



Repair and Generation of Formal Models Using Synthesis

Joshua Schmidt^(✉), Sebastian Krings, and Michael Leuschel

Institut für Informatik, Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
joshua.schmidt@uni-duesseldorf.de,
{krings,leuschel}@cs.uni-duesseldorf.de

Abstract. Writing a formal model is a complicated and time-consuming task. Usually, one successively refines a model with the help of proof, animation and model checking. In case an error such as an invariant violation is found, the model has to be adapted. However, finding the appropriate set of changes is often non-trivial.

We propose to partially automate the process by combining synthesis with explicit model checking and implement it in the context of the B method: Guided by examples of positive and negative behavior, we strengthen preconditions of operations or relax invariants of the model appropriately. Moreover, by collecting initial examples from the user, we synthesize new operations from scratch or adapt existing ones. All this is done using user feedback, yielding an interactive assistant. In this paper, we present the foundations of this technique, its implementation using constraint solving for B, and illustrate the technique by synthesizing the formal model of a process scheduler.

1 Introduction

Writing and adapting formal models is a non-trivial task, difficult for beginners and time-consuming even for trained developers. Often, one iterates between changing a model and proof or model checking. Once an error has been detected, the model has to be adapted.

The premise of this paper is that, to some extent, this correction phase can be automated, using negative and positive examples provided by a model checker or by the user. For example, we can synthesize corrected preconditions or invariants in order to repair invariant violations. If deciding to allow an invariant violating state, we know that we need to synthesize relaxed invariants using the given I/O examples. Otherwise, the precondition of the affected operation needs to be strengthened to exclude the state from the model. Moreover, deadlocks can be repaired either by generating a new operation or strengthening the precondition of an existing operation.

When model checking has been exhaustive without finding any invariant violation or deadlock state, we are able to extend the model by synthesizing new

transitions based on state pairs for input and output. In case the machine already contains an operation providing the desired behavior, we relax its precondition or the invariants if necessary. Otherwise, a completely new operation is synthesized.

The tool mainly aims at providing better access to formal methods for beginners. By allowing to define behavior by means of I/O examples the user might be able to learn from the synthesized code. Moreover, if finding an invariant violation or a deadlock state, an automated repair eases the workflow for any user.

In this paper, we present this technique in the context of the B formal method based on our previous publication [32]. In particular, the extensions include:

- extended interactive workflow for the repair of deadlocks and the adaption of existing operations or machine invariants (Sect. 3)
- thorough presentation of the technique, along with support for if-statements and operation parameters (Sect. 4)
- performance improvements due to dynamic expansion of the search space, parallelization, randomized search and symmetry reduction (Sect. 5)
- graphical user interface (Sect. 6)
- performance evaluation (Sect. 8).

2 A Primer on the B-Method

The formal specification language B [1] follows the correct-by-construction approach and is based on first-order-logic and set theory. A formal model in B consists of a collection of machines starting from an abstract specification and successively refining the behavior. The development in B is thus incremental, which increases the maintainability and eases the specification of complex models. In this paper, we always refer to B formal models. The synthesis workflow is applied to a single B machine. A machine consists of variable and type definitions as well as possible initial states. A state is defined by the current evaluation of the machine variables. By defining machine operations, one is able to specify transitions between states. A machine operation has a unique name and consists of B substitutions (aka statements) defining the machine state after its execution, i.e., the values of a set of machine variables are assigned. An operation can have a precondition, allowing or prohibiting execution based on the current state. For instance, a valid machine operation o is defined by $o = \text{PRE } x > 0 \text{ THEN } x := x + 1 \text{ END}$ using the single assignment substitution of B. Several variables can be assigned either in parallel or in sequence. A state s is called a deadlock if it has no successors, i.e., no operation is enabled. To ensure certain behavior, the user can define machine invariants, i.e., safety properties that have to hold in every reachable state. Hence, the correctness of a formal model refers to the specified properties. In addition to the types explicitly provided by the B language like INTEGER or BOOL, one can provide user-defined sets. These sets can be defined by a finite enumeration of distinct elements (the set is then referred to as an enumerated set) or left open (called deferred sets). For instance, by defining a set $S = \{s1\}$ the element $s1$ is of type S and can be accessed by name within

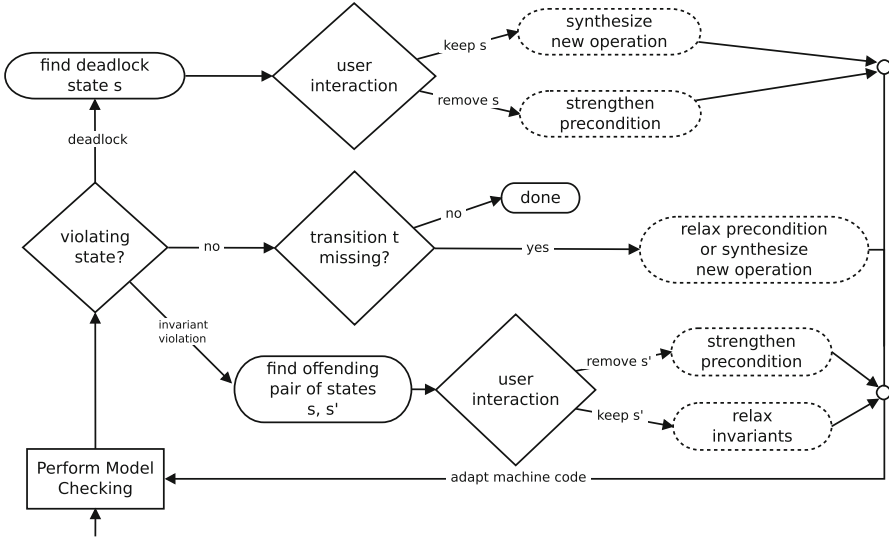


Fig. 1. Interactive workflow to repair and generate formal models using synthesis

the machine. Deferred sets are assumed to be non-empty during proof and also finite for animation.

