Model-Driven Development with the jABC

CONFERENCE PAPER · JANUARY 2006

DOI: 10.1007/978-3-540-70889-6_7 · Source: DBLP

Bernhard Steffen
Technische Universität Dortmund
374 PUBLICATIONS 6,677 CITATIONS
SEE PROFILE

Margaria Tiziana
University of Limerick and Lero - The Irish S…
297 PUBLICATIONS 2,049 CITATIONS
SEE PROFILE

Ralf Nagel
Fraunhofer Institute for Software and Syste…
12 PUBLICATIONS 220 CITATIONS
SEE PROFILE

Christian Kubczak
Technische Universität Dortmund
26 PUBLICATIONS 240 CITATIONS
SEE PROFILE

Model-Driven Development with the jABC

Bernhard Steffen[1], Tiziana Margaria[2], Ralf Nagel[1], Sven Jörges[1], and Christian Kubczak[1]

[1] Chair of Programming Systems, University of Dortmund, Germany
{steffen,nagel,joerges,kubczak}@ls5.cs.uni-dortmund.de
[2] Chair of Service and Software Engineering, University of Potsdam, Germany
margaria@cs.uni-potsdam.de

Abstract. We present the jABC, a framework for model driven application development based on Lightweight Process Coordination. With jABC, users (product developers and system/software designers) easily develop services and applications by composing reusable building-blocks into hierarchical (flow-) graph structures that are executable models of the application. This process is supported by an extensible set of plugins providing additional functionalities, so that the jABC models can be animated, analyzed, simulated, verified, executed and compiled. This way of handling the collaborative design of complex software systems has proven to be effective and adequate for the cooperation of non-programmers and technical people, and it is now being rolled out in the operative practice.

1 Lightweight Process Coordination

jABC[2] is a mature framework for service development based on Lightweight Process Coordination [29]. Predecessors of jABC have been used since 1995 to design, among others, industrial telecommunication services [30], Web-based distributed decision support systems [19], and test automation environments for Computer-Telephony integrated systems [16].

jABC allows users to easily develop services and applications by composing reusable building-blocks into (flow-) graph structures. This development process is supported by an extensible set of plugins that provide additional functionality in order to adequately support all the activities needed along the development lifecycle like animation, rapid prototyping, formal verification, debugging, code generation, and evolution. It does not replace but rather enhances other modelling practices like the UML-based RUP (Rational Unified Process, [3,15]), which are in fact used in our process to design the single components.

Lightweight Process Coordination (LPC) [29] as a service-oriented, model-driven development approach, offers a number of advantages that play a particular role when integrating off-the-shelf, possibly remote functionalities:

– **Simplicity.** jABC focuses on application experts, who are typically non-programmers. The basic ideas of our modelling process have been explained in past projects to new participants in less than one hour.

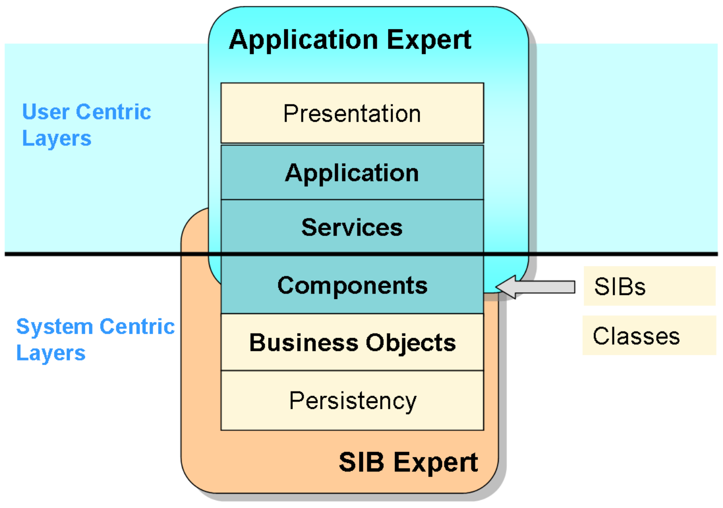


Fig. 1. Layered Architecture of jABC Applications

– **Agility.** We expect requirements, models, and artefacts to change over time, therefore the process supports evolution as a normal process phase.
– **Customizability.** The building blocks which form the model can be freely renamed or restructured to fit the habits of the application experts.
– **Consistency.** The same modelling paradigm underlies the whole process, from the very first steps of prototyping up to the final execution, guaranteeing traceability and semantic consistency.
– **Verification.** With techniques like model checking and local checks we support the user to consistently modify his model. The basic idea is to define local or global properties that the model must satisfy and to provide automatic checking mechanisms.
– **Service orientation.** Existing or external features, applications, or services can be easily integrated into a model by wrapping the existing functionality into building blocks that can be used inside the models.
– **Executability.** The model can have different kinds of execution code. These can be as abstract as textual descriptions (for example in the first animations during requirement capture), and as concrete as the final runtime implementation.
– **Universality.** Thanks to Java as largely platform-independent, object-oriented implementation language, jABC can be easily adopted in a large variety of technical contexts and of application domains.

The basic idea of Lightweight Process Coordination is to add a coordination layer to the generally well established three tier architecture. This coordination layer spans the application and services layers of Fig. 1. It is a purely model driven development layer, created and managed within a graphical interactive tool: the *Java Application Building Center* (jABC). In jABC, users build coordination models by arranging predefined building blocks simply by drag and drop. These basic building blocks are called *SIBs* (Service Independent Building Block). SIBs have one or more outgoing edges (*branches*), which depend on the different outcomes of the execution of the functionality represented by the SIB.

