DOI 10.1007/s00165-010-0169-9
BCS © 2010
Formal Aspects of Computing (2011) 23: 589–606

Formal Aspects of Computing

Assuring property conformance of code generators via model checking

Sven Jörges[1], Tiziana Margaria[2] and Bernhard Steffen[1]

[1] Chair of Programming Systems, Technische Universität Dortmund, Dortmund, Germany. E-mail: sven.joerges@tu-dortmund.de
[2] Service and Software Engineering, Universität Potsdam, Potsdam, Germany

Abstract. Automatic code generation is an essential cornerstone of today's model-driven approaches to software engineering. Thus a key requirement for the success of this technique is the reliability and correctness of code generators. This article describes how we employ standard model checking-based verification to check that code generator models developed within our code generation framework Genesys conform to (temporal) properties. Genesys is a graphical framework for the high-level construction of code generators on the basis of an extensible library of well-defined building blocks along the lines of the Extreme Model-Driven Development paradigm. We will illustrate our verification approach by examining complex constraints for code generators, which even span entire model hierarchies. We also show how this leads to a knowledge base of rules for code generators, which we constantly extend by e.g. combining constraints to bigger constraints, or by deriving common patterns from structurally similar constraints. In our experience, the development of code generators with Genesys boils down to re-instantiating patterns or slightly modifying the graphical process model, activities which are strongly supported by verification facilities presented in this article.

Keywords: Extreme Model-Driven Development, Code generation, Model checking, Verification

1. Introduction

Automatic code generation is a key feature of model-driven approaches to software engineering. It has several advantages such as the elimination of manual coding errors, and it provides a fast track to a deployable and testable system/application. Furthermore, it disburdens developers from writing boilerplate code, which is often a highly repetitive and cumbersome task, and by doing this, it shifts the attention back to the primary concern, the application-level logic. Of course, an indispensable requirement for the success of this approach is the reliability and correctness of the corresponding code generators.

In [JMS08], we presented Genesys, a framework for the high-level construction of code generators along the lines of the Extreme Model-Driven Development (XMDD, see Sect. 2) paradigm. In this framework, code generators are modeled on the basis of an extensible library of well-defined building blocks. We showed that constructing code generators this way offers several advantages, such as the high potential for reuse:

Correspondence and offprint requests to: S. Jörges, E-mail: sven.joerges@tu-dortmund.de

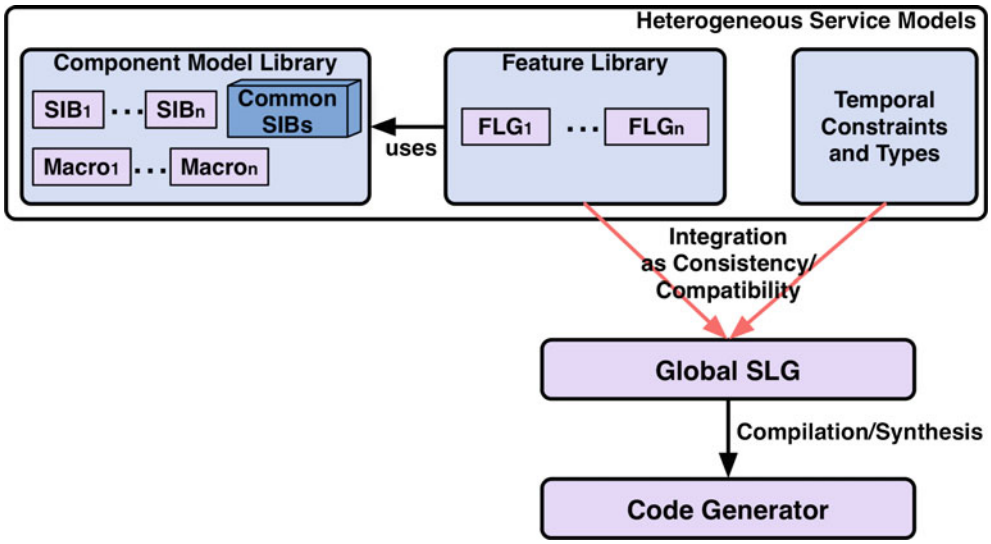

Fig. 1. XMDD with jABC and Genesys

The building blocks, as well as whole parts of code generators covering features or aspects like error-handling or input/output-related tasks, can easily be reused for constructing new code generators. This enables short development cycles and a fast evolution of the generation library. We also showed that the approach perfectly integrates with other classical, well-established concepts, such as bootstrapping from the field of compiler construction.

In this article we focus on another advantage. The XMDD approach comprises model checking-based [CGP01, QS82] verification, which can be applied to the code generators: As the code generators are realized as formal models, they are amenable to such techniques. This enables us to check code generation rules or constraints, e.g. guaranteeing the complete processing of all input data or the correct order of generation steps. Genesys contains a large and steadily growing library of such constraints, which greatly improves the overall quality and reliability of the code generators.

In [JMS08], we already briefly motivated the use of model checking in Genesys with several simple example constraints. As the main contribution of this article, we want to elaborate on this by examining more complex constraints for code generators, which even span entire model hierarchies. We also show how we constantly extend our knowledge base for code generation by e.g. combining constraints to bigger constraints, or by deriving common patterns from structurally similar constraints. For our experiments, we used the model checker GEAR [BMRS07a, BMRS07b] to check whether the code generators conform to the constraints, which we specified graphically using the FormulaBuilder [JMS06, JMS08].

In our experience, the development of code generators with Genesys boils down to re-instantiating patterns or slightly modifying the graphical process model. These activities are strongly supported by the application of model checking presented in this article, as it helps leveraging the increasing body of domain knowledge during code generator construction.

In the following sections, we first will describe jABC, which is a basic framework that enables the development according to XMDD (Sect. 2). Afterwards, we present Genesys, which is based on the jABC (Sect. 3), and we briefly recapitulate the main ideas presented in [JMS08]. Section 4 outlines the verification facilities of the jABC, which we apply to check the property conformance of Genesys' code generators. Subsequently Sect. 5, the main section, describes and exemplifies the verification of code generators along an elaborate example. Finally, we discuss some related work (Sect. 6), before we conclude with Sect. 7.

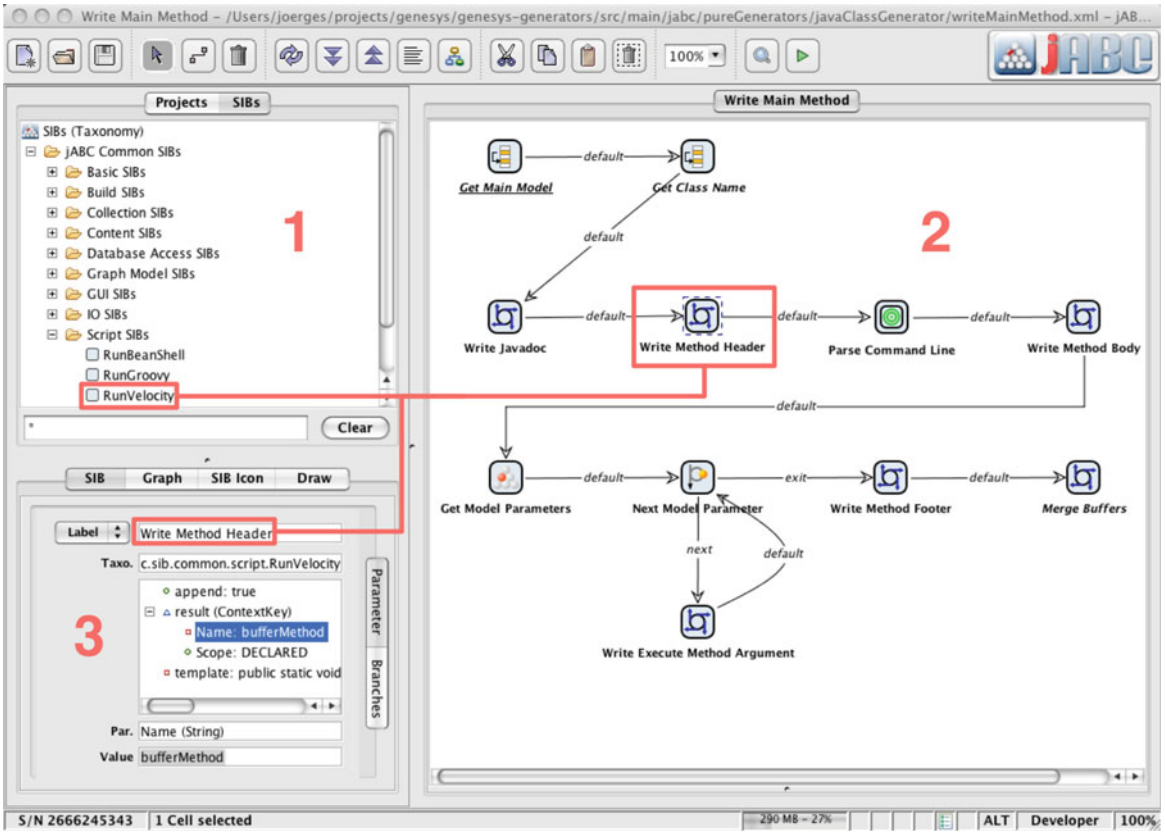

Fig. 2. The user interface of jABC

2. The jABC framework

The code generation framework Genesys is an integral part of jABC, which follows the ideas of XMDD [MS08]. XMDD is a new development paradigm designed to continuously involve the customer/application expert throughout the whole system's life cycle. In technical practice, user-level models are successively enriched and refined from the user perspective, until a sufficient level of detail is reached. At this level, elementary services, solving tasks at the application level, can be implemented. The realization of the individual services should typically be simple and is often based on functionality provided by third-party and standard software systems. As the continuously enriched model is the central and sole artifact of this methodology, which comprises all relevant information including documentation, access policies, versioning, etc., we also call this the "One-Thing Approach" [SN07, MS09].

