Theory of Hybrid Systems
Informatik 2
RWTH AACHEN

Diploma Thesis

# Parallel user-defined strategies for QFNRA

**Henrik Schmitz**

*Supervisors:*
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

*Advisor:*
Florian Corzilius

April 2013

**Abstract**

Satisfiability modulo theory (SMT) solvers exist for various theories of first-order logic, whereas the theory of quantifier-free nonlinear real arithmetic (QFNRA) problems is quite unexplored. Corresponding state-of-the-art SMT solvers are still actively developed in order to achieve further efficiency for their underlying implemented methods.

SMT solvers are developed towards well-known problems, but seem to behave less sophisticated in unknown fields, especially those which have emerged through practical issues. This downside can mostly be vanished, when rearranging the utilization order of the used approaches. The strategy challenge proposes combinations of them, which can flexibly and easily be defined by the user in order to improve the efficiency of an SMT solver for new problem classes.

The SMT toolbox for real arithmetic, short SMT-RAT, allows the creation of SMT solvers for QFNRA problems according to a user-defined strategy. This thesis introduces an improved strategy approach for this toolbox and presents a user-friendly GUI for creating them. The new approach entails the possibility for the user to intend alternative combinations of solving procedures, depending on dynamically determined conditions on the input. Furthermore, it is capable to apply these procedures even in parallel, which can be considered as a novel achievement in this field of research.

# Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

(Henrik Schmitz)
Aachen, 3. April 2013

# Danksagung

Ich bedanke mich für die Förderung von meinen Eltern, die mir das Studium an der RWTH Aachen ermöglicht haben. Darüber hinaus möchte ich mich bei meinen Freunden und Kommilitonen für die gegenseitige Unterstützung während des Studiums bedanken, insbesondere bei Florian Corzilius.

Ich bedanke mich ebenfalls bei Prof. Ábrahám und Florian Corzilius für das abwechslungsreiche und spannende Thema meiner Diplomarbeit. Mein Dank gilt an dieser Stelle auch dem gesamten Team der Hybrid Systems Gruppe, die mich immer herzlich willkommen hießen.

Abschließend möchte ich mich auch bei meiner Universität bedanken, die mein Studium durch viele zusätzliche Veranstaltungen, Angebote und vor allem mein Auslandsjahr an der Keio Universität in Japan vielschichtig erweitert hat.

# Contents

# Chapter 1

# Introduction

In computer science, *decision problems* pose the question whether formulas from a given logic are satisfiable or not, i.e., whether there exists a type-correct assignment of values to variables of a formula such that the formula evaluates to true. Algorithmic solutions to decision problems, called *decision procedures*, are available for a variety of different logics. *SAT solvers* are highly efficient tools implementing decision procedures for propositional logic. They are widely used in both academia and industry, e.g., for the analysis of circuits. The efficiency of SAT solvers enabled the technique of *satisfiability modulo theories* (*SMT*) solving. In this approach, quantifier-free first-order logic formulas over some theories can be checked for satisfiability with the help of a SAT solver, which handles the Boolean structure of the formula, in combination with a *theory solver*, which checks consistency of constraints from the underlying theory. SMT solvers for different logics like equality logic or linear real arithmetic are widely used for, e.g., the analysis of programs or real-time systems.

In this thesis *quantifier-free nonlinear real arithmetic* (QFNRA or simply NRA) is considered, for which a great number of solving procedures exist, such as the *cylindrical algebraic decomposition* (CAD) method [6], *Gröbner bases* [24], *virtual substitution* (VS) [23] or *interval constraint propagation* (ICP) [14]. The SMT solver *Z3* [16], for instance, implements the CAD method, whereas *iSAT* [15] is based on the incomplete ICP method. *QEPCAD B* [5] is a highly tuned implementation of the CAD method without working in SMT fashion and the *REDLOG package* [11] involves a fast combination of Gröbner bases, the virtual substitution and CAD methods. The SMT solver *CVC3* [3] can only handle very few nonlinear instances by a simplification of the input formula to the supported linear fragment of NRA. *RAHD* [20] combines different existing implementations of some of the aforementioned methods in *Maxima* [19] by a user-defined strategy to obtain a theory solver for NRA. However, these implementations and therefore the obtained theory solvers are not *SMT-compliant*, which means that they support *incrementality*, *backtracking* and *infeasible subset generation*. The *satisfiability modulo theories-real arithmetic toolbox* (SMT-RAT) [7], which is subject of this thesis, can be utilized to build on the one hand SMT solvers, which handle existential fragments of NRA and, on the other hand, theory solvers to enable other SMT solvers to solve *NRA problems*. It maintains several of the mentioned decision procedures for checking the satisfiability of *NRA formulas* and simplifying methods, which can be composed

in a strategy in order to exploit the advantage of their combined power.

As efficiency is one of the important aspects, which must be regarded when designing SMT solvers, it is meaningful to compare different tool implementations by means of efficiency. Benchmarks, e.g. from SMT-LIB [2], are provided for the purpose of comparing the efficiency of SMT solvers and to motivate their further development. Unfortunately, such benchmarks have the negative side effect, that SMT solver implementations tend to be optimized for the well-known problems of these benchmarks, but do not behave sophisticated enough for new classes of problems, especially those which occur in practice, as stated in [9]. Interestingly, in many cases the implemented decision procedures and simplifier methods of SMT solvers are sophisticated enough such that for a given problem instance often more efficient compositions can be found. For this reason, [9] proposes the *strategy challenge*, which intends the user of an SMT solver to take control over how to compose the different simplifying and solving approaches.

The strategy challenge motivated that the implementation of SMT-RAT composes its decision procedures and simplifier methods in a user-defined way. Moreover, it is also the key ingredient for the first part of this thesis, which applies an enhancement of the already provided capabilities of SMT-RAT in strategy building in order to allow further flexibility. Using the new strategy approach, the user is enabled to define compositions containing alternative decision procedures and simplifier methods, which can be invoked dynamically. SMT solver tools, which allow the adaptation of the underlying composition are for example RAHD and Z3, whereas the latter one fulfills the adaptation through a great number of parameters, which is rather a less straight-forward solution for the user. This thesis pursues a more receptive way by introducing a *graphical user interface* (GUI), in its second part, in order to enable the creation of strategies in an easy and self-explanatory manner, as proposed by [9].

To my knowledge, none of the state-of-the-art SMT solvers for NRA contains parallel implementations. This is a remarkable fact, as virtually all modern computer architectures offer multiple CPU cores, which enable the ability for multiprocessing and thereby additional hardware resources, which can be exploited. This motivates the third part of this thesis, which applies a parallelization approach on SMT-RAT in order to execute several decision procedures and simplifier methods concurrently. This approach is designed to exploit the new strategy component, as it allows to set several alternatives in one strategy, which can be executed in parallel. In this way, it is guaranteed, that the most efficient alternative is always involved in the solving process. As all simplifying and solving approaches in SMT-RAT are implemented *SMT-compliant*, it is ensured in the parallelization of SMT-RAT to support these features.