Using Atelier B [12] or PROB [24–26] one can verify a B model and analyze its state space. In particular, PROB allows the user to animate formal models, providing a model checker and constraint solver. PROB’s kernel [24] is implemented in SICStus Prolog [7] using the CLP(FD) finite domain library [8]. Alternatively, a constraint solver based on Kodkod [33] is available [31]. Furthermore, an integration with the SMT solver Z3 [28] can be used to solve constraints [22].

Below, we will focus on classical B [1] for software development, but our approach also works for Event-B and could be extended to other languages supported by PROB such as TLA^+ [23].

3 Interactive Workflow

The process as outlined in Fig. 1 is guided and enforced by PROB. The workflow itself is quite mature and has been fully implemented within the system. Repair is performed successively, that means, we loop until no error can be found anymore and the user is satisfied with the model. Each step starts with explicit model checking performed by PROB. To that effect, the user at least needs to provide a B machine defining variables and an initial state. The dotted nodes mark the parts of the workflow where synthesis is applied. There are three possible outcomes.

First, an invariant violation might be found. We then identify the machine variables that violate the invariants and reduce the examples obtained by the

model checker if possible. The user then can decide to disable the last transition, leading from a state satisfying the invariants to one violating it, by synthesizing a stronger precondition. Alternatively, the system can generate weaker invariants, allowing the violating state.

As a second outcome, the model checker may have uncovered a deadlock state s . The user can then decide between two options:

- Remove s by strengthening the precondition of the involved operation
- Keep s by synthesizing a new operation or adapting an existing one enabling to transition from s to another state s' .

Third, the model has been checked and no error was found, that means, model checking was exhaustive or a timeout occurred. We then query the user if state transitions are missing. In case any operation is able to reach the missing states but its precondition is too restrictive, we synthesize a relaxed precondition covering the new state transitions. Otherwise, we synthesize a completely new operation. In general, we need to verify generated programs using the model checker by restarting the workflow. There is no fixed order that determines if an invariant violation or a deadlock state is found first. This depends on the state space and the order of its traversal.

Besides generating an operation from scratch, the user is able to modify an existing operation. The tool initially provides some sample transitions covering the behavior of an operation. This results in providing positive transitions describing the behavior of the operation's substitution. In case the operation provides a precondition, negative transitions are presented describing the behavior of the precondition. The user is then able to provide new transitions either strengthening or relaxing the precondition. Additionally, positive transitions can be provided to modify the substitution of the operation. The machine invariants can be modified in the same manner.

4 Synthesis Technique

The task of (semi-)automatically generating executable programs from a given specification is called program synthesis. There are different approaches in specifying the behavior of a program, for instance, in the form of pre- and postconditions or partial implementations. Jha et al. [19] presented a synthesis technique that uses explicit I/O examples of positive and negative behavior to synthesize loop-free programs that are correct for a set of examples.

In order to restrict the search space, the approach resorts to a library of program components D , each defined in a single static assignment represented by a formula $output = f(inputs)$. For example, for an addition instruction, a constraint would ensure that $o_1 = i_1 + i_2$ holds. Each component is unique and located in a single line of the program. Each line is characterized by its own output variable. The program inputs are also represented as program lines, characterized by their own variable and located in the first lines of the program. The program outputs are located in the last lines of the program overlapping with

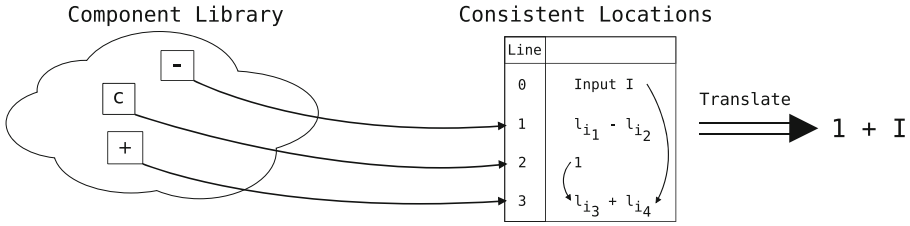


Fig. 2. An example for a possible location mapping between components

components, i.e., the program outputs are defined by compositions of components. In a synthesized program each component is assigned to a unique program line. Given N program inputs, the generated program thus has $M = |D| + N$ lines of code. Each component is used in the generated program but does not necessarily has to participate in the output, i.e., we might generate dead code which is ignored when translating the synthesized program.

Input and output variables are connected using location variables referring to other components. An output location l_{o_k} , $k \in \mathbb{N}$, describes the program line a component is defined in, while an input location l_{i_k} can be interpreted as the line of the program defining its value. Given a set of I/O examples E , synthesis searches for a mapping of locations L between inputs and outputs of components. Afterwards, the locations participating in the output of the synthesized program can be collected and translated to a corresponding abstract syntax tree.

For instance, we search for a program with one input and one output, considering three components describing *addition*, *subtraction*, and an *integer constant*. The set of I/O examples E consists of several examples describing an incrementation of an integer by one. A possible solution is illustrated in Fig. 2, where the solver enumerated the constant c to the value of 1. The single line arrows represent the mapping of the location variables (a solution for L) that participate in the output of the program. Since line one of the program does not participate in the output, the subtraction component is dead code which is ignored when translating the program. In case a program has several outputs, we collect the partial programs from the last lines of the synthesized solution representing the program outputs. Afterwards, the partial programs are combined using the parallel execution substitution of B.

We adapted this technique to synthesize B expressions and predicates using PROB [25, 26] as a constraint solver [21]. In order to synthesize B expressions, we use explicit state transitions transforming input to output values and preserving their types. In case of predicates, we replace output states by the evaluation of the desired predicate using the corresponding input states. In this context, an I/O example thus assigns an input state to output either *true* or *false*.

