Abstracts

of the

1st International Joint Symposium on Program and Model Transformations,

PAMT 2014

Anya Helene Bagge & Yngve Lamo (eds.)
Contents

Rogardt Heldal:
   Challenges and Opportunities of Model Transformations in Industry  1

Wendy MacCaull:
   The Need for Multimodelling: a Healthcare Perspective  1

Piotr Kamiarczak, Truls Pedersen and Thomas Gotnes:
   NorMC: a Norm Compliance Temporal Logic Model Checker  2

Kent Inge Fagerland Simonsen and Lars Kristensen:

Ekkart Kindler:
   Coordinating Interactions: The Event Coordination Notation  5

Fazle Rabbi and Yngve Lamo:
   Metamodel integration: A formal approach to system integration  7

Xiaoliang Wang:
   Verification of model transformations using Alloy  10

Eivind Jahren and Anya Bagge:
   Applying abstract interpretation with SMT solvers to compiler optimization  12

Lukas Härtel, Johannes Härtel and Ralf Lämmel:
   Test-data generation for Xtext  12
Gabriele Taentzer:
On the relation of meta-modeling and grammars 12

Volker Stolz:
Model querying with graphical notation of QVT relations 13

Ralf Laemmel and Marcel Heinz:
A classroom approach to DSML implementation 14

Florian Mantz:
Formalizing Model Migration Approaches by Algebraic Graph Transformations 14

Uwe Egbert Wolter:
Structure and behaviour – dual aspects of systems 18

Adrian Rutle:
A World of Models: Structure and Behavior 18

Tero Hasu:
Exploiting Macros in Source-to-Source Compiler Implementation 18

Admar Ajith Kumar Somappa and Kent Inge Fagerland Simonsen:
Model-Transformations to Assist in Protocol Design for Wireless Sensor Actuator Networks 21
Challenges and Opportunities of Model Transformations in Industry

Rogardt Heldal

Abstract. Model Driven Engineering (MDE) has large potential do increase productivity and improve quality of software. However, to achieve this automation in form of good transformation is required. In industry this often mean transformation from models to code. In this talk I will share the results from a study focused on collecting experiences regarding working with transformations from two large international companies. The main finding indicates that transformations need to be agile, to be able to modify transformations when needed according to new syntax and platforms. This is not the case today: transformation can be hard to modify due to complicated languages and structures of the code. Moreover, in some cases only tool vendors can modify the transformations or are allowed to create new ones. This is particular problematic for new transformations which might have buggy code or create too slow code. Unfortunately, the solution is often to change the code directly which is not in the line with the aims of MDE.

The Need for Multimodelling: a Healthcare Perspective

Wendy MacCaull

Abstract. Rising costs, ageing populations and increased expectations are making the current healthcare systems unsustainable. Information technology has the potential to support healthcare but its application has not nearly reached its full potential. Barriers include the lack of interoperability even among systems in one hospital, the high degree of customization required to serve local conditions, the fact that change is constant in healthcare settings, the fact that healthcare is a complex process involving many players in a variety of ways; the fact that data and information is complex and siloed, and the fact that software engineering needs strategies to ensure that systems for such safety critical applications behave correctly. Based on the lessons learned from a multi-year project at St
Francis Xavier University, I will briefly outline several scientific problems underlying the development of flexible, intelligent, and failsafe information technology suited to healthcare, and argue that MDE, vis a vis a multimodelling framework, can address challenges in software engineering relating to the productivity, flexibility and reliability of healthcare systems.

NorMC: a Norm Compliance Temporal Logic Model Checker

Piotr Kaźmierczak, Truls Pedersen and Thomas Ågotnes

Abstract. We describe NorMC, a model checker for Norm Compliance CTL, a temporal logic for reasoning about compliance in normative systems, implemented in the Haskell programming language. NorMC is intended as a tool for students, researchers, and practitioners to learn about and understand normative systems, and as an exploratory tool for researchers in multi-agent systems. The objectives of the paper are twofold. First, to give a system description of NorMC. Second, to argue and demonstrate that the Haskell programming language is a natural and useful alternative for model checking multi-agent systems; in particular that the full power of Haskell makes it easy to describe arbitrary multi-agent state-transition models in a natural way.
A Pragmatic Approach for Transforming Coloured Petri Net Models into Code:
A Case Study of the IETF WebSocket Protocol

