ABSTRACT
In this paper we present the FormulaBuilder, a flexible tool for graph-based modelling and generation of formulae. The FormulaBuilder allows easy and intuitive creation of formulae by using basic components called Formula Building Blocks (FBBs) and arranging them as graphs according to the syntactic structure of a formula. Such a graph can then be validated and used to generate the corresponding formula on the basis of a specific syntax which is chosen from a list of syntaxes supported by the FormulaBuilder.

An important application of the FormulaBuilder is the formal specification of properties that describe the requirements of a system. Such property specifications are usually needed by verification tools like model checkers, that help software engineers to detect errors in a specified system. The FormulaBuilder allows users to model property specifications as formula graphs by using commonly-occurring specification patterns.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications; D.2.4 [Software/Program Verification]: Formal methods

General Terms: Languages, Verification

Keywords: Property specification, specification patterns, formula generation

1. INTRODUCTION
Finite-state verification techniques like e.g. model checking [3] provide a powerful way to determine whether a given system is consistent with a specified property. Such properties usually are described by using specification formalisms such as temporal logics or regular expressions.

Writing properties with a specification formalism such as e.g. CTL (Computation Tree Logic) is often a difficult task. For a software engineer who is used to formulate properties by means of UML state diagrams or even natural language, learning such a specification formalism is a big and time-consuming obstacle. Furthermore, the resulting formulae that describe specific properties can get very complex and hard to understand, even for an expert.

Users of finite-state verification tools are always faced with a clash between accessibility and precision. While specification formalisms usually can describe a property in a very precise way, they are less accessible because of the reasons mentioned above. In contrast to this, natural language is easily accessible for everyone, while being too ambiguous and imprecise to be useful for property specifications.

The goal of the FormulaBuilder [10, 11] is to provide an intuitive way for users to create property specifications or, to be more general, formulae. We achieve this by utilizing two basic concepts. First, a formula is modelled and visualized as a graph, so that its syntactic structure can be intuitively understood. Second, the FormulaBuilder incorporates a system of specification patterns which serve as abstract parameterizable building blocks of such a formula graph.

2. SPECIFICATION PATTERNS
Dwyer, Avrunin and Corbett proposed a system of patterns [7, 6] to help users on their way from the natural description to the formal specification of a property. In the style of Design Patterns [8] this system uses simple abstractions of commonly used specifications. Dwyer et al. found out that already a small set of basic patterns is sufficient to cover most formal specifications. These patterns are parameterizable and independent from the concrete formalism that is used. Every pattern consists of a name, a description of its purpose and a body. From the description it should be clear which system property or behaviour is described by the pattern. The body contains

- a set of mappings to corresponding specification formalisms,
- a list of example and known uses,
- a description of the relationships to other patterns and
- a scope which describes where in the system the specified property has to hold.
Dwyer et al. identified five scopes: global (throughout the system), before (before a certain state/event), after (after a certain state/event), between (between two particular states/events) and after-until (between two particular states/events, even though the second event/state might never occur).

Example: The Precedence Pattern

The Precedence pattern describes a property in which one state/event \( S \) always has to precede another state/event \( P \). This pattern is often useful in the specification of concurrent systems, e.g. in a web service where the logout of a user is only possible if preceded by a login of this user. For different scopes the Precedence pattern can now be mapped into different specification formalisms. A corresponding CTL formula for the scope global would e.g. look like this: \( A \geq WPuS \) (note that \( S \) has to precede \( P \)). For detailed information on the pattern-system the interested reader may refer to [7, 6].

3. FORMULABUILDER

As can be seen in Fig. 1 the FormulaBuilder framework consists of four components: the formula graphs, the Formula Building Blocks (FBBs), the target syntaxes for generation and the FormulaBuilder Plugin for the jABC environment. The plugin provides access to the algorithms for validation and generation of formula graphs as well as a GUI for interaction with the user.

In its current version the FormulaBuilder is realized on the basis of the jABC [16, 14, 13], which is a platform-independent framework that supports graphical modelling, analysis, verification and implementation of software systems. It provides an abstract modelling layer, so that no programming knowledge is needed. For modelling an envisioned system a user utilizes readymade coarse-granular components called SIBs (Service Independent Building Blocks). These components are arranged as hierarchical graphs called SLGs (Service Logic Graphs) which represent models for the desired system.

SLGs are directed graphs whose nodes are represented by SIBs. The labels of a directed edge depend on the edge’s starting node and are called branches. Speaking roughly, branches could also be called the “exits” of a SIB. Each SIB can have an arbitrary number of branches, and the branches then are assigned to outgoing edges. SIBs are also parameterizable, i.e. each SIB can have an arbitrary number of SIB parameters. A SIB is implemented as a simple Java class.