Initially, we are given a set of I/O examples by the user describing the complete desired behavior as well as a list of library components D to be considered during synthesis, which is either prepared automatically or provided by the user. In B, the I/O examples are states of the current machine, i.e., values of machine

variables. The authors refer to P and R as the set of input and output variables of the used library components. Let $I_i \subseteq P$ be the set of input variables of a specific component with the output $O_i \in R$, $1 \leq i \leq N$. We assume that we derive the formula of the i -th library component using $\Phi_i(I_i, O_i)$. The library is then encoded by the following constraint:

$$\Phi_{lib}(P, R) := \bigwedge_{i=1}^N \Phi_i(I_i, O_i)$$

For instance, having two components *addition* and *subtraction* the library is encoded by $(o_1 = i_1 + i_2) \wedge (o_2 = i_3 - i_4)$.

Let E_I be the set of input values and E_O the set of output values of a specific I/O example. Let L be the set of integer valued location variables of inputs and outputs of components $d \in D$. Moreover, L contains locations referring to program parameters, that means, inputs of the overall program. A constraint $\Psi_{wfp}(L, P, R, E_I, E_O)$ defines the control flow of the program to be synthesized and ensures well-formedness. This constraint consists of several parts.

For consistency, output locations are made unique by asserting inequalities between each two locations in R , which is encoded by a constraint $\Psi_{cons}(L, R)$. Component input parameters have to be defined before they are used to prevent cyclic references, which is encoded by

$$\Psi_{acyc}(L, P, R) := \bigwedge_{i=1}^N \bigwedge_{I_i \subseteq P, x \in I_i, y \in R, y \equiv O_i} (l_x < l_y). \quad (1)$$

Otherwise, a cyclic expression like $1 + (1 + (\dots))$ would be part of the search space where the location of the right input parameter maps to the addition component itself.

The approach defines program input parameters to be located in the first lines and component outputs in the ensuing lines of a program. To that effect, all components are able to access the program input parameters with respect to the acyclic constraint defined in Eq. (1). Program output parameters are defined in the last lines of the program in order to be able to access all components if necessary. To reduce the overhead, we additionally set each program output parameter to a fixed position by enumerating their positions to one of the last lines, which is achieved by a constraint $\Phi_O(E_O)$. The complete well-definedness constraint is then encoded by

$$\begin{aligned} \Psi_{wfp}(L, P, R, E_I, E_O) := & \bigwedge_{x \in E_I} (0 \leq l_x < |E_I|) \wedge \bigwedge_{x \in P} (0 \leq l_x < M) \\ & \wedge \bigwedge_{x \in R} (|E_I| \leq l_x < M) \wedge \Psi_{cons}(L, R) \wedge \Psi_{acyc}(L, P, R) \wedge \Phi_O(E_O). \end{aligned}$$

We furthermore extend the well-definedness constraint Ψ_{wfp} to ensure well-defined programs according to B. For example, sequences have to be indexed from 1 to n where n is the cardinality of the sequence.

Component inputs can either refer to a program input parameter or another component's output. By setting up constraints for each location, the authors define valid connections between program parameters and components as well as in between components. This includes ensuring type compatibility, that means, only defining connections between locations referring to the same type. We explicitly add constraints preventing connections between differently typed locations to support the PROB constraint solver in finding a solution for the mapping of location variables L . Let $\bar{L} = L_1, \dots, L_n$, $n > 0$, be a partition of the set of location variables divided by the types they refer to. We then assert:

$$\Psi_{conn}(L) := \forall_{L_1 \in \bar{L}} \left(\bigwedge_{l_x, l_y \in L_1} l_x = l_y \Rightarrow x = y \right) \wedge \forall_{L_1, L_2 \in \bar{L} \wedge L_1 \neq L_2} \left(\bigwedge_{l_x \in L_1, l_y \in L_2} l_x \neq l_y \right)$$

By combining these constraints, the behavior for a single example with a set of inputs E_I and outputs E_O is encoded by

$$\Phi_{func}(L, E_I, E_O) := \exists P, R : \Psi_{wfp}(L, P, R, E_I, E_O) \wedge \Phi_{lib}(P, R) \wedge \Psi_{conn}(L).$$

The overall behavior for a set of examples E containing tuples of input and output is then defined by asserting $\Phi_{func}(L, E_I, E_O)$ for each single example, which is referred to as the behavioral constraint:

$$Behave(E)_L := \bigwedge_{(E_I, E_O) \in E} \Phi_{func}(L, E_I, E_O) \quad (2)$$

When solving the behavioral constraint, we derive an explicit solution for the integer valued location variables in L describing a candidate program satisfying the provided behavior. Afterwards, another semantically different solution \hat{L} is searched by excluding the solution for the location variables L from the behavioral constraint defined in Eq. (2). Of course, we could also use the first solution as is without a further search. However, the user may forget edge cases when providing the set of I/O examples resulting in an ambiguous behavior. We thus want to guide the user to the desired solution as much as possible.

When finding another solution \hat{L} , the user chooses among the solutions based on a distinguishing example. That is, a program input where the output of both programs differs, which can be described by

$$\begin{aligned} \exists E_I, E_O, \bar{E}_O : Behave(E)_L \wedge Behave(E)_{\hat{L}} \wedge \Phi_{func}(L, E_I, E_O) \\ \wedge \Phi_{func}(\hat{L}, E_I, \bar{E}_O) \wedge E_O \neq \bar{E}_O. \end{aligned} \quad (3)$$

If no distinguishing example can be found, we assume both programs to be equivalent and choose the smaller one. The system iterates through further solutions in the same fashion. Continuous search for distinguishing inputs provides additional I/O examples, eventually leading to a semantically unique solution. Once found, the synthesized program is returned. It should be noted that searching for another solution possibly results in a solver timeout. In practice, the uniqueness of a synthesized program is therefore only as far as we can decide using the currently selected solver timeout.