**Structure of the thesis** Chapter 2 introduces basic definitions about SMT solving, which are required for the comprehension of the further course of the thesis. The SMT-RAT application and the components of its framework are presented in Chapter 3. The capabilities of the new strategy for SMT-RAT are explained in Chapter 4, whereas the new GUI for creating such strategies is presented in Chapter 5. Chapter 6 introduces the parallelization of SMT-RAT in order to exploit the new strategy design. In Chapter 7 some experimental results based on the aforementioned features are presented, while Chapter 8 motivates further goals for future releases of the toolbox.

# Chapter 2

# SMT solving for NRA

The problem of SMT solving covers the process of solving a decision problem for *SMT formulas*. An SMT formula is a Boolean combination of constraints from one or more theories of first-order logic. This thesis confines itself to the theory of NRA and thereby on *NRA formulas*.

**Definition 2.0.1** (NRA formula, direct subformula, degree of polynomial)
*An* NRA formula $\varphi$ *is a Boolean combination of (NRA theory)* constraints $c$ *and Boolean variables* $b$*. A constraint* $c$ *is, w.l.o.g., a comparison of a* polynomial $p$ *with* $0$*. A polynomial* $p$ *can either be a combination of polynomials, using the operators for addition, subtraction and multiplication, a real valued variable* $x$ *or a constant.*

$$\begin{array}{lllllllllll} \varphi & ::= & (\neg\varphi) & | & (\varphi \wedge \varphi) & | & (\exists x \varphi) & | & c & | & b \\ c & ::= & p = 0 & | & p < 0 & & & & & & \\ p & ::= & p + p & | & p - p & | & p * p & | & x & | & 0 \quad | \quad 1 \end{array}$$

*Let* $\varphi := \bigwedge_{i=0}^{n} \varphi_i$*, then* $\varphi_i$ *is a* direct subformula *of* $\varphi$ $(0 \leq i \leq n)$*. Furthermore, let* $p := \Sigma_{i=0}^{n} a_i \Pi_{j=0}^{m_i} x_j^{e_{i,j}}$*, then* $deg(p) := max(\{\Sigma_{j=0}^{m_i} e_{i,j} \mid 0 \leq i \leq n\})$ *is the* degree of polynomial $p$*.*

The operators '$>$','$\leq$', '$\geq$', '$\vee$', '$\Rightarrow$', and so on are defined as syntactic sugar, whereas the standard semantics for NRA formulas is used for all mentioned operators.

An SMT solver decides whether a given formula is satisfiable or not. In case of satisfiability, it provides a satisfying *assignment* for the formula. The following presented definitions orientate towards [18].

**Definition 2.0.2** (Assignment)
*An* assignment *for an SMT formula* $\varphi$ *is a function* $\alpha$*, which maps the real valued and Boolean variables of* $\varphi$ *to elements of the domains of the reals* $\mathbb{R}$ *and Booleans* $\mathbb{B}$*, respectively. An assignment is* full*, if all variables of* $\varphi$ *are assigned, otherwise it is* partial*.*

**Definition 2.0.3** (Satisfiability)
*An SMT formula* $\varphi$ *is* satisfiable*, if there exists an assignment* $\alpha$ *for* $\varphi$ *under which* $\varphi$ *evaluates to* true*, which is denoted by* $\alpha \vDash \varphi$*. It is* unsatisfiable*, if there exists no such* $\alpha$ *for* $\varphi$*.*

When transforming or simplifying an NRA formula $\varphi$, it results in an equisatisfiable NRA formula $\varphi'$, which possibly contains more or less variables than $\varphi$.

**Definition 2.0.4** (Equisatisfiability)
*Given two SMT formulas they are* equisatisfiable *if they are both satisfiable or both unsatisfiable.*

An example is Tseitin's encoding [22], which transforms an arbitrary Boolean combination to *conjunctive normal form* (CNF) by introducing linearly many fresh Boolean variables. A formula is in CNF, if it is a conjunction of disjunctions of *literals*. A literal is an *atom* or its negation, where an atom is either a Boolean variable or a constraint. Furthermore, a formula is in *negation normal form* (NNF), if a negation only appears in front of Boolean variables. This can be achieved by using De Morgan's laws

$$\neg(x \vee y) \equiv \neg x \wedge \neg y, \quad \neg(x \wedge y) \equiv \neg x \vee \neg y$$

and inverting a constraint, if a negation appears in front of it, by transforming the relational symbol according to

$$= \to \neq, \quad \neq \to =, \quad > \to \leq, \quad \leq \to >, \quad < \to \geq, \quad \geq \to < .$$

An SMT solver is able to solve SMT formulas of different underlying theories by maintaining a *Davis-Putnam-Logemann-Loveland-style* (DPLL-style) [8] SAT solver and decision procedures for each of the targeted theories. The input formula is transformed into CNF and NNF, as mentioned before. In order to decide the satisfiability of the resulting formula, it is abstracted to its Boolean skeleton by replacing each of its constraints with a fresh *propositional* variable, where either the value true or false can be assigned. The SMT solver exploits a DPLL-style SAT solver in order to find satisfying assignments for the Boolean skeleton. The assignment is then checked for *consistency* with the underlying theory by utilizing an appropriate decision procedure. As the formula is in NNF, only those constraints need to be checked, for which the corresponding Boolean abstraction is assigned to true. This achieves a sophisticated way of deciding the satisfiability of SMT formulas containing theories richer than propositional logic. In this context a decision procedure implementation is also called a *theory solver* in order to emphasize its usage for a given theory.

SMT solving can be distinguished by two main variants. In the *full lazy SMT solving* approach, an SMT solver utilizes the implemented SAT solver in order to find a full satisfying assignment for the Boolean abstraction of a given SMT formula, which is then checked for consistency by the theory solver. In case of consistency, the SMT formula is satisfiable. In case of inconsistency, the SAT solver is required to create another full assignment. This process is repeated until a consistent assignment can be found or until the SAT solver is not able to create further assignments, which implies the unsatisfiability of the given SMT formula.

In the *less lazy SMT solving* approach, which is depicted by Figure 2.1, a SAT solver utilizes the theory solver to check the consistency of a partial assignment. Partial assignments are built for each *decision level* according to [8]. As a partial assignment is extended from one decision level to the next, the set of constraints