The semantics of such a SLG is defined by plugins which can be easily added to the jABC framework. The FormulaBuilder is realized as such a plugin and, in this context, SLGs are interpreted as formulae. This special interpretation of a SLG is called a formula graph.

3.1 Formula Graphs & their Building Blocks

A formula graph is a special SLG that reflects the syntactic structure of a formula. In comparison to normal SLGs formula graphs have two restrictions. First, formula graphs are actually acyclic, i.e. trees or DAGs (directed acyclic graphs) and second, only SIBs of a specific class called Formula Building Blocks (FBBs) can be used to build up a formula graph.

FBBs usually represent the parts of a formula, i.e. operands, operators, etc. That way the FBBs inherit all the advantages of the SIB concept, i.e. that they are reusable, parameterizable, fully documentable, etc. Fig. 2 shows an example of a formula graph that models the simple CTL formula \((P \land Q) U R\). In this example, the CTL operator “always strong until” is represented by a FBB with two branches (Argument1 and Argument2, analogous for the contained conjunction), while an atomic proposition is represented by a FBB with a single parameter that contains the value of the particular proposition (in this case \( P, Q \) and \( R \)). The example also shows that the correct bracketing of the formula arises from the structure of the formula graph in a natural manner, which is an important advantage of this visualization approach.

The FormulaBuilder offers a large library of readymade standard FBBs which can be used for modelling formula graphs. Among these FBBs there are logical, arithmetic, comparison and set operators as well as common operands like e.g. strings, numbers or boolean values.

To support the easy and intuitive specification of formal properties as formula graphs, the FormulaBuilder also offers the specification patterns described in Section 2. For every
pattern there are five FBBs, one for each scope. The exemplary pattern *Precedence* with the scope *global* is here realized as a FBB with two branches that represent the two slots (*P* and *S*) that have to be filled to instantiate this pattern. Fig. 3 shows a formula graph using a concrete instantiation of the pattern, where *P* and *S* have the values “Logout” resp. “Login”. This graph models the property that (e.g. in a web application) a login globally has to precede a logout. Further FBBs can easily be added by implementing corresponding Java classes.

The FBB’s function is a purely representative one. What will be generated from a specific FBB and how the result of the generation looks like depends entirely on the syntax that is selected by the user.

### 3.2 Target Syntaxes

The FormulaBuilder allows generating formulae for arbitrary target syntaxes. Formula graphs are, just like the specification patterns, formalism-independent and can be validated and generated according to different syntaxes.

New target syntaxes can easily be added to the FormulaBuilder. For this purpose we provide an uniform interface that allows defining target syntaxes in a BNF-like style. Each target syntax is identified by a unique name and a textual description which should describe the syntax as exactly as possible. Among other things, this description is the basis for online help that is provided for the user at runtime.

The definition of a target syntax is in most cases contained in a single Java class. Such a class is the central place for modifications of a syntax, and the interface provides several possibilities for customizing syntax options and even for adding new functionality.

New functionality which might be necessary for a specific target syntax can be added to the FormulaBuilder. For this purpose the uniform interface provides several extension points to influence and extend the standard validation and generation algorithms. That way it is e.g. possible to implement additional validation tasks for a target syntax, like checking the monotony of a formula for the modal μ-calculus. Depending on the complexity of such extensions it is possible that a target syntax consists of more than one Java class.

The FormulaBuilder already offers a set of target syntaxes which are ready to be used for the generation of formulae. The currently supported syntaxes are Linear Time Logic (LTL), Computation Tree Logic (CTL), the modal μ-calculus and monadic second-order logic (M2L).

We will now elaborate on these syntaxes to show the abilities and the extensibility of the FormulaBuilder while skipping LTL and M2L for brevity reasons. For further information the interested reader may refer to [18], our environment for analysis and verification with M2L.

**CTL** is a temporal logic which can be used to formulate temporal properties of a model (usually a transition system) which e.g. are concerned with the reachability of certain states. For the specification of such formulae CTL offers the path quantifiers *A* and *E* as well as the temporal operators *F*, *U* or *G*. CTL belongs to the branching-time logics, i.e. its temporal operators quantify over paths that start in a specific state of the model. CTL formulae can be constructed as follows (*p* represents an atomic proposition):

\[
\phi ::= p | \neg \phi | \phi_1 \land \phi_2 | \phi_1 \lor \phi_2 | \Box \phi | \Diamond \phi | \phi_1 \mathrm{U} \phi_2 \\
\]

With the help of this minimal grammar all the other CTL operators can easily be constructed, e.g.:

\[
AX\phi = \neg EX(\neg \phi), EF\phi = E[true \ U \ \phi_1], AF\phi = \neg EG(\neg \phi)
\]

For the semantics of such CTL formulae please refer to [3].

