MODELA DBT: Model-Driven Elaboration Language Applied to Dynamic Binary Translation

F. Salgado, A. Martins, D. Almeida, T. Gomes, J. Monteiro A. Tavares
Centro ALGORITMI, University of Minho, Portugal
Corresponding author: filipe.salgado@algoritmi.uminho.pt

***Abstract*—Industrial solutions design is a highly complex topic due to the challenge of integrating multiple technologies into a single solution, the inherent complexity of the problems to be solved and also because the proposed solutions often require a great level of interoperability among their components and also the outside world. Dynamic Binary Translation has been used as a tool to deal with such interoperability issues, e.g., legacy support, virtualization and secure execution, among others. However its integration in the industry as an end-product is hampered by the intricate variability management required in this subject. To address these issues and in an attempt to power DBT utilization as an interoperability-providing tool, we propose a model-driven DSL modeling language for DBT architectures. The developed DSL proved to be efficient to model an in-house DBT engine, and MODELA DBT, a framework for ready-to-use DBT solutions was obtained. MODELA DBT provides design validation, easy configuration of customizable DBT parameters and components, as well as code generation features.**

***Index Terms*—Domain-Specific Language (DSL), Dynamic Binary Translation (DBT), Model-driven Development, Generative Programming**

I. Introduction

Some of the big difficulties in todays' industrial solutions design is the high complexity of the systems and the variability management challenge [1] that the integration of multiple technologies poses. This is partially caused by the inherent complexity of the problems that need to be solved but also because the solutions often require a great level of interoperability between the system's elements and the outside world. A long known solution to deal with such interoperability issues is binary translation, which consists in a process of disassembling and translating source binary code into target binary code that will be executed for emulation purpose. It is classified as static binary translation which translates code offline and dynamic binary translation (DBT) which translates code at run-time. DBT may also provide system functionality expansion by adding new features which were not originally available (e.g., communication capabilities) in obsolete systems. In this IoT advent, DBT may become an emerging technology due to the interoperability it may provide to heterogeneous industrial environments, and also for its usability as a tool to support competitors products's firmware on a market race perspective [2,3]. In spite of being successfully deployed in general-purpose systems to support cross-ISA binary compatibility, dynamic optimization, profiling, virtualization, secure execution or debugging environment, in embedded systems, binary translation has been avoided mainly due to performance, memory and power overheads [3]–[5]. StrataX [4] is an existing DBT deployment for embedded devices, developed to evaluate the challenges on bringing DBT to such resource constrained systems. The authors state that in order to obtain satisfactory results, the shifting requires the application of numerous performance enhancing techniques, which highly depend on the target system characteristics.

However, DBT utilization as an end-product in the industry has been hampered by the complexity of the subject and its associated variability management, which brings configuration challenges into the final solution [5]. The accessible and profitable use of DBT requires design automation paradigms and variability management solutions, expanding its usage for DBT laymen. In this sense, a DBT framework must provide support for several source and target architectures, optional execution features (e.g., code profiling, dynamic optimization, etc.), resource utilization settings (e.g., memory size), design validation and consequent automatic code generation. In [6], Kondoh and Komatsu propose a specialization framework to generated host code to exploit a limited number of characteristics (MMU, bi-endianess and register banks). Their contribution however does not offer the necessary flexibility to be applied on different translators or to support other configurations than the ones provided. To achieve such type of tool, a robust base system model is required. This kind of model must comprise not only the components of the translator and their characterization, but also specify interfacing rules and foreknow the accepted variations of the model on a higher abstraction level that is subsequently lowered until reaching the implementation code files and the executable binaries. A DBT framework model is hard to describe using existing modeling languages (general-purpose, business, project management, software development), thus a domain-specific language (DSL) is required for such task. LLDSAL [7] and PACT:U [8] are two DSLs applied to DBT for automatic translation generation. Despite both approaches applying DSL in DBT, the modeling of the DBT architecture and the generation of translation for mismatching source/target pairs are not supported. This is due to their application scope, which is code instrumentation [8] and application security [7]. For the best of the authors knowledge, there are no existing DSL modeling solution or frameworks aimed towards industrial DBT application domain.

