Simplicity Principles for Plug-In Development: The jABC Approach

Stefan Naujokat, Anna-Lena Lamprecht
Chair for Programming Systems
TU Dortmund, 44227 Germany
stefan.naujokat@cs.tu-dortmund.de
anna-lena.lamprecht@cs.tu-dortmund.de

Bernhard Steffen, Sven Jörges
Chair for Programming Systems
TU Dortmund, 44227 Germany
steffen@cs.tu-dortmund.de
sven.joerges@tu-dortmund.de

Tiziana Margaria
Chair for Service and Software Engineering
University Potsdam, 14482 Germany
margaria@cs.uni-potsdam.de

***Abstract*—In this paper we present our experiences from a decade of plug-in development in the jABC framework, that is characterized by rigorous application of *simplicity principles* in two dimensions. First, the *scope* of the plug-in development is clearly defined: The jABC readily provides a sophisticated graphical user interface, which has been tailored to working with all kinds of directed graphs. Within this scope, plug-in development can deliberately focus on the actual functionality, like providing semantics to graphs, without having to deal with tedious but semantically irrelevant issues like user interfaces. Second, *plug-in functionality* can be itself conveniently modeled as a workflow within the jABC. We illustrate our approach by means of two mature plug-ins: Genesys, a plug-in that adds arbitrary code generator functionality to the jABC, and PROPHETS, a plug-in that eases user-level definition of workflows by completing model sketches by means of synthesis capabilities, so that they become complete and executable. We summarize our experience so far and derive general design principles for "lightweight plug-in development", that we are going to realize in the next generation of the jABC, which will be implemented itself as a collection of Eclipse plug-ins.**

I. Introduction

For many frameworks, developing plug-ins is a technically complex task that usually involves the need for knowledge of many technical details about the target framework. This is often due to the fact that frameworks try to achieve a maximum of generality, and the task gets more complicated the more general the target framework is.

On the contrary, to grant ease of extension, the jABC framework is therefore itself built to follow the 'simplicity paradigm' [1], [2]: Its field of application is limited to efficient handling of graph-like data structures consisting of nodes and edges, but it provides easy-to-handle APIs to introduce or enhance the semantics of such models. Thus, due to the simplicity of the core platform the development of the plug-in mechanics is more 'lightweight' than in other platforms, and the bulk of plug-in development can concentrate on the actual things the plug-in should do.

Plug-in development for the jABC is in fact simplified beyond mere technical interface definitions, as common in many other frameworks. jABC provides a plug-in mechanism that may be used even by non-programmers, as they can rely on jABC's modeling itself for the construction of the plug-in's processes and automatically generate those into executable code. The decision to consistently make jABC easy to extend has been a design decision *ab initio*: plug-ins are since 2001 the mechanism we adopted to provide *variability at the framework level*, and this paper sums up the conclusions of our 10 years of experience in developing the jABC framework and tools.

Over the jABC generations, versions, and releases, we have consistently simplified the core framework and the way to extend it, favouring ease of evolution and extension over the power of the core itself. This led us to follow a consistent simplicity-driven approach also to the realization of plug-ins, that can be reflected in an 80/20 rule: it should be made very easy to create plug-ins that cover 80% of the extensions one wishes for the core jABC. This simplicity tradeoff is claimed by other frameworks as well, but here we proceed more rigorously: jABC is a framework for working with directed graphs, where the general functionality, common to all kinds of graphs and all domain-specific extensions or specializations, is therefore conveniently provided, and where specific functionality can be very easily plugged in.

In the spirit of end-user software development [3], the jABC framework is fully targeted to be used by non-programmers. Consequently, we wish that no programming skills are necessarily required also for adding functionality to the jABC, such as for instance a plug-in that automates a particular task. Our approach to extending the core functionality is conceptually comparable to the definition of custom 'macros', that have been possible since the early years of MS Office products: Technically, those macros are scripts in the Visual Basic programming language, but user-friendly macro-definition options like the simple recording of UI clicks enable the definition of custom scripts for a wide field of MS Office non-technically savvy users, who can even less be considered programmers than in our case.