jABC is a flexible framework designed to support systematic development according to the XMDD paradigm (see Fig. 1). It allows users to develop service-oriented systems by composing reusable building blocks into flow graph structures, called *Service Logic Graphs* (SLG). The building blocks are called *Service Independent Building Blocks* (SIBs) [ITU93, IT92], and may represent a single atomic service or also a whole subgraph (i.e. another SLG, called *features* in Fig. 1). Thus SLGs can be hierarchical, which facilitates the refinement along the lines of the One-Thing Approach. This also grants a high reusability not only of the building blocks, but also of the models themselves, within larger systems. Besides SIBs and hierarchical SLGs, there are *constraints* defining e.g. the structure of the underlying meta model, global frame conditions and business rules, and which can be verified by formal methods like model checking (see Sect. 4). Finally, the modeled SLG can be compiled to a stand-alone running system. This compilation part is the task of Genesys. Furthermore, an extensible set of jABC plugins provides additional functionality that adequately supports all the activities needed along the development life-cycle.

Figure 2 shows a screenshot of jABC's user interface. The GUI consists of three main parts (indicated by the numbers):

1. the *project and SIB explorers*, which enable the user to browse available jABC projects and the library of building blocks that can be used for modeling,
2. the *graph canvas*, which is used for composing SLGs, and
3. the *inspectors*, which provide detailed information about selected SIBs and may be used by plugins to add further functionality.

Several application domains have been successfully covered with the jABC, including complex supply chain management with IKEA [HMM+08], modeling and execution of bioinformatics workflows [MKS07], the Semantic Web Service challenge [KMSN08], dataflow analysis of Java programs [LMS06a], a management framework for remote intelligent configuration of systems [BM06], online decision support systems [KM06] and graphical construction of game strategies [BJM09]. These projects were very different in nature, nevertheless we observed a surprisingly high potential of synergy, which was due to the XMDD approach.

In the following sections, we will elaborate on how SIBs and SLGs are defined, constructed and organized.

2.1. Service Independent Building Blocks

A SIB is an abstract and generic representation of some service or functionality. In order to provide configurability, each SIB contains a set of *formal parameters*. Via those parameters, a SIB's behavior can be customized depending on the current context of use. In a model, SIBs are then wired based on the possible results of their execution, reflected by a list of outgoing *branches*. Roughly speaking, branches can be considered the "exits" of a building block. SIBs and their constituents are described using simple Java classes.

For the graphical representation to the jABC user, each SIB consists of an icon and a documentation of its constituents and its purpose. Furthermore, SIBs are semantically classified in terms of *taxonomies*. A taxonomy is a directed acyclic graph: sinks represent SIBs, which are atomic entities in the taxonomy, and intermediate nodes represent groups, that is sets of modules satisfying some basic property (expressed as predicates). This structure is freely defined by domain experts, who organize, label and preconfigure SIBs so that they fit the actual domain.

A SIB's implementation (i.e. its execution behavior) is realized by one or more *service adapters*. Each service adapter contains execution code for a particular SIB. In most cases, this includes data type conversions and calls to third-party libraries that implement the runtime functionality of the SIB. As a SIB's execution behavior may be different depending on the execution environment, an arbitrary number of service adapters can be assigned to one SIB. For instance, imagine a SIB that reads in customer data for a shop application: in a staging or testing environment, this information might come from a text file containing dummy data, whereas in a live scenario the customer data might be stored in a database system. These different behavioral patterns would be realized by two service adapters, one for the staging system and one for the live system. Service adapters assure that the execution of a SIB is entirely platform-independent: the same SIB may be executable in a standard Java environment, a .NET setup and on a mobile phone, if it provides corresponding service adapters. Thus from the modeling perspective, the usage of a SIB completely virtualizes the target platform on which it will be executed.

Figure 2 contains an example of a SIB and its constituents. In the canvas (2), we highlighted a SIB instance labelled "Write Method Header". This is an instance of the SIB "RunVelocity", which can be found inside the taxonomy displayed in the SIB explorer (1). In this taxonomy, the SIB is labeled as part of the "Script SIBs", which in turn are classified as "jABC Common SIBs". The latter denotes a big SIB library that is shipped with jABC, and that provides ready-made SIBs for very general, typically quite low-level functionality useful for almost any application [The08]. The task of the "RunVelocity" SIB is to employ the Velocity template engine [Apa07] to evaluate a template, which is specified by setting one of the SIB's parameters. Such a template is written in the Velocity Template Language (VTL), and is basically a textual skeleton containing placeholders which are filled with dynamic content as soon as the SIB is executed. Afterwards, the result of the evaluation is pushed into the so-called execution context. For the scope of this article, it is sufficient to think of the execution context as a shared memory, which allows SIBs to communicate with each other during the execution.

As visible from the SIB inspector in Fig. 2 (3), the "RunVelocity"-SIB takes three parameters:

- *template*: The Velocity template to be evaluated. In the case of the "Write Method Header" instance, this template is used to generate the signature and first lines of a Java `main` method.
- *result*: The key which is used to store the evaluation result in the execution context. For "Write Method Header", this key is `bufferMethod`.
- *append*: If this boolean flag is set to `true`, and if there is already a buffer in the execution context identified by the key "result", the evaluation result is appended.

Furthermore, the SIB has two branches (not visible in Fig. 2): *default*, if the template was evaluated successfully, and *error*, if the template could not be evaluated (e.g. because of syntax errors). As most of Genesys' code generators are template-based, the "RunVelocity" SIB is used very frequently in this context.

2.2. Service Logic Graphs

In the jABC, Service Logic Graphs (SLGs) are internally modeled as Kripke Transition Systems (KTS, [MOSS99]) whose nodes represent elementary SIBs and whose edges represent branching conditions (see Fig. 2):

Definition 1 (Kripke Transition System, KTS) A KTS $(V, AP, I, Act, \rightarrow)$ consists of a set of nodes V and a set of atomic propositions AP describing basic properties for a node. The interpretation function $I : V \rightarrow 2^{AP}$ specifies which propositions hold at which node. A set of action labels Act is used to designate branching conditions. The possible transitions between nodes are given by the relation $\rightarrow \subseteq V \times Act \times V$.

Through this non-standard abstraction in our model we obtain a *separation of concerns* between

- the control-oriented modeling layer, where the user is not troubled with implementation details while designing or evaluating the applications, and
- the underlying data-oriented communication mechanisms enforced between the participating subsystems, which are hidden in the SIB implementation.

As a key characteristic, jABC facilitates hierarchical design. SLGs are allowed to make full use of other already existing SLGs (cf. [SMBK97] for a detailed discussion). Figure 3 shows an example of a hierarchical SLG with a three-level hierarchy: SIBs with a green dot in the middle (e.g. "GenerateSIBGraphCells") are so-called *macros* which reference a submodel and thus realize hierarchy. Just like any other SIB, such a macro has parameters (*model parameters*) and outgoing branches (*model branches*). The parameters of a macro can be mapped to (selected) parameters of the underlying SIBs. Similarly, the set of (un-set) outgoing branches of the underlying SIBs defines the outgoing branches of the macro. For a detailed description of the hierarchical SLG in Fig. 3 please refer to Sect. 3. Please note that, in contrast to comparable approaches like e.g. the Business Process Execution Language (BPEL, [OA07]), a big advantage of SLGs is their direct and clean formal semantics.

3. The Genesys framework

Once the SLG of a system or application is fully designed and all SIBs are implemented, it is ready for deployment. The SLG has to be transformed into an executable and deployable artifact, usually a piece of code in a desired programming language suitable for a particular execution environment. This is the task of Genesys [JMS08], which provides means for the construction of highly specialized code generators that are made available to the user via a jABC plugin.

Furthermore, Genesys is a framework that enables the construction of code generators along the lines of XMDD. Accordingly, all code generators in Genesys are themselves designed and built as SLGs within the jABC. In [JMS08], we already described this concept, and we also discussed the various advantages arising from it, such as the possibility of bootstrapping and a high potential for reuse of already existing components. As the focus of this article is rather on the verification of Genesys' code generators, we will only provide a short overview of the main ideas and concepts presented in [JMS08].