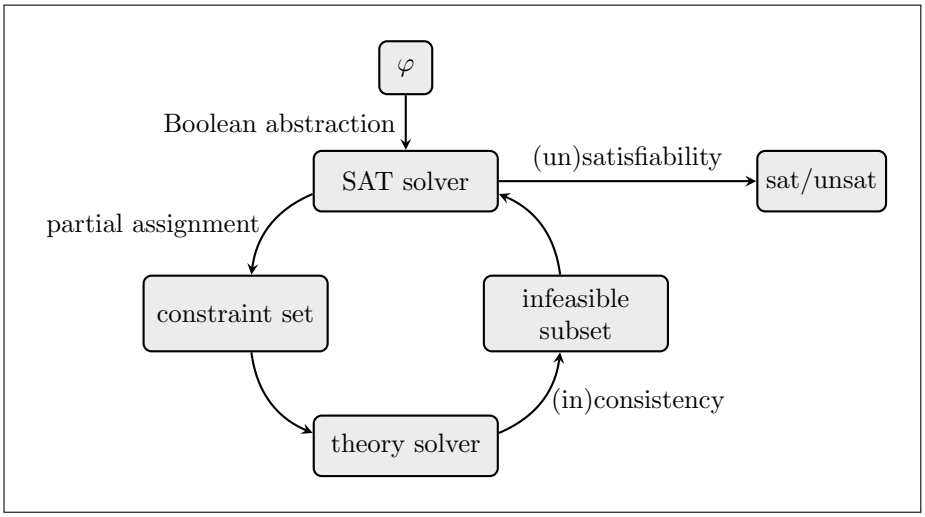



Figure 2.1: The less lazy SMT solving approach.

to be checked is extended as well. For efficiency reasons it is significant, that the theory solver is able to reuse results from its previous checks. This approach is called *incrementality*. In case a checked assignment is consistent, the SAT solver continues, otherwise, the theory solver provides *infeasible subsets*, which are reasons for the inconsistency in form of a set of infeasible subsets of the checked constraints. The SAT solver can utilize these sets in order to prevent from getting into the same conflicts again. Choosing a new partial assignment involves an adjustment of the Boolean assignment of the SAT solver, which might undo decision levels. This means that some of the constraints, held by the theory solver, must subsequently be removed. When removing these constraints, the theory solver tries to keep as many of the still relevant results of the previous check, which are referred to as the *backtracking* ability. The three mentioned abilities, incrementality, backtracking and infeasible subset generation, make theory solvers for less lazy SMT solving *SMT-compliant*.

# Chapter 3

# SMT-RAT

The SMT-RAT application, as presented in [7], enables the development of customized SMT solvers. The core of an SMT solver constructed with SMT-RAT consists of implementations of transformations, simplifications and decision procedures for NRA formulas, which in the context of SMT-RAT are called *modules*. The toolbox offers a various range of modules and its modular framework is designed in order to be extended by additional implementations. Further customizations are achieved by utilizing user-defined compositions of the contained modules. A composition specifies a set of modules and defines the way of their interaction in order to complete an SMT solver. There exists a wide variety of NRA problems, which ask for tailor-made solutions for efficient SMT solving. Therefore essential concerns of SMT-RAT are to provide versatile and powerful SMT-RAT modules and to allow their custom-made application. Furthermore, a resulting solver is SMT-compliant, which gives the opportunity to embed it as a theory solver into a DPLL-based SMT solver, although it can as well be utilized as a standalone application.

The main components of the toolbox are the modules, a strategy and the manager. Their designated usage is outlined in the following sections. When mentioning SMT solvers in the following of this thesis, they refer to customized SMT solvers created with the SMT-RAT application, unless stated otherwise.

## 3.1 Module

The development of the SMT-RAT application includes providing efficient SMT-RAT *modules* for checking the consistency of NRA formulas. The different module implementations apply different approaches in order to either check the satisfiability of an NRA formula or to transform or simplify its underlying form. The currently provided SMT-RAT modules are introduced at the end of this section.

### 3.1.1 Module interfaces

When including an SMT-RAT module into an existing SMT solver, a key ingredient for efficiency is that a theory solver implementation composed by SMT-RAT modules is SMT-compliant. Besides that, the interaction of different mod-

ule instances within any composition of SMT-RAT as well benefits from the fact, that their interfaces are implemented according to the concepts of SMT-compliance, as discussed in the previous chapter. The following interfaces are defined for each SMT-RAT module:

`check(formula` $\varphi$`):` This method contains the respective consistency check implementation of an SMT-RAT module for the given formula $\varphi$, which is referred to as *received formula*. Note that the received formula is, w.l.o.g., a conjunction of formulas. When invoking the module, it checks for the consistency of its received formula $\varphi$. The result can be of one of the following three states: `sat`, if the formula is satisfiable, `unsat` for the contrary case and `unknown` in case the implementation is not capable of checking the satisfiability $\varphi$.

`addFormula(formula` $\varphi_i$`):` This method is utilized in order to fulfill the incrementality aspect of SMT-compliance. This method is invoked in order to append the direct subformula $\varphi_i$ to the received formula $\varphi$ of the module (by conjunction).

`removeFormula(formula` $\varphi_i$`):` This method is the counterpart of the previously explained method as it is invoked to remove direct subformulas $\varphi_i$ from the received formula $\varphi$ of the module. This method requires the ability of backtracking.

`getInfeasibleSubsets():` The aspect of retrieving infeasible subsets is fulfilled by this method. In case a module detects the unsatisfiability of its received formula $\varphi$, this method can be invoked to retrieve the corresponding infeasible subsets. In SMT-RAT, infeasible subsets are returned in form of a set of subsets of the set of direct subformulas $\varphi_i$.

### 3.1.2 Module implementations

The toolbox contains a set of SMT-RAT module implementations, which can be extended with more implementations either by the user or by future releases of SMT-RAT. Currently the toolbox provides the following modules.

`CADModule` The `CADModule` contains a complete procedure, which implements the CAD method. The implementation can solve any conjunctions of constraints, but is often inefficient, especially when equations occur.

`CNFerModule` The implementation of the `CNFerModule` transforms a given NRA formula into CNF using the aforementioned Tseitin's encoding. The module is then supposed to pass the result to a further module.

`GroebnerModule` The `GroebnerModule` employs Gröbner bases in order to detect the unsatisfiability of equations, and under certain circumstances also inequations, of a given NRA formula. In case of being unsuccessful, it supports the ability of invoking a further module on a simplified version of its received formula.

`LRAModule` The name `LRAModule` derives from the fact, that it implements the Simplex method of [12], which tries to solve the *linear real arithmetic* (LRA) fragment of an NRA formula. In case this module detects the satisfiability of this LRA fragment, but the found assignment does not satisfy the NRA fragment of the formula, it can exploit a further module implementation.

`PreprocessingModule` The `PreprocessingModule` is a general simplifier. It as well implements a method to transform an NRA formula into CNF with some additional preprocessing before and after the transformation. The preprocessing of a formula includes also the weighting and thereby prioritizing of constraints for a further invoked module.

`SATModule` It is meaningful to prefix the `SATModule` with the `CNFerModule`, as it can only work on NRA formulas in CNF. The implementation transforms the received formula internally into its Boolean abstraction, where each constraint is replaced by a fresh Boolean variable. The result is solved with a DPLL-style less lazy SAT solver [13] and every time it finishes a decision level, the `SATModule` invokes a further module on the constraints corresponding to the Boolean abstraction assigned to true.

`SimplifierModule` The `SimplifierModule` is a simplifier for constraint conjunctions and bases on smart simplifications as proposed in [10]. As a simplifier module it is supposed to forward the simplified received formula to a further module.