Kent Inge Fagerland Simonsen1,2 and Lars Michael Kristensen1

1 Department of Computing, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
2 DTU Compute, Technical University of Denmark, Denmark
Email: {kisi}@dtu.dk

Extended Abstract

The vast majority of software systems today can be characterised as concurrent and distributed systems as their operation inherently relies on protocols executed between independently scheduled software components and applications. The engineering of correct protocols can be a challenging task due to their complex behaviour which may result in subtle errors if not carefully designed. Ensuring interoperability between independently made implementations is also challenging due to ambiguous protocol specifications. Model-based software engineering offers several attractive benefits for the implementation of protocols, including automated code generation for different platforms from design-level models. Furthermore, the use of formal modelling in combination with verification and model checking provides techniques to support the development of reliable protocol implementations.

Coloured Petri Nets (CPNs) [3] is formal language combining Petri Nets with a programming language to obtain a modelling language that scales to large systems. In CPNs, Petri Nets provide the primitives for modelling concurrency and synchronisation while the Standard ML programming language provides the primitives for modelling data and data manipulation. CPNs have been successfully applied for the modelling and verification of many protocols, including Internet protocols such as the TCP, DCCP, and DYMO protocols [1, 4]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols considered and have resulted in improved protocol specifications. However, earlier work has not fully leveraged the investment in modelling by also taking the step to automated code generation as a way to obtain an implementation of the protocol under consideration.

In earlier work [5], we have proposed the PetriCode approach and developed a supporting software tool [7] for automatically generating protocol implementations based on CPN models. The basic idea of the approach is to enforce particular modelling patterns and annotate the CPN models with code generation pragmatics. The pragmatics are bound to code generation templates and used to direct a template-based model-to-text transformation that generates the protocol implementation. As part of earlier work, we have demonstrated the use
of the PetriCode approach on small protocols. In addition, we have shown that
our approach supports code generation for multiple platforms, and that it leads
to code that is readable and also compatible with other software [6].

In the present work we consider the application of our code generation ap-
proach as implemented in the PetriCode tool to obtain protocol software im-
plementing the IETF WebSocket protocol [2] protocol for the Groovy language
and platform. This demonstrates that our approach and tool scales to industrial-
sized protocols. The WebSocket protocol is a relatively new protocol developed
by the IETF. The WebSocket protocol makes it possible to upgrade an HTTP
connection to an efficient message-based full-duplex connection. WebSocket has
already become a popular protocol for several web-based applications such as
games and media streaming services where bi-directional communication with
low latency is needed.

The contributions of our work include showing how we have been able to
model the WebSocket protocol following the PetriCode modelling conventions.
Furthermore, we perform formal verification of the CPN model prior to code gen-
eration, and test the implementation for interoperability against the Autobahn
WebSocket test-suite [8] resulting in 97% and 99% success rate for the client and
server implementation, respectively. The tests show that the cause of test fail-
ures were mostly due to local and trivial errors in newly written code-generation
templates, and not related to the overall logical operation of the protocol as
specified by the CPN model. Finally, we demonstrate in this paper that the
generated code is interoperable with other WebSocket implementations.

References

col Verification. In Lectures on Concurrency and Petri Nets, volume 3098 of Lecture
   modelling and validation of concurrent systems. International Journal on Software
   Functional Validation of Protocol Designs. In Transactions on Petri Nets and Other
   from CPN Models Annotated with Pragmatics. In Formal Methods: Foundations
   Models. In SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok,
   MoKMaSD, and OpenCert, volume 8368 of LNCS, pages 151–166. Springer.
Coordinating Interactions: 
The Event Coordination Notation
– Abstract –

Ekkart Kindler

DTU Compute
Technical University of Denmark
ekki@dtu.dk

Abstract. The Event Coordination Notation (ECNO) allows modelling the behaviour of a software system on a high level of abstraction that is close to the application’s domain. Still, these models contain all necessary details for generating fully functioning software from these models. A tool that supports all modelling concepts of ECNO and is based on Eclipse and EMF has recently been published. Along with this ECNO Tool, a technical report was published, which discusses the objective and motivation of ECNO, its concepts and notations, some of its design choices, as well as the use of the ECNO Tool and its programming framework.