Bootstrapping and evolution Via a jABC plugin called Tracer [SMN+06], SLGs can be directly executed (animated), which enables rapid prototyping and debugging of modeled systems. In other words: the Tracer is an interpreter for SLGs, which also enables the immediate execution of code generators modeled in jABC. This allows us to perform bootstrapping: by tracing a code generator's SLG, we can apply this code generator to itself and gain the same code generator, implemented in another language. The further evolution then mostly boils down to simple parameterization: As currently most of the generators provided by Genesys are template-based, the typical way to build a new generator is to modify the templates (that is, to parameterize the SIBs) of an existing generator accordingly.

Reuse of components In [JMS08], we described the genealogy of Genesys' code generators, showing how the different generators emerged from each other. Corresponding to the ideas of XMDD, a rich pool of existing components accelerates the development of new code generators. These components can be:

- SIBs, which mostly come from jABC's Common SIBs (see 2.1), or in rare cases have to be newly implemented, and
- SLGs, which model features or aspects that are reusable among different code generators.

For instance, we derived a code generator for C# by almost entirely reusing a code generator for Java Servlets. Besides parameterization of contained SIBs and SLGs (e.g. modification of templates to contain C# rather than Java syntax), virtually no modifications of the SLG's workflow were necessary.

Extruders and Pure Generators During the early development of Genesys, we recognized that the code generators can be divided into two different classes: Extruders and Pure Generators. These classes differ in the applied strategies for mapping SLGs into executable source code, as well as in the extent to which jABC features are required for the execution of the generated source code.

Generators belonging to the Extruder class simply use the Tracer for the execution of the generated code. This is perfectly possible as the Tracer provides an API for interpreting SLGs without the jABC user interface. Furthermore, Extruders directly use jABC's graph data structures to represent the SLGs in the generated code. Both, the use of the Tracer as well as the use of jABC's data structures, cause the generated code to depend on the jABC framework during compilation and runtime. Furthermore, the execution with the Tracer causes some runtime overhead. All this may be problematic for application domains with strong performance requirements or memory limitations as it is often the case for embedded systems.

The big advantage of Extruders is their simplicity, as they support all advanced Tracer features, like e.g. thread and event handling. Furthermore, Extruders automatically profit from every new Tracer feature. For Pure Generators, neither the jABC Tracer nor the jABC Framework are required. They solely base on functionality provided by the considered runtime environment, like e.g. the Java Runtime Environment (JRE). Moreover, code produced by Pure Generators is usually smaller and faster. These advantages come of course at a rather high price: to profit from complex Tracer features, they need to be specifically programmed.

Figure 3 shows a hierarchical SLG that models a part of a code generator for Java classes. This SLG is shared by all code generators following the Extruder approach and targeting a Java-based platform (such as the Java Standard Edition, JSE). The depicted SLG hierarchy is responsible for processing the input SLGs for which code will be generated. In the Extruder approach, a Java method is generated for each input model. Each method then contains code for all constituents of an SLG, e.g. for SIBs, parameters, edges, etc. When such a method is executed, it basically reconstructs the corresponding SLG using jABC's graph data structures, which is understood by and passed to the Tracer. Please note that the depicted SLG hierarchy is truncated for the sake of presentability. This is sufficient as our example focuses on the generation of code for SIB parameters, leaving out all the other constituents of an SLG.

The SLG marked with number 1 in Fig. 3 first iterates over all input models (SIB "Next Model"). Via an instance of the "RunVelocity" SIB introduced in Sect. 2.1, it then generates the header of the method ("GenerateMethodHeader"). Afterwards, there are two macros (see Sect. 2.2) containing further SLGs that generate code for the SIBs (the SLG marked with 2), for the SLG's edges and for its model parameters. Finally, another "RunVelocity" SIB is used to generate the remainder of the method ("GenerateMethodFooter").

SLG 2 is responsible for generating code for all SIBs contained in an input model. First, the current input model and its SIBs are retrieved. Afterwards, the SLG simply iterates over the SIBs ("Next SIB Graph Cell"), and generates code for each by calling SLG 3, which is again embedded via a macro.

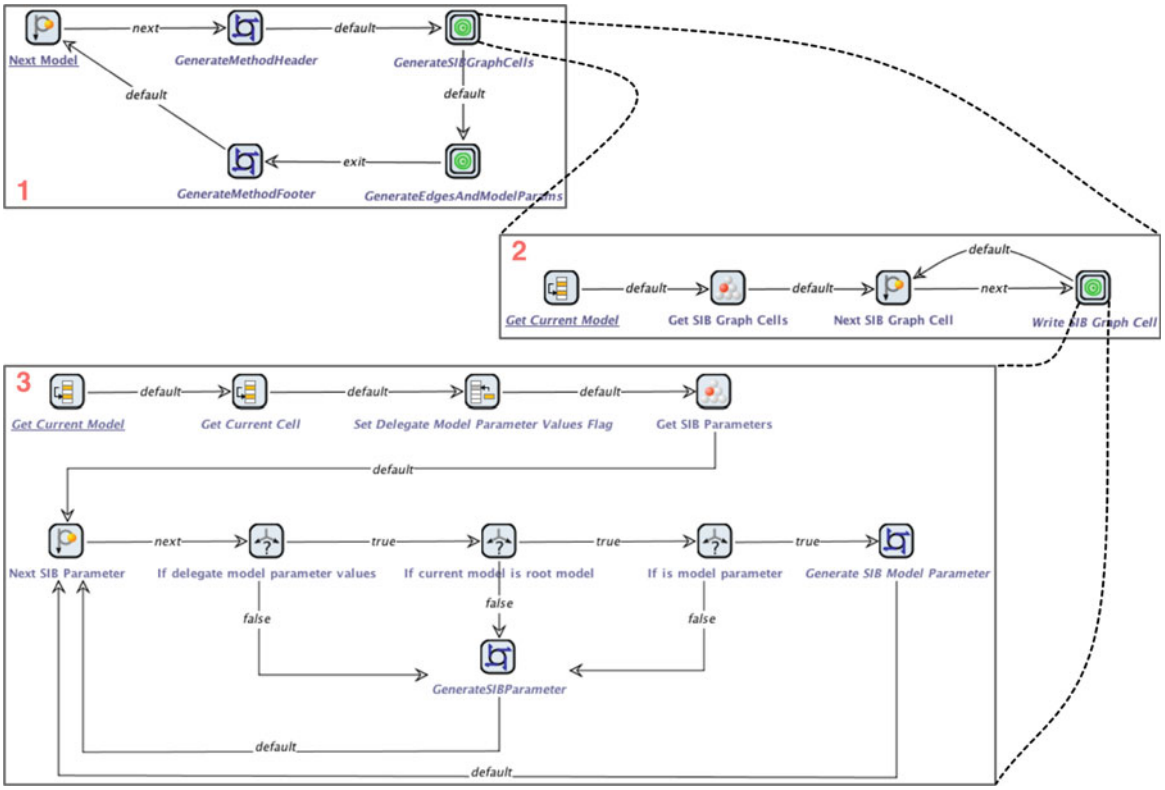

Fig. 3. A hierarchical code generator SLG (shared by all Extruders)

SLG 3 first retrieves the current model, the current SIB and its parameters. Then it iterates over the SIB's parameters ("Next SIB Parameter") and generates corresponding code via "RunVelocity" SIBs, depending on whether the current parameter is a normal SIB parameter ("GenerateSIBParameter") or exported as a model parameter ("Generate SIB Model Parameter").

For this article, it is not necessary to understand all details of this code generation process. The modus operandi of all Genesys code generators is pretty standard: The generator reads the input models and then iterates all contained elements in order to retrieve all information necessary to produce corresponding code. However, the example in Fig. 3 provides an impression of how such generation processes are expressed as jABC models, and finally forms the basis for the case study presented in the following sections.

Besides the advantages given above, the Genesys approach to the development of code generators also enables the use of sophisticated verification techniques like model checking, which we already motivated in [JMS08]. In Sect. 5, we will further elaborate on how we used model checking to assure the property conformance of code generators.

4. Model verification

As visible from Fig. 1, safeguarding the consistency and compatibility during the integration of a system is key to the XMDD paradigm. Thus constraints or business rules, which usually arise from the given application domain, have to be defined. While assembling a global system SLG from SIBs, macros and other SLGs, these constraints have to checked continuously, in order to ensure that the created system is executable and translatable to working code.