The paper is structured as follows. Sect. II gives a brief introduction to the core ideas of the jABC framework and how it is used for modeling. Sect. III then provides more detail on jABC's plug-in architecture and extension points. Sect. IV details on using jABC's own modeling methods to generate parts of plug-ins, followed by the presentation of two example plug-ins (in Sect. V and VI), which are based on this self-application. Finally, Sect. VII details on related approaches and Sect. VIII concludes with some general recommendations on plug-in architectures and our plans on developing the next generation of jABC, which will itself be based on the Eclipse framework.

II. The jABC Framework

In its simplest form, the jABC is basically a graphical editor for constructing directed, hierarchical graphs by placing nodes (called *Service Independent Building Blocks*, or simply *SIBs*) on a canvas and connecting them with labeled edges (called *branches*). As explained in greater detail in, e.g. [4], the design of its GUI largely corresponds to that of similar frameworks: the available SIBs are listed in a browser (upper left), from where they can be dragged onto the "drawing" area (right), where the model construction takes place. Different inspectors (lower left) can be used for the detailed configuration of components and models. With this graphical interface, models can be reconfigured and extended very easily.

Additional functionality is brought to the framework by means of plug-ins. In particular, any kind of interpretation of the model is subject to specific plug-ins. The basic jABC framework can be extended by two principal kinds of plug-ins:

1) *Semantics plug-ins* provide functionality for specific interpretations of the graphical model or its components. Examples of such plug-ins are the Tracer, Genesys and PROPHETS.
2) *Feature plug-ins* extend the range of general, domain-independent functionality that is available in the jABC. Examples of such plug-ins are the TaxonomyEditor, the GEAR model checker, the SIBCreator and the Layouter.

Most semantics plug-ins interpret the graphical models as *Service Logic Graphs* (*SLGs*) [5], [6], that is, as executable control-flow models composed of services [7]. The Tracer plug-in, for example, provides an interpreter and execution environment, allowing for the stepwise or complete execution of the SLGs, and the use of breakpoints. Similarly, the Genesys plug-in is a jABC plug-in for using the Genesys code generator framework [8], [9] that can be used to automatically compile any executable process model into a single deployable application that can be run independently of the jABC. Genesys interprets the jABC models as control flow graphs, using principally the same mechanisms as the Tracer does. Also the PROPHETS plug-in (Process Realization and Optimization Platform using Human-readable Expression of Temporal-Logic Synthesis) [10], enabling (semi-) automatic workflow composition according to the *loose programming* paradigm [11], assumes control-flow semantics.

Examples for semantics plug-ins that apply another kind of semantics to the jABC's SIB graph models are the FormulaBuilder plug-in [12] for the graphical definition of logic formulae, and the DBSchema plug-in for graphical specification of relational database schemata.

Prominent examples of jABC feature plug-ins are the *TaxonomyEditor* plug-in [13] (for organizing SIB libraries according to arbitrary, user-defined criteria, instead of simply to their location in the file system), the *GEAR model checking plug-in* [14] (for verification of global, temporal-logic properties of a workflow model), the *SIBCreator* plug-in (assisting in the development of new SIBs), and the *Layouter* plug-in (providing a variety of layout algorithms for application to SIB graph models).

Plug-ins are in fact the principal components of the jABC product line: particular products of the variant-rich jABC product family are typically defined by combining different sets of plug-ins. For instance, the Bio-jETI [15], [16] incarnation of the jABC framework comprises a set of plug-ins that is adequate for dealing with bioinformatics workflows (esp. Tracer, Genesys, PROPHETS, TaxonomyEditor, GEAR), the LearnLib Studio [17], [18] for management of automata learning processes combines the Tracer and Layouter plug-ins with an additional LearnLib plug-in, and ConnectIT [19] is mainly constituted by a specific plug-in that facilitates the intuitive design and evaluation of game strategies for Connect Four, but makes use of the Tracer for the execution of matches with the developed strategies.

III. Developing Plug-Ins in jABC

As introduced before, jABC is a specialized framework for tools that work with directed graph structures. Therefore, its plug-in architecture is designed to handle those structures easily, instead of being more generic at the cost of a more complex implementation.