In this paper we propose a model-driven DSL modeling

978-1-5386-1127-2/17/$31.00 ©2017 IEEE

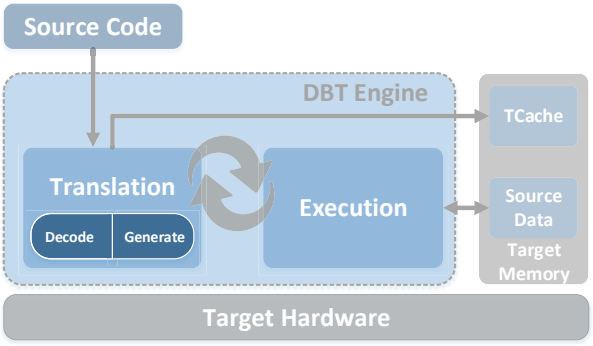

Figure 1. DBT architectural overview.

language for DBT architectures, aiming to improve complexity management, design validation and industry interoperability. Along with the DSL modeling language a framework named MODELA DBT was also developed, for ready-to-use DBT solutions, providing easy configuration and code generation features even for DBT laymen users. The remain of this paper is organized as follows: Section II introduces an essential DBT architecture; Section III presents the modeling Elaboration Language (EL) and the MODELA DBT framework; the DBT modeling process with the EL is explained in Section IV; Section V describes the implementation of the DBT system using the obtained framework and in Section VI and VII the work evaluation is performed and the conclusion is exposed, respectively.

II. DBT ARCHITECTURE

In this section a brief technical background is provided on DBT and the typical architecture of a dynamic binary translator is represented in Figure 1. In a functional approach, the execution flow is as follows: a source code is loaded into a source code cache (CCache). This cache exists for performance purposes, so that only the code fragment that is being translated is located on fast random access memory. The DBT engine, running on top of the target processor, fetches source binaries from the CCache, proceeds to its translation and stores the resulting instructions in a Translation cache (TCache). The code stored in TCache is then natively executed in the target architecture.

From a modeling perspective, the translator can be split into source-specific components, target-specific components and DBT engine-components. The DBT engine itself has two profiles: the Translation and the Execution. The Execution profile refers to the native execution of the translated code, while the Translation profile can be further split into the Decoder and the Generator sub-components. The Decoder, which is source ISA specific, is responsible for decoding the source binary into an Intermediate Representation (IR) for the Generator component. In its turn, the Generator, which is target ISA specific, is responsible for generating target machine code from the IR of the source binaries. The source binaries execution requires emulation support for architectural elements like the data and external memories, or the program counter and other function registers. This support is generally provided through allocated variables that emulate the source elements' behavior.

To evaluate DBT on embedded systems, an in-house re-sourceable and re-targettable dynamic binary translator was developed. The details of this translator are outside of the scope of this paper, however the obtained deployment will be used as the subject of this study.

III. ELABORATION LANGUAGE (EL)

A domain-specific language (DSL) is a programming language that targets a specific problem domain. A DSL should not provide features to solve every kinds of problems found in a certain domain, but instead should make it easier to deal with the problems of the domain it is specific for [9]. The usage of DSLs over general purpose languages (GPLs) is justified by several advantages such as gains on expressiveness on the target domain, ease of use, enhanced productivity, reliability, maintainability, easier reasoning and validation, and the direct involvement of domain experts [10]. The effort required to develop a DSL is however quite hard, as it requires a lot of technical experience and great understanding of the domain. Nonetheless, after its implementation, the development cost easily pays off [11]. The interest in DSLs for generative programming (GP) [12] and in model-driven development (MDD) [13] is becoming wider, as they promote software reuse and fast development through a high abstraction level. GP purpose is to automatically generate a system given a set of specifications [12]. MDD is an approach used to create extensive and descriptive system models on a higher implementation abstraction, thus simplifying development and testing activities [13]. Together, these two techniques promote software reutilization and automatic code generation, powering the DSL to map different models together and elaborate the final system code [14].

To pursue such goal, a model-driven DSL, named Elaboration Language (EL), was developed to automatically generate code from the source files of a given system's model. It is based on the Service-Component Architecture (SCA) standard, which specifies that a model's components should follow a composite pattern [15]. SCA features six key elements: composite, component, service, reference, property and wire. A complete reference architecture can be constructed by identifying system components and their interactions, as well as the properties associated to each component. The EL grammar was developed using Xtext [16], which is a framework for programming languages and domain-specific languages development, offering a full development infrastructure including parser, linker, type-checker and compiler [17]. An auxiliary tool, Xtend [18], was also used to implement the language validators, the code generation software and other additional features. Xtend is a flexible and expressive dialect of Java, which compiles into readable Java 5 compatible source code [19]. Both Xtend and Xtext are widely used for DSL development and integrable with Eclipse IDE [9]. The EL