During the synthesis of an operation, the user is able to change the output state of a distinguishing example. That means, we do not include an explicit discovered state transition in the set of examples and maybe find it again afterwards. To guarantee unique distinguishing transitions we memorize all results to exclude them from the distinguishing constraint defined in Eq. (3).

When synthesizing explicit if-statement, i.e. using an expression, we have to mix the generation of expressions and predicates leading to a larger search space and a worse performance. To that effect, we also provide an implicit representation of if-statements and implement them as follows: We successively synthesize new operations for each example of a given set of state transitions if necessary, yielding the desired behavior split into several operations. Each operation presents a conditioned block of the statement which is semantically equivalent to explicitly providing if-statements. We start with the first example $(E_I, E_O) \in E$ and synthesize an operation probably with an appropriate precondition. That is, we solve the functional constraint $\Phi_{func}(L, E_I, E_O)$. Afterwards, we decide according to the next example and the so far synthesized operations:

- A previously synthesized operation’s substitution fits the current example but the precondition is too restrictive. Hence, we relax the precondition.
- The example can be executed by a synthesized operation. We skip this example since there is nothing to do.
- No operation executes the transition. We generate a new operation only using this example for initialization.

In B, custom types are often accessed via operation parameters. In order to support parameters when synthesizing an operation we add additional components for each custom type. These components are implemented as constants which values are set locally for each example, that means, for each functional constraint of the behavioral constraint defined in Eq. (2). When synthesizing a precondition for an operation that uses parameters, we extend the I/O examples by adding each operation parameter. That means, we view each parameter as a machine variable. The behavioral constraint then considers all necessary information when generating an appropriate precondition. For instance, assuming a machine violates an invariant caused by an operation that uses one parameter and we want to synthesize a strengthened precondition. Furthermore, the machine defines one machine variable. We then extend the states obtained by the model checker by computing the operation parameter for each example and use these extended examples for synthesis.

The synthesis technique by Jha et al. [19] relies on two oracles. The *I/O oracle* is used to define the desired output of the program to be synthesized based on a given input. We replace it by the user. The *validation oracle* is used to check if a synthesized program is correct. To provide it, we apply the synthesized changes to the model and use the PROB model checker for verification.

Moreover, the technique is specialized on synthesis of loop-free programs. However, loops are a special case of the B formal method that are not necessary to be used. A finite loop can be unfolded to several operations providing the

same semantics. In B, one can mistakenly define an infinite loop which, however, is detected by PROB and prevented from execution.

5 Performance Considerations

5.1 Concerned Machine Variables

As we have to consider different combinations of program components, the search space grows exponentially with the number of involved variables. When repairing an invariant violation, we automatically reduce the examples from the model checker to those taking part in the violating state. For instance, assuming we have two machine variables m_1, m_2 . Only the variable m_1 is involved in the violated invariant, while m_2 is not. On the one hand, the user can decide to allow the violating state by synthesizing a relaxed invariant. We then only consider the variable m_1 when modifying the machine and do not change any code that involves m_2 . However, the synthesized changes may indirectly affect the behavior of the variable m_2 . On the other hand, the user can decide to remove the violating state from the model by strengthening the precondition of the operation leading to the violating state. We then additionally consider the machine variables the operation refers to. For instance, if the current precondition of the operation refers to m_2 , we consider both variables during synthesis. If synthesis fails, we can consider all machine variables as a last resort. As described in Sect. 4 the *validation oracle*, i.e. the PROB model checker, verifies the modified machine. In B, each operation may access all machine variables. In case of generating an operation from scratch or repairing a deadlock, we cannot draw any conclusions regarding the variables in use. To counter this, we allow the user to mark machine variables that are known not to take part for being skipped.

5.2 Component Library Configuration

Given that B is strictly typed, we are able to reduce the component library to a subset of B. The performance when solving the synthesis constraint itself highly depends on the library configuration. For example, if we do not need arithmetic but only logical operators, the unnecessary operators expand the search space exponentially. To that effect, we consider the types of the variables that are involved in the given examples and only use corresponding operators. Additionally, we statically provide several library configurations for each type. By default, we start with a restricted library configuration to search for simple programs at first, i.e., programs using as few components as possible. In case we do not find a solution, we successively expand the library and restart synthesis. For example, for integers we can start with operators like *addition* and *subtraction*, while not considering constants at first. If this configuration is not sufficient we successively increase the amount of used constants and consider additional operators. When all library configurations failed and no solution can be found using the current timeout, synthesis fails. At this stage, the user may choose to increase the timeout or provide more concise I/O examples.

Of course, in order to synthesize a complex program we probably need to expand the library several times. Therefore, we parallelize synthesis for different library configurations to overcome the loss of performance. In detail, we run $C = |CPU|$ instances of PROB at the same time, where $|CPU|$ is the amount of logical cores that are available to the JVM. All instances have loaded the same model and are always in the same state. When running synthesis, we call the backend C times using distinct library configurations. We listen to the instances and decide as follows for each single instance:

- Success: we return the program and cancel synthesis for the other instances
- Failure: we try another library configuration or do not restart this instance in case there is no library configuration left
- Distinguishing example found: synthesis on this instance is suspended, we present the example to the user and restart the suspended instance after the example has been validated.

To prevent enumerating constants to be synthesized without an upper or lower bound we restrict each constant domain according to the initial examples, which is automatically encoded in the behavioral constraint defined in Eq. (2). As a last resort, we widen such domains if no solution can be found.

5.3 Avoiding Redundancy

To reduce the search space, we implement symmetry reduction for adequate operators. This is done in a preliminary step and directly encoded within the synthesis constraint. Let D be the set of library components and $\bar{D} \subseteq D$ the subset of symmetric components. Assuming n is the amount of input variables of a specific component $d \in D$, we refer to $d(i)$, $i = 1, \dots, n$, as the i -th input of d . $L(d(i))$ is referred to as the location variable an input $d(i)$ is mapped to whilst $L(d)$ refers to the output location variable of the component. We encode symmetry reduction on the level of operands by the following constraint:

$$\forall d \in \bar{D} : \bigwedge_{i=1}^{n-1} L(d(i)) < L(d(i+1))$$

When considering an *addition* having two inputs i_1 and i_2 this results in $L(i_1) < L(i_2)$. That is, we consider only $o_1 = i_1 + i_2$ and avoid $o_1 = i_2 + i_1$.