To create a new plug-in for the jABC, one needs to implement the `Plugin` interface that is shipped with the framework, add it to the runtime class path and enable it for loading within the jABC plug-in configuration. Implementing classes then provide `start()` and `stop()` methods where the plug-in can register and deregister additions to the jABC, such as menu entries, toolbar buttons for frequently required actions, and other UI elements. Via global handlers provided by the framework, such additions then can access anytime, for instance, the currently open models or project.

Technically, SIBs are arbitrary Java classes that contain a special `@SIBClass` annotation. By convention, all public fields are interpreted as parameters of a SIB for which the framework automatically provides an editor (the SIB inspector). Plug-ins that add additional semantics to the jABC provide a dedicated interface via the `Plugin#getPlugininterface()` method that has to be implemented by all SIBs used in the graphs that shall be interpreted according to those semantics. With this approach, the plug-in developer can focus on the 'interesting' part, i.e. the semantic interpretation of the models, because the user interface and event handling are already provided by the jABC framework. This is

- a *lightweight* approach, because it just extends at need those SIBs that are affected by the plug-in,
- it is *incremental*, because it is easy to add the interface at need to more SIBs and thus retrofit existing components, and it is
- *aspect oriented* in its spirit, because it allows one to make accessible to a plug-in in a concern- and purpose-driven fashion a variety of SIBs that may span over the entirety of the SIB collections. This way, cross-cutting concerns are easily overlaid on existing domains and applications.

In addition to the generic inspectors that are provided by the jABC framework, plug-ins may add own inspectors for specific views on model and SIB data. Conveniently, as all plug-ins use the common model data structure of the framework, they can get notified automatically about the events happening on the process model. In particular, this makes it easy for a plug-in's inspectors to display context-dependent information, e.g. on the currently selected SIB.

Finally, a plug-in may provide a dynamically rendered *overlay icon* that can be used to graphically display status information for all SIBs in the current model. This is implemented via the `Plugin#getOverlayIcon(SIBGraphCell)` method. This icon is then overlaid in a corner of the SIB icon. If more than four loaded plug-ins provide such overlay icons, the user may choose between them and configure their position within the jABC plug-in configuration. Common usage scenarios for overlay icons are warning and error notifications of the LocalChecker plug-in or breakpoints and the *currently executing* marker of the Tracer plug-in (cf. Fig. 3, top left and bottom right).

IV. How to Model Plug-Ins

The previous section detailed on the lightweight nature of plug-in development for the jABC. With a tool like jABC at hand, that allows for modeling workflows as SLGs and for their generation into Java source code (among others), even non-programmers are empowered to modify or enhance plug-ins. Workflows that naturally exist within plug-ins can be defined by modeling them as jABC Service Logic Graphs and are therefore easily modifiable by users at any time. This can be considered as a powerful means to carry out plug-in configuration, providing more flexibility as commonly used key/value pairs.

It is even possible to realize a full plug-in-generating code generator (cf. Fig. 1). With this, any SIB graph model could be easily exported into a plug-in that provides the modeled process as a custom action (e.g. via a menu entry or a button). Plug-ins that provide additional functionality to SIBs and SLGs can be modeled with the existing common SIB libraries for the handling of SIB graph models, which provide SIBs for the modification of SLGs and their components (i.e. SIB nodes and branches). Possible use cases could be 1) the export all error edges 2) domain-specific set, preset or reset some SIBs' parameters or 3) detect under-specification using the model checker and automatically mark unassigned branches as loosely specified.

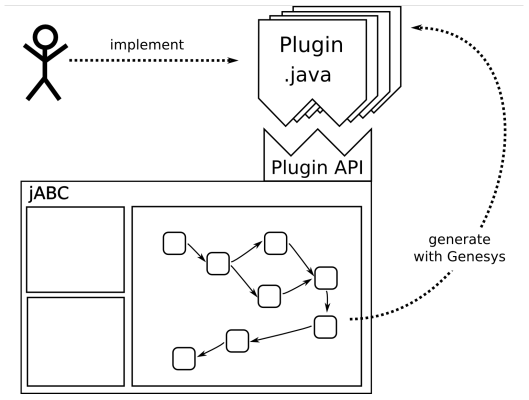

Figure 1. Model plug-ins in jABC and generate them with Genesys

The following sections detail on two semantics plug-ins, namely Genesys and PROPHETS, that make use of jABC processes for the definition of their core functionality. These two plug-ins have been developed according to the current 'state of the art' of our constantly evolving framework and plug-in development based on the concept of simplicity. Most other existing plug-ins (esp. the feature plug-ins introduced in Sect. II) are 'older' and have been developed conventially, i.e. manually implemented against the jABC plug-in API. They will be redesigned according to our recent findings as soon as the need for their extension arises.