`VSModule` The virtual substitution method is implemented in the `VSModule`. The implementation searches for possible candidate solutions of the variables of a given NRA formula, whereas only a finite number must be considered for each constraint. The VS method cannot create any candidate solutions for constraints, which have a variable degree greater than 2, but in such a case a further module is invoked for support.

When referring to the term *Modules* in the following, the set of the aforementioned modules are meant.

## 3.2 Strategy

A composition of SMT-RAT modules is expressed by a *strategy*, which defines a sequential call hierarchy among the intended modules. For this purpose, a strategy needs to define

- the set of intended modules,
- which of the modules should be invoked as a further module in order to continue the consistency check, and
- under which *condition* such an invocation can be permitted.

The structure of conditions is explained in the next subsection, whereas the syntax and semantics for building strategies are presented afterwards.

### 3.2.1 Conditions

Conditions make a strategy dynamic, which means, that the choice of the module to solve a given formula bases on the formula itself. It is of great importance, to have this possibility as it often highly depends on the structure and the contents of the formula, which module might perform best to solve it. For instance, it is senseless to invoke the `LRAModule` on an NRA formula, which does not contain any LRA fragments. A condition consists of a Boolean combination of propositions, which state demanded properties of NRA formulas. The set of all possible conditions is denoted by *Conditions*. In case a formula satisfies a given condition the corresponding module becomes *available* and can be invoked, otherwise not. The SMT-RAT application provides several kinds of propositions: There exist propositions, which can be used to describe properties of the Boolean structure of an NRA formula, for example if a formula is in CNF, if it is a pure conjunction, a clause, a literal or an atom, or if it contains Booleans. Further propositions address the properties of the constraints contained in a given NRA formula, for instance, if equations or (weak/strict) inequalities are contained. At last, propositions can be related to the polynomials of the constraints, for instance, whether they are linear, nonlinear, multivariate or of a degree less than a certain bound.

### 3.2.2 Strategy scheme

The possibility of an SMT solver to utilize user-defined strategies of SMT-RAT modules is one of its main distinctive features in comparison with the standard DPLL-based SMT solver model, which does not use module compositions. It is therefore interesting to explain how strategies can be built for SMT-RAT and how a standard DPLL-based SMT solver can be constructed by an SMT solver created with SMT-RAT. Before explaining the semantics of strategies by an example, which constructs the standard DPLL-based SMT solver as introduced in the last section, the syntax of a strategy is presented:

**Definition 3.2.1** (Syntax of strategies)
*A strategy $s$ can be constructed with the following abstract grammar:*

$$s ::= (c \ ? \ s : s) \mid m,$$

*where $c \in Conditions$ and $m \in Modules$.*

Such a strategy $s_1 := (c_1 \ ? \ m_1 : s_2)$ is evaluated according to a given formula $\varphi$ by checking if $\varphi$ satisfies condition $c_1$. If the case is given, module $m_1$ is used to check the satisfiability of $\varphi$, otherwise it must be continued with strategy $s_2$ and $\varphi$ in the same manner. It must be stated, that even in case no condition of a strategy holds, a module will still be determined at the end.

### 3.2.3 Strategy example constructing the standard DPLL-based SMT solver model

Using the provided syntax, the standard DPLL-based SMT solver model can be constructed with the following strategy:

(`is_conjunction` ? `CADModule` : (`is_in_cnf` ? `SATModule` : `CNFerModule`))

As can be seen by this example, a strategy consists of a list of pairs of a condition and a module, whereas the last element of the list is just a module. When processing a strategy, the list is traversed from outside to inside until one of the conditions is satisfied by the given NRA formula. For the given example, this means, that if, on the one hand, a given NRA formula is a conjunction, it can directly be checked by the `CADModule`, as its implemented decision procedure underlies a complete algorithm. If, on the other hand, an NRA formula is not a conjunction and also not in CNF, it is processed by the last module, the `CNFerModule`, because the last module is always applied in case no condition is satisfied by the formula. After applying the `CNFerModule`, the formula can be checked by the `SATModule` implementation, as the formula would then be in CNF. After being processed by the `SATModule` the formula is received by the `CADModule`, which can check the formula, as the `SATModule` asks for the consistency of a conjunction of constraints.

The first objective of this thesis, describes the enhancement of the capabilities of the strategy component and is presented in the next chapter.

## 3.3 Manager

SMT solvers designed with SMT-RAT are instances of a manager, which holds all the aforementioned components together and also handles their interactions. It maintains a frontend to receive NRA problems from the environment and to return the results of the satisfiability check on this formula. This environment could either be a user, who applies SMT-RAT on example files defining an NRA formula, or another SMT solver. On the basis of the strategy the manager decides whether a *backend* can be used by an invoking module or not. A backend is as well a module, but its term emphasizes the relation to another module, which can invoke this backend. Such a *module-backend-relation* is used, in case a module is unable to apply a satisfiability check independently, because its algorithm is incomplete for instance, or when its implementation intends the transformation or simplification of a given NRA formula only. A module can invoke backends by calling the interface `runBackends(..)`, which is provided by the manager.

`runBackends(formula` $\varphi'$`):` While checking a formula $\varphi$ a module can utilize a backend in order to ask for the satisfiability of a formula $\varphi'$, which in this context is called *passed formula*. For this reason, the method `runBackends(..)` is invoked on $\varphi'$, which is provided by the manager and which is responsible for forwarding the communication to the backend. In case no backend is intended by the given strategy it returns `unknown`. If a backend is available, the manager invokes its interface `check(..)` on the passed formula $\varphi'$, which becomes the received formula of that backend. Afterwards the result is returned to the invoking module, which can then use this result and continue checking $\varphi$. In case the result of checking $\varphi'$ is unsatisfiable, the invoking module can use `getInfeasibleSubsets()` of the backend for retrieving reasons, which can influence the further progress of checking $\varphi$.

A more detailed description of the complete procedure, which as well integrates the newly introduced strategy component of the next chapter, is given at

the end of Chapter 6.

# Chapter 4

# Strategy graph

As mentioned before, SMT-RAT utilizes user-defined strategies, as proposed by [9], to compose several single SMT-RAT modules of the toolbox into one powerful SMT solver. Previous releases of SMT-RAT allow only the usage of strategies, which pursue sequential executions of the composed modules. The first task of this thesis covers the enhancement of the available capabilities of the already implemented strategy component of SMT-RAT. This enhancement should provide the opportunity to compose given SMT-RAT modules for sequential as well as for *parallel* executions.

This chapter begins by motivating the reason for renewing the existing strategy component of SMT-RAT and by specifying the objectives for the enhancement process.

## 4.1 Motivating parallel strategies

When creating solutions for solving SMT problems, efficiency is an important aspect and, hence, the main motivation to develop strategies, which are additionally capable of guiding executions of composed modules in parallel. Upgrading the existing strategy component of SMT-RAT with parallelism, gives the potential for more flexible and more powerful module compositions.