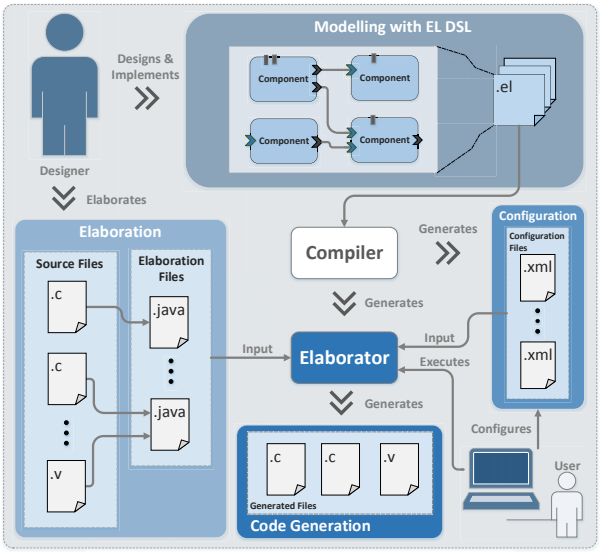

Figure 2. Elaboration Language framework workflow.

development will not be covered in this paper. To validate the EL's usability as a solution for embedded systems modeling a case study was elaborated on the modeling of a dynamic binary translator. The obtained framework was called MOdel-Driven Elaboration Language Applied to Dynamic Binary Translation - MODELA DBT.

A. *MODELA DBT - EL framework overview*

The EL framework workflow, shown in Figure 2, is composed by four main stages: modeling, elaboration, configuration and code generation. During each stage a set of artifacts is created and then used in the following stages. The modeling stage goal is to create the model that will be used as the system's reference architecture. In this stage every component must be identified, as well as their dependencies, properties, interfaces and relations with other components. This model must be described through the EL. Then, the model representation goes through the model compiler, which proceeds to syntactic and semantic validations. After a successful compilation, an architecture-specific Java Elaborator is generated, together with the XML configuration files, and abstract elaboration and Java classes for each component. The elaboration stage also includes the definition of the annotated source files and the implementation of the elaboration classes (which are based on the abstract elaboration classes). The elaboration files gather the information from the whole component's behavior and how the source code for that component must be generated. If there is more than one implementation available to a given component, the desired implementation should be specified in the correspondent configuration file. Only one elaboration class per component is executed by the Elaborator. The annotation process is eased by an API that was created to fetch the desired values from the configuration files and replace them within the source code files. During the configuration stage, it is possible to modify the component properties' values and the elaboration file that will be loaded into the Elaborator. It is also possible to have specific properties for each elaboration which are not presented in the reference architecture. Finally, the generated elaborator is executed. During this process, components' properties are fetched and the elaboration classes are loaded using Java reflexion.

B. *EL's Constructs*

As previously said, the EL follows a SCA, which means that an EL file contains the following constructs: interfaces, languages and components. An interface is a set of functions that implement a service provided by a component. Components can be connected through bindings of services and references that follow the same interface type. Every declared component must have an implementation language. The language used by these components should have, in addition to a name, an annotation section where the user defines a marker that will be used to specify annotations in the source files. A component can be composed by a set of other components, properties, references, services and a free section used to make assignments or promoting services and/or references. A component can also inherit another component. This operation transfers all the content of the inherited component to the top-level component. Each property may have its own restriction list or range, for value filtering purposes. Later, the specified properties will receive their values in the assignments section or at the code generation step, through the replacement of the established annotations. By the definition of a composite design pattern, a component can provide one or more services to other components. Since a service is implemented by interface functions, a component can not have more than one service on the same interface. A reference should be created only when a component requires to access a composite's component service. The `assignments` field allows the user to set the value of components properties. In the `promotes` section the user can promote references or services of a composite's component in order to use them in their top-level component. Furthermore, it is also possible to bind references to services in the `binds` section and to specify the top-level component on the hierarchy using the keyword `compile`. This specification also indicates the classes invocation order to the Elaborator program. Table I presents the available EL's keywords and their respective description.

IV. Modeling the DBT

The DBT system's model building was supported on the theoretical and implementation background on dynamic binary translation, obtained from the in-house DBT engine development. After a thoughtful analysis of the existent deployment, several components and interfaces were identified. Moreover, several configuration points were found and transposed to the model through properties. The end goal of the model was to automate the system configuration, to perform architecture validation and to generate the final source code for the end system.