As an example we may use a SIB called `CreateBooking`, which prepares a modification in a database. This SIB could have three branches, labelled *Successful*, *DataError* and *DatabaseError*, showing the difference between a correct execution, an error caused by invalid featured data, and an error caused by a problem with the database.

Two groups of users work collaboratively on a LPC standard model:

- **SIB Experts**, who are (Java) developers with detailed knowledge about the development of SIBs and appropriate plugin interfaces, and
- **Application Experts**, who have detailed knowledge about the process or application under realization, but are not programmers and may not even have a technical background.

As shown in Fig.1, application experts model the business logic of the application from existing SIBs that correspond to components or basic services, and from instances of a special SIB used as placeholder for functionalities not yet implemented. If an application needs additional SIBs, the application expert can use the placeholder to define name, appropriate parameters and branches on his own, using the SIBCreator Plugin. Adding real functionality to the SIB is done in cooperation with the SIB expert, on the basis of the specification of the SIB and possibly also of the business logic model (called Service Logic Graph or SLG) of the application.

SIB experts take care of implementing missing SIBs, of the integration of legacy systems and components at the SIB level, and of the persistency layer.

Feature-Oriented Descriptions. The terminology SIB and SLG is taken from the context of Intelligent Networks, a successful telecommunications domain which was among the first to standardize a service-oriented architecture and development methodology [23,24], also in connection with features (here seen as basic services). In fact, the jABC methodology instantiates those concepts as follows:

Definition [Feature-oriented Description]

1. A *feature-oriented service description* of a complex service specifies the behaviours of a *base system* and a set of *optional features.*
2. The behaviour of each feature and of the basic system are given by means of Service Logic Graphs (SLGs) [24].
3. The realization of each SLG bases on a library of *reusable components* called Service Independent Building-Blocks (SIBs).
4. The feature-oriented service description includes also a set of *abstract requirements* that ensure that the intended purposes are *met.*

5. *Interactions* between features are regulated *explicitely* and are usually expressed via *constraints* in temporal logics.
6. Any *feature composition* is allowed that does not violate any constraint.

Hierarchy and Refinement. Each SLG model can be wrapped into a single coarser-grained SIB, and may be used on another hierarchical level of modelling. Similarly, each SIB can be refined into an own model, showing a more detailed view on the represented feature. This way we support both a top-down and bottom-up application modelling process.

In the remainder of the paper we present the basic components of the complete jABC toolbox. Sect. 2 and 3 give a more detailed overview of the jABC. In Sect. 4 and 5 we focus on the included verification and analysis tools (the local- and model checker). The jABC Tracer, used to animate, simulate, interpret, and debug at the coordination level is presented in Sect. 6. In Sect. 7 we present the jABC code generator, which is itself constructed by means of the jABC, as a LPC process. In Sect. 8 we discuss related approaches and in Sect. 9 we present our conclusions.

2 The Java Application Building Center

The jABC is meanwhile the fourth generation of this framework [34], with C++ precursors dating back to 1992 [33]. Thanks to Java we are largely platform independent: jABC runs wherever a JVM is available, solving this way many portability and interoperability issues of its precursors. jABC is at the same time used as a commercial product in several projects with industry, and as a teaching and experimental platform for our students. This is possible thanks to a plugin framework concept which supports the easy replacement of almost every part of the system and the easy addition of new (customer-specific) features.

Handling Basic Services. Java simplifies the handling of SIB components too - a single Java class contains all the required or optional information:

– the *name* is represented by the Java class name,
– the *parameters* are defined as the *public* fields of the class,
– the *branches* are represented by the reserved field *branches*, which can be optionally flagged as *final*,
– the *graphical representation* in the drawing canvas is the picture returned by the *getIcon()* method,
– the *online documentation* for the SIB, its parameters and branches, are retrieved via the *getTooltipText()* method, and
– optional information for animation, simulation, analysis, etc., are encapsulated with plugin-specific interfaces. For example, the interface `Tracer` consists of the defined method *onTrace()*.