Furthermore, we implement symmetry reduction on the level of the same operators to prevent changing the location of components without changing the semantics of the program. This is encoded by the following constraint:

$$\forall d, \bar{d} \in D, d \equiv \bar{d} : L(d) < L(\bar{d})$$

For example, using two components $+_1, +_2$, each representing an addition, this results in $L(+_1) < L(+_2)$, i.e., only $x +_1 y +_2 z$ can be part of the synthesized program while $x +_2 y +_1 z$ can not.

Besides that, the search of a semantically different program for a synthesized solution is another performance bottleneck. There can be numerous equivalent programs to discover, before we are able to determine the uniqueness of a solution using the current timeout or find a program yielding a different semantics. By design, a specific component can only have one output within a synthesized program, and, thus, has to be duplicated if necessary. For instance, a synthesized program uses an union encoded by $o_1 = i_1 \cup_1 i_2$. With respect to this component, synthesis may find a solution mapping a program input parameter p_1 and an enumerated constant to the component inputs, like $p_1 \cup_1 \{1, 2, 3\}$. However, we may need another union with a different output, for instance, to union p_1 with a program input parameter p_2 . We then need to use another distinct component \cup_2 . Given that, the ongoing search possibly results in swapping the inputs of \cup_1 and \cup_2 , i.e., generating $p_1 \cup_2 \{1, 2, 3\}$, but providing the same semantics. Let $L_{out}(d(i))$ be the output location variable that is mapped to the i -th input location of the component d . Given a solution for L , we prevent swapping the inputs of the same operators by asserting the following constraint to hold for the new solution \hat{L} :

$$\forall d, \bar{d} \in D, d \equiv \bar{d} : \bigwedge_{i=1}^n L_{out}(d(i)) \neq \hat{L}(\bar{d}(i))$$

Unfortunately, the preliminary symmetry reduction is not strong enough to exclude symmetric changes when searching for further solutions. Given the example from above, we assert $l_{i_1} < l_{i_2}$ to hold. This does not prevent the components p_1 and $\{1, 2, 3\}$ to swap locations when searching for another semantically different program, resulting in $\{1, 2, 3\} \cup_1 p_1$ with $l_{i_1} < l_{i_2}$ being satisfied. Moreover, we need to prevent symmetric changes between equivalent components. That means, $\{1, 2, 3\} \cup_2 p_1$ should not be part of the solution. Therefore, we additionally implement a stronger symmetry reduction when searching for further solutions. Let $\bar{D}_{sol}(L)$ be the set of symmetric components that have been used in the solution for L . We assert the following constraints to hold:

$$\forall d \in \bar{D} : \bigwedge_{i=1}^n \left(\bigwedge_{j=1}^n L_{out}(d(i)) \neq \hat{L}(d(j)) \right)$$

$$\forall d, \bar{d} \in \bar{D}_{sol}(L), d \equiv \bar{d} : \bigwedge_{i=1}^n \left(\bigwedge_{j=1}^n L_{out}(d(i)) \neq \hat{L}(\bar{d}(j)) \right)$$

Given a solution for L , the first constraint ensures that no component output that is mapped to an input of a symmetric component is mapped to any other input of this component. The second constraint ensures the same behavior but in between the same symmetric components, which is only necessary if a symmetric component is included in the component library several times like $+_1, +_2$.

Another factor that has proven to speed up search is to increase variance in synthesized programs and distinguishing examples. To do so, we randomize enumeration order when solving constraints. Using a linear order often causes

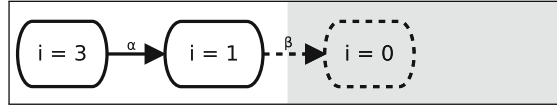


Fig. 3. Abstract visualization of an invariant violation in the user interface

only small syntactical changes among synthesized solutions resulting in more equivalent programs to be generated. Moreover, finding dissimilar distinguishing examples it is less likely to get stuck in some part of the search space where there is no solution [14].

6 User Interface

The graphical user interface is implemented in Java using the JavaFX framework. We use the PROB Java API [4]¹ providing an interface to the PROB Prolog kernel to animate and verify formal models and utilize the synthesis backend. The application can be found on Github².

When starting the application, the user is able to load a classical B machine. The UI presents a main view split in two areas defining valid and invalid states or transitions. States and transitions are represented as nodes that can be resized and are connected respectively. The workflow starts with explicit model checking as described in Sect. 3. The environment has two different states depending on the result from the model checker: If the model is erroneous, we display the invariant violating trace. Initially, we use a shortened version of the trace containing valid and invalid states. Upon user request, we show further successor and predecessor states. Manually added states are tentative by default and can be validated using PROB. States from the model checker are immutable but can be deleted. The final distribution of the nodes determines the type of the synthesized program as abstracted in Fig. 3. Here, the model checker found an invariant violation in the state $i = 0$ which is presented to the user. The states violating an invariant are assumed to be invalid and thus set to be a negative example by default. In the presented setting, the precondition of the operation β will be strengthened to exclude the invariant violating state from the model. If deciding to allow the state by moving it on the side of valid states, the machine invariants will be relaxed. Graphically, a node which state violates an invariant on the side of the pane presenting valid states or vice-versa leads to modifying the invariants. Otherwise, the precondition of the affected operation will be strengthened. If repairing an invariant violation by strengthening a precondition, the considered operation is the one leading to the first state of the trace provided by the model checker that is set to be invalid. During synthesis, distinguishing states may be presented to the user who is asked to place them according to the desired behavior.