The purpose of this extended abstract is to motivate you to have a look into the technical report and get started with some of the examples, which are deployed together with the ECNO Tool.

Keywords: Model-based Software Engineering, Event coordination, Interactions, ECNO Tool.

ECNO: Motivation

The purpose of a domain model is to concisely capture the concepts of an application’s domain, and their relation among each other. Even though the main purpose of domain models is not on implementing the application, major parts of an application can be generated from the application’s domain models fully automatically with today’s technologies. The focus of today’s code generation technologies, however, is mostly on the structural aspects of the domain; the domain’s behaviour is often not modelled at all, or implemented manually based on some informal models, or the behaviour is modelled on a much more technical level.

The Event Coordination Notation (ECNO) allows modelling the behaviour of an application on a high level of abstraction that is closer to the application’s domain than to the software realizing it. Still, these models contain all necessary details for actually executing the models and for generating code from them.

In order to be able to model the behaviour of a domain, the ECNO makes the events in which the different elements of the domain could engage explicit.
The local behaviour of an element defines at which time an element can engage or participate in an event. The global behaviour of the application results from different elements jointly engaging in such events, which is called an interaction. Which events are supposed to be jointly executed and which elements need to join in is defined by so-called coordination diagrams of the ECNO. Together, the models for the local and the global behaviour define the overall behaviour of the domain.

**ECNO: more information**

Many more details as well as instructions on how to obtain and install the ECNO tool are available at the ECNO Homepage: http://www2.imm.dtu.dk/~ekki/projects/ECNO/.

In particular, a detailed technical report on ECNO [1] is available from the ECNO Homepage. In this technical report, we discuss the main idea and philosophy of ECNO and its notation as well as all the subtle details and concepts – and we motivate the decisions made for its design. Moreover, we discuss the prototypical implementation of ECNO, which consists of a modelling environment based on Eclipse and the Eclipse Modeling Framework (EMF) and an execution engine, which fully supports all the concepts and features of ECNO discussed in the technical report.

All the examples deployed with the ECNO Tool are based on EMF, but the ECNO Engine can also be used with different other platforms or object-oriented code across different platforms, once some adapters are provided.

**References**

Metamodel integration: A formal approach to system integration

Fazle Rabbi and Yngve Lamo

1 Bergen University College, Bergen, Norway.
Fazle.Rabbi@hib.no

2 Bergen University College, Bergen, Norway.
Yngve.Lamo@hib.no

1 Introduction

Modern programming languages such as C++, Java, and Python raises the level of abstraction of programming languages and techniques. Although these high level programming languages made the software development process faster and less error prone, they do not provide sufficiently powerful support for analysis and verification. One of the latest steps has led to the usage of models in the developing process. Models of a system not only raises the abstraction level but also facilitates the use of analysis techniques which are essential to build reliable systems. Modelling languages may be categorized in some major categories, such as entity designing, concurrent system modelling, workflow modelling. For entity designing purpose ER diagram [5], UML class and object diagrams [11] are well known languages. Petri net [7], Kripke structure [3], UML sequence diagram, Activity diagram [11], and State machines may be considered as modelling languages for designing concurrent and/or distributed systems. There are several workflow and process modelling languages such as YAWL [14], BPMN [16], CWML [9], DERF [13]. Tools have been developed to produce models with these modelling languages; some tools are also equipped with analysis and verification techniques. For example the Petri net language is being supported by many tools and available for both commercial and research use [2].

Every modelling language has their own grammar for syntax and semantics. Using metamodelling for defining a modelling language is a new approach of modelling where a metamodel defines the abstract syntax of a modelling language. The abstract syntax (i.e., a metamodel) describes the set of modelling concepts, their attributes, their relationships, and the rules for combining these concepts to specify valid models [1]. This formal approach for precisely defining a modelling language has been studied in DPF and a formalisation of concepts in MDE has been proposed based on category theory and graph transformation [12] [10]. A metamodelling language may be recursively defined by a modelling language (see Fig. 1) where a specific model serve as the corresponding metamodel of another modelling language.