V. Genesys Code Generation

Genesys [8], [9] is a general framework for constructing code generators for arbitrary source and target platforms in a model-driven and service-oriented way, based on the jABC. Consequently, in this framework code generators are modeled as SLGs [20], using specific SIBs which provide services that are typically required for code generation, such as template engines (e.g., StringTemplate [21]), data type mapping and variant management. Genesys is distributed as a dedicated jABC product (cf. Sect. II), comprising, among other things, models, services and tools that support the integrated development of code generators.

Apart from being based on the jABC, Genesys is also used for providing code generation capabilities for the jABC itself, i.e., code generators which translate SLGs into executable code for target languages such as Java, C# or Objective-C (in the following, we will refer to this specific class of code generators as "jABC code generators"). Accordingly, Genesys aims at both generator developers as well as generator users. In order to fulfill this double role, Genesys heavily exploits jABC's plug-in architecture.

For generator developers, the *developer tools* add helpful functionality for the construction of code generators, such as rich editors for templates or tools for benchmarking generation performance, all realized as feature plug-ins.

For generator users, the *Genesys jABC Plug-In* provides convenient access to any existing jABC code generator. Via the plug-in, the user is able to select a jABC code generator for a desired target language, and to configure it according to his needs. After this one-shot configuration step, code can be generated from any SLG at the push of a button. For inspecting the progress of the code generation, the plug-in also provides an integrated console which visualizes information and error messages.

Since all jABC code generators interpret SLGs according to the control flow semantics defined by the Tracer [9], the Genesys jABC Plug-In is clearly a semantics plug-in.

From the plug-in design perspective, the way in which jABC code generators are made available through the Genesys jABC Plug-In is particularly interesting. As the plug-in does not make any assumptions about which and how many code generators are supported nor on how they are configured individually, the jABC code generators are integrated via a *generic plug-in interface* that is similar to the one provided by the jABC: this means that technically they are *plug-ins of a plug-in* (cf. Fig. 2).

In combination with the fact that those code generators are again developed with the jABC via the Genesys framework, this shows that although jABC's plug-in mechanism is deliberately kept simple, it allows the realization of powerful and sophisticated plug-in scenarios, nicely organized in families according to what they share.

VI. PROPHETS Synthesis

The PROPHETS plug-in introduces loose specification [11] and synthesis for process models defined in the jABC. Its central application is for a user friendly and knowledge-driven workflow and process definition. PROPHETS is built

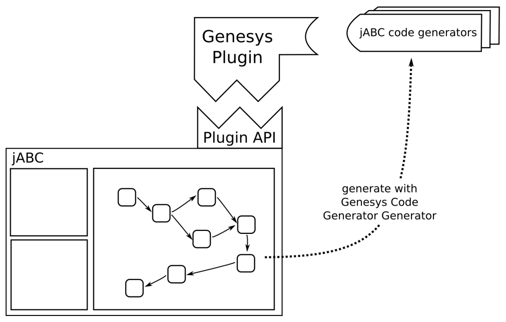

Figure 2. Genesys Plug-In: additional plug-in API for jABC code generators.