Table I
AVAILABLE EL'S KEYWORDS

Keyword	*Description*
annotation	Defines the character that limits the annotations.
as	Renames a promoted reference or service.
bind	Binds a reference to a service.
compile	Tells to compiler which is the top level component.
component	Defines a component.
final	Defines that a component has a concrete elaboration.
import	Imports the content of the specified file.
interface	Defines a set of functions used by a service or pointed by a reference.
is	Inherits the specified component.
language	Defines a language.
promote	Promotes a reference or service from a composite's component to the top component.
properties	Defines the properties set of a component.
reference	Defines the reference used in a promote or in a bind operation.
references	Defines the references set of a component.
restrict	Restricts the values that a property can take to a user's defined set.
service	Defines the service used in a promote or in a bind operation.
to	Connects a reference to a service in a bind operation.

A. Reference Architecture

In Figure 3 is represented the reference architecture model for the DBT with a simplified representation of the most relevant interfaces. The model components are represented as blocks and the properties as black diamonds. The services are identified by gray polygons and references by blue polygons. Dashed lines are used to represent interfaces between components. The composite DBT is made by all the other components and composites of the dynamic binary translator reference architecture. Through its modeling, five main components were identified, all providing the main features of the system: CCache, TCache, Source Cluster, Target Cluster and DBT Engine. The Source Cluster is the composite that aggregates the software blocks associated with the source architecture, which are Source Environment (also composed by the data memory), Source Architecture and Decode. The Target Cluster is also a cluster of related software blocks, but in this case, related to the target architecture. It is composed by the Generator and Target Architecture components. The

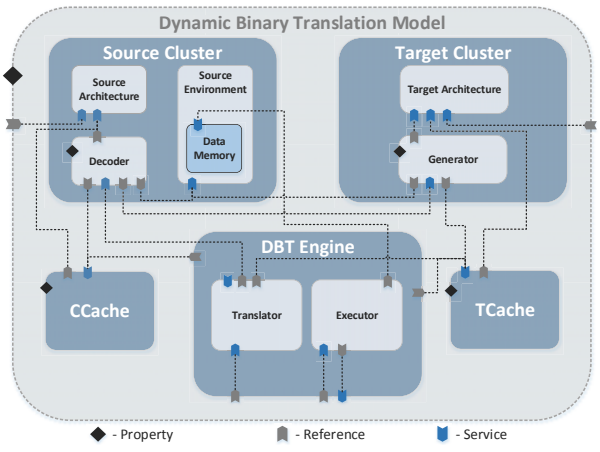

Figure 3. Reference Architecture.

DBT Engine represents the heart of the translator. It models the intermediary layer created in run-time that implements the translation of source code and execution of the target code. Therefore, it is a composite of other two components: Translator and Executor.

V. IMPLEMENTATION

A. Model Representation

Based on the DBT reference architecture, an EL code representation was created for all the components and interfaces. Listing 1 depicts the code representation of the `TranslationCache` component with its properties, references and services. The component's properties, `TCache_Size` and `cacheType`, have default values that can be later modified by the end user. These properties refer to the total memory size that will be allocated to store the translated code, and the type of eviction mechanism to be used when the TCache is full, respectively. The `TranslationCache` has a reference, `r_ISA`, to the target architecture to define the Translation Cache word size and the service `s_TCache` provided by the Translation Cache.

```
1  component TranslationCache(cpp)
2  {
3    properties:
4      int TCache_Size : 8192  // 20Kbytes
5      string cacheType : "fullFlush"
6    services:
7      i_TCache s_TCache
8    references:
9      i_ISA r_ISA
10 }
```

Listing 1. EL representation of Translation Cache component.

The implementation of this component is written in `C++` language, therefore a language type entity `cpp` was created, where the meta-characters used to annotate the source files are defined as `'@@'`, as exemplified in Listing 2. These meta-characters are used in the source code preceding and proceeding the unique identifier of an annotation that will be replaced by its respective value during the elaboration process of the final sources.

B. Elaboration

After the EL files being compiled, the elaboration files in Java and configuration files in XML are generated. In this phase the annotated source code files and configuration files are used in the elaboration process of the final files.

1) Configuration Files: The configuration files provide to the user the capability of modifying the default values of the component's properties and the system's behavior through the modification of the default elaboration files.