The DPF framework may be used to develop different modelling languages having different metamodeling hierarchies. A workflow modelling language called DERF [13] was developed and afterwards it was extended in [15] to define (static) semantics for timed and compensable workflow models and defined the dynamic semantics of models by a transition system where the states are instances and transitions are applications of transformation rules. An important observation of software systems is that very often they have different aspects such as user access, process flow, notification systems. In [8] we introduced the use of multiple metamodelling for the user access modelling, health process, modelling process monitoring, user interface modelling, and modelling of the data sources, and integrate them by using morphisms (see Fig. 2). The directed arcs from one metamodel to another in Fig. 2 represent the bindings...
between metamodels. Using separate metamodels for modeling different aspects of a system gives us the flexibility for remodelling and also makes models more readable.

Currently we are investigating how an integrated system model may be translated to an algebraic specification to realize the static and dynamic semantics of DPF specifications and the integration of multiple metamodels. Although other executable systems may be used, to simulate the software model designed in DPF we propose the use of Maude system [4, 6] which is a high–level programming/specification/modeling language based on rewriting logic theory.

References


Verification of model transformations using Alloy

Xiaoliang Wang

Bergen University College

xwa@hib.no

In model driven engineering (MDE), models are the basis for software development. They are used to specify the domain under study, to generate program code, to document software, etc. As a program should conform to its programming language, every model should conform to a metamodel which defines the syntax and semantics of a certain set of models. In each phase of a software development process, models are specified according to various metamodels for different purposes. Ideally, models in the next development phase can be generated automatically from models in the current development phase with use of model transformations. According the definition in [4], model transformations translate a source model into a target model according to a transformation definition. The transformation definition, a set of transformation rules, specifies how artefacts in a source metamodel are transformed into artefacts in target one. The source and the target metamodel may be different. As stated in [2], model transformation is similar to program transformation. "While program transformation systems are typically based on mathematically oriented concepts such as term rewriting, attribute grammars, and functional programming, model transformation systems usually adopt an object-oriented approach for representing and manipulating their subject models." The automation of model transformation makes MDE appealing by increasing productivity. However, the quality of the model transformations should be ensured. Otherwise, errors in model transformations may be propagated to subsequent phases and may cause erroneous models and software at the end. Thus, verification of model transformations is a necessary task to ensure correctness, i.e., the produced models or the transformations satisfy some expected properties.

In this work, we present our approach to the verification of graph-based model transformations [3] using Alloy [1]. The idea is to encode model transformations automatically as an Alloy specification. Then the specification is analyzed by the Alloy Analyzer to verify several kinds of properties. Each component, metamodel (including structure and constraints) and model transformation rules, can be encoded in relational logic. The structure of a metamodel is encoded as functions and predicates, representing all the possible model instances typed by the graph. Besides, the semantics of model transformations, or how to execute a transformation, is also considered in the approach. In each direct model transformation, according to the applied rule and the DPO approach, some elements in the source model are deleted while some elements in the target model are added. Except those elements the rest of the source model is preserved in the target model. In this way, a direct model transformation can be encoded as the following two functions \( \text{add} : T \rightarrow S \) and \( \text{delete} : S \rightarrow T \):

\[
\text{add}(e) = \begin{cases} 
NULL & \text{if } e \text{ is added to } T \\
e & \text{otherwise}
\end{cases}
\]

\[
\text{delete}(e) = \begin{cases} 
NULL & \text{if } e \text{ is deleted from } S \\
e & \text{otherwise}
\end{cases}
\]

After the model transformations are encoded as Alloy specification, we use the Alloy Analyzer to verify several kinds of properties of transformations by analyzing the Alloy specification. We are concerned with three kinds of properties:

**Conformance** A model should conform to its metamodel. Model transformations should preserve conformance, i.e., given a valid source model, every sequence of direct model transformations from the model can produce a valid target model. This is performed by checking if the
system satisfy the Direct Condition, i.e., every direct model transformation is valid. A direct model transformation is valid if it produces a valid target model from a valid source model. We verify conformance by finding counterexample, i.e., any invalid target model is produced from a valid source model. This could be also used to verify safety properties which is requested to be satisfied all the time.