Nowadays, processor manufacturers accelerate software executions rather by offering multiple cores within one CPU than by increasing the clock rates of single core processors. In order to be able to benefit from accelerations, software engineers must also adapt their algorithms to multiprocessing. Preceding releases of SMT-RAT can just build SMT solvers, which only exploit one single CPU core. Thereby the toolbox dismisses valuable hardware resources, when being executed on computer architectures containing multiple cores. Introducing parallel strategies in SMT-RAT is a first step to exploit more available resources and to improve efficiency by the usage of parallelism.

The way how more available resources can be exploited emerges, when the different properties of the SMT-RAT modules, which have been described in the previous chapter, are considered. As different SMT-RAT modules underlie different mathematical approaches, a perfect composition of them does not exist in general. The efficiency of a composition also depends on the properties of the presented NRA formulas, which should be checked for satisfiability. For

instance, for certain kinds of formulas it makes sense to utilize a `CADModule`, whereas in other cases, it might be more efficient to use a `GroebnerModule`. It is also possible, that the best approach might be to use a composition, which enables a `GroebnerModule` to invoke a `CADModule` as its backend, in case of being unsuccessful. The choice of a module or composition of modules is a critical aspect, which should be considered, when regarding efficiency. It is desirable to dynamically choose the intended module or composition, which can check a given formula most efficiently. Sequential hierarchies allow a module to invoke a backend, but they cannot offer several backends to allow alternatives. Parallel strategies improve this approach, as they enable modules to utilize a set of alternative backends, which are supposed to be executed concurrently. An NRA formula can thereby be checked with different mathematical approaches simultaneously. This ensures, that the process of checking a given NRA formula with the intended modules or compositions is performed in an optimal fashion, as they are all involved in the process.

## 4.2 Objectives for enhancing the strategy component

Before explaining the concept of the underlying structure of the new strategy component, it should be clearly stated what the component must be capable of.

**Module maintenance** First of all, the component must be able to define, which SMT-RAT modules can be utilized. This involves maintenance of their corresponding *module instances* and conditions. In the case of the new strategy component, several SMT-RAT modules of the same type can be used within one SMT solver, which is the reason for keeping instances of modules and not modules. As mentioned before, an SMT-RAT module requires a condition in order to decide, whether it is useful to invoke it as a backend of another module.

**Module-backend-relations** Moreover, it must be possible to save directed relations of pairs of the maintained module instances to point out their intended call hierarchy in a composition of module instances. This means, that these relations represent the module-backend-relations of the composed module instances in the strategy. The component must be enabled to manage either *sequential*, *parallel* or *combined strategies*. A sequential strategy allows no selection of alternative backends for any of the module instances in a composition. Only a single or no backend at all can be assigned to each module instance. A parallel strategy does offer alternative backends at intended points in the strategy. When using the term combined strategy, it emphasizes the fact, that a given strategy contains sequential as well as parallel parts.

**Prioritized backend executions** In case a parallel strategy is used, sets of backends are assigned to certain module instances. A parallel working SMT solver is supposed to execute all backends of such a set concurrently. Furthermore, several currently executed module instances of the strategy might invoke their backends concurrently. The SMT solver should then execute all backends of all invoked sets in parallel. When executing several

backends concurrently, the occasion might arise, that not enough hardware resources are available to process all backends efficiently in parallel. The reason for this case is outlined in Chapter 6. The SMT solver must then be enabled to decide over an execution order of all currently intended backends, as it can only process subsets of them successively. The new strategy component therefore expands the set of module attributes by a *priority value*, which allows to prioritize executions of any module instance in the strategy after a predefined order.

**Backend retrieval** Besides the task of maintaining strategies, the renewed strategy component must as well entail functionality. A method must be provided, which returns the intended set of backends for any module instance of the underlying composition. Moreover, this method must dynamically filter those backends, which are actually available for a current NRA formula.

On the one hand, the above listed objectives present the requirements for the enhanced strategy implementation. The manager, on the other hand, also requires enhancements to enable concurrently running SMT-RAT modules. They are described in Chapter 6, which examines how parallelization is implemented inside SMT-RAT.

## 4.3 Grammar for strategy graphs

The preexisting strategy component is replaced by a new component named *strategy graph*. Its name highlights its underlying concept of using graph structures, as it will be fully explained later in this chapter. The term strategy graph is from now on synonymously used for strategies, which are capable of maintaining sequential, parallel and combined compositions. The plain term *strategy* is then again used as a general term for any type of strategy. Derivations of the following introduced Grammar $\mathcal{SG}$ are abstractions of the internally maintained compositions of strategy graph instances.

**Definition 4.3.1** (Grammar $\mathcal{SG}$ for strategy graphs)
*Abstract strategy graphs derive from the formal Grammar*

$$\mathcal{SG} = (N, \Sigma, R, G),$$

*whereas the nonterminal symbols are given by the set*

$$N = \{G, G', P, S, B, C, M\},$$

*and the terminal symbols are denoted by the set*

$$\begin{aligned}\Sigma = \; & \mathcal{L}(\mathcal{C}) \cup \\ & \{\texttt{start[0]}, \texttt{.}, \texttt{/}, \texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}, \Rightarrow, i\} \cup \\ & \{\texttt{cad}, \texttt{cnf}, \texttt{groebner}, \texttt{lra} \\ & \texttt{prepro}, \texttt{sat}, \texttt{simplifier}, \texttt{vs}\},\end{aligned}$$

*which is the union of the language of conditions, which will be introduced later by Grammar $\mathcal{C}$ of Chapter 5, the set of auxiliary terminal symbols and the set of all currently available SMT-RAT modules. $G \in N$ is the start symbol and the production rules are denoted by the set $R$, which are as follows:*

$$
\begin{array}{lcl}
G & \rightarrow & \texttt{start[0]}.G' \\
G' & \rightarrow & P \mid S \\
P & \rightarrow & (S/G') \\
S & \rightarrow & B.G' \mid B \\
B & \rightarrow & C \Rightarrow M\,\texttt{[i]} \\
C & \rightarrow & \textit{start symbol of Grammar } \mathcal{C}\textit{, see Definition 5.3.1} \\
M & \rightarrow & \texttt{cad} \mid \texttt{cnf} \mid \texttt{groebner} \mid \texttt{prepro} \mid \\
 & & \texttt{lra} \mid \texttt{sat} \mid \texttt{simplifier} \mid \texttt{vs}
\end{array}
$$

*The grammar uses the placeholder variable* $\texttt{i}$*, where* $\texttt{i} \in \mathbb{N}$ *is a unique priority value within the strategy graph. For a given sub-strategy graph* $C \Rightarrow M\,\texttt{[}i_n\texttt{]}.G'$ *it holds, that* $\texttt{minPriority}(G') > i_n$*, with* $\texttt{minPriority}(G')$ *being the minimum of all priorities appearing in sub-strategy graph* $G'$*.*

### 4.3.1 Semantics of strategy graphs

For a better comprehension of the grammar the semantics of the terminal and nonterminal symbols is explained. The letters of the nonterminals are standing for the following: $G$ is the start symbol of the production rules and as strategy graphs are derived it simply stands for graph. The nonterminal $G'$ is used to derive *sub-strategy graphs*, which are subgraphs of a strategy graph. A subgraph contains the same construction as a strategy graph derived from $G$, with the exception of not possessing the *Start module* as its *root*. Subgraphs become handy, when explaining the terminal symbols of the grammar or traversing a strategy graph, which is for instance done by Algorithm 1. $S$ furthermore stands for sequential, $P$ for parallel and $B$ for backend. A backend or module instance assembles from a condition, a type of module instance and a priority value, symbolized by the nonterminals $C$, $M$ and the placeholder variable `i` respectively. As mentioned before, a condition is a Boolean combination of propositions to demand properties of NRA formulas. No concrete production rules for the related nonterminal $C$ are given at this point, because conditions are derived from Grammar $\mathcal{C}$, whose description is deferred to Definition 5.3.1 of the next chapter.