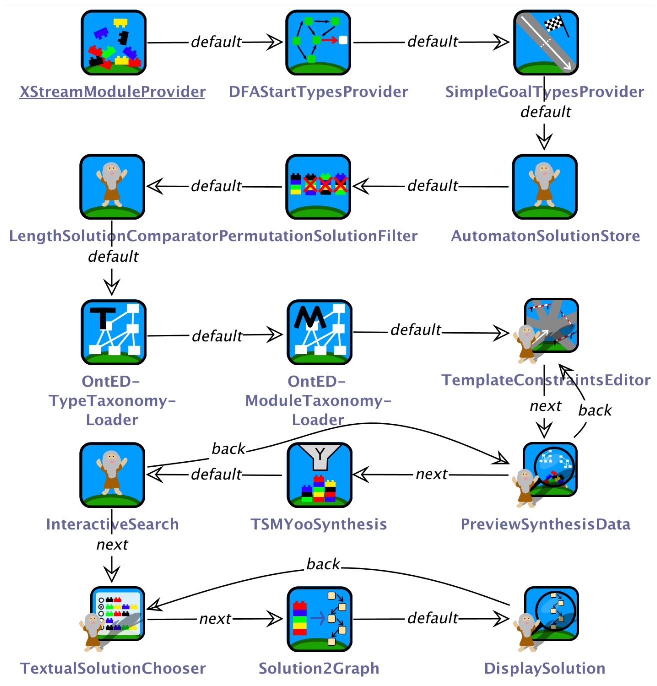

Figure 3. User-defined synthesis process for the PROPHETS plug-in

upon the experience previously gathered in workflow synthesis [22], [23], [24], [25].

Rather than requiring workflow designers to model everything in the workflow explicitly, the workflow model is only specified roughly, in terms of ontologically defined "semantic" entities. These loose specifications are then automatically concretized to fully executable concrete workflows by means of a combination of different formal methods, namely data-flow analysis, LTL synthesis and model checking.

Regarding the previously introduced classification of feature and semantics plug-ins, PROPHETS falls in the latter class. But instead of defining a completely different semantics to the models, it enhances the control-flow graph semantics of Genesys and the Tracer by adding a new type of branch that explicitly states the model's under-specification at those locations where the user knows that something is missing. We call it a knowledge-driven way of defining loose workflows because the user models what he/she knows, and asks PROPHETS to complete the workflow. For those *loosely specified branches* the synthesis process searches for appropriate inserts (typically a SIB or an entire sub-workflow) and replaces them accordingly, yielding (where possible) a fully executable workflow.

This synthesis process itself that constitutes the core of PROPHETS is modeled with the jABC [26], generated using the Genesys Java class generator and packaged into the PROPHETS plug-in. However, this builtin process can be dynamically overwritten at runtime. If (by name convention) a special PROPHETS SLG is present within the current jABC project, it will be used instead of the generated one (see Fig. 3 for one such user-defined process). This is conceptually comparable to Genesys' approach of adding jABC code generators into the plug-in. Technically, however, the generators are generated and compiled, whereas the user-defined synthesis process is interpreted at runtime using the Tracer plug-in. In addition to the flexible configuration possibilities that arise for the user at his level of expertise, following the jABC paradigm of simplicity, it enables the domain modeler to add domain-specific SIBs for the synthesis process, e.g. a SIB that loads domain-specific synthesis constraints from a database.

VII. Related Approaches

The concept of full generation of plug-ins is also present in other frameworks. For instance in the context of the Eclipse Modeling Framework (EMF) [27], the automatic generation of a tree-based editor plug-in for a given meta-model can be realized within a few clicks. However, the specification of a meta-model is something that we do not consider to be on the user's level of expertise. Furthermore, this editor suits well for prototyping, but for many meta-models a less generic editor or even a graphical one seems to be the more reasonable choice. The latter is often realized with GMF [28, Chap. 11, 12], which is even less at user-level and has other drawbacks that more and more drive the community to use the Graphiti framework [29]. But this is a pure programming API and therefore only accessible to programmers.

One other aspect clearly differentiates between the jABC approach and the Eclipse approach: with the Genesys framework we postulate the full generation of the (plug-in) code, explicitly without the need (or admission) to make changes to the generated code. Changes are subject to the SLG model or SIB implementation and a re-generation always leads to fully operational code. This solves the round-trip problem [30] at the cost of some flexibility. On the contrary, EMF spends extensive efforts into the automatic tracking of code changes and merging to minimize the cost of round-tripping. But this cannot be achieved in a fully automatic way, so the developer has to assist there, e.g. by following certain name conventions or changing annotations of the EMF generator.