Reachability The Direct Condition is quite strong in the sense that, in a sequence of direct model transformation, no invalid intermediate model is allowed. However, some systems accept such intermediate models and only require that the final target model should be valid. In such a situation, we weaken the correctness condition by checking the Sequential Condition, i.e., for each counterexample \( S \rightarrow T_0 \), a sequence of direct model transformations \( T_0 \rightarrow \cdots \rightarrow T_n \) can produce a valid target model \( T_n \). If the condition is satisfied, we can assure that, given any valid source model, a valid target model can be produced after some sequences of direct model transformations. It is weaker since it does not promise to produce a valid target model after every sequence of direct model transformations.

Liveness The liveness property generally requires that something eventually happens. Given a liveness property \( p \), if for each possible sequence of transformations \( M_0 \rightarrow M_1 \cdots \rightarrow M_k \), some model \( M_i (i \in \{0..k\}) \) satisfies \( p \), the system satisfies the liveness property \( p \). In the approach, we consider the negation, i.e., we try to find a loop sequence. That is, a sequence of transformations \( M_0 \rightarrow M_1 \cdots \rightarrow M_k \) where \( M_k \) equals to one of the preceding models \( M_i (i \in \{0..k\}) \) and no model on the sequence satisfies the property. If we can find such a sequence, we say that the liveness property is not satisfied. Otherwise, the search continues for length \( k + 1 \) until the length reaches a user-defined limit \( K \).

With the approach, we can find defects in model transformation definition and use the feedback given by the Alloy Analyzer tool to revise the model transformations. However, since the Alloy Analyzer performs check with a user-defined scope, the approach is incomplete and bounded. Besides, the Alloy is based on first-order relational logic, the verification ability is highly related with the relations in the metamodel and the transformation rules. For systems with relation of high arity, the approach is not applicable. In the future work, we may use other solver, like SMT solver, to solve the problem.

References

Applying abstract interpretation with SMT solvers to compiler optimization

Eivind Jahren and Anya Bagge

Abstract. Similar to program verification, producing the optimal code for a given program is undecidable. However, unlike compiler optimization where ad-hoc rules guides optimization, program verification often rely on general solver routines such as SMT solvers, enabling the programmer to hypothesize about the program’s behaviour. We explore the possibility of incorporating well-known techniques from program verification into compiler optimization, that permit optimizations not only based on rigid program invariants, but also based on assumptions (axioms) made by the programmer.

Test-data generation for Xtext

Lukas Härtel, Johannes Härtel and Ralf Lämmel

Abstract. We describe a method and a corresponding tool for grammar-based test-data generation. The method is inspired by existing work on controlled combinatorial test-data generation. The method breaks down generation into two phases. The first phase returns skeletons of data to be post-processed by the second phase, which in turn leverages different kinds of iterators to instantiate the placeholders in the skeletons. A DSL for grammar transformation is used to actually transform a given grammar, meant for parsing, into one serving for test-data generation. The implementation is tailored to Xtext.
On the relation of meta-modeling and grammars

Gabriele Taentzer

Abstract. Meta modeling is a wide-spread technique to define visual languages, with the UML being the most prominent one. Despite several advantages of meta-modeling such as ease of use, the meta-modeling approach has one disadvantage: it is not constructive, i.e., it does not offer a direct means to generate meta-model instances. This disadvantage poses a severe limitation on certain applications. For example, when developing model transformations, it is desirable to have enough valid instance models available for large-scale testing. Producing such a large set by hand is tedious. In the related problem of compiler testing, a string grammar together with a simple generation algorithm is typically used to produce words of a given language automatically. In the context of meta-models, we need model grammars to generate instances and thereby to overcome the main deficit of the meta-modeling approach for defining modeling languages. A model grammar has to cope with model structures as well as constraints.