2) Annotated Sources: While building the model, the designer identifies several configuration points that should be annotated in the source code with the meta-characters defined in the EL. The Listing 2 shows the annotations

TCacheSize, cacheType and CC_type, that will be replaced by user defined values during the elaboration process. These values will replace the (@@TransCache_Size@@, @@TransCache_Type@@) and @@CC_type@@ annotations, respectively. TCacheSize and cacheType refer to the TCache available size to store translated code and the type of cache management used, respectivelly, while the CC_type expresses the method used in the DBT to emulate the source condition codes.

```
#define TCacheSize @@TransCache_Size@@
#define cacheType @@TransCache_Type@@
#define CC_type @@CC_type@@
```

Listing 2. Example of Annotations.

3) Elaboration Files: The specific elaboration files for each source file has the annotations that should be replaced and their corresponding value. The elaboration API provides methods to replace the annotations and generate the final files. The annotations that were previously explained are presented in the Listing 3 with the replacement methods.

```
openAnnotatedSource("defines.h");
replaceAnotation("TransCache_TSize",target.get_TCache_Size());
replaceAnotation("TransCache_Type",target.get_cacheType());
replaceAnotation("CC_type",target.get_CC_type());
```

Listing 3. Elaboration File.

In the header file it is necessary to change the annotation TransCache_Size, TransCache_Type and CC_type. These annotations should be replaced by the value of their respective properties, previously configure by the user. The elaboration file also contains methods that return the names of the implemented services specific for each elaboration.

VI. Evaluation

In order to demonstrate the contribution of the EL, a reference DBT architecture framework was created, the MODELA DBT. After the system's configuration, all source files are automatically generated and ready to be compiled. The reference model abstracts the component's implementation to the user but requires the system designer to specify the real implementation. An XML configuration file for the Translation Cache component is created, where SpecificTransCacheElabTemplate is specified as its implementation. The translation cache XML configuration file is shown in Listing 4. Every property requires a value as input, which can be specified by the end user or retrieved by its default value. Without these, it is not possible to generate the final DBT source code. The translation cache default size is 8192, but it was configured as 4096, so the second value will be used in the compilable source code. The cache type is left to default (full flush eviction). The generator component is also configured with the condition codes evaluation type, so the respective XML configuration file was created.

```
<component type="TranslationCache">
  <elaboration default="SpecificTransCacheElabTemplate">
    SpecificTransCacheElabTemplate
  </elaboration>
  <properties>
    <property type="int" name="TCache_Size" default="8192">
      <value>
        <element>4096</element>
      </value>
    </property>
    <property type="string" name="cacheType" default="fullFlush">
      <value>
        <element></element>
      </value>
    </property>
  </properties>
</component>
```

Listing 4. Translation Cache specific XML configuration file.

SpecificGeneratorElaborator is specified as the generator component's implementation. The generator XML configuration file is shown in Listing 5. The method for conditions codes emulation was modified from *lazy* to *standard evaluation*. The optimizations were also left to default and remained disable.

After generation, the DBT source code was compiled and the result deployed on a ARM Cortex-M3, integrated on a Microsemi's SmartFusion2 Advanced Development Kit. This evaluation module was used to run benchmark binaries from the BEEBS suite [20] compiled for the MCS-51 architecture. The output is shown in Figure 4, where the configuration inputs result can be seen. The translation cache size 4096 bytes (0x1000) and the STANDARD EV condition codes emulation settings are output to the console prior to the binaries execution.

```
<component type="Generator">
  <elaboration default="SpecificGeneratorElaboratorTemplate">
SpecificGeneratorElaborator
  </elaboration>
  <properties>
    <property type="bool" name="optimizations" default="false">
      <value>
        <element></element>
      </value>
    </property>
    <property type="string" name="conditionCodes" default="LAZY">
      <value>
        <element>STANDARD</element>
      </value>
    </property>
  </properties>
</component>
```

Listing 5. Specific XML configuration file.

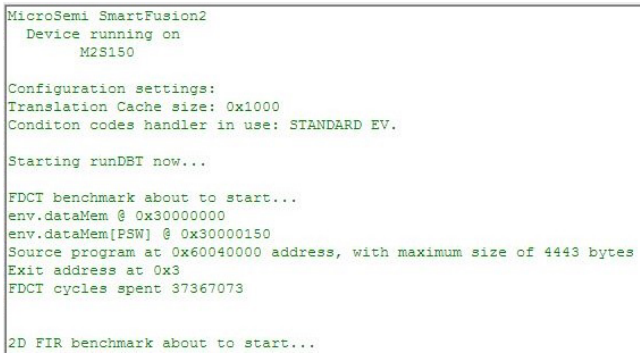

Figure 4. The compiled DBT source files run on the evaluation board, translating a program compiled for MCS-51 architecture.