¹ The documentation is available online <http://www.prob2.de>.

² <https://github.com/joshua27/bsynthesis>.

```

MACHINE scheduler
SETS PID = {p1,p2,p3}
VARIABLES active, ready, waiting
INVARIANT active <: PID & ready <: PID & waiting <: PID &
            (ready /\ waiting) = {} & // inv1
            active /\ (ready \/ waiting) = {} & // inv2
            card(active) <= 1 & // inv3
            (active = {} => ready = {}) // inv4
INITIALISATION active := {} || ready := {} || waiting := {}
OPERATIONS
del(p_PID) = PRE p_PID : PID & p_PID : waiting THEN
            waiting := waiting - {p_PID} END;
new(p_PID) = PRE p_PID : PID & p_PID /: active THEN
            waiting := waiting \/ {p_PID} END
END

```

Fig. 4. A partial scheduler used as a starting point

In case model checking was exhaustive, the user is able to synthesize a new operation. An operation is specified by creating transition nodes and placing them according to the desired behavior. Transition nodes consist of explicit input and output states referring to the variables that should be considered during synthesis. Furthermore, it is possible to modify an existing operation or the machine invariants as described in Sect. 3.

If synthesis succeeds, the generated program is presented to the user who can approve or discard the changes. On approval, the changes are applied to the model followed by a complete run of the PROB model checker. The user interface also presents a list of all B operators that are currently supported by the tool.

7 Example

As an example, we synthesize a B machine managing the states of several processes, which we refer to as a scheduler. Since there are no similar approaches to the semi-automated repair and generation of B formal models we are not able to compare the results. Instead, this example should illustrate the workflow described in Sect. 3.

Initially, we have started from the model shown in Fig. 4 defining only the enumerated set *PID* containing processes, the three machine variables for the different states of a process and their types as well as the initialization state. We have already synthesized four invariants and two machine operations to create a new process and to delete an existing one from the set of waiting processes. There is no required order and we could have started by generating other operations.

The workflow starts with explicit model checking. Since no erroneous state is found, we proceed to synthesize a new machine operation to activate a waiting task. As a user, we provide six examples which we set to be valid. The machine

```

set_active(p_PID) =
  PRE p_PID : PID & p_PID : waiting & active = {} THEN
    waiting := waiting - {p_PID} || active := {p_PID} END;
set_ready(p_PID) =
  PRE p_PID : PID & p_PID : waiting & active /= {} THEN
    waiting := waiting - {p_PID} || ready := ready \ / {p_PID}
  END;
active_to_waiting = PRE active /= {} & ready = {} THEN
  waiting := waiting \ / active || active := {} END;
ready_to_active(p_PID) =
  PRE p_PID : PID & p_PID : ready & active /= {} THEN
    waiting := waiting \ / active ||
    active := {p_PID} || ready := ready - {p_PID} END

```

Fig. 5. Operations that have been synthesized each at a time

invariant *inv3* specifies that there can only be one activated process at a time. To that effect, we additionally provide invalid examples for that the operation should block execution, that is, states where either no process is in the set of waiting tasks or there already is an activated task. Synthesis of the substitution using the valid examples succeeds without any further interaction. In contrast to that, the invalid examples do not describe unique behavior. The generation of the precondition thus provides three distinguishing examples that we validate according to the desired behavior. Afterwards, the operation *set_active* shown in Fig. 5 is returned. The operation *set_ready* has been synthesized in the same manner.

When running model checking, a violation of the invariant *inv1* caused by the operation *new* is found, and the user interface presents the trace leading to the violating state. Moreover, the tool automatically decided that only the machine variables *waiting* and *ready* are involved in this invariant violation. We are presented four states that are set to be valid since they do not violate any invariant and one invalid state. We do not change or add any states and run synthesis. This results to synthesizing a strengthened precondition for the operation *new* to remove the invariant violating state from the model. Without any further interaction synthesis terminates and the predicate `p_PID /: ready` is added to the precondition of the operation *new*. Another run of the model checker is exhaustive. We furthermore synthesized two operations to swap a task from being active to either waiting or ready as shown in Fig. 5.

8 Performance Evaluation

In the following we will evaluate the synthesis backend regarding its runtime for several examples. For each program, we provided a complete set of I/O examples describing unique semantics. Consequently, synthesis terminates without any interaction with the user. The synthesized programs *eval_i* can be found in the

Table 1. An evaluation of the runtime of the synthesis backend

Program	Exact library (in seconds)		Used timeout	Default library (in seconds)		Used timeout	Amount of examples
	no sym.	sym.		no sym.	sym.		
eval ₁	11.180	2.569	2.5	⊥	18.370	5.0	4
eval ₂	6.090	0.830	2.5	⊥	57.260	30.0	4
eval ₃	⊥	9.506	2.5	⊥	⊥	max	5
eval ₄	⊥	10.670	8.0	⊥	11.320	8.0	6
eval ₅	⊥	463.860	240.0	⊥	⊥	max	6
inv ₁	0.750	0.070	0.5	10.445	9.893	0.5	6
inv ₂	⊥	1.630	1.0	433.245	229.340	30.0	8
inv ₃	0.054	0.050	0.5	1.775	1.560	0.5	5
inv ₄	0.690	0.170	0.5	4.162	2.460	1.0	7
del	0.236	0.230	0.5	1.254	0.929	0.5	6
new	0.943	0.180	0.5	1.925	1.850	0.5	8
new_pre	⊥	0.046	0.5	⊥	2.609	1.0	8
set_active	1.485	0.880	0.5	6.173	4.950	1.0	8
set_ready	3.433	1.010	0.5	9.928	8.540	1.0	9
active_to_waiting	2.964	0.590	0.5	7.135	6.910	1.0	11
ready_to_active	2.792	1.730	1.0	11.459	9.210	1.0	10

Github repository mentioned in Sect. 6. The programs inv_i refer to the invariants of the machine defined in Fig. 4. We will use the average time of ten independent runs using the exact library that needs to be used to synthesize a program and the default library configuration without parallelization as described in Sect. 5.2. We used a maximum solver timeout of 10 min indicated by *max*. \perp indicates a timeout considering the used timeout of a specific benchmark. The used solver timeout and the amount of examples needed to synthesize a certain program are listed in Table 1. Furthermore, we investigate the impact of symmetry reduction suggested in Sect. 5.3. We use the same timeout when synthesizing a program with and without symmetry reduction. All presented times are measured in seconds. The benchmarks were run on a system with an Intel Core I7-7700HQ CPU (2.8 GHz) and 32 GB of RAM.