An approach is presented that allows to translate a meta-model to a so-called meta-graph, i.e., a type graph with a set of graph constraints, and further a meta-graph to a graph grammar which can be directly used to deduce a model grammar. Having clarified the big picture, this talk focuses on the translation of Core OCL invariants to graph constraints and further on the deduction of grammar rules.

Model querying with graphical notation of QVT relations

Volker Stolz

Abstract. As a standard high-level model transformation language, QVT Relations defines a graphical notation, which provides a concise, intuitive way to specify transformations. However, QVT Relations relies only on the textual language OCL for model querying, leading to verbose and complicated OCL
expressions. Here, we present a graphical model query facility based on the checking semantics and pattern matching of QVT Relations. The query facility also borrows from QVT Relations the graphical notation. In addition we propose an approach to map the queries into XSLT to facilitate their execution. We have developed a tool for designing the queries and automatically generating the XSLT programs.

This is joint work with LI Dan and LI Xiaoshan, published in the UML&FM workshop 2012.

A classroom approach to DSML implementation

Ralf Laemmel and Marcel Heinz

Abstract. We report on our experiences with a design for a course on Software Language Engineering (SLE). Specifically, we discuss in more detail a course assignment which aims at the implementation of a Domain Specific Modeling Language (FSML). This assignment establishes competences with regard to parsing, interpretation, constraint checking, and code generation. An important aspect is here that the students have to research different technologies and come up with proper arguments as to why and how they are going to use their platform of choice. To give an example of an assignment solution, we present one that involves model-driven engineering: it uses TGraphs and JGraLab.
Formalizing Model Migration Approaches by Algebraic Graph Transformations

Florian Mantz
Bergen University College
fma@hib.no

Model-driven engineering [5] (MDE) is a software engineering discipline which employs models as primary artifacts in the software development process. Model-transformations are used to automate recurring development tasks as well as to generate software artifacts for different runtime environments and testing. In MDE modeling languages evolve often. This implies (usually) that models need to be migrated correspondingly (see. Fig. 1). Since this challenge is time-consuming and error-prone, it has become a hot topic in MDE research [18, 17, 2, 7, 15].

\[
\begin{array}{c}
\text{Modeling Language} \\
\Downarrow \text{evolution} \\
\text{Model} \\
\Downarrow \text{migration} \\
\rightarrow \\
\text{Model'} \\
\text{conforms to} \\
\text{conforms to}
\end{array}
\]

Figure 1: Model co-evolution: Modeling language evolution and model migration

In our work we consider modeling languages that are based on graphs and described by class models (such as in the Object Management Group (OMG) standard [14]). Models of such “meta-models” are described by corresponding object models (abstract syntax) and visualized as diagrams in which elements of the object model are presented by shapes\(^1\) (concrete syntax). To migrate models three different kind of migration approaches appear in related work [15]:

- **Manual specification approaches** [17, 10, 15] consider two meta-model versions as given and migrate models by copying elements from the previous model version to the new one. Elements automatically copied are those that have unchanged or compatibly changed types. New elements of a meta-model have to be taken into account in manually defined migration specifications.

- In **operator-based** approaches [7, 21], a meta-model is evolved using pre-defined operators. The evolution history is tracked as a sequence of changes. Usually, a library of coupled evolution-migration operators is supported to stepwise evolve meta-models and migrate models accordingly.

- **Meta-model matching** approaches [3, 6] consider two versions of a meta-model as given. An evolution history i.e. a sequence of evolution steps, is (semi-)automatically derived from the difference of two meta-model versions. Afterwards, all detected meta-model evolution steps are (semi-)automatically mapped to predefined migration operations.

It is a fact that MDE is still in its infancy and many techniques and tools are not always mature. Therefore, we develop in our work theory to support MDE. In particular, we focus on the problem of model migration. In our previous work [20, 12, 11, 13, 19] we have contributed a

\(^{1}\)Such that an object diagram is e.g. presented as Petri Net [8].
A formal framework for model migration which can be used to formalize all three kinds of model migration approaches above.