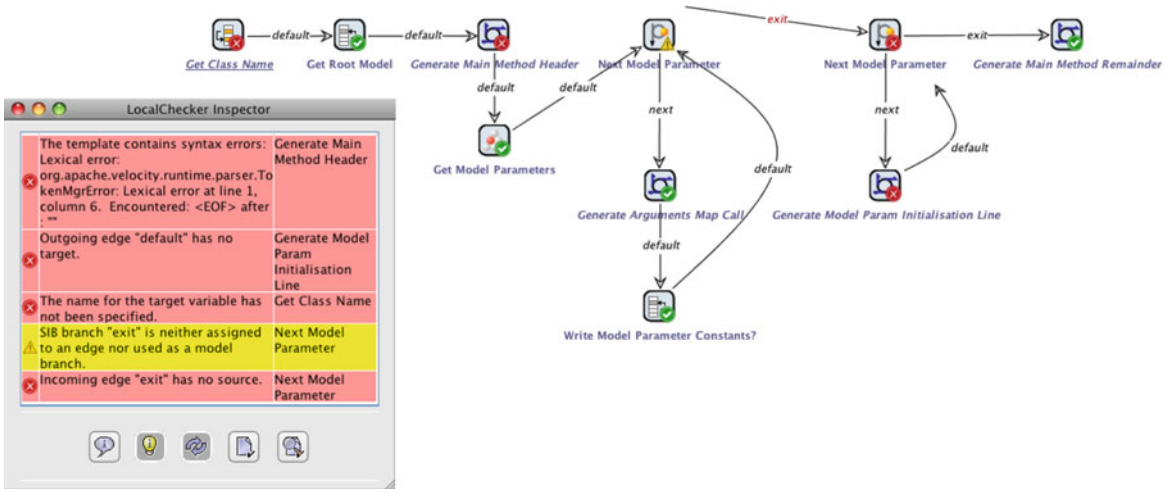

Fig. 4. Checking local constraints with the LocalChecker

For the verification of SLGs, we distinguish between *local* and *global* constraints. jABC allows to check both types of constraints via corresponding tools (jABC plugins), the LocalChecker and the model checker GEAR. In the following, we will briefly introduce both types of constraints and the respective check tools.

4.1. Checking local constraints with the LocalChecker

Local constraints relate to specific parts of the application, i.e. to its components. In the context of SLGs, these components are the SIBs used for modeling. Constraints for SIBs usually have to do with e.g. correct parameterization, the valid assignment of branches etc.

Such simple local conditions are verified using the LocalChecker [SMN+06] plugin of jABC. The check conditions are explicitly implemented in the Java class representing a SIB. If conditions are not fulfilled while modeling, the LocalChecker immediately informs the user with messages categorized according to their severity, e.g. as warnings, errors or fatal errors, as displayed on the left side of Fig. 4. This ensures the correct usage of building blocks and thus prevents common modeling mistakes.

For assuring the correct appliance of SIBs in Genesys' code generators, we enforce the use of all standard checks that are provided out-of-the-box by the LocalChecker. This impedes common modeling errors such as edges without any targets (which is tantamount to a breach in the execution flow), missing branch assignments (leading to inaccessible execution paths) or the use of SIBs which are not suitable for building code generators (because they e.g. originate from another, incompatible application domain).

An important example of using local checks is the "RunVelocity" SIB. As already outlined in Sect. 2.1, this SIB takes a Velocity template as a parameter. One necessary local check of course is to check for the bare existence of a corresponding template, another application is to check the validity of the template's syntax. A result from the latter check is visible in Fig. 4: The first message in the inspector on the left reports a syntax error in the Velocity template belonging to the SIB "Generate Main Method Header", which can be found in the SLG on the right. Figure 4 also exemplifies several other messages typically emitted by the LocalChecker. As a further example, we use local checks to control whether certain project guidelines are followed properly. For instance, SIBs which are not documented always lead to a warning.

4.2. Checking global constraints with GEAR

In contrast to local constraints, global constraints usually span the entire application, sometimes even the whole domain or frame conditions enforced by the corresponding meta model. The ideas formulated by such constraints seem often very simple, as they mostly require the consistency between actions taken at different points of the configuration or functioning of the system, e.g. in order to guarantee well-formedness, executability, etc.

Take for instance the "RunVelocity" SIB we presented in Sect. 2.1, which demands a Velocity template as a parameter. Of course, it is essential for a code generator which e.g. generates Java code, that only SIBs that specify Java templates are used—a constraint which has to be satisfied in order for the code generator to work properly.

If the system is designed in a model-driven way, there are automated ways of proving this consistency of actions. Model checking [CGP01, QS82] is a powerful approach to automatic verification of behavioral models, as it provides an effective way to determine whether a given system model is consistent with a specified (temporal) property. The jABC framework incorporates this technique via the model checker GEAR [BMRS07a, BMRS07b], which is integrated as a plugin. Intuitively, any system modeled as SLGs can be verified with this tool: As described in Sect. 2.2, SLGs can be seen as KTS including atomic propositions and actions. Please note that the results presented below are not restricted to the use of GEAR—any model checker such as SPIN [Hol03] or SMV [CGP01] could be easily integrated into jABC using its plugin mechanism.

Global constraints usually are defined using appropriate formalisms. In the case of GEAR, these are temporal logics, for example *CTL* (Computation Tree Logic). As we will use CTL for textually describing the constraints that are to be discussed later on in Sect. 5, we will now briefly introduce the logic.

CTL can be used to formulate temporal constraints of a model which e.g. are concerned with the reachability of certain states. For the specification of such formulas CTL offers the path quantifiers A and E as well as the temporal operators F, U (and its weak variant WU), X and G. CTL belongs to the branching-time logics, i.e. its temporal operators quantify over computation paths that start in a specific state of the model. CTL formulas can be constructed as follows (p represents atomic propositions):

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid EX\phi \mid EG\phi \mid E[\phi_1 U \phi_2]$$

The intuition behind these operators is as follows: the validity of an atomic proposition p is checked relative to a valuation function associating atomic propositions with states. The following two operators are the standard negation and disjunction, while

- $EX\phi$ requires ϕ to hold in one successor state,
- $EG\phi$ requires ϕ to hold continuously along one leaving path, and
- $E[\phi_1 U \phi_2]$ requires ϕ_1 to hold until ϕ_2 eventually holds along one leaving path.

For the formal semantics of CTL formulas please refer to [CGP01].

With the help of the minimal grammar given above, all the other CTL operators can easily be constructed [CGP01]:

$$\phi_1 \wedge \phi_2 \equiv \neg(\neg\phi_1 \vee \neg\phi_2) \qquad \phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$$
$$AX\phi \equiv \neg EX(\neg\phi) \qquad EF\phi \equiv E[true\ U\ \phi]$$
$$AG\phi \equiv \neg EF(\neg\phi) \qquad AF\phi \equiv \neg EG(\neg\phi)$$
$$A[\phi_1\ U\ \phi_2] \equiv \neg E[\neg\phi_2\ U\ (\neg\phi_1 \wedge \neg\phi_2)] \wedge \neg EG(\neg\phi_2) \qquad A[\phi_1\ WU\ \phi_2] \equiv \neg E[\neg\phi_1\ U\ \neg\phi_2]$$
$$E[\phi_1\ WU\ \phi_2] \equiv \neg A[\neg\phi_1\ U\ \neg\phi_2]$$

The $A[\phi_1\ WU\ \phi_2]$ will be used in the complex constraints discussed in Sect. 5. It denotes a weak variant of the until operator, that requires ϕ_1 to hold until ϕ_2 holds along all leaving paths. Please note that in contrast to the strong until operator, ϕ_2 is not required to occur in cases where ϕ_1 continues to hold indefinitely.

GEAR also supports the specification of constraints using other logics like e.g. Allen's Temporal Logic (ATL, [All83]). Internally, these input logics are mapped to modal μ-calculus [Koz83]. Furthermore, GEAR's input syntax is extensible via so-called macros (not to be confused with macro SIBs that are used for hierarchical SLGs!). Basically, a macro is an abbreviation or a pattern that represents a specific formula, thus leading to more readable and concise formulas. For instance, GEAR's macro mechanism is used to incorporate the property specification patterns proposed by Dwyer et al. [DAC99], and it is also used for mapping CTL and ATL to modal μ-calculus.