***Start* module** The terminal symbol '`start[0]`' expresses the inevitable *Start* module, which is the root of each strategy graph. It does not implement any solving approach itself, but is utilized to receive the *initial NRA formula* of a given NRA problem. Its task is to forward the initial NRA formula to the *initial module instances* of a strategy graph. Maintaining the initial module instances as backends of the *Start* module has the advantage of enabling the usage of *parallel subgraphs* at the direct beginning of a strategy graph, which can then be invoked concurrently.

**Sequential operator** The *sequential operator* is denoted by the terminal symbol '`.`' and outlines a sequential call hierarchy of module instances. Having the sub-strategy graph $G_1 := \texttt{TRUE} \Rightarrow M_1\,\texttt{[1]}.\texttt{TRUE} \Rightarrow M_2\,\texttt{[2]}.\texttt{TRUE} \Rightarrow M_3\,\texttt{[3]}$, where $M_1$, $M_2$ and $M_3$ are module instances and `TRUE` denotes the condition, that always holds. It means, that $M_3$ is the one and only backend of $M_2$, which is the one and only backend of $M_1$. If available, $M_1$ can invoke $M_2$ on its passed formula, which then again can invoke $M_3$ on its passed formula as well.

There are no backends intended for $M_3$ in this strategy. When the sequential operator is used, all invocations are intended to be executed sequentially one after the other. The sequential operator can also be utilized to indicate sequential call hierarchies of subgraphs. The module instances $M_2$ and $M_3$ can be composed into the subgraph $G_2 := M_2.M_3$ and the above stated strategy can then be changed into $M_1.G_2$. It outlines that all module instances of the subgraph $G_2$ are executed after module instance $M_1$. The sequential operator is right-associative, which means that the sub-strategy graph $G_1$ is interpreted as `TRUE` $\Rightarrow M_1$`[1].(TRUE` $\Rightarrow M_2$`[2].TRUE` $\Rightarrow M_3$`[3])`.

**Parallel operator** The terminal symbol '`/`' represents the *parallel operator*, which outlines a parallel call hierarchy of invocable backends or subgraphs of a module instance. The sub-strategy graph $G_1 :=$ `TRUE` $\Rightarrow M_1$`[1].(TRUE` $\Rightarrow$ $M_2$`[2]/TRUE` $\Rightarrow M_3$`[3])` expresses, that the module instances $M_2$ and $M_3$ are both backends of the module instance $M_1$. If both are available, $M_1$ can call them on its passed formula, which is then supposed to be executed in both of them in parallel. In order to ease the algorithmic processing of a strategy graph, the parallel operator is binary. This is realized by the usage of the terminal symbols '`(`' and '`)`' for each pair of backends or subgraphs, which should be executed in parallel. A set of three backends for $M_1$ is expressed by the subgraph $G_1 :=$ `TRUE` $\Rightarrow M_1$`[1].(TRUE` $\Rightarrow M_2$`[2]/(TRUE` $\Rightarrow M_3$`[3]/TRUE` $\Rightarrow M_4$ `[4]))`, where $M_4$ is a module instance as well.

**Types of module instances** The grammar contains several terminal symbols, for example `cad` or `lra`, for each type of the currently implemented SMT-RAT modules of the toolbox. This set of terminal symbols can vary between releases of the toolbox and the user of SMT-RAT is as well enabled to integrate own module implementations to achieve desired customizations. This becomes again important, when discussing the implementation of the GUI in Chapter 5. Note again, that if the same module occurs at different positions of the strategy, it refers to different instances of this module.

**Priority values** The terminal symbols of priority values are expressed by the placeholder variable `i`. As said, priority values are utilized to indicate the order of intended module instance executions, in case hardware resources are not sufficiently available. Chapter 6 is reserved to explain further details. The lower a priority value, the higher the priority of a module instance. The sub-strategy graph $G_1 :=$ `TRUE` $\Rightarrow M_1$`.(TRUE` $\Rightarrow$ `cad[1]/TRUE` $\Rightarrow$ `vs[2])` is given as an example. In case resources remain only for the concurrent execution of one further module instance, the module instance of type `cad` will be executed first. After its execution or after sufficient hardware resources become available again, the execution of the module instance of type `vs` will be initialized, because it has the next highest priority. Moreover, it might be the case, that several concurrently executed module instances are invoking their sets of backends. The execution order of all backends of all the sets must then be prioritized. For this purpose, the priority values must be unique throughout a whole strategy graph.

As the priority values denote the execution order of module instances, the *Start* module, as the root of all strategy graphs, has the highest priority, which is denoted by the lowest priority value `0`. Moreover, Definition 4.3.1 constitutes,

that priority values must have an ascending order, when the sequential operator is used. This is meaningful, as module instances will never invoke module instances, which are predecessors of themselves. Furthermore, the later introduced GUI implementation displays all priority values together with all module instances and the user can thereby easily follow the execution flow of a strategy graph. In the strategy graph $G_1\texttt{.}G_2$, all priority values of $G_2$ must be higher than those of $G_1$. When the parallel operator is used, the order of the priority values of the subgraphs are independent from each other. In the strategy graph $G_1\texttt{.(}G_2\texttt{/}G_3\texttt{)}$, the contained priority values of subgraph $G_2$ are ordered completely independent of the priority values of subgraph $G_3$ and vice versa. As both subgraphs are supposed to be executed concurrently, this allows to alternate the execution of module instances in both subgraphs, in case not sufficient hardware resources are present to execute all module instances of both parallel subgraphs concurrently. It must not be dismissed, that the priority values of both subgraphs must still be higher than those of subgraph $G_1$, as they are executed afterwards and the sequential operator is used.

**Auxiliary terminal symbols** The grammar includes some auxiliary terminal symbols, which are not necessarily required to fulfill the tasks of the grammar, but which improve the readability of the derivations and the later introduced algorithms.

- The terminal symbol '⇒' outlines the execution of a module instance. An example would be $C_1 \Rightarrow M_1\texttt{[5]}$. If the condition $C_1$ evaluates to true for a given NRA formula, the manager forwards the passed formula to a module instance of type $M_1$, which is then executed with a priority of `5`.

- A pair of an opening square bracket '`[`' and a closing square bracket '`]`' is used to clutch and thereby mark priority values within the derivation.