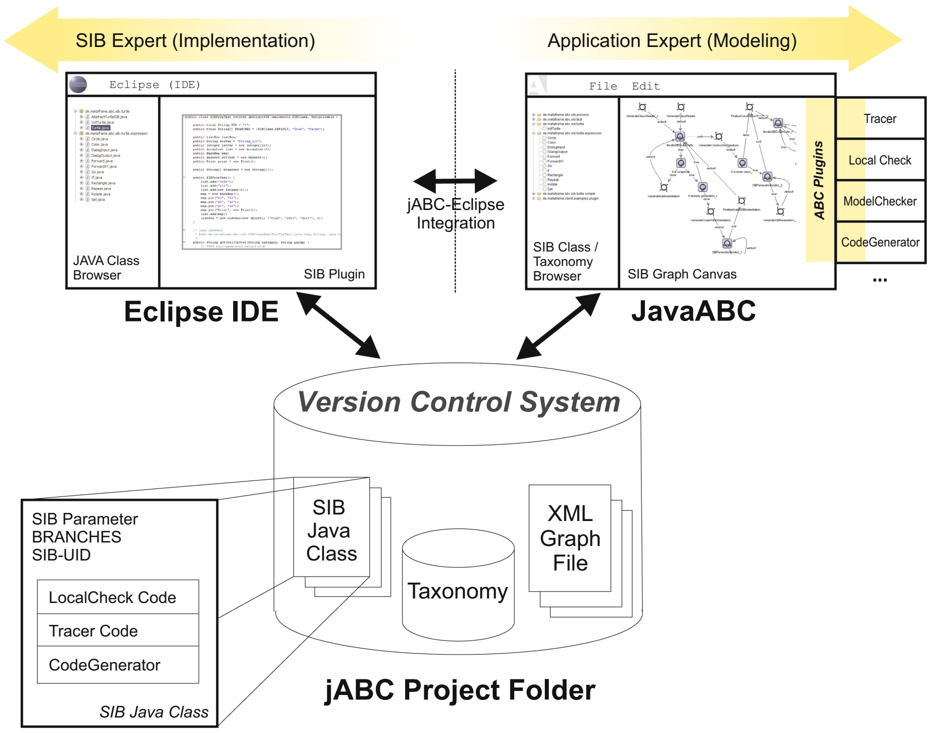
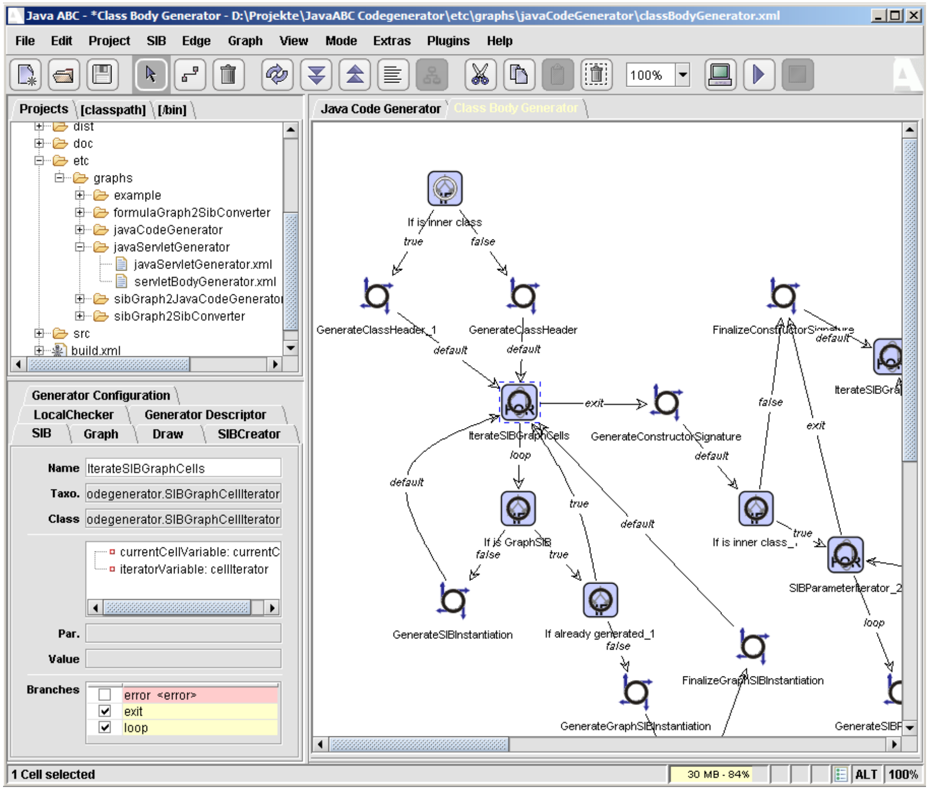


Fig. 2. jABC Big Picture

SIB Palettes as Taxonomies. At runtime, the jABC discovers and analyses the compiled SIB class files and generates a structured representation of the available SIBs for the application experts. The SIBs are presented as a taxonomy, as in Fig. 3(upper left), which shows on the canvas the model of the code generator. This taxonomy is a tree representation of a directed acyclic graph. A SIB can thus appear there several times, even with different names.

Different to standard Java classes, the physical class package of the SIB is irrelevant for the jABC: the SIB description achieves this decoupling. The jABC replaces unavailable or deleted SIB classes with the *ProxySIB*, a specific placeholder, and protects the model from information loss. If the graph is stored again, the information of the missing SIB is kept; if the SIB becomes available again, the model will automatically load the correct SIB. Even after refactoring a SIB class, older models referencing such a SIB will use the correct class.

Meaning of the Coordination Graphs. The basic jABC System does not define a standard semantics for graphs: at this point, SLGs are purely structural descriptions that can be printed, layouted, edited, but have no meaning. This meaning is given by different jABC-plugins, like the Tracer already mentioned. The Tracer interprets an SLG as a flow graph with one or more distinguished start nodes and is able to execute it. The Tracer defines an own Java interface that contains



Name IterateSIBGraphCells

Taxo. odegenerator.SIBGraphCellIterator

Class odegenerator.SIBGraphCellIterator

currentCellVariable: currentC
iteratorVariable: cellIterator

Par.

Value

Branches
☐ error <error>
☑ exit
☑ loop

1 Cell selected 30 MB - 84% ALT 100%

Fig. 3. Working with the jABC: Modifying the SLG of the Code Generator

all methods it can execute for a SIB. To support a Tracer execution, a SIB must implement this Tracer interface. Missing interface implementations are semantically empty, thus the corresponding plugin simply stops executing.

3 Overview of the jABC Architecture

Fig. 2 shows an overview of the complete jABC development system. The work with the jABC is organized in projects, which are seen as local storage folders. Users can define multiple projects, but only one at a time is active. A project folder contains all the elements (files and materials) needed for a model, even if not necessarily needed by the jABC. By versioning this folder with a versioning system (like CVS) it is possible to retrieve arbitrary older versions of a model and to distribute model changes to all project members. Currently, the rights and roles management within a project is delegated to the underlying versioning system.

There is no prescribed development environment for the jABC SIBs: it is possible to use any compatible Java development application (even *vi* and *javac*).