The FormulaBuilder provides a target syntax which allows generating CTL formulae. This syntax supports FBBs for atomic propositions, for the ten CTL operators and for the specification patterns. Each pattern can easily be mapped to a corresponding CTL formula (see [6] for a complete listing of all mappings).

Just like CTL, the modal μ-calculus [12] belongs to the branching-time logics, but it is much more expressive than CTL. μ-calculus formulae can be constructed as follows:

\[
\phi ::= true | false | [a]\phi | (\mu X \phi) | (\nu X \phi) | \phi_1 \lor \phi_2 | \phi_1 \land \phi_2 | X | \mu X \phi | \nu X \phi
\]

In this grammar, *X* represents a variable while *a* ranges over a set of actions. Free occurrences of *X* are bound by the two fixpoint operators μ*X* (the least fixpoint) and ν*X* (the greatest fixpoint). There are no negations present in the above grammar because in every fixpoint formula μ*X*φ resp. ν*X*φ the meaning of *φ* has to depend monotonically on *X*. To achieve this, every free occurrence of *X* in *φ* has to appear under an even number of negations. More detailed information and the definition of the semantics of μ-calculus formulae can be found in [15, 3].

Properties to be used by the jABC model checker [2] to verify systems which are modelled as SLGs have to be written as μ-calculus formulae. The FormulaBuilder provides a target syntax to allow modelling and generating such properties. This syntax supports FBBs for all operators and operands described in the above grammar for the modal μ-calculus, as well as FBBs for the specification patterns. A complete list of all mappings of the patterns into the modal μ-calculus can be found in [11].

The syntax also supports FBBs for CTL because all CTL modalities can be easily translated to μ-calculus formulae ([15, 3]), e.g.:

\[
E[\phi_1 \ U \ \phi_2] = (\nu X.((\phi_1 \lor X) \land (\phi_2 \lor \Diamond X))) \\
EG(\phi) = (\mu X.((\phi \lor \Box X) \land (\phi \lor \Diamond false)))
\]
Furthermore, the syntax offers a set of syntactic extensions like e.g. regular expressions, quantifiers or SIB expressions. The latter are powerful constructs that allow making assertions about values of particular SIB parameters of SIBs contained in the model that is to be verified. All syntactic extensions are realized by using the FormulaBuilder’s possibilities of adding and extending functionality. The syntax also uses the uniform interface to extend the standard validation algorithm with an additional validation step that checks the monotony of the generated \( \mu \)-calculus formulae.

### 3.3 The Plugin

As mentioned earlier, the FormulaBuilder is realized as a plugin for the jABC framework. Besides providing access to the algorithms for validation and generation, it offers a GUI for interaction with the user (see Fig. 3). The existing facilities of the jABC GUI already provide a complete but generic graphical environment for modelling SLGs. The FormulaBuilder plugin extends it with additional GUI components that assist the user while modelling a formula graph and supports the selection of the target syntax that should be used for the generation.

### 4. RELATED WORK

Many related research projects are concerned with simplifying the specification of system properties.

The PROPEL-System [17] tries to increase the precision of the specification patterns by offering additional options to cover alternatives in the behaviour 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 FormulaBuilder evolved in the context of the jABC framework, where users typically are used to work with SLGs, our approach focuses on the graph visualization to provide a fully-integrated solution.

The pattern system also plays an important role in the Bandera project [4] which facilitates model checking of Java source code. There, properties are specified in a language called BSL (Bandera Specification Language) that is based on the pattern system and independent from the concrete model checker that is used. This is very similar to our approach as properties specified as formula graphs can be generated according to arbitrary target syntaxes for various purposes, so that different model checkers and even other tools can be supported. Nevertheless, we believe that formula graphs in combination with the pattern system form a more intuitive approach to the specification of properties, especially for jABC users.

Beside the pattern system there are also other approaches that try to simplify the specification of formulae. Graphical Interval Logic (GIL) [5] 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. PuPROSPER [9] or Attemop Controlled English [1] try to employ restricted (“controlled”) versions of natural language for the specification of properties. The FormulaBuilder uses the specification patterns because, due to their parameterizability and reusability, they fit well into the concept of SIBs and are suitable as components of SLGs.

### 5. CONCLUSIONS

In this paper we have presented the FormulaBuilder, a flexible tool for the generation of formulae. By modelling formulae as formula graphs and allowing to generate their concrete incarnation according to a selectable syntax, the FormulaBuilder offers a convenient and intuitive way to design formulae for many purposes. This is especially interesting for the formal specification of system properties, which is additionally simplified with the help of commonly-used specification patterns. The FormulaBuilder has already been successfully used by our students in Dortmund and in Göttingen (in courses at bachelor and master level), who gained a more intuitive access to complex formal specification languages like CTL, the modal \( \mu \)-calculus and monadic second-order logic by experimenting with the tool.

### 6. REFERENCES


[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