## 4.4 Strategy graph examples

A few practical examples are presented in the following to explain the way how strategy graphs can be derived from the Grammar $\mathcal{SG}$. For simplicity, all examples contain only the already introduced condition '`TRUE`'.

A strategy graph, which intends the usage of only a single SMT-RAT module can be meaningful, when just one complete decision algorithm should be used, as can be seen in the first example.

**Example 4.4.1** (Strategy graph with one SMT-RAT module)
*An example for a strategy graph, which contains one single module instance would be:*

$$w_1 = \texttt{start[0].TRUE} \Rightarrow \texttt{cad[1]}$$

The next example shows how an instance of the new strategy graph component can maintain a sequential call hierarchy of several SMT-RAT modules, as it was already possible with the preceding strategy component. This allows to exploit the power of several mathematical solving and simplification approaches.

**Example 4.4.2** (Sequential strategy graph)
*An example of how SMT-RAT modules can be composed for a sequential execution order would be:*

$$w_2 = \texttt{start[0].TRUE} \Rightarrow \texttt{lra[1].TRUE} \Rightarrow \texttt{groebner[2].TRUE} \Rightarrow \texttt{vs[3].TRUE} \Rightarrow \texttt{cad[4]}$$

The new capability of the strategy graph component of building parallel strategies is presented in the next example. As explained, a parallel strategy enables an SMT solver to follow up several approaches for a single NRA problem simultaneously.

**Example 4.4.3** (Parallel strategy graph)
*This example illustrates how a strategy graph can assemble several SMT-RAT modules for a parallel execution:*

$$w_3 = \texttt{start[0].(TRUE} \Rightarrow \texttt{groebner[1]/(TRUE} \Rightarrow \texttt{cad[2]/TRUE} \Rightarrow \texttt{vs[3]))}$$

A combined strategy graph allows the most flexibility, as both approaches of sequential and parallel hierarchies can be exploited. It allows to build several sequential hierarchies of SMT-RAT modules, which can be invoked in parallel and can then again split into several further sequential hierarchies, and so on.

**Example 4.4.4** (Combined strategy graph)
*A more complex strategy graph combining sequentiality and parallelism could be:*

$$w_4 = \texttt{start[0].(TRUE} \Rightarrow \texttt{cad[1]/(TRUE} \Rightarrow \texttt{prepro[2].TRUE} \Rightarrow \texttt{sat[3].(TRUE} \Rightarrow \texttt{cad[4]/TRUE} \Rightarrow \texttt{vs[9])/TRUE} \Rightarrow \texttt{lra[5].TRUE} \Rightarrow \texttt{groebner[6].TRUE} \Rightarrow \texttt{vs[7].(TRUE} \Rightarrow \texttt{groebner[8]/TRUE} \Rightarrow \texttt{cad[10])))}$$

The last example again emphasizes, that priority values of parallel subgraphs can be distributed independently from each other. This is useful in order to allow a switching of the execution of module instances in different subgraphs, in case no sufficient hardware resources are available, as it was stated in the previous section.

## 4.5 Concept and capabilities of the grammar for strategy graphs

The strategy graph $w_4$ of Example 4.4.4 contains a manageable number of only ten module instances, but is already complex to overlook. A graphical illustration of the strategy graph, which can be seen in Figure 4.1, counters this problem. It reveals the underlying concept of the strategy graph component and why it actually deserves its name: Its structure is an acyclic, directed and weakly connected graph, where in a weakly connected graph each node can be reached from every other node, when changing the directed into undirected edges. Moreover, this graph is actually a tree, but the component keeps its name for future planned applications. Chapter 8 proposes a purpose for the usage of cyclic graphs. The concept of a graph has been chosen, as it can easily

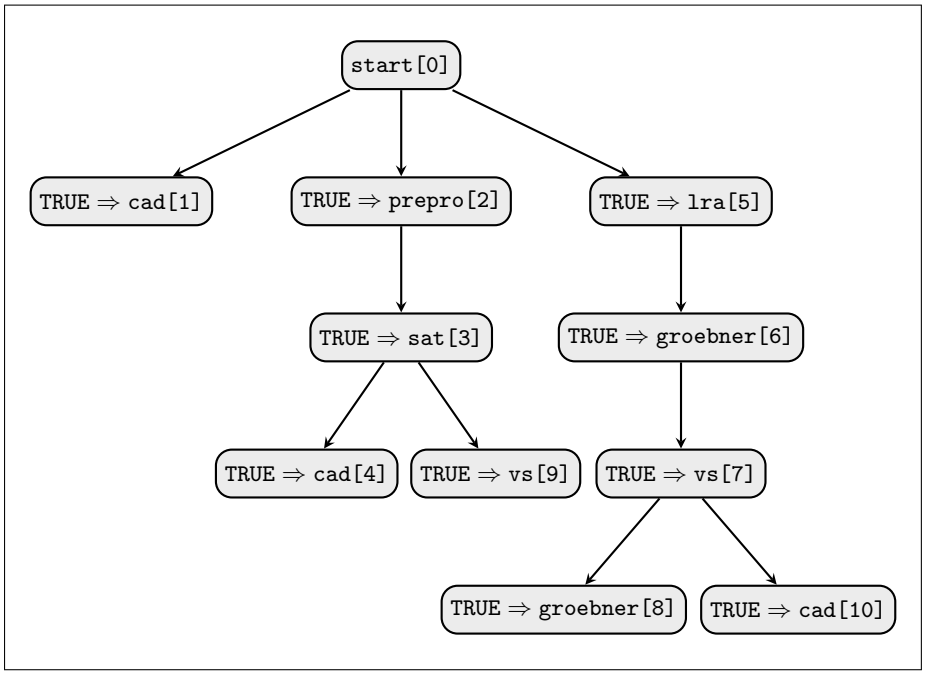



Figure 4.1: A graphic illustration of strategy graph $w_4$ of Example 4.4.4.

be utilized to express relations among nodes. Within the graph structure of a strategy graph, the nodes are module instances. An instance of a strategy graph stores a module instance by its set of attributes. As mentioned, such a set contains the type of the module instance, the condition and the newly introduced priority value. The attributes can easily be seen in the figure, when hiding the auxiliary terminal symbols, which were '⇒', '`[`' and '`]`'. Directed edges between module instances express their related module-backend-relations, the successor being the backend. Thereby a graph is able to maintain a composition of module instances, either a sequential, parallel or combined one.

It should be emphasized, that the strategy graph component applies a different way of invoking module instances than as it was done by the preceding strategy scheme, which has been explained in subsection 3.2.2 of the previous chapter. When an SMT solver utilizes a strategy graph to invoke backends for a module instance, it can invoke those backends, whose related conditions are satisfied by the currently processed NRA formula. Backends, whose conditions are not satisfied, are not invoked and no further actions are applied. For example, it does not check a set of conditions to find one being satisfied by the formula in order to invoke a different backend then, as it was done by the preceding strategy component.

