdapter	TIL generated	TIL manually added	manual baseline
UML	1409	59	1313
Simulink	2030	1118	3077
Safety	3833	317	2359
Sum	7272	1494	6749
Percentage		22%	100%

The size of the generated code is not a significant indicator for the quality of the generated code. This is why we study the comparison of the manually added LOC (in step 5) with the LOC of the manual baseline. Both codes are manually created, realize the same functionality and their sizes can thus indirectly give clues about the quality of the generated code. On average, only 22% of the source code from the manual baseline needed to be implemented manually with the TIL approach.

B. Generator for Transformations

The generator for transformations introduced in Section V is a heuristics. It is the nature of heuristics to approximate the optimal solution and it cannot be guaranteed that the calculated result is the optimal solution. It is thus important to measure the quality of the results. In the following, we measure the quality of the results of applying our matching algorithm on a number of tool metamodels. The matching algorithm is based on two simplifying assumptions: (i) The transformation, which is part of the DataChannel between two ToolAdapters is intended to be semantics preserving. (ii) A semantics preserving transformation maps elements, which are similar regarding structure or naming. Assumption

 ISBN: 978-1-61208-230-1

Table IV
RELATIONSHIP BETWEEN SEMANTICS PRESERVATION AND STRUCTURAL/NAMING SIMILARITY

		semantics preservation	
		yes	no
structural/naming similarity	yes	1	2
	no	3	4

(i) is part of the semantics of the DataChannel in TIL (see Section III). In the following, we will evaluate assumption (ii). We analyze the relationship between semantics preservation and structural/naming similarity in Table IV and distinguish four situations.

Since assumption (ii) correlates semantics preservation with structural/naming similarity, the algorithm only distinguishes between situations 1 and 4 in Table IV. Situations 2 and 3 are in the "blind spot" of the algorithm, as semantics preservation may not be correlated with structural/naming similarity. The impact of situation 2 is measured by the precision metric, the impact of situation 3 is measured by the recall metric, which are defined as follows.

$$Precision = \frac{|\{correctmatches\} \cap \{foundmatches\}|}{|\{foundmatches\}|} \quad (1)$$

$$Recall = \frac{|\{correctmatches\} \cap \{foundmatches\}|}{|\{correctmatches\}|} \quad (2)$$

where $correctmatches$ is defined as the correct, semantics-preserving mapping, and $foundmatches$ is the mapping that was identified by the matching algorithm. We use the precision/recall measure and present statistics of the number of false positives and false negatives in the mappings.

We measure the quality of the calculated mappings of all six possible combinations between the three metamodels presented in Figure 4. The resulting precision/recall measurements are displayed in Table V. On average, the synthesis method returns mappings that have a high precision (93%), but only an average recall (56%).

Table V
PRECISION/RECALL METRIC FOR THE COMPUTED MAPPING OF UML, SIMULINK AND HIP-HOPS

Source	Target	Precision	Recall
UML	Simulink	1	0.56
UML	HiP-HOPS	0.89	0.53
Simulink	UML	1	0.67
Simulink	HiP-HOPS	0.9	0.6
HiP-HOPS	UML	0.89	0.53
HiP-HOPS	Simulink	0.9	0.5
Average		0.93	0.56

In situation 2, no mapping should be found since there is no semantic equivalent, but the algorithm finds a mapping due to structural similarity; this would result in a low precision. The measurements in Table V show a high precision metric. This means that the generated mappings are correct and only need to be changed seldomly. The mappings that need to be changed have a stable skeleton for manually added mappings.

In situation 3, there are semantically equivalent metamodels, for which no structural or naming similarity can be detected. If it is not possible to deduce clues about the semantic equivalence from the structural features of the metamodels, the automated algorithm does not have sufficient data to make mapping decisions. In this situation, either additional user data would need to be provided as input, e.g., via annotations, or the missing mappings need to be manually added after the algorithm is finished. Situation 3 is captured by the recall metric. The average recall metric in Table V is largely due to missing attribute mappings. Such attribute mappings concern only one value and are relatively easy to add manually, since no other mappings depend on them. An example for the transformation between UML and Simulink is the mapping (Class.name,Block.name).

Due to its high precision (93%) and average recall (56%) characteristic, the matching algorithm can be classified as a conservative method. The algorithm rather does not include a mapping into the result than produce a wrong mapping. The mappings that are included in the result are almost all correct, maximally one of the mappings is incorrect. The mappings that are not found automatically by the matching algorithm can be manually added to the result.

IX. RELATED WORK

The contribution of this paper is in the intersection of several fields, namely model-based tool integration, metamodel matching and rapid prototyping. Related work can be found in each of these fields. We list the approaches by fields and point out approaches that are in the intersection of two or more fields.