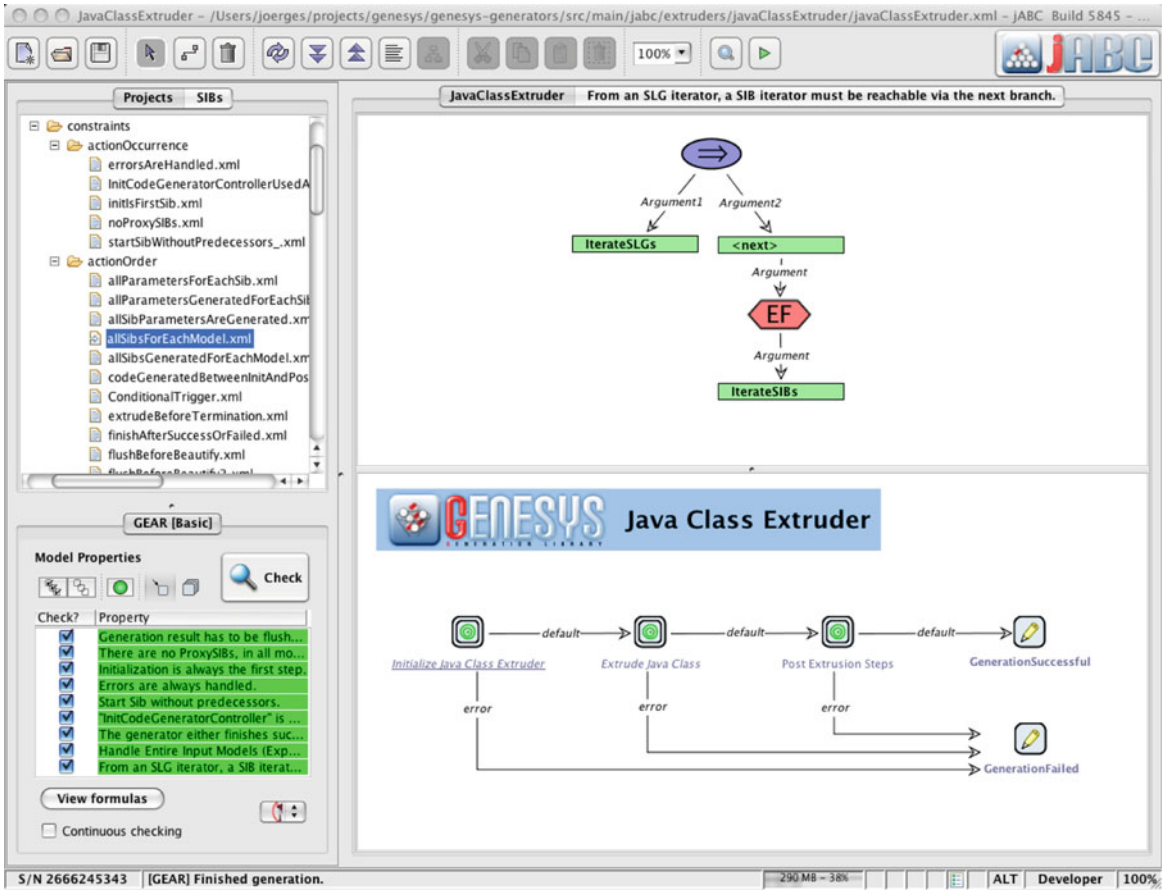

Fig. 5. Verifying the Java Class Extruder with GEAR and the FormulaBuilder

To further ease the specification of constraints, a tool called the FormulaBuilder [JMS06, JMS08] enables users to model the constraints themselves as SLGs. Users can thus create both the system and the constraints in the same environment. As such graphically modeled constraints are a special kind of SLG, we call them *formula graphs*. In comparison to normal SLGs there are two restrictions on formula graphs. First, as formula graphs reflect the syntactic structure of constraints, they usually are trees or directed acyclic graphs. Second, only specific SIBs, called Formula Building Blocks (FBBs) can be used to model constraints this way. FBBs represent the parts of a formula, i.e. operators, operands, etc. The FormulaBuilder provides a large library of FBBs for creating constraints, including logical, arithmetic, comparison and set operators as well as all specification patterns and other GEAR macros. Furthermore, jABC's hierarchy feature allows the creation of composite constraints and new patterns. We will elaborate on this later on in Sect. 5.

Figure 5 shows an example of how we use GEAR and the FormulaBuilder for checking the Java Class Extruder, whose topmost SLG is depicted on the bottom right of the image. The graph on the top right represents a constraint that will be translated by the FormulaBuilder into the following formula meaning "*From an iterator that processes SLGs, an iterator that processes SIBs has to be reachable via the 'next' branch.*":

$$\mathsf{IterateSLGs} \Rightarrow < next > EF(\mathsf{IterateSIBs})$$

In this formula, $< conditions > \phi$ indicates the diamond operator (originating from Hennessy–Milner logic [HM85, Mil89]), while $conditions$ represents an (optional) set of branch names. The given formula is true, if ϕ holds for at least one of the successor states reachable via a branch specified in $conditions$.

This formula graph can be dragged into the GEAR inspector depicted on the bottom left of Fig. 5. It is then added to the constraint library of the Java Class Extruder and translated on-the-fly by the FormulaBuilder, each time the constraint is required. Via the GEAR inspector and while modeling, the user is always able to check whether all constraints attached to an SLG and its sub-models are currently satisfied. In the following section, we will discuss more complex constraints used to verify Genesys' code generators.

5. Checking the property conformance of code generators

As the main contribution of this article, this section will focus on how we use jABC's verification facilities (outlined in Sect. 4) to support quality and reliability in the Genesys framework.

Section 4.1 already exemplified the application of the LocalChecker in the context of Genesys. In the following, we will concentrate on the verification of code generators via the model checker GEAR. In [JMS08] we already provided some simple example constraints that motivated the usage of model checking-based verification in the context of Genesys. We will further elaborate on this by examining more complex constraints, which span entire model hierarchies.

These constraints are part of a library of requirements which apply to all SLGs contained in Genesys. Since the publication of [JMS08], this library of constraints has been steadily growing. This is due to the fact that new code generators usually raise new constraints, which then are added to the library and often can even be reused and recombined for checking other code generators. The following sections will also show how we modularized constraints in order to broaden their applicability.

As motivated in [JMS08], jABC's integrated model checker GEAR (see Sect. 4.2) enables checking global constraints that address the well-formedness of the SLGs contained in Genesys. These constraints are collected in a library, which serves as a corpus of rules and guidelines for constructing code generators. There are several occasions which cause this library to grow steadily, e.g.:

- A new code generator is developed and raises new constraints.
- A bug that traces back to a modeling mistake which has been found in a code generator. A new constraint is then added to the library to assure that such mistakes will not happen again.
- Existing constraints are recombined to form new, in most cases stronger and more strict, constraints.

As depicted in Fig. 5 at the top left, the library is organized according to the constraint's semantics. This structure is inspired by the structure of the well-known specification pattern system presented by Dwyer et al. in [DAC99]. Accordingly, most constraints are currently distinguished by whether they say something about the occurrence of actions ("`actionOccurrence`") or the relative order ("`actionOrder`").

5.1. Occurrence constraints

Figure 6 shows three example constraints belonging to the category `actionOccurrence`. The depicted formula graphs represent the following constraints (translated into natural language and into a syntax close to GEAR's input syntax; the numbering corresponds to Fig. 6):

(1) *All errors are handled or at least logged, otherwise the generation fails or finishes successfully*:

$$< error > true \ \vee \ \mathsf{GenerationSuccessful} \ \vee \ \mathsf{GenerationFailed} \ \vee \ \mathsf{Log}$$

(2) *No proxy SIBs*:

$$absence_{globally}(SIB.isProxySib) \ \wedge \ absence_{globally}(SIB.isReplaced)$$

(3 and 4) *SIB* `InitCodeGeneration` *is used and is a start SIB without any predecessors*:

$$existence_{globally}(SIB.class == .^{*}InitCodeGeneration)$$
$$\wedge((SIB.class == .^{*}InitCodeGeneration) \Rightarrow (SIB.isStartSib \ \wedge \ \overline{[]} \ false))$$

In constraint **(1)**, the subformula using the diamond operator demands that every node in the SLG has an outgoing edge labeled with the branch *error* and pointing to a valid successor, while GenerationSuccessful, GenerationFailed and Log are atomic propositions. An early version of this constraint has already been published in [JKPM07], but during the further evolution of Genesys, it has been refined by adding the special cases of logging and the successful (error-free) code generation.

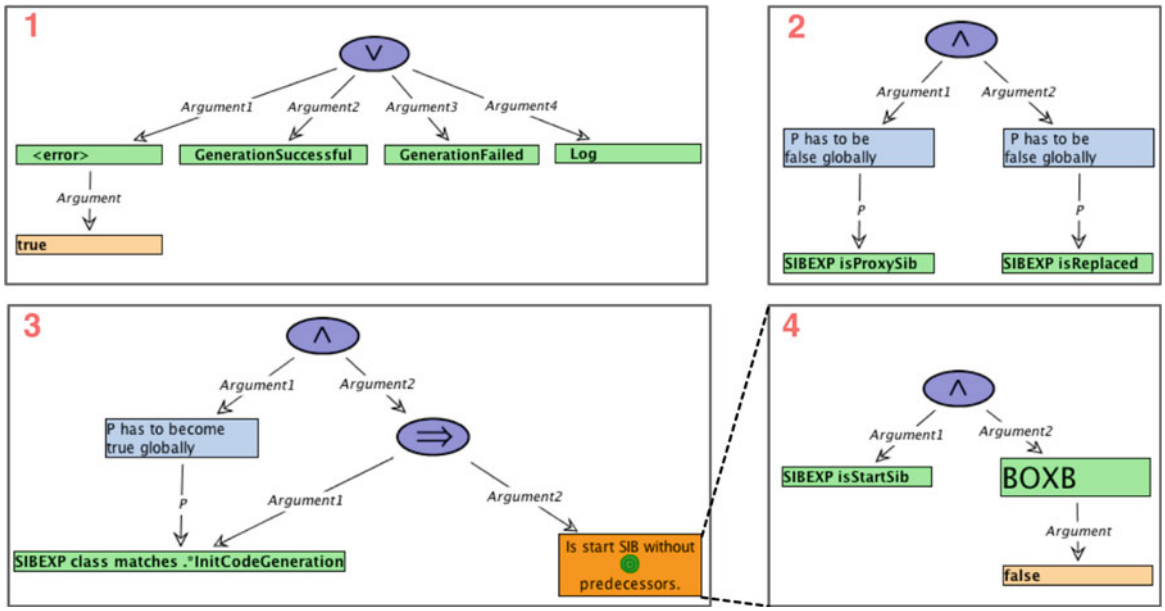

Fig. 6. Three constraints from the category `actionOccurrence`: **1** "All errors are handled or at least logged, otherwise the generation fails or finishes successfully.", **2** "No proxy SIBs." **3**, **4** SIB `InitCodeGeneration` is used and is a start SIB without ay predecessors

Example **(2)** is a constraint which is not only applicable to code generators, but to all kinds of executable SLGs. It refers to a jABC feature that ensures the integrity of SLGs, even if required SIBs cannot be loaded (e.g. because a SIB has been deleted, or the current user simply does not have access to a SIB). In such a case jABC replaces the affected SIBs by so-called *proxy SIBs*. Via these placeholders, the user is able to do all modeling work as usual: the affected SLGs can be loaded, and the parameters and branches of proxy SIBs can be edited. However, when it comes to the code generation for an SLG, it must not contain any proxy SIBs because they lack the required execution code. To ensure that no such placeholders are contained in an SLG, constraint 2 uses GEAR macros to check two cases:

- $SIB.isReplaced$ is true if a SIB has been automatically replaced by a proxy SIB.
- $SIB.isProxySib$ is true if a user misapplied a proxy SIB for modeling (which was technically possible in earlier versions of jABC).