The more module instances are composed in one strategy graph, the more the readability and comprehension of the corresponding derivation is impaired. The reason is not an unsophisticated grammar, but the form of its representation. Whereas sequential strategies can still easily be read consecutively, parallel strategy graphs split into subgraphs, which are scattered along the flat structures of their derivations. A graphic illustration of strategy graphs as presented

in Figure 4.1 is much more receptive. The figure gives a clearly understandable overview of the strategy graph and is self-explanatory. The second task of this thesis, which Chapter 5 deals with, constitutes the creation of a GUI to build customized strategy graphs. Preparing derivations in a graph form as in the figure exposes an illustrative and apprehensible way for the user of this GUI.

## 4.6 Retrieving available backends

The strategy graph examples reveal, that most of the targeted objectives of Section 4.2 have been accomplished already. Module instances, their attributes and their relations can be stored by the usage of the underlying Grammar $\mathcal{SG}$. A function is still required, which an SMT solver can utilize to determine the set of available backends for a given module instance, formula and strategy graph. An implementation for such a function is introduced by Algorithm 1.

An SMT solver retrieves available backends of a module instance by invoking the method `getAvailableBackends(..)`, which requires three parameters. The first parameter, denoted by $G$, is a strategy graph, which needs to be a derivation of Grammar $\mathcal{SG}$. The second parameter is the priority value $p$ of the module instance, whose available backends should be retrieved. Priority values are unique within a strategy graph and this can be exploited when traversing a strategy graph to search for a module instance. As described, the priority value of the *Start* module is always `0`. If $p$ equals `0`, the set of initial module instances of the strategy graph are returned by method `getAvailableBackends(..)`. The last parameter is an NRA formula $\varphi$, which is checked against the conditions of all backends of module instance $p$ to filter only the available ones.

The procedure of the method `getAvailableBackends(..)` traverses the passed strategy graph in order to find the position of the module instance, whose backends should be retrieved. Once the position has been fixed, the backends of the module instance are contained in the succeeding subgraph, which is passed to the auxiliary method `filterBackends(..)`. A set of pairs of filtered backends and corresponding priority values is returned by the method `getAvailableBackends(..)` to the manager. Traversing the graph is achieved by checking the root of the passed strategy graph $G$. If the priority value of the root does not match $p$, it means, that the desired module instance is not in the root, which can be dismissed. The search is then continued by recursively calling the method on maybe several subgraphs, whose roots contain the backends of the just dismissed module instance. While traversing a strategy graph, the appearance of the top-level of its currently checked subgraphs alternates and the method distinguishes four different kinds:

1. The root is the *Start* module, which is denoted by `start[0].`$G_{sub}$, as can be seen in Line 1. If the priority value $p$ equals `0`, the set of available initial module instances of the strategy graph $G$ must be returned with help of the method `filterBackends(..)`. Otherwise the search is continued in the subgraph $G_{sub}$.

2. The input graph is sequentially composed at the top-level, which is described by $C \Rightarrow$ `M[i].`$G_{sub}$ in Line 7. If the priority value $i$ of the first module instance equals $p$, the searched module instance is found and its backends must be returned. If $p$ is greater than $i$, the search is continued

in the subgraph by calling the method `getAvailableBackends(..)` on $G_{sub}$. Otherwise $p$ is less than $i$ and the search can be stopped by returning an empty set at this point. The desired module instance cannot be contained in the remaining subgraph, as its contained priority values must all be greater than $p$, as given by Definition 4.3.1. The algorithm reaches this case, when traversing the different parallel subgraphs of a strategy graph.

3. The input graph consists of a branch containing parallel subgraphs, as it is outlined in Line 15 by $(G_{sub_1}/G_{sub_2})$. The search is independently continued in the subgraphs and their results are joined afterwards as it can be seen in Line 16. In Grammar $\mathcal{SG}$ the binary operator has been defined for this case in order to enable a unique recognition of branches, even if they contain more than two parallel subgraphs. Further parallel subgraphs are then contained in $G_{sub_2}$.

4. Only one module instance remains in the graph. In this case, it does not have any backends and thus an empty set is returned.

As an auxiliary method, `filterBackends(..)` has the task to collect and return all module instances, which are stated at the top-level of a passed substrategy graph $G$, as they are the backends of the module instance, which has been found with method `getAvailableBackends(..)`. The availability of these backends is given in case their conditions are satisfied by the formula $\varphi$. The procedure of the method `filterBackends(..)` is similar as above. It also recursively traverses the passed graph to collect the backends and must therefore as well distinguish different types of the traversed subgraphs, as they are outlined above.

1. The input graph is sequentially composed at the top level and the root is a desired backend, which must be checked for availability, as can be seen in the Lines 1 and 2.

2. The input graph consists of a branch containing parallel subgraphs. The procedure is independently continued in both subgraphs, because the root of each subgraph contains a backend. Afterwards the union of both results is returned. This is described in the Lines 3 and 4.

3. Only one module instance remains in the graph. It must be checked whether $\varphi$ satisfies its condition $C$ or not. If it does, the backend is available and can be returned. This is outlined by the Lines 5 to 7. If not, the empty set is returned.

**Algorithm 1** Algorithm to retrieve the set of all available backends for a given strategy graph $G$, a module instance, denoted by its priority value $p$, and an NRA formula $\varphi$.

```
backends getAvailableBackends(graph G, priority p, formula φ)
begin
    if G = start[0].G_sub then                                          (1)
        if p = 0 then                                                   (2)
            return filterBackends(G_sub, φ);                            (3)
        else  \\ p > 0                                                  (4)
            return getAvailableBackends(G_sub, p, φ);                   (5)
        end if                                                          (6)
    else if G = C ⇒ M[i].G_sub then                                     (7)
        if p = i then                                                   (8)
            return filterBackends(G_sub, φ);                            (9)
        else if p > i then                                              (10)
            return getAvailableBackends(G_sub, p, φ);                   (11)
        else                                                            (12)
            return ∅;                                                   (13)
        end if                                                          (14)
    else if G = (G_sub1 / G_sub2) then                                  (15)
        return getAvailableBackends(G_sub1, p, φ) ∪                     (16)
                                    getAvailableBackends(G_sub2, p, φ);
    else  \\ G = C ⇒ M[i]                                               (17)
        return ∅;                                                       (18)
    end if                                                              (19)
end

backends filterBackends(graph G, formula φ)
begin
    if G = C ⇒ M[i].G_sub then                                          (1)
        return filterBackends(C ⇒ M[i], φ);                             (2)
    else if G = (G_sub1 / G_sub2) then                                  (3)
        return filterBackends(G_sub1, φ) ∪ filterBackends(G_sub2, φ);   (4)
    else  \\ G = C ⇒ M[i]                                               (5)
        if C ⊨ φ then                                                   (6)
            return {M};                                                 (7)
        else                                                            (8)
            return ∅;                                                   (9)
        end if                                                          (10)
    end if                                                              (11)
end
```

# Chapter 5

# SMT-XRAT

The previous chapter dealt with the development of the parallel strategy graph component of SMT-RAT. The creation of a GUI in order to facilitate the building and management of customized strategy graph instances constitutes the second task of this thesis.

Preceding releases of the toolbox