Plugin	**Description**	**local**	**global**	**internal**	**interface**
BeanShell	Scripting facilities	√			√
CodeGenerator	Model compilation		√		√
FormulaBuilder	Visual formula modelling		√		√
DBSchema	ER-diagram modelling		√		√
Docbook	Documentation for jABC projects	√	√		√
Eclipse	Eclipse integration for jABC	√	√		√
GEAR	mu-calculus Model checker		√	√	
jETI	Integration of remote services	√			√
JEEWAB	Web technologies support (e.g. J2EE)		√		√
jMosel	Verification with M2L		√		√
LearnLib	Automata Learning & Experimentation		√		√
LocalChecker	Local SIB verification	√		√	
SIBCreator	Automatic SIB generation	√			√
Taxonomy-Editor	SIB taxonomy customization	√		√	
Tracer	Model execution	√	√	√	

Fig. 4. Summary of available jABC Plugins

We use Eclipse [1] because it utilizes a similar plugin approach. SIB experts implement SIB Java classes in close collaboration with the application experts. Different to the usual CVS setup, the SIB expert commits both SIB source and class files to the project repository. In fact, jABC does not compile any source files, it just scans for available SIB classfiles, which are then retrieved from the common project CVS.

The application expert uses the jABC at a completely graphical level to model the application. The pure modelling activity can be complemented by analysis, animation, verification, and execution, which come together with a set of plugins. Usually the set of plugins corresponds to the implemented interfaces in the SIBs.

In the following, we describe the central plugins, which are part of the standard jABC distribution. Fig. 4 shows a list of currently available jABC plugins, here classified according to distinguishing criteria for their nature and aim:

– SIB-level plugins are *locally scoped* in the sense that they concern single SIBs, while SLG-level plugins are *globally scoped*, since they handle whole SLGs,
– jABC *internal* plugins which are used inside the jABC, and *interface* plugins, that provide functionality that interfaces the jABC framework with other worlds, like Webservices, databases, or ERP systems like SAP.

According to the full lifecycle of applications built with the jABC, depicted in Fig. 5, we see that the jABC core, which includes the Local Checker and the Tracer plugins, is used along the full development cycle. Other plugins are more phase specific, like the ITE (Integrated Test Environment) [16] and the LearnLib [7] plugins, which focus on the runtime.

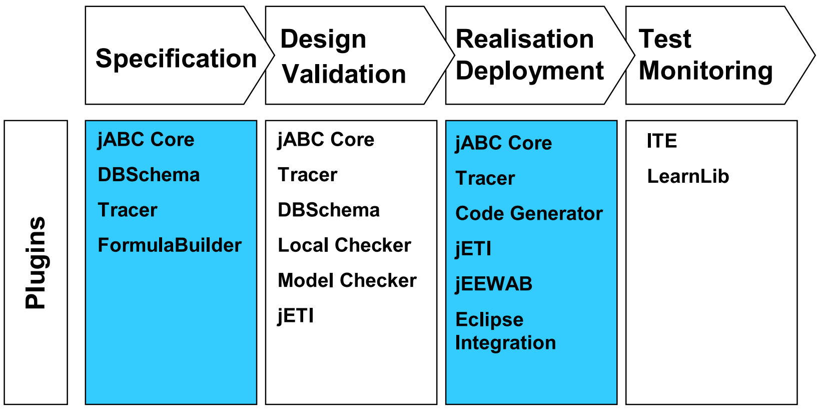


Fig. 5. Model Based Lifecycle Management in jABC

4 The LocalChecker

The LocalChecker-plugin checks whether some assertions (expressed in Java) concerning the single instance of this SIB hold. These assertions are locally specified and concern the correct use of this SIB in its immediate neighbourhood. They concern single instances of user-defined SIBs. They can be used to check the value of a SIB parameter or some conditions of a branch. While relating to many different parameters or branches of a SIB-instance at once, the assertions can be free in complexity yet always simple in the definition formalism. They are formulated in plain Java code, therefore the whole power of the Java language could be used to implement and analyse such properties.

The main advantage of using Java code for local check definitions is that it enables runtime checking of non-trivial properties of a SIB-instance, more involved than standard abstract interpretations. One could for instance need to know whether a given attribute value is contained in an external database. This can be simply solved by embedding the database query in the local check code. For many common checks relating to SIBs and their branches the corresponding rule implementation is already provided with the LocalChecker-Plugin itself, simplifying the use and helping users to reduce the amount of additional work.

As an example consider a SIB representing a table in a database: the name of this table must neither be empty nor too long, it should only consists of valid characters and may not be a reserved word of SQL. In the LocalCheck code the SIB experts test all these different criteria with corresponding Java statements and create warning or error messages, which are presented by the jABC GUI to the user and help users to correct the problems.

5 Model Debugging Via the ModelChecker

Even at an early modelling stage requirements, properties, and general frame conditions emerge which have to be fulfilled by a system or an application.

Besides very local requirements, which only relate to particular parts of a system and which can easily be checked using the LocalChecker, there are also global requirements which are associated with the entire system. These requirements are often very intuitive, are independent of the concrete model, often are part of the rules of the game for an application domain, and can be easily expressed by the application expert during the business logic modelling. E.g., in a web application a logout can only be performed if there was a login at some earlier point in time.