The idea of developing jABC plug-ins from reusable core assets – SLGs and SIBs – is in line with classical *generative programming* [31], [32]. The central concept of this approach is the generative domain model, which essentially consists of a problem space, a solution space and a mapping between both. Applied to this conceptual framework, SLGs and SIBs are ready-made components situated in the solution space. Modeling an actual plug-in happens in the problem space, and the corresponding model is again an SLG composed of SIBs, resembling a particular configuration of solution space components. The final mapping between the two spaces is provided by a suitable Genesys code generator, which translates the SLG into the actual plug-in.

VIII. Conclusion

We showed that the process of developing plug-ins can be enabled for non-technical users, if the target framework is specifically designed. With jABC this was made possible. We do not claim to have within the jABC the best or most powerful plug-in mechanism, but our experience of meanwhile over 10 years has shown that its simplicity-first approach is valuable for lightweight, small footprint plug-ins and we think that this is something that other developers of plug-in frameworks might wish to take into consideration.

In comparison to other frameworks with rich plug-in support, for instance Eclipse PDE [33], it is obvious that jABC does not offer as many possibilities. But the features that are within the scope of jABC are this way far more easily realizable for programmers who are not experts on the respective framework. Due to the consistent application within jABC of model driven design plus code generation to write extensions to the framework, plug-in generation and plug-in modification can be made accessible even to non-programmers, as we have shown for Genesys and PROPHETS. We consider this capability to be a core asset of jABC, especially in its perspective use as a Web 3.0 platform, that turns non-IT users into "prosumers" (producers and not only consumers) of process-oriented service engineering applications. Therefore, we are planning to develop the next generation of jABC following a hybrid approach:

- we will develop it on top of the Eclipse framework, which meanwhile offers a significant wealth of functionality that is both stable, thus reliable enough to use in the framework core, and also community-maintained, thus cutting the amount of own development and of maintenance effort,
- but holding on to the simplicity paradigm for what concerns jABC's user-friendliness for non-technically skilled user communities, in particular concerning the eXtreme Model Driven Design approach (XMDD [34], [35], [36]) and the ease of extensibility via lightweight plug-ins we have right now.

This enables framework experts to exploit all features the Eclipse world has to offer (e.g. different meta-models via EMF [27], tool support for Java development [37], VCS integration etc.), but enhancements to the jABC core features will be as simple as in the current version.

References

[1] T. Margaria and B. Steffen, "Simplicity as a Driver for Agile Innovation," *Computer*, vol. 43, pp. 90–92, 2010.

[2] "IT[t] Simply Works - Recommendation Document, 2011." http://www.cs.uni-potsdam.de/sse/ITSy/files/ITSy_Recommendation.pdf.

[3] A. J. Ko, R. Abraham *et al.*, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, pp. 21:1–21:44, Apr. 2011.

[4] B. Steffen, T. Margaria *et al.*, "Model-Driven Development with the jABC," in *Hardware and Software, Verification and Testing*, ser. LNCS, E. Bin, A. Ziv *et al.*, Eds. Springer Berlin / Heidelberg, 2007, vol. 4383, pp. 92–108.

[5] T. Margaria and B. Steffen, "Lightweight coarse-grained coordination: a scalable system-level approach," *STTT*, vol. 5, no. 2-3, 2004.

[6] B. Steffen, T. Margaria *et al.*, "Heterogeneous Analysis and Verification for Distributed Systems," *Software - Concepts and Tools*, vol. 17, no. 1, pp. 13–25, 1996.

[7] T. Margaria, B. Steffen *et al.*, "Service-Oriented Design: The Roots," in *ICSOC*, 2005, pp. 450–464.

[8] S. Jörges, T. Margaria *et al.*, "Genesys: service-oriented construction of property conform code generators," *ISSE*, vol. 4, no. 4, pp. 361–384, 2008.

[9] S. Jörges, "Genesys: A Model-Driven and Service-Oriented Approach to the Construction and Evolution of Code Generators," Ph.D. dissertation, TU Dortmund, 2011.