The constraint demands that these two cases do not occur in all SLGs by using the specification pattern "Absence" with the scope "globally" [DAC99]. This pattern means "P has to be false globally" and translates to the CTL formula $AG(!\mathsf{P})$.

The formula graphs 3 and 4 in Fig. 6 exemplify the use of hierarchy when modeling constraints with the FormulaBuilder. Formula graph 4 corresponds to the formula $SIB.isStartSib \wedge \overline{[\,]}false$, which is true for any start SIBs without any predecessors (usually applies to the very first SIB of an SLG hierarchy). In this formula, $SIB.isStartSib$ again is a GEAR macro, while $\overline{[\,]}$ (resp. $BOXB$ in formula graph 4) is the backward variant of the box operator. By using jABC's hierarchy feature, this formula graph is embedded into formula graph 3 in order to produce a new constraint.

The constraint composed of graphs 3 and 4 demands the use of a SIB called `InitCodeGeneration`, which is obligatory for all code generators. This SIB initializes several data structures and helper functionality essential for any code generation process, e.g. for impeding the multiple generation of identifier names (which usually leads to uncompilable code). For this purpose, the constraint uses $SIB.class == .^*InitCodeGeneration$ which is true for all SIBs whose class name matches the regular expression $.^*InitCodeGeneration$. Furthermore, another specification pattern is used: "Existence" with scope "globally" means "P has to become true globally" and translates to $AF(\mathsf{P})$. Via the embedded formula graph 4, the constraint additionally demands that `InitCodeGeneration` is the very first SIB of any code generator.

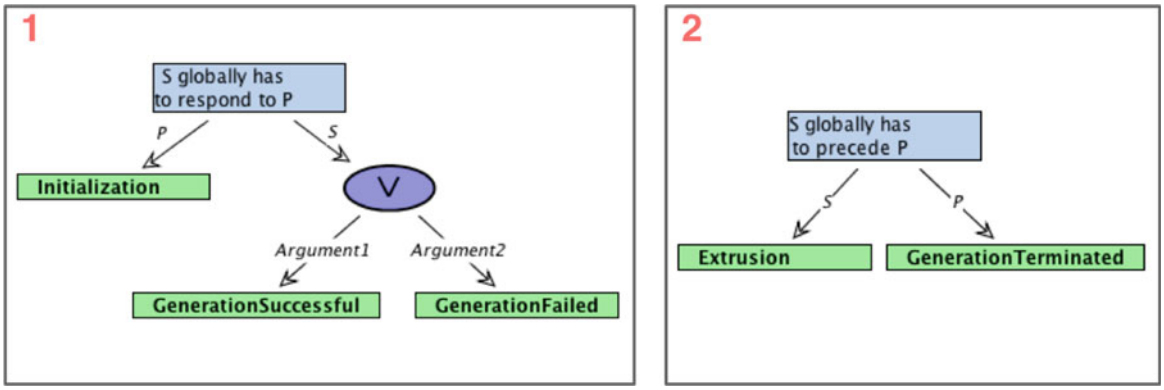

Fig. 7. Two constraints from the category `actionOrder`: **1** "After it is initialized, the generator either finishes successfully or fails." **2** "The extrusion phase has to be executed before the generation terminates"

When comparing the formula graphs 3 and 4 in Fig. 6 with the corresponding textual formula given above in **(3 and 4)**, the benefit of the graphical representation becomes apparent. Due to the structural reuse that is typical for directed acyclic graphs and the hierarchy features, formula graph representations are particularly beneficial for defining increasingly complex formulas.

5.2. Order constraints

Figure 7 shows two examples from the `actionOrder` category, which represent the following constraints:

(1) *After it is initialized, the generator either finishes successfully or fails*:

$$(\mathsf{GenerationSuccessful} \vee \mathsf{GenerationFailed})\ respondsto_{globally}\ \mathsf{Initialization}$$

(2) *The extrusion phase has to be executed before the generation terminates*:

$$\mathsf{Extrusion}\ precedes_{globally}\ \mathsf{GenerationTerminated}$$

Constraint **(1)** uses the specification pattern "Response" with scope "globally" to specify the allowed outcomes of a generation run, once the code generator has been initialized. This pattern means "S globally has to respond to P" and translates to $AG(\mathsf{P} \Rightarrow AF(\mathsf{S}))$.

The constraint in example **(2)** is concerned with the order of phases in the generation run. It assures that the extrusion phase, which contains the processing of the input models and the code generation itself, is definitely executed before the generation is terminated. Again, a specification pattern is used: It is called "Response" with scope "globally", means "S globally has to precede P" and translates to $A[\neg \mathsf{P}\ WU\ \mathsf{S}]$.

5.3. Deriving patterns and composing constraints

As many requirements for code generators lead to very similar formulas, new patterns can often be derived from structural similarities of the corresponding formula graphs.

For instance, imagine a constraint that should ensure the complete treatment of all the constituents of the SLGs that serve as an input for a code generator. A corresponding constraint must not only check that for every input SLG all contained SIBs are guaranteed to be processed, but also the SIB's parameters and branches, etc. It appeared that all these checks follow a similar new pattern, which we call Handle By. It is partly based upon "Precedence" and thus belongs to Dwyer et al.'s class of "Order patterns". Basically, the new pattern adds to "Precedence" the possibility to describe conditional cause-effect relationships. In the description style of [DAC99], the pattern's intent is:

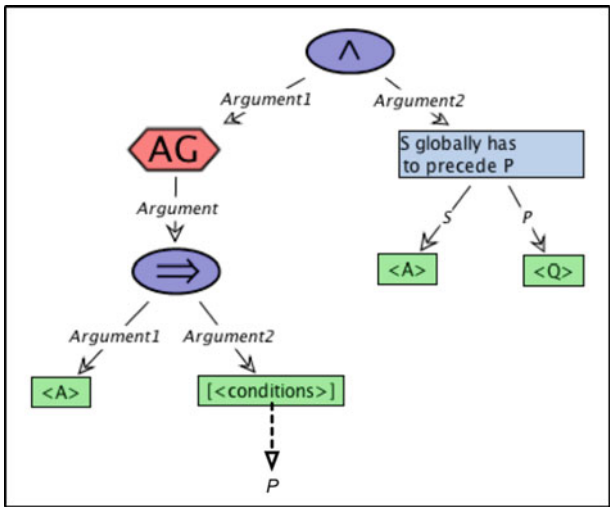

Fig. 8. The "Handle By" pattern with scope "before", meaning "Handle every A by P before Q"

Definition 2 (Intent of "Handle By") "To describe cause-effect relationships between a pair of events/ states on certain conditions. Under certain conditions, an occurrence of the first, the cause, must be handled by an occurrence of the second, the effect."

In its current version, the pattern only exists with the scope "before", as other scopes were not required yet. In textual form, this version of the pattern is to be read as "Handle every A by P before Q". The formula graph in Fig. 8 shows the pattern, which is basically modeled as an incomplete formula graph with:

- one model branch P (indicated in Fig. 8 by a dotted arrow) as well as
- three model parameters A, Q and conditions.

As A and Q are currently realized as parameters of the pattern, these slots can only be filled by atomic propositions, not by entire formulas (as it is the case with P). This basically was an intentional and pragmatic decision, as in all our scenarios, A and Q had always been atomic propositions. Nevertheless the pattern could be quickly and easily adjusted for supporting formulas in the slots A and Q. conditions provides a list of branch names that represent the conditions under which A will be handled by P.

To use the pattern, it can be simply embedded into a formula graph using jABC's hierarchy feature, as depicted by the left graph in Fig. 9. The nodes labelled 1 to 3 of this graphs use the "Handle By" pattern in the before scope. For instance, node 1 instantiates the pattern with:

- the atomic proposition NextModel as the *cause*,
- node 2 as the effect,
- "next" as the branch condition, and
- the atomic proposition GenerationTerminated for the "before" scope.