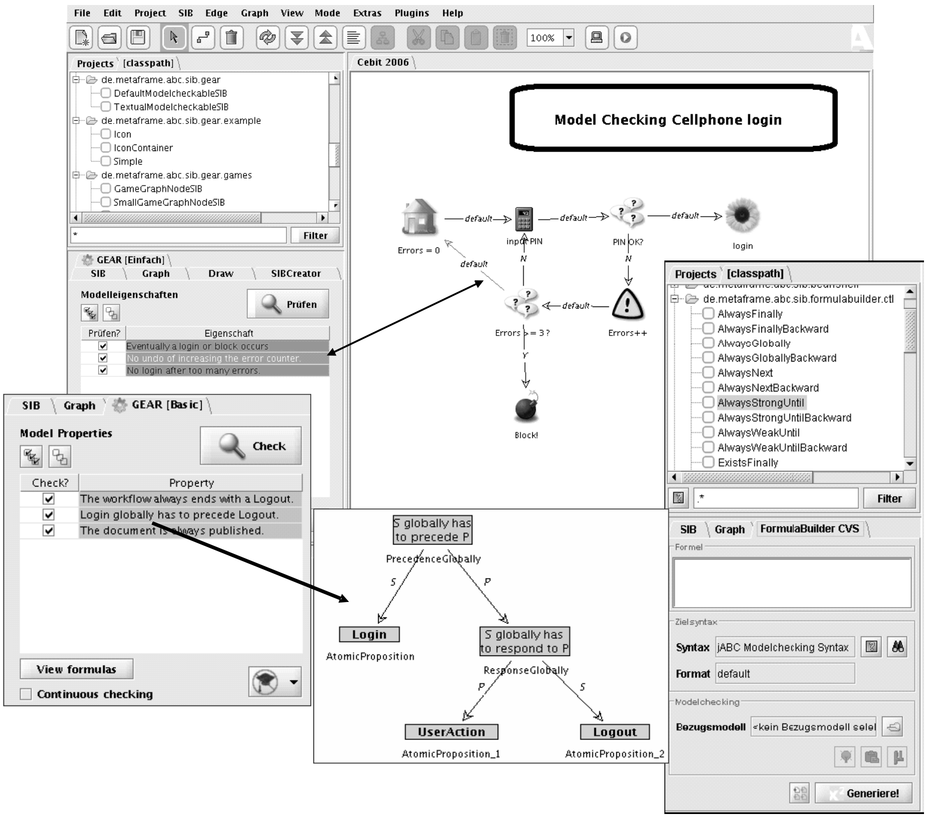
In the last years model checking [8,32] has established itself as a powerful approach to automatic verification of systems. It provides an effective way to determine whether a given system is consistent with a specified property. The jABC framework incorporates this technique via a core plugin called GEAR [6]. Intuitively, any system modelled as SLGs can be verified with this plugin: SLGs consist of SIBs holding labels that the model checker interprets as atomic propositions (for example the SIB names), and have edges annotated with branch names that for the model checker represent actions. Fig. 3 shows an example of such a SLG which models the core part of the jABC code generator for Java classes (see Sect. 7). Properties of such a model have to be specified using appropriate formalisms, in the case of GEAR these are temporal logics, for example CTL (Computation Tree Logic) or the modal μ-calculus [21].

GEAR augments the GUI of the jABC with specific functionalities that enable the user to:

- equip the individual SIBs of a model with atomic propositions,
- add and describe properties,
- check properties for a particular model and
- interactively investigate the error diagnosis information by playing model checking games.

As formal specifications of properties are basically formulas in a temporal logic, which are usually difficult to create and to understand without the required theoretical knowledge, GEAR provides two separate views for using the plugin.

- The *basic view* is designed for users that are unfamiliar with temporal logic-based specification formalisms. Properties here are displayed as natural language descriptions which are tagged according to their validity in the model of interest. E.g., green highlighted properties in Fig. 6 are verified and red highlighted are disproved.
- The *advanced view* addresses experts who know how to create properties using CTL or the modal μ-calculus. If the user enters a formula, its syntax tree is immediately visualized, so that all corresponding subformulae are directly visible. The user may invoke model checking for the whole formula or just for one of its subformulae - the satisfying nodes in the model are then marked accordingly. It is also possible to do *reverse checking*, by using the model checker in the opposite direction. By selecting a subset of nodes in the model of interest it is immediately possible to see which subformulae are satisfied by the selected nodes.



Prüfen?	Eigenschaft
✓	Eventually a login or block occurs
✓	No undo of increasing the error counter.
✓	No login after too many errors.

Check?	Property
✓	The workflow always ends with a Logout.
✓	Login globally has to precede Logout.
✓	The document is always published.

Syntax: jABC Modelchecking Syntax
Format: default
Bezugsmodell: <kein Bezugsmodell selel

Fig. 6. Using the GEAR Model Checker and the FormulaBuilder

To further improve the accessibility of property specifications, the FormulaBuilder plugin [18,17] can be used in conjunction with GEAR. With the FormulaBuilder, also formulae can be modelled as SLGs. Fig. 6 right shows the collection of SIBs for CTL operators. Users can create such graphs also by using commonly-occurring specification patterns based on a system proposed by Dwyer, Avrunin and Corbett [12,11].

Fig. 6 bottom left shows an example of pattern usage: this formula graph corresponds to the template used for the third property checked on this graph: *No login after too many errors*. It uses the patterns *Global Precedence* and *Global Response*. The *Global Precedence* pattern expresses the following property: *Globally, the occurrence of S has to be an enabling condition for the occurrence of P.* In Fig. 6 S and P are instantiated to the values *Login* resp. *Global Response*, so the whole graph models a property that uses hierarchical patterns. From this simple and intuitive graph visualization the FormulaBuilder generates a mu-calculus formula that can be stored, or directly used for model checking in GEAR, for example to verify whether a web application modelled with the jABC satisfies the property.

The manual creation and the comprehension of such formulas is a known difficult task, thus GEAR and the FormulaBuilder together provide an expressive and accessible way for users, even for application experts lacking the theoretical background, to enjoy the benefits of model checking. This goes well in line with the basic goals of LPC we described in section 1.

Being a game based model checker, GEAR also supports interactive error diagnosis by computing and animating winning strategies for modal *mu*-calculus model checking games. Details on this use go beyond the scope of this paper, and are available in [35,31,36].