The framework builds on algebraic graph transformations which have been successfully used earlier as theoretical foundation of model transformation [16, 1]. In particular we formalize modeling language evolution steps and corresponding model migration steps as coupled transformations. In our work graph transformations operate in weak adhesive High-Level-Replacement (HLR) [9, 4] instead of classical (directed-multi) graphs. This makes it flexible to work with different kinds of graph formalizations (e.g. graph formalizations supporting attributes or inheritance [4]).

In the talk core results of this work are summarized and related to existing kinds of model migration approaches (above). Figure 2 (a) shows the core tasks which have been identified to formalize model migration approaches. Figure 2 (b) sketch the developed framework and relates it to identified tasks above. Figure 2 (c) relates the developed framework to exiting formal frameworks it is based on.

Figure 2: Overview coupled transformation framework

References


2In previous work also called co-transformations
Formalizing Model Migration Approaches

Florian Mantz


Structure and behaviour – dual aspects of systems

Uwe Egbert Wolter

Abstract. The talk is not about transformations but presents and discusses more informally the crucial observation that structure and behaviour are dual aspects of the systems we are modelling.

This observation becomes highly relevant when we combine models for structure and behaviour, respectively, and consider transformations for those combined models.

A World of Models: Structure and Behavior

Adrian Rutle

Abstract. In software engineering, models represent real world objects and prescribe software which may provide functionalities for various systems such as banking, public management, healthcare, etc. These models function as a link between the description of the real world and the implementation of the software systems. Model-driven engineering strives to formalize this link and create best practices in delivering high quality software out of models. In this paper, I will first briefly outline software modeling, especially in the view of the Diagram Predicate Framework, then, I will give some details of the syntax and semantics of models. The main focus will be on the use of models to describe both the structure and the behavior of software. The structure is defined by graphs together with constraints while the dynamic semantics is given by a transition system which is described by couple model transformation rules. Rarely these two aspects of software are linked formally, thus I will try to give a short introduction to a novel way to specify the link between structure and behavior.
Exploiting Macros in Source-to-Source Compiler
Implementation *

Tero Hasu

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
tero@ii.uib.no

Abstract

A sufficiently feature-rich general-purpose programming language with an expressive macro system can play multiple roles in the implementation of a source-to-source compiler: it can host the language being compiled, and expose its own macro system to make the hosted language user extensible; it can embed macro-implemented language for the domain of program transformations; and its general-purpose features and libraries allow for the entire compiler to be implemented based on the same language technology. I discuss some potential uses of the “programmable programming language” Racket in the implementation of source-to-source compilers.

1 Introduction

A source-to-source compiler (or transcompiler for short) is a programming language implementation outputting source code. Transcompiled languages can be useful in reusing target language infrastructure and abstracting over target language variability, and—particularly when translated into human-readable code—their adoption need not entail high risk [3].

Transcompilers and Lisp-style macro expanders are conceptually similar in that they translate between languages, and usually operate on abstract syntax. This suggests that a macro system might be used to do some of the work of a transcompiler, particularly on the front end side, which is where macros are designed to operate. The Racket programming language [2] has a particularly expressive macro system, and it has also been designed to host other languages defined as Racket libraries [8]. With some language design compromises for Racket compatibility, a Racket-hosted language may get significant reuse out of the host language facilities. These, in turn, can serve as a convenient basis e.g., for surface syntax implementation, desugaring transformations, and an extension mechanism for the hosted language.

There are also many potential uses for domain-specific languages (DSLs) in the program transformation domain, as suggested by the Spoofax language workbench [5] for instance, with its selection of DSLs for defining various aspects of language implementations and their tools support. In transcompiler implementations DSLs are commonplace in the specification of parsers and data structures for representing programs being transformed. While a Racket-hosted language can get parsing almost “for free”, a program object model (POM) is still a likely implementation requirement. A POM includes at least a data structure used to represent a program, and a programming interface (or API) for manipulating the data.

In the case of a transcompiler, an abstract syntax tree (AST) would typically be a useful POM structure. As each syntactic construct in the transformed language usually gets its own AST

*This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.
node data type, an AST implementation tends to involve repetitive code. There are a number of existing tools (e.g., ApiGen [1] and GOM [6]) capable of generating syntax tree definitions from a language grammar description (or similar). A language like Racket is also capable of performing the required code generation with macros, thus enabling declarative programming of ASTs within the language. One might even consider making it possible to declaratively specify abstractions that are not restricted to grammar structure, but rather reflect some other structural or conceptual similarity in the constructs of the transformed language [3].