Besides using hierarchy to simplify constraints via the use and creation of patterns, constraints can be combined with and embedded into each other to capture big requirements of whole SLG hierarchies. All code generators in Genesys are hierarchical SLGs, with a typical code generator model forming a hierarchy of around 20 SLGs, each of them containing around 20 SIB. This means that a typical code generator consists of about 400 SIBs.

An example of a composite constraint is depicted in Fig. 9. It elaborates on the idea outlined above: checking a complete processing of all SLGs that serve as an input to a code generator. This should ensure that no information is lost or forgotten when generating code from a hierarchical SLG. The constraint focuses on the generation of code for SIB parameters and contains the following requirements:

(1) *Handle every input SLG before the generation terminates, by*

(2) *handling every SIB before handling the next SLG, by*

(3) *handling every SIB parameter before handling the next SIB, by*

(4) *generating code for either a "normal" SIB parameter or for a model parameter before handling the next SIB parameter*.

As a textual formula, this constraint would translate to (again in a syntax close to GEAR's input syntax):

$$
\begin{aligned}
&AG(\mathsf{NextModel} \Rightarrow [next](\\
&\quad AG(\mathsf{NextSIB} \Rightarrow [next](\\
&\quad\quad AG(\mathsf{NextSIBParameter} \Rightarrow [next](\\
&\quad\quad\quad A[!\mathsf{NextSIBParameter}\ WU\ (\mathsf{GenerateSIBParameter} \vee \mathsf{GenerateModelParameter})] \\
&\quad\quad)) \wedge A[\neg\mathsf{NextSIB}\ WU\ \mathsf{NextSIBParameter}] \\
&\quad)) \wedge A[\neg\mathsf{NextModel}\ WU\ \mathsf{NextSIB}] \\
&)) \wedge A[\neg\mathsf{GenerationTerminated}\ WU\ \mathsf{NextModel}]
\end{aligned}
$$

This formula certainly is not very intuitive, and it is rather laborious for a user to create and understand it. When modeling the constraint explicitly as a formula graph, as depicted on the right side of Fig. 9, it gets much more concise due to the consequent use of jABC's hierarchy feature, and of specification patterns like "Precedence". Furthermore, if patterns like "Handle By" can be identified and created in the fashion exemplified above, the constraints become simple, revealing the true intent of the designer.

By combining and embedding constraints this way, Genesys' constraint library for code generators grows rapidly, and the constraints tend to be more and more concise and effective.

6. Related work

There are various approaches that aim at assuring the reliability of automatic code generation. A "verifying compiler", as proposed by Tony Hoare and Jay Misra in their grand challenge [HM05, Hoa03], is one possible solution to achieve this goal. According to this approach, the compiler is able to determine the correctness of the problem by using additional information, like e.g. assertions or annotations, in the source code.

A lot of research has been done on this, e.g. taking the form of proof-carrying code [Nec97, App01] or evidence-based approaches [DF06]. Other work aims at verifying the compiler itself, like e.g. the Verifix project [GZ99], or at validating the translation especially for optimizing compilers [Nec00]. In contrast to these rather *analytical* techniques, *constructive* approaches postulate a more systematic development process, e.g. based on the adoption of accepted standards [SWC05] or the generation of code from specifications [CG05].

All these approaches have in common that they work more or less on the *source code level*. As all code generators in the Genesys framework are modeled as SLGs, the verification approach presented in this article is applied on the *design* or *process level*, which is one of the key ideas of the XMDD paradigm. This allows us to formulate constraints which are not only applicable to one code generator, but to whole families of code generators.

Our graphical way of constructing constraints or rules is also comparable to several related research projects which are concerned with simplifying the specification of such properties. For instance, the PROPEL-System [SACO02] tries to increase the precision of Dwyer et al.'s specification patterns by offering additional options to cover alternatives in the behavior that is to be specified. The patterns are presented by means of two complementary views: the first is based on a restricted subset of natural language called DNL (disciplined natural language), the other uses finite-state automata for visualization. As the formula graphs used in our solution are based on SLGs, we profit from the powerful hierarchy mechanisms that allow us to easily derive new, more precise and even domain-specific patterns.

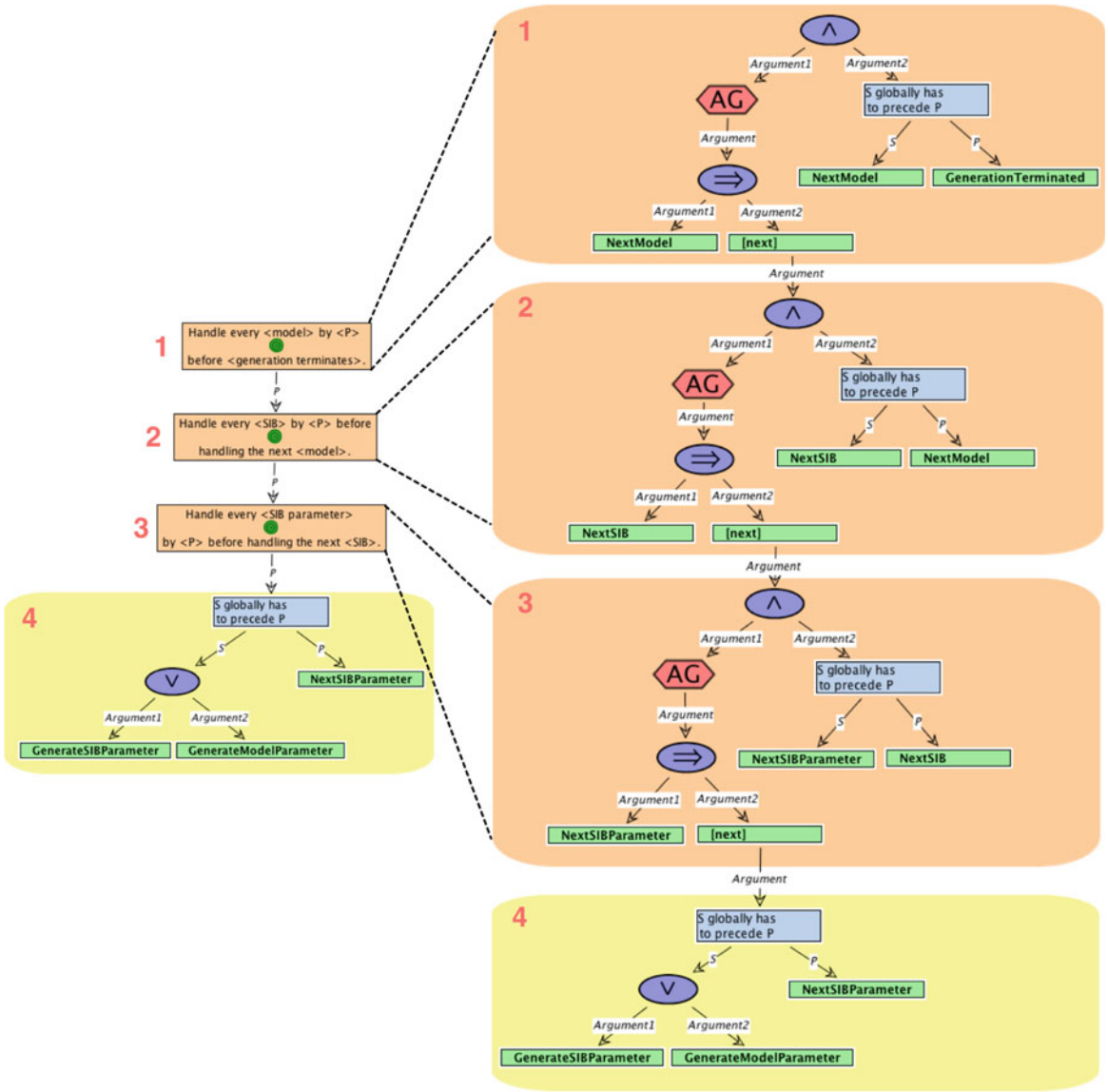

Fig. 9. A constraint checking for the complete handling of input SLGs, using the new "Handle By" pattern

Beside the specification pattern system, there are also other approaches that try to simplify the specification of formulas. Graphical interval logic (GIL) [DKM+94] provides a graphical notation which can be used to model temporal properties of a system. Here intervals serve for the definition of contexts in which particular properties hold. Other projects like e.g. PROSPER [Hol99] or Attempto Controlled English [FSS98] try to employ restricted ("controlled") versions of natural language for the specification of properties. Our approach uses the specification patterns as a fundament, because due to their parameterizablity and reusability they fit well into the concept of SIBs and are suitable as components of SLGs.

7. Conclusion and future work

We have presented our approach to the model checking-based verification of code generators in Genesys. Genesys is a framework for the high-level construction of code generators along the lines of the XMDD paradigm. In our experience, the development of code generators with Genesys boils down to re-instantiating patterns or slightly modifying the graphical process model. These activities are strongly supported by our verification facilities, which help leveraging the increasing body of domain knowledge in terms of constraints, features and patterns during code generator construction. We have shown how we support the build-up of this body of knowledge along the construction and structuring of a complex constraint, which enforces a certain notion of completeness of the code generation process.