6 The Tracer

The Tracer plugin adds an execution layer for SLGs to the jABC. The model is thereby interpreted as a *directed control flow diagram* which can be traced comparable to a standard debugger in *run mode* or *step mode* and using *breakpoints* or *pause* to stop the execution. This is done by the Tracer by taking the SIBs from a model as an input to create execution threads, called *Tracer threads*. Each thread contains a number of linear execution steps and is executed by the *Execution environment* of the Tracer, as illustrated in Fig. 7(a).

6.1 The Execution Environment

The execution environment as mentioned above is a runtime environment for the Tracer. For each Tracer thread to be executed, the execution environment uses exactly one *execution stack* containing a single *execution context*. By doing so, the Tracer can execute recursive threads, as variables and invariants are available inside each context. In addition to the execution contexts bound to a single thread there are other contexts with a more global scope. They are available to different SIB instances and they can mutually communicate. Thereby, a global context could also be available through remote interfaces like RMI or JNDI, causing a communication between locally separated models or SIBs. The global execution contexts are usable for different kinds of tasks, for example a context for all available objects or one for just some special kinds of objects.

6.2 Parallel Execution

In general a SIB represents an application feature within the jABC, but it cannot modify the control flow of a modelled application. While talking about a standard SIB, all branches are treated as *alternatives*. A *ControlSIB* overwrites this default setting of a branch and allows choosing two branches in parallel, synchronizing branches, or passing messages between SIBs.

The Tracer is able to execute threads in parallel by using two *ControlSIBs*: *ForkSIB* and *JoinSIB*. A fork divides a thread in any number of subthreads which are independently executed (see Fig. 7(b)) by stopping the current thread and starting a number of new threads. The Tracer waits for each single subthread

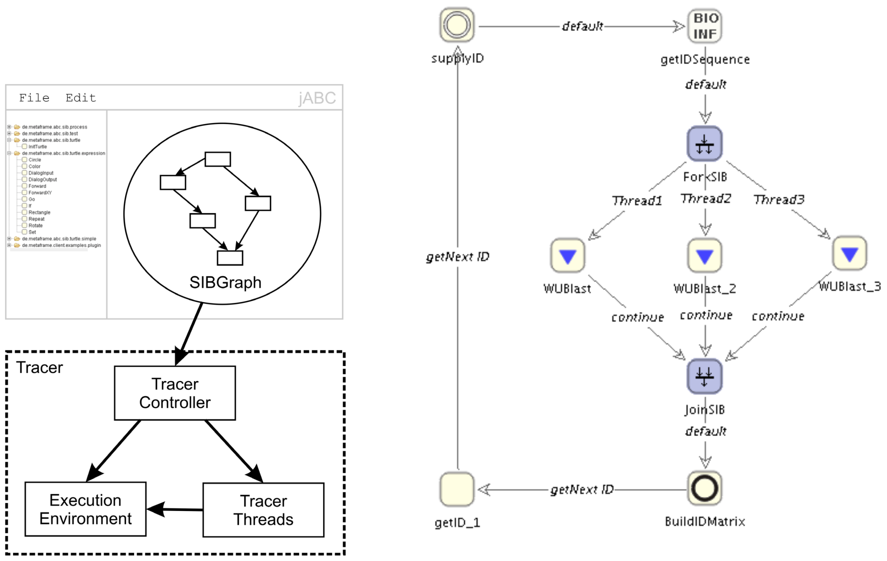


Fig. 7. The Tracer: (a) overview and (b) fork/join parallelism

to end, then the JoinSIB terminates all the subthreads and starts a new single thread to continue the execution of the whole model.

Finally, a whole model can be transformed into a single SIB, called a *Graph-SIB*. These ControlSIBs could be seen in terms of control macros, as they obfuscate complex structures to the user by building a hierarchical control structure within large models.

6.3 Remote Debugging Tool

To comfortably handle the Tracer, the *Remote debug tool* plugin provides a GUI that offers all the features of the Tracer, like starting an execution or executing only a single step. The viewed hierarchy can be chosen freely, allowing the user to control a special scope of the traced model. To visualize the tracing of a graph, the SIBs visited by the Tracer are highlighted during execution. By supporting different underlying communication protocols (like RMI, SOAP, CORBA) the debug tool truly gets ”remote” and therefore it could be even used to control separate tracing processes on different machines (e.g. over the internet).

7 The Code Generator

Once a jABC application is ready for deployment, the current version of the model is often transformed into an executable and deployable piece of code in a desired programming language. This is supported by the jABC Codegenerator plugin, which currently allows generating the control flow of a model into a standard Java class or a Java servlet.

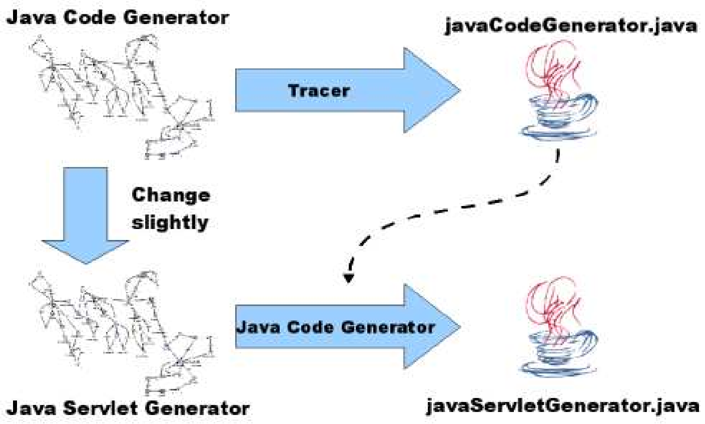


Fig. 8. The development process for the family of Code Generators

The Codegenerator is itself a direct application of the Lightweight Process Coordination approach: it was not programmed by hand, but itself modelled as a SLG within the jABC.