A. Tool Integration

Early work on tool integration focuses on identifying the scope of tool integration in form of aspects [29] and patterns [14]. A number of integration frameworks have been defined to support building tool chains, such as the one from Vanderbilt [13] and jETI [19]. Model-based integration frameworks focus on data integration, the other integration aspects (such as control, process, platform and presentation) defined by Wasserman [29] are excluded or a secondary issue. Examples are MOFLON [1] or ModelCVS [12]. These related approaches use metamodeling for describing the tool data. However, these approaches provide neither concepts to model a complete tool chain nor concepts to describe the architecture of the tool chain. The related approaches assume that tool data is available in the form of models and that the tool adapters are implemented manually. Only the MOFLON approach mentions code generation for tool adapters. The related approaches also use model transformations to translate between the metamodels of different tools, but the transformation usually has to be specified manually.

 ISBN: 978-1-61208-230-1

Tool integration platforms, such as ModelBus [11] or Jazz [9], mainly provide support for executing the tool chain, or generic building blocks, so constructing tailored, user-defined tool chains requires a lot of work. We automate the construction of such tailored tool chains.

B. Metamodel Matching

Matching metadata on data structures has been studied in the field of databases as schema matching [20], [26], [27]. These matching algorithms have been adopted in the modeling community, where metamodels are matched instead of schema definitions. Algorithms based on similarity flooding and naming similarity are described [6], [8]. Different metamodel matching algorithms are compared in [15] and formalized into a DSL for metamodel matching [10]. Model weaving approaches [6] can leverage metamodel matching to create weaving models that express the correspondence.

Del Fabro shows how metamodel matching can be applied for data migration between two bugtracking tools [6]. The approach assumes that the tool data is already available in a model format and focuses on the use of metamodel matching for weaving models. We use metamodel matching in an integration scenario and focus on a comprehensive approach for the creation of a complete tool chain.

C. Prototyping

Prototyping approaches focus on the early synthesis of an executable system from a high-level specification. Bernstein stresses the importance of prototyping [3] and lists advantages of the approach, among them he sees prototyping as a vehicle to better understanding the environment and the requirements, to validate requirements with the user and to study the dynamics of a system. We distinguish between throwaway and evolutionary prototyping [17]. In throwaway prototyping, the prototype is built to learn a specific thing and is discarded before a completely new prototype is built. In evolutionary prototyping, one prototype is refined over several iterations. The technology proposed in this paper can be used for either prototyping approach.

Many prototyping systems employ a prototyping language in combination with code generation techniques. These approaches are usually specialized to certain domains, such as CAPS and DCAPS [18] for embedded systems, information systems and user interfaces [30], component based systems [28] or data mining systems [21].

X. Future Work and Conclusion

The creation of tool chains is usually regarded as a completely manual implementation task. The presented approach shows that the tool chain implementation for a prototype can be automatically created with generative techniques. Different generative techniques need to be combined to produce the heterogeneous parts of the tool chain: code generation for ToolAdapters and a heuristic matching algorithm for transformations. The generated code for the ToolAdapters ensures compliance with standards and serves test data for prototyping.

This code is also the basis for production software as it provides a skeleton that needs to be refined with manually written code that interacts with the API of the integrated tool. In our case study the generated code for ToolAdapters makes up 78% of the total production software. The generated transformation code for the DataChannels provides a precise mapping for the data elements (93% precision), but does not cover all data elements (56% recall). Due to the conservative characteristic of the approach (high precision, average recall), the generated mapping can be be extended into a comprehensive mapping. The generated artifacts can serve as a starting point for manual extensions and refinements of the generated tool chain implementation.

The proposed approach for automated synthesis of both source code and transformations makes it possible to systematically and rapidly create an executable prototype of a tool chain. This allows the user to test and iteratively modify the tool chain prototype, before investing time to extend the prototype into the final production software.

An important next step is a further assessment of the practical applicability of this approach. We will apply our approach in additional case studies, which cover a broader set of development tools. This will allow us to further narrow down the conditions, in which the approach can achieve the best mapping, measured with the precision/recall metric. In addition, we will examine if the algorithm can also be applied to support the evolution of a tool chain, when tool A in the tool chain is exchanged against a similar tool B. The here presented algorithm might be applicable for realizing the migration of the data from tool A to tool B.

Acknowledgement

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement 100203.

References

[1] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08*, pp. 807–810, 2008.

[2] E. Armengaud, M. Biehl, Q. Bourrouilh, M. Breunig, S. Farfeleder, C. Hein, M. Oertel, A. Wallner, and M.s Zoier. Integrated tool chain for improving traceability during the development of automotive systems. In *ERTS2 2012 — Embedded Real Time Software and Systems*, pp. 30–46, 2012.

[3] L. Bernstein. Importance of software prototyping. *Journal of Systems Integration*, 6(1), pp. 9–14, 1996.