A source-to-source compiler implementation language equipped with a sufficiently expressive macro system makes it possible to get the convenience of (program transformation) domain specific language without losing the flexibility of a general-purpose programming language. Macro-enabled language malleability also opens up opportunities for hosting the implemented language on top of the implementation language, which allows for varying degrees of sharing of language infrastructure between the two languages; the achievable level of sharing depends on the used language integration approach [4, 7], as well as the design of the hosted language.

References

Model-Transformations to Assist in Protocol Design for Wireless Sensor Actuator Networks

A. Ajith Kumar S.¹,² and K.I.F Simonsen¹,³

¹ Bergen University College, Bergen, Norway  
² University of Agder, Grimstad, Norway  
³ Danish Technical University, Copenhagen, Denmark  
{aaks,kifs}@hib.no

Wireless Sensor-Actuator Networks (WSANs) consists of a network-connected sensors and actuators working towards a specific mission. It is a branch of the existing Wireless Sensor Network (WSN) systems. Sensors and actuators in WSAN are resource constrained small devices usually powered by batteries. WSAN has several application domains such as process automation and factory automation, and is thus widely applicable in an industrial setting. One important setting in which WSAN is applicable is the control-loop of automation processes. Systems with control-loops have stringent requirements, and these applications are often safety-critical. This implies that it is important to have a sound design methodology for developing software solutions to be deployed on the sensors and actuators. Quite often the design methodology includes modeling of the protocols. Later these models are manually converted to simulation code and further analyzed. Some development approaches also includes model-checking to check for correctness. As the last step, models are transformed to the implementation platform code. This approach towards design of WSAN solutions can be combined with existing software engineering approaches, to strengthen the reliability of the software generated and to reduce the time for development.

Model-Driven Software Engineering (MDSE) is one such approach that has long been seen as a prominent approach for software engineering. MDSE uses models as primary artifacts. MDSE is an extensively used methodology across several domains for development of applications. Advantages of MDSE approach include shorter time from design to implementation, verified and validated models used for automatic code generation. In MDSE approaches, the initial model is an abstract and platform independent representation of the protocol. This abstraction allows the designers to focus on creating a model with proper functioning. Combining this abstraction step with formal approaches allows further improving the verification and validation process. The abstract models can be model-checked for verification and can be further simulated to obtain initial performance assessment results. This minimizes the possibility of errors in the code generated to a further extent.

Our MDSE approach is illustrated in figure 1. The starting point of the approach is an abstract model of the given protocol. For modeling purposes, we use CPN Tools [2]. Along with modeling and verification, CPN Tools allows for initial simulation. Using CPN Tools, developers can create a formal executable model of a protocol. Developers can also perform behavioral and functional verification, state-space analysis, and initial simulation. Based on the analysis, developers can verify and validate the given model, based on its requirements specification. CPN has been previously used for modeling and verification of network protocols [1]. In the next step, the refined CPN model is converted to code models using the PetriCode tool [5]. PetriCode is a code generation tool that is designed to automatically generate implementations of network protocols for various platforms. The code generation approach of PetriCode is shown in figure 2. PetriCode requires the models to be annotated with code generation pragmatics. These pragmatics are structural annotations on CPN models that are bound to code-generation pragmatics.
templates via template bindings. In PetriCode the pragmatics and the template bindings as well as the structure of the CPN model is used to guide code generation without needing to translate or interpret the ML code of any given CPN model. Using the PetriCode tool, developers can automatically generate code for model-checkers, network simulators, and hardware platforms. The implementation code generated for simulation is based on the MiXiM [3] platform, and the hardware implementation code is based on the TinyOS [4] platform. The challenge here is to obtain two different implementation models from one base abstract model. The MiXiM platform uses implementation based on C++, and TinyOS is based on the nesC programming language. nesC is a C like language but has many different features that are designed to suit hardware implementations.

References