Already our first code generator, the generator for standard Java classes, was completely modelled using the jABC. To assure the correctness of the design, the LocalChecker and the ModelChecker were constantly used during the modelling phase. The result is a SLG that, after parameterization, generates an executable Java class from another SLG. We bootstrapped the generator by executing the model with the Tracer: the execution produces an executable Java class *of itself* - this way we obtained our first code generator as a standalone Java application.

This generator forms the basis for our family of code generators. The generator for Java servlets was achieved by slight modifications to the SLG of the initial Java class generator and subsequent code generation from this SLG via the original Java class generator. The development process for these two is depicted in Fig. 8.

Following this approach, we have a general and easy way of creating new code generators. We envisage here generating J2ME MIDlets, C++ code or even one single SIB from a whole SLG (implemented). By exploiting the advantages of the LPC approach we can profit directly from the code generators already available, achieving high reusability and increasing the efficiency of development.

8 Related Work

There are numerous powerful verification environments/tools that work at the programming language level, like SLAM, Bandera or ESC/Java[1], and others that focus for example on protocol aspects like CADP[2] or on the development of

[1] See the websites at http://research.microsoft.com/slam/, http://bandera.projects.cis.ksu.edu/, and http://secure.ucd.ie/products/opensource/ESCJava2/, respectively.

[2] See the website at http://www.inrialpes.fr/vasy/cadp/

embedded systems like Autofocus[3]. The jABC with its LPC approach is different from them in that it operates on a much higher level of abstraction. We will therefore concentrate the following discussion on approaches which more closely resemble this characteristic trait of the jABC.

Our way of aggressive model-driven development closely relates to the concepts of generative software development, where "a given system can be automatically generated from a specification written in one or more textual or graphical *domain-specific languages*st, as in [10]. Both concepts focus on achieving application-domain and technical flexibility. The notation for specifying a system in our approach is given through the SIBs, which can be considered a domain-specific language since they offer domain-specific functionalities both as concepts in the taxonomy and as design primitives. Thanks to taxonomies and high customizability, the user is free to resort to familiar terminology and to graphical representations that fit the specific application domain. By defining rules for local checking and model checking domain-specific error-checking can be performed, and the framework character of the jABC allows to add any domain-specific tool support by providing new plugins. Technical variability is gained among others through different code generators, which generate a running and deployable system in such a way that it fits virtually any target platform.

Another approach very close to ours is presented in [4]. This approach proposes the use of coordination contracts to promote the separation of the coordination aspects that regulate the way objects interact in a system from the way objects behave internally. As with us, their main concern is supporting evolutionary aspects of the whole system. In their work, contracts fulfill a role similar to architectural connectors: they make these coordination features available as first-class citizens, so that it is possible to treat them distinctly from the functionality of the components.

Contracts are based on superposition mechanisms [20] for supporting forms of dynamic reconfiguration of systems. These mechanisms enable contracts to be added or replaced without the need to change the objects to which they apply. CDE, an environment for developing coordination contracts in Java, is described in [13]. The CDE approach is still programming oriented: unlike our coordination graphs, contracts must be programmed; they do not (yet) support macros or hierarchy, and no automatic verification of contracts is available.

Subject-oriented design [9] is another approach comparable to LPC. The basic idea behind it is the decomposition of standard design models into smaller units called design subjects, which are very close to our SIBs. Just like SIBs, design subjects encapsulate "a single, coherent piece of functionality" [9] and may be built so that they fit the structure of application-specific requirements. Thus subject-oriented designs help to bridge the gap between requirements, which may relate to certain features, and implementation code, which may utilize object-oriented terminology. This is also achieved by our LPC approach, as it propagates a notation and the corresponding tools and framework, which explicitly can be adequately used by both non-programmers and programmers.

[3] See the website (in German) at http://autofocus.in.tum.de/Infos/afinfo-de.html

While the aim of most of the approaches of the five categories named above is very similar to our LPC, their realization is quite different. They work still at much lower level of abstraction, and they typically do not support formal methods-based *verification mechanisms* like model checking.

9 Conclusion

In this paper we have presented the jABC Framework, which incarnates the Lightweight Process Coordination development method. We also introduced the core plugins of the jABC (LocalChecker, the ModelChecker, the Tracer, and the Codegenerator). The set of available jABC plugins is constantly growing and covers a broad range of topics (see [37], [28]). Besides enhancing the range of applicability, these plugins focus on improved validation technology, like improved diagnosis for model checking and run time verifcation based on our integrated test environment.

Several application domains have been successfully covered so far with the jABC: complex supply chain management with IKEA [15], modelling and execution of bioinformatics workflows [27], the Semantic Web Service challenge [22], and dataflow analysis of Java programs [26], a management framework for remote intelligent configuration of systems [5], and an application development platform for Galileo services [14]. These projects were very different in nature, nevertheless we observed a surprisingly high potential of synergy, which was due to the jABC approach.

References

1. *Eclipse Website*. `http://www.eclipse.org/`.
2. *jABC Website*. `http://www.jabc.de`.
3. *Rational Unified Process*. `http://www-306.ibm.com/software/awdtools/rup/`.
4. L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes, M. Wermelinger. Coordination technologies for component-based systems. In Proc. *Integrated Design and Process Technology*, 2002.
5. M. Bajohr, T. Margaria: *MATRICS: A Management Tool for Remote Intelligent Configuration of Systems*, Innovations in System and Software Engineering - a NASA Journal, Springer Verlag, July 2006.
6. M. Bakera and C. Renner. *GEAR - A Model Checking Plugin for the jABC framework*. `http://www.jabc.de/modelchecking/`.
7. T. Berg, H. Raffelt, B. Steffen: *LearnLib: A Library for Automata Learning and Experimentation*, Proc. FMICS'05 (ACM 10th Int. Worksh. on Formal Methods for Industrial Critical Systems), Sept. 2005, Lissabon (P), ACM Press.
8. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.
9. S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, 1999.
10. K. Czarnecki. Overview of generative software development. In *UPP*, pages 326–341, 2004.
11. M. Dwyer, G. Avrunin, J. Corbett. *Specification Patterns Website*. `http://patterns.projects.cis.ksu.edu/`.

12. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In Proc. *ICSE'99*, pp. 411–420, Los Alamitos, CA, 1999. IEEE CS Press.
13. J. Gouveia, G. Koutsoukos, L. Andrade, J. L. Fiadeiro. Tool support for coordination-based software evolution. Proc. *TOOLS'01: Technology of Object-Oriented Languages and Systems*, p.184, Washington DC, 2001. IEEE CS Press.
14. M. Högl, T. Margaria, B. Steffen: *The GalileoGate Solution Factory for Location-Based Integrated Services*, Proc. IDPT 2006, Int. Conf. on Integrated Design and Process Technologies, San Diego, June 2006.
15. M. Hörmann, T. Margaria, T. Mender, R. Nagel, M. Schuster, B.Steffen, H. Trinh: *The jABC Appraoch to Collaborative Development of Embedded Applications*, CCE'06, Int. Workshop on Challenges in Collaborative Engineering - State of the Art and Future Challenges on collaborative Design, Prag (CZ), April 2006 (Industry day).
16. H. Hungar, T. Margaria, B. Steffen: *Test-Based Model Generation for Legacy Systems*, Proc. IEEE ITC'03, Charlotte, 2003, IEEE CS Press, pp.971–980.
17. S. Jörges. *FormulaBuilder Website*. `http://www.jabc.de/formulabuilder/`.
18. S. Jörges, T. Margaria, and B. Steffen. Formulabuilder: A tool for graph-based modelling and generation of formulae. In Proc. *ICSE'06* Shanghai, May 2006.
19. M. Karusseit, T. Margaria: *Feature-based Modelling of a Complex, Online-Reconfigurable Decision Support Service*, WWV'05, 1st Int. Worksh. Automated Specif. and Verification of Web Sites, Valencia, March 2005, ENTCS N. 1132.
20. S. Katz. A superimposition control construct for distributed systems. *ACM TOPLAS.*, 15(2):337–356, 1993.
21. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
22. C. Kubczak, R. Nagel, T. Margaria, B. Steffen: *The jABC Approach to Mediation and Choreography*, Semantic Web Services Challenge 2006, Phase I-III Workshops, DERI, Stanford University, U. of Georgia, March-November 2006.
23. ITU: *General recommendations on telephone switching and signaling - intelligent network: Introduction to intelligent network capability set 1*, Recommendation Q.1211, Telecommunic. Standardization Sector of ITU, Geneva, Mar. 1993.
24. ITU-T: *Recommendation Q.1203. "Intelligent Network - Global Functional Plane Architecture"*, Oct. 1992.
25. ITU-T: *Recommendation Q.1204. "Distributed Functional Plane for Intelligent Network Capability Set 2: Parts 1-4"*, Sept. 1997.
26. A.L. Lamprecht, T. Margaria, B.Steffen: *Data-Flow Analy-sis as Model Checking within the jABC*, Proc. CC'06, 15th Int. Conf. on Compiler Construction, Vienna (A), March 2006, LNCS, 3923, Springer Verlag, pp. 101-104.
27. T. Margaria, C. Kubczak, M. Njoku, B. Steffen: *Model-based Design of Distributed Collaborative Bioinformatics Processes in the jABC*, Proc. ICECCS 2006, Stanford Univ., CA (USA), August 2006, IEEE CS Press.
28. T. Margaria, R. Nagel, B. Steffen: *Remote Integration and Coordination of Verification Tools in JETI*, Proc. IEEE ECBS 2005, April 2005, Greenbelt (USA), IEEE CS Press, pp. 431–436.
29. T. Margaria and B. Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. *STTT*, 5(2-3):107–123, 2004.
30. T. Margaria, B. Steffen, M. Reitenspieß: *Service-Oriented Design: The Roots*, IC-SOC 2005: 3rd ACM SIGSOFT/SIGWEB Int. Conf. on Service-Oriented Computing, Amsterdam, Dec. 2005, LNCS 3826, pp. 450-464, Springer Verlag.

31. M. Müller-Olm and H. Yoo. Metagame: An animation tool for model-checking games. In *TACAS 04, LNCS 2988*, pages 163–167. Springer-Verlag, 2004.
32. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
33. B. Steffen, B. Freitag, A. Claßen, T. Margaria, and U. Zukowski. *Intelligent Software Synthesis in the "DaCapo" Environment*In Proc. 6th /Nordic Workshop on Programming Theory/, Aarhus (DK), October 1994, BRICS Report N. 94/6, December 1994.
34. B. Steffen, T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In *Correct System Design - Recent Insights and Advances*, LNCS N. 1710, State-of-the-Art Survey, pp. 390–415. Springer-Verlag, 1999.
35. C. Stirling and P. Stevens. Practical model-checking using games. Proc. *TACAS 98*, LNCS N.1384, pp. 85–101. Springer-Verlag, 1998.
36. W. Thomas. On the synthesis of strategies in infinite games. Proc.*STACS'95*, LNCS N.900, pp.1-13. Springer-V., 1995.
37. C. Topnik, E. Wilhelm, T. Margaria, B. Steffen: *jMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str)*, Proc. SPIN'06, 13th Int. Works. on Model Checking of Software, Vienna, April 2006, LNCS 3925, Springer V., pp.293-298.