Amongst other things, the complexity of the synthesis constraint depends on the amount of considered machine variables. However, this also depends on their types which directly affect the components to be considered during synthesis. For instance, if we consider five variables that are all of the same type, it is more complex to find a unique mapping of location variables since the components overlap and can be used at several positions. In case of considering the same

amount of variables but all referring to different types, the possible locations a component can be mapped to are more restricted leading to better performance.

Besides the amount of considered machine variables, the runtime of the synthesis tool also depends on the selected solver timeout. If a solution is found, we search for another semantically different program by excluding the previous solution from the synthesis constraint. On the one hand, this might lead to finding a contradiction. On the other hand, we might need to exhaust the full solver timeout to conclude that we cannot find another solution with the current settings. We then definitely have a runtime higher than the selected timeout. For instance, solving the synthesis constraint for the program *eval₁* using the exact library that is necessary with symmetry reduction provides a solution after a few milliseconds. Afterwards, we exhaust the solver timeout when searching for another semantically different solution leading to the presented runtime.

When evaluating the impact of symmetry reduction as suggested in Sect. 5.3, one can see that symmetry reduction gains performance for each benchmark. For instance, synthesizing the program *inv₁* is around ten times faster using symmetry reduction. Of course, the impact of symmetry reduction also depends on the current settings like the library configuration or the solver timeout.

The program *eval₅* uses two explicit if-statements so that it is necessary to mix the generation of expressions and predicates. By default, B does not feature if-statements. However, the extended version of B understood by PROB provides an *if-then-else* expression. When synthesizing a program, we use program constructs like expressions backwards. That means, given an output, we search for matching inputs. Of course, PROB is not optimized in doing so for all operators, especially for extensions like if-statements. However, the native B operators are handled efficiently by the PROB constraint solver, which can be seen at the runtimes using the exact library components that are necessary.

When synthesizing the program *eval₂* or *inv₂*, we have a large difference between using the exact library and the default library configuration. Of course, this highly depends on the configured library expansions. In this case, the tool at first uses several library configurations that are not sufficient, which is either indicated by finding a contradiction or by exhausting the full solver timeout. Eventually, a sufficient library configuration is found. However, this configuration uses several unnecessary components so that a larger timeout needs to be used leading to the presented runtime. In practice, we parallelize synthesis for different library configurations as described in Sect. 5 leading to better results.

Finally, the performance mostly depends on the amount of program lines which is influenced by the amount of considered library components and program inputs as described in Sect. 4. This can be seen when comparing the runtimes using the exact library with the default library configuration.

9 Related Work

There are many other approaches to program synthesis which could in theory be used to synthesize formal models as well. For instance, techniques to create

divide and conquer algorithms using proof rather than constraint solving [30]. Inductive logic programming [29] is related in the sense that it also starts from positive and negative examples, but is normally not user-guided and is less based on constraint solving but on measures such as information gain. Compared to approaches like [3, 9] we synthesize entirely new programs based on input and output values instead of transforming an existing model.

Beyer et al. [6] present a constraint-based algorithm for the synthesis of inductive invariants expressed in the combined theory of linear arithmetic and uninterpreted function symbols. As an input for synthesis, the user specifies a parameterized form of an invariant. In theory, this approach can also be used to synthesize B machine invariants. However, in order to partially automate the development process our workflow is based on explicit model checking providing traces of machine states. Moreover, we do not demand invariants to be inductive and also want to synthesize preconditions and complete operations.

Gvero et al. [15] present a tool to synthesize Java code based on a statistical model derived from existing code repositories. The suggested approach uses natural language processing techniques to accept free-form text queries from the user and infer intentioned behavior from partial or defective Java expressions. The tool learns a probabilistic context-free grammar which is used to generate code. Finally, the user is offered a set of possible solutions ranked by the most frequent uses in the training data. In contrast, we intend to find a unique solution covering exactly the described behavior derived from explicit I/O examples.

In CEGAR [11], spurious counterexamples are used to refine abstractions. Our synthesis tool is guided by real counterexamples and provides an interactive debugging aid for model repair. Moreover, we not only rely on the model checker to find counterexamples but also use PROB as a constraint solver. This leads to more flexibility in model repair and generation, i.e., we can avoid or allow specific states and even extend a machine in case model checking has been exhaustive.

Synthesis can also be applied to functional programming. For instance, Feser et al. [13] present a tool synthesizing functional recursive programs in a λ -calculus. The suggested synthesis approach resorts to a set of higher-order functions as well as language primitives and constants. The tool specializes on synthesizing data structure transformations from explicit I/O examples. The authors define a cost model assigning a positive value to program constructs to find programs with minimal costs using deductive reasoning and a best-first search.

In addition to synthesizing formal models, one can use model checkers and model finders for program synthesis. For instance, Mota et al. [27] use the model finder Alloy [18] to synthesize imperative programs. Programs are described in terms of pre- and post-conditions together with an abstract program sketch defined in Alloy*. To that effect, this approach for specifying programs is more concise than using explicit I/O examples resulting in a smaller search space. While this provides better performance, the user needs more knowledge about the language specification and the program to be synthesized.

10 Future Work

While the example performed in Sect. 7 shows that our approach is feasible in practice, we still have to overcome performance limitations. The B-Method is quite high-level, featuring constructs like sequences, functions or lambda expressions. Powerset construction and arbitrary nesting is allowed as well, affecting the performance of the synthesis tool.