[4] M. Biehl, J. El-Khoury, F. Loiret, and M. Törngren, "On the Modeling and Generation of Service-Oriented Tool Chains," *Journal of Software and Systems Modeling*, vol. 0275, 2012.

 ISBN: 978-1-61208-230-1

[5] M. Biehl, J. El-Khoury, and M. Törngren. High-Level Specification and Code Generation for Service-Oriented Tool Adapters. In *Proceedings of the International Conference on Computational Science (ICCSA2012)*, pp. 35–42, Jun. 2012 [Online]. Available: http://dx.doi.org/10.1007/s10270-012-0275-7

[6] M. Del Fabro, J. Bézivin, and P. Valduriez. Model-Driven Tool Interoperability: An Application in Bug Tracking. In*On the Move to Meaningful Internet Systems 2006*, LNCS, vol. 4275, pp. 863–881, 2006.

[7] J. El-khoury, O. Redell, and M. Törngren. A Tool Integration Platform for Multi-Disciplinary Development. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 442–450, 2005.

[8] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Model Driven Engineering Languages and Systems*, LNCS, vol. 5301, pp. 326–340, 2008.

[9] R. Frost. Jazz and the Eclipse way of collaboration. *IEEE Software*, vol. 24, no. 6, pp. 114–117, 2007.

[10] K. Garcés, F. Jouault, Pi. Cointe, and J. Bézivin. A Domain Specific Language for Expressing Model Matching. In *Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09)*, pp. 30–45, 2009.

[11] C. Hein, T. Ritter, and M. Wagner. Model-Driven Tool Integration with ModelBus. In *Workshop Future Trends of Model-Driven Development*, pp. 1–12, 2009.

[12] E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, and M. Wimmer. On Models and Ontologies - A Layered Approach for Model-based Tool Integration. In *MOD2006*, pp. 11–27, 2006.

[13] G. Karsai and J. Gray. Component generation technology for semantic tool integration. vol. 4, pp. 491–499, 2000.

[14] G. Karsai, A. Lang, and S. Neema. Design patterns for open tool integration. *Software and Systems Modeling*, vol. 4, no. 2, pp. 157–170, 2005.

[15] L. Lafi, S. Issam, S. Hammoudi, and J. Feki. Comparison of two metamodel matching techniques. In *4th Workshop on Model-Driven Tool & Process Integration (MDTPI2011)*, pp. 54–65, 2011.

[16] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Doklady Akademii Nauk SSSR*, 163(4), pp. 845–848, 1965.

[17] V. Luqi, V. Berzins, M. Shing, R. Riehle, and J. Nogueira. Evolutionary Computer Aided Prototyping System (CAPS). In *Technology of Object-Oriented Languages and Systems*, pp. 363, 2000.

[18] V. Luqi, J. Berzins, J. Ge, M. Shing, M. Auguston, B. Bryant, and B. Kin. DCAPS-architecture for distributed computer aided prototyping system. In *Rapid System Prototyping, 12th International Workshop on*, pp. 103–108, 2001.

[19] T. Margaria, R. Nagel, and B. Steffen. jETI: A Tool for Remote Tool Integration Tools and Algorithms for the Construction and Analysis of Systems. In *TACAS*, LNCS, vol. 3440, pp. 557–562, 2005.

[20] S. Melnik, H. Garcia-molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm, 2002.

[21] I. Mierswa, M. Scholz, R. Klinkenberg, M. Wurst, and T. Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In *In Proceedings of the 12th ACM SIGKDD*, pp. 935–940, 2006.

[22] OMG. MOF Model to Text Language (MTL). Technical report, OMG, 2008.

[23] OMG. MOF 2.0 Query / View / Transformation. Technical report, OMG, 2009.

[24] OSLC Core Specification Workgroup. OSLC core specification version 2.0. Technical report, Open Services for Lifecycle Collaboration, 2010.

[25] Y. Papadopoulos and J. McDermid. Hierarchically Performed Hazard Origin and Propagation Studies. In *SAFECOMP*, LNCS, vol. 1698, pages 139–152, 1999.

[26] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[27] P. Shvaiko, J. Euzenat. A Survey of Schema-Based Matching Approaches Journal on Data Semantics IV. In *Journal on Data Semantics IV*, LNCS vol. 3730, pp. 146–171, 2005.

[28] M. Tkachuk, A. Zemlyanoy, and R. Gamzayev. Towards Prototyping-Based Technology for Adaptive Software Development Information Systems and e-Business Technologies. In LNBIP, vol. 5 , pp. 508–518, 2008.

[29] Anthony I. Wasserman. Tool Integration in Software Engineering Environments. In *Software Engineering Environments, International Workshop on Environments Proceedings*, LNCS, pp. 137–149, 1989.

[30] W. Zhou. A Rapid Prototyping System for Distributed Information System Applications. *Journal of Systems and Software*, vol. 24, no. 1, pp. 3–29, 1994.

 ISBN: 978-1-61208-230-1