VII. Conclusions and Future Work

This paper presented the MODELA DBT framework, the result of a modeling DSL for embedded systems that targets dynamic binary translation architectures. The developed DSL supported by the composite pattern was used to modeled a dynamic binary translator, creating a higher abstraction level for system description. The attained functionality demonstrate that despite a DSL being hard to create, its potential contribution to the modeling, configuration and code generation of a certain domain tend to pay off, achieving higher productivity, lower development time and ease of use. The proposed framework, MODELA DBT, aims to aid design automation, decreasing configuration efforts and promoting the use of DBT techonology in the industry as a ready-to-use solution. To demonstrate the proposed contributions, an in-house DBT deployment for embedded systems was modeled using the EL and integrated into the MODELA DBT framework. The translator was configured through the framework at two levels, the first one comprising the data support components size (i.e., TCache size) and the second one at a functional level, with the configuration of the CC emulation method. The configured translator's end source files were obtained from the framework without manually changing the source code, neither adding any code overhead, and used to run the DBT system on a development kit.

Further testing and validation of the framework depend on broader configuration options offered by the subject DBT system. In the future, it is planned to improve the MODELA DBT by introducing a graphic user interface (GUI) to ease system configuration and reduce the number of code generation steps. Currently, the DSL semantics are also being improved, with the usage of semantic technology to describe the domain knowledge. A semantically enhanced DSL will improve the model validation and reduce the elaboration development efforts. This will contribute to greater system scalability, configuration granularity, code generation efficiency and design verification.

VIII. Acknowledgments

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - *Fundação para a Ciência e Tecnologia* within the Project Scope: UID/CEC/00319/2013.

F. Salgado is supported by FCT (grant SFRH/BD/81681/2011).

References

[1] N. Cardoso, P. Rodrigues, J. Vale, P. Garcia, P. Cardoso, J. Monteiro, J. Cabral, J. Mendes, M. Ekpanyapong, and A. Tavares, "A generative-oriented model-driven design environment for customizable video surveillance systems," *EURASIP Journal on Embedded Systems*, vol. 2012, no. 1, p. 7, dec 2012.

[2] Qualcomm Technologies, Inc., "Enabling the Next Mobile Computing Revolution with Highly Integrated ARMv8-A based SoCs," Tech. Rep., 2014.

[3] G. Ottoni, T. Hartin, C. Weaver, J. Brandt, B. Kuttanna, and H. Wang, "Harmonia," in *Proceedings of the 8th ACM International Conference on Computing Frontiers - CF '11*. New York, New York, USA: ACM Press, 2011, p. 1.

[4] J. a. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Enabling dynamic binary translation in embedded systems with scratchpad memory," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 4, pp. 1–33, dec 2012.

[5] W. Chen, Z. Wang, and D. Chen, "An emulator for executing IA-32 applications on ARM-based systems," *Journal of Computers*, vol. 5, no. 7, pp. 1133–1141, 2010.

[6] G. Kondoh and H. Komatsu, "Dynamic binary translation specialized for embedded systems," *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '10*, p. 157, 2010.

[7] M. Payer, B. Bluntschli, and T. R. Gross, "LLDSAL," in *Proceedings of the seventh workshop on Domain-Specific Aspect Languages - DSAL '12*. New York, New York, USA: ACM Press, 2012, p. 15.

[8] S. Makarov, A. D. Brown, and A. Goel, "An Event-Based Language for Dynamic Binary Translation Frame Works," *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 499–500, 2014.

[9] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[10] M. Voelter, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*, 2010-2013.

[11] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[12] K. Czarnecki, *Overview of Generative Software Development*, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[13] M. V. Thomas Stahl, *Model-Driven Software Development*. John Wiley & Sons, Ltd, 2006.

[14] J.-P. Tolvanen and M. Rossi, "MetaEdit+: Defining and Using Domain-specific Modeling Languages and Code Generators," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 92–93.

[15] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*, ser. Independent Technology Guides. Pearson Education, 2009.

[16] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, and S. Zarnekow, "Xtext User Guide," http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf, 2010, accessed: 2017-03-11.

[17] "Xtext - Language Engineering Made Easy!" https://www.eclipse.org/Xtext/, accessed: 2017-03-11.

[18] "Xtend introduction," http://www.eclipse.org/xtend/documentation/, accessed: 2017-03-11.

[19] "Xtend - Modernized Java," https://www.eclipse.org/xtend/, accessed: 2017-03-11.

[20] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," pp. 1—-12, 2013.