We proposed a default library configuration starting with a small library for each used type and successively considering more components if no solution can be found. In practice, we are not able to efficiently decide for which type the library needs to be expanded. Balog et al. [2] have shown that deep learning techniques can be used to predict the components that are necessary to synthesize a program for a given set of I/O examples, which we also intend to implement for our tool.

One long-term vision would be to combine our approach to model repair with generated models. Clark et al. [10] presented an extension to the internal domain specific language of the ProB Java API which can be used to define classical imperative algorithms. The tool generates an Event-B model describing the algorithm, which can then be processed by PROB and the synthesis tool. If finding an error, we can use our synthesis tool to repair the machine interactively without the need for the end user to know formal methods.

Of course, one could extend our approach to other formal languages such as TLA^+ [23]. As there is an automatic translation of TLA^+ in B [16] and vice-versa [17], we could directly use our implementation inside PROB.

11 Discussion and Conclusion

When enforcing the interactive workflow, model checking is the bottleneck for performance. In order to validate a synthesized program, we need a complete run of the model checker. The performance in discovering a violating state depends on the chosen search strategy as well as the current state of the machine. In order to increase performance in validating synthesized programs it is possible to use distributed model checking [5, 20] for models with finite state spaces.

One concern about the suggested approach is that the repeated reparation of a model using generated code affects its comprehensibility and maintainability. Of course, the generated code will be biased to the used library configuration. To counter this, we can use a B simplifier or pretty printer. Moreover, the user could provide short comments for synthesized changes that are added to the code. Besides affecting the code, an automated repair of formal models using synthesis might be considered sceptical since such changes should be made wisely. Using the suggested approach, a synthesized program always fulfils the provided behavior without false positives. As described in Sect. 4, we want to guide the user towards a unique solution as much as possible. However, in practice, we might miss further solutions with a different semantics when searching for a distinguishing example due to a solver timeout. Nevertheless, the user will either

derive a program exactly supporting the described behavior or no solution at all in case synthesis fails. In general, the user should provide an elaborated set of I/O examples each describing different semantics of the desired program, and, in the best case, covering all corner cases that overlap with semantically different programs. Since B is based on states, the representation of system behavior using explicit I/O examples seems to be justified. Nevertheless, the evolution of complex models using synthesis needs to be evaluated in a more detailed way.

Independent from the actually used synthesis technique, we believe that an interactive modelling assistant like the one we outlined above will have its merits both for teaching and for professional use.

References

1. Abrial, J.R.: *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA (1996)
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: 5th International Conference on Learning Representations (ICLR 2017) (2017)
3. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 326–340. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_30
4. Bendisposto, J., et al.: ProB 2.0 Tutorial. In: Butler, M., Hallerstede, S., Waldén, M. (eds.) Proceedings of the 4th Rodin User and Developer Workshop. TUCS Lecture Notes, vol. 18. TUCS (2013)
5. Bendisposto, J.M.: Directed and distributed model checking of B-Specifications. Ph.D. thesis. Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2015)
6. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
7. Carlsson, M., Mildner, P.: Sicstus prolog-the first 25 years. *Theory Pract. Log. Program.* **12**(1–2), 35–66 (2012)
8. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0033845>
9. Chatzieftheriou, G., Bonakdarpour, B., Smolka, S.A., Katsaros, P.: Abstract model repair. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 341–355. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_32
10. Clark, J., Bendisposto, J., Hallerstede, S., Hansen, D., Leuschel, M.: Generating Event-B specifications from algorithm descriptions. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 183–197. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_11
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15

12. ClearSy: Atelier B, User and Reference Manuals. Aix-en-Provence, France (2014). <http://www.atelierb.eu/>
13. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 229–239. ACM, New York, USA (2015)
14. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of AAAI/IAAI, pp. 431–437. American Association for Artificial Intelligence (1998)
15. Gvero, T., Kuncak, V.: Interactive synthesis using free-form queries. In: Proceedings of ICSE, pp. 689–692 (2015)
16. Hansen, D., Leuschel, M.: Translating TLA to B for Validation with PROB. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_3
17. Hansen, D., Leuschel, M.: Translating TLA⁺ to B for validation with PROB. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_3
18. Jackson, D.: Software Abstractions: Logic Language and Analysis. MIT Press, Cambridge (2006)
19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings ICSE, pp. 215–224 (2010)
20. Körner, P., Bendisposto, J.: Distributed model checking using PROB. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 244–260. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_18
21. Krings, S., Bendisposto, J., Leuschel, M.: From failure to proof: the PROB disprover for B and Event-B. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 199–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_15
22. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
23. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2002)
24. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: the ProB constraint solver 10 years on Chapter 14. In: Boulanger, J.L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, pp. 427–446. Wiley ISTE, Hoboken, NJ (2014)
25. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
26. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
27. Mota, A., Iyoda, J., Maranhão, H.: Program synthesis by model finding. Inf. Process. Lett. **116**(11), 701–705 (2016)
28. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

29. Muggleton, S., Raedt, L.D., Poole, D., Bratko, I., Flach, P.A., Inoue, K., Srinivasan, A.: ILP turns 20. *Mach. Learn.* **86**(1), 3–23 (2012)
30. Nedunuri, S., Cook, W.R., Smith, D.R.: Theory and techniques for synthesizing a family of graph algorithms. In: *Proceedings First Workshop on Synthesis, EPTCS*, vol. 84, pp. 33–46 (2012)
31. Plagge, D., Leuschel, M.: Validating B,Z and TLA⁺ Using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) *FM 2012. LNCS*, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31
32. Schmidt, J., Krings, S., Leuschel, M.: Interactive model repair by synthesis. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) *ABZ 2016. LNCS*, vol. 9675, pp. 303–307. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_25
33. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007. LNCS*, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49