[10] S. Naujokat, A.-L. Lamprecht *et al.*, "Loose Programming with PROPHETS," in *FASE 2012*, ser. LNCS, vol. 7212. Springer Heidelberg, 2012, pp. 94–98.

[11] A.-L. Lamprecht, S. Naujokat *et al.*, "Synthesis-Based Loose Programming," in *QUATIC*, Sep. 2010.

[12] S. Jörges, T. Margaria *et al.*, "FormulaBuilder: a tool for graph-based modelling and generation of formulae," in *ICSE*, 2006.

[13] T. Margaria and B. Steffen, "Second-Order Semantic Web," in *Proc. of 29th Annual IEEE/NASA Software Engineering Workshop*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 219–227.

[14] M. Bakera, T. Margaria *et al.*, "Tool-supported enhancement of diagnosis in model-driven verification," *Innovations in Systems and Software Engineering*, vol. 5, pp. 211–228, 2009.

[15] T. Margaria, C. Kubczak *et al.*, "Bio-jETI: a service integration, design, and provisioning platform for orchestrated bioinformatics processes," *BMC Bioinformatics*, vol. 9 Suppl 4, p. S12, 2008.

[16] A.-L. Lamprecht, T. Margaria *et al.*, "Bio-jETI: a framework for semantics-based service composition," *BMC Bioinformatics*, vol. 10 Suppl 10, p. S8, 2009.

[17] M. Merten, B. Steffen *et al.*, "Next Generation LearnLib," in *TACAS*, 2011.

[18] M. Merten, F. Howar *et al.*, "Demonstrating Learning of Register Automata," in *TACAS*, 2012.

[19] M. Bakera, S. Jorges *et al.*, "Test your Strategy: Graphical Construction of Strategies for Connect-Four," in *ICECCS*. Washington, DC, USA: IEEE, 2009.

[20] S. Jörges, B. Steffen *et al.*, "Building Code Generators with Genesys: A Tutorial Introduction," in *GTTSE'09*. Springer, 2010, pp. 364–385.

[21] T. Parr, "Enforcing strict model-view separation in template engines," in *Proceedings of the 13th international conference on World Wide Web, WWW'04*. ACM, 2004, pp. 224–233.

[22] B. Steffen, T. Margaria *et al.*, "Module Configuration by Minimal Model Construction," Fakultät für Mathematik und Informatik, Universität Passau, Tech. Rep., 1993.

[23] T. Margaria and B. Steffen, "Backtracking-Free Design Planning by Automatic Synthesis in METAFrame," in *Fundamental Approaches to Software Engineering*, 1998, pp. 188–204.

[24] B. Steffen, T. Margaria *et al.*, "Automatic synthesis of linear process models from temporal constraints: An incremental approach," *In ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97)*, 1997.

[25] T. Margaria and B. Steffen, "LTL Guided Planning: Revisiting Automatic Tool Composition in ETI," in *SEW*. IEEE, 2007, p. 214–226.

[26] S. Naujokat, A.-L. Lamprecht *et al.*, "Tailoring Process Synthesis to Domain Characteristics," in *ICECCS*, 2011.

[27] D. Steinberg, F. Budinsky *et al.*, *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.

[28] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2008.

[29] "Graphiti - a Graphical Tooling Infrastructure," http://www.eclipse.org/graphiti/.

[30] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[31] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[32] K. Czarnecki, *Overview of Generative Software Development*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3566, pp. 326–341.

[33] "Eclipse Plug-in Development Environment," http://www.eclipse.org/pde/.

[34] T. Margaria and B. Steffen, "Agile IT: Thinking in User-Centric Models," in *ISoLA*, T. Margaria and B. Steffen, Eds., vol. 17. Springer Berlin Heidelberg, 2009, pp. 490–502.

[35] ——, *Handbook of Research on Business Process Modeling*. IGI Global, 2009, ch. Business Process Modelling in the jABC: The One-Thing-Approach.

[36] ——, "Continuous Model-Driven Engineering," *IEEE Computer*, vol. 42, no. 10, pp. 106—109, Oct. 2009.

[37] "Eclipse Java development tools," http://www.eclipse.org/jdt/.