Currently, we are also investigating data flow analysis via model checking (DFA-MC [Ste91, LMS06a]) as a technique for further increasing the quality of our code generators. Interestingly, DFA-MC is applicable for us on the component level as well as on the process level. As our components, the SIBs, are Java classes, we can directly use DFA-MC on them, just as described in [LMS06a]. At the process level, we need to annotate the contained components with data flow information to be able to perform such analyses. This is part of the corresponding domain modeling, which typically requires manual effort, but we also envisage the support by further analyses or by employing a systematic way of incorporating service-level agreements.

To complement our verification facilities, we also envision a more seamless combination of code generation and testing. It is e.g. imaginable to attach test cases (themselves modeled as SLGs [NSM+01]) to the SLGs of a modeled application. Along with the source code for the application, the code generators could then also translate the test cases to source code for a common testing framework like e.g. JUnit [Obj07] as part of the quality management process—again modeled with the jABC framework.

References

[All83] Allen JF (1983) Maintaining knowledge about temporal intervals. Commun ACM 26(11):832–843

[Apa07] Apache Software Foundation (2007) Velocity Website, 2007. http://velocity.apache.org/

[App01] Appel AW (2001) Foundational Proof-Carrying Code. In Proceedings of LICS '01. IEEE Computer Society, p 247

[BJM09] Bakera M, Jörges S, Margaria T (2009) Test your strategy: graphical construction of strategies for connect-four. In: Proceedings of the 14th IEEE international conference on engineering of complex computer systems, ICECCS 2009. IEEE Computer Society, pp 172–181

[BM06] Bajohr M, Margaria T (2006) Matrics: a service-based management tool for remote intelligent configuration of systems. Innov Syst Software Eng 2(2):99–111

[BMRS07a] Bakera M, Margaria T, Renner C, Steffen B (2007) Property-driven functional healing: playing against undesired behavior. In: 10th CONQUEST

[BMRS07b] Bakera M, Margaria T, Renner C, Steffen B (2007) Verification, diagnosis and adaptation: tool supported enhancement of the model-driven verification process. In: Workshop: formal methods in avionics, space and transport (ISOLA), pp 85–98

[CG05] Coglio A, Green C (2005) A constructive approach to correctness, exemplified by a generator for certified Java Card Applets. In: Proceedings of VSTTE

[CGP01] Clarke EM, Grumberg O, Peled DA (2001) Model checking. MIT Press, CA

[DAC99] Dwyer M, Avrunin G, Corbett J (1999) Patterns in Property specifications for finite-state verification. In: Proceedings of ICSE '99. IEEE CS Press, pp 411–420

[DF06] Denney E, Fischer B (2006) Extending source code generators for evidence-based software certification. In: Proceedings of ISOLA '06

[DKM+94] Dillon LK, Kutty G, Moser LE, Melliar-Smith PM, Ramakrishna YS (1994) A graphical interval logic for specifying concurrent systems. ACM Trans Software Eng Methodol 3(2):131–165

[FSS98] Fuchs NE, Schwertel U, Schwitter R (1998) Attempto controlled english—not just another logic specification language. In: LOPSTR '98: Proceedings of the 8th international workshop on logic programming synthesis and transformation, pp 1–20

[GZ99] Goos G, Zimmermann W (1999) Verification of Compilers. In correct system design. Springer, New York, vol 1710, pp 201–2309

[HM85] Hennessy M, Milner R (1985) Algebraic laws for nondeterminism and concurrency. J ACM 32(1):137–161

[HM05] Hoare CAR, Misra J (2005) Verified software: theories, tools, experiments. Vision of a Grand Challenge Project. In: Proceedings of VSTTE, Zürich, Switzerland. Springer, New York

[HMM+08] Hörmann M, Margaria T, Mender T, Nagel R, Steffen B, Trinh H (2008) The jABC approach to rigorous collaborative development of SCM applications. In Proceedings of ISoLA, pp 724–737

[Hoa03] Hoare CAR (2003) The verifying Compiler: a grand challenge for computing research. J ACM 50(1):63–69

[Hol99] Holt A (1999) Formal verification with natural language specifications: guidelines, experiments and lessons so far. S Afr Comput J 24:253–257

[Hol03] Holzmann GJ (2003) The SPIN model checker: primer and reference manual. Addison-Wesley Professional, MA

[IT92] ITU-T (1992) Recommendation Q.1203. Intelligent network—global functional plane architecture. Technical report, Standardization Sector of ITU

[ITU93] ITU (1993) General recommendations on telephone switching and signaling—intelligent network: introduction to intelligent network capability set 1, Recommendation Q.1211. Technical report, Standardization Sector of ITU, Geneva
[JKPM07] Jörges S, Kubczak C, Pageau F, Margaria T (2007) Model driven design of reliable robot control programs using the jABC. In: Proceedings of EASe '07, pp 137–148
[JMS06] Jörges S, Margaria T, Steffen B (2006) FormulaBuilder: a tool for graph-based modelling and generation of formulae. In: Proceedings of ICSE '06
[JMS08] Jörges S, Margaria T, Steffen B (2008) Genesys: service-oriented construction of property conform code generators. Innov Syst Software Eng 4(4):361–384
[KM06] Karusseit M, Margaria T (2006) Feature-based modelling of a complex, online-reconfigurable decision support service. Electr Notes Theor Comput. Sci 157(2):101–118
[KMSN08] Kubczak C, Margaria T, Steffen B, Nagel R (2008) Service-oriented mediation with jABC/jETI
[Koz83] Kozen D (1983) Results on the propositional mu-Calculus. Theor Comput Sci 27:333–354
[LMS06a] Lamprecht A-L, Margaria T, Steffen B (2006) Data-flow analysis as model checking within the jABC. In: Compiler construction, pp 101–104
[Mil89] Milner R (1989) Communication and concurrency. Prentice Hall international series in computer science. Prentice-Hall, Englewood Cliffs
[MKS07] Margaria T, Kubczak C, Steffen B (2007) Bio-jeti: a service integration, design, and provisioning platform for orchestrated bioinformatics processes. In BioMed Central (BMC) Bioinformatics supplement dedicated to network tools and applications in biology 2007 workshop (NETTAB 2007), vol 9
[MOSS99] Müller-Olm M, Schmidt DA, Steffen B (1999) Model-checking: a tutorial introduction. SAS, pp 330–354
[MS08] Margaria T, Steffen B (2008) Agile IT: thinking in user-centric models. In Procedings of ISoLA 2008, CCIS N.17. Springer, New York, pp 493–505
[MS09] Margaria T, Steffen B (2009) Business process modelling in the jABC: the one-thing approach. Handbook of research on business process modeling. IGI Global, PA
[Nec97] Necula GC (1997) Proof-carrying code. In: Proceedings of POPL '97, ACM Press, New York, pp 106–119
[Nec00] Necula GC (2000) Translation Validation for an Optimizing Compiler. ACM SIGPLAN Notices 35(5):83–94
[NSM+01] Niese O, Steffen B, Margaria T, Hagerer A, Brune G, Ide H-D (2001) Library-based design and consistency checking of system-level Industrial test cases. In: Proceedings of FASE, volume 2029 of LNCS. Springer, New York, pp 233–248
[OA07] OASIS (2007) WS-BPEL 2.0 Specification, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
[Obj07] Object Mentor (2007) JUnit Website. http://www.junit.org/
[QS82] Queille J-P, Sifakis J (1982) Specification and verification of concurrent systems in CESAR. In: Proceedings of 5th Colloquium on international symposium on programming, Springer, London, pp 337–351
[SACO02] Smith RL, Avrunin GS, Clarke LA, Osterweil LJ (2002) PROPEL: an approach supporting property elucidation. In: ICSE '02: Proceedings of the 24th international conference on software engineering, ACM Press, New York, pp 11–21
[SMBK97] Steffen B, Margaria T, Braun V, Kalt N (1997) Hierarchical service definition. In: Annual review of communication. International Engineering Consortium Chicago (USA), IEC, pp 847–856
[SMN+06] Steffen B, Margaria T, Nagel R, Jörges S, Kubczak C (2006) Model-driven development with the jABC. In: HVC—IBM Haifa Verification Conference, LNCS N.4383, Springer, New York
[SN07] Steffen B, Narayan P (2007) Full life-cycle support for end-to-end processes. IEEE Comput 40(11):64–73
[Ste91] Steffen B (1991) Data flow analysis as model checking. In: TACS '91: Proceedings of the international conference on theoretical aspects of computer software. Springer, New York, pp 346–365
[SWC05] Stürmer I, Weinberg D, Conrad M (2005) Overview of existing safeguarding techniques for automatically generated code. In: Proceedings of SEAS '05, ACM Press, New York, pp 1–6
[The08] The jABC Team (2008) jABC Common Sibs. http://www.jabc.de/sib

Received 17 February 2009
Revised 1 September 2010
Accepted 20 September 2010 by Daniel Kröning and Jim Woodcock
Published online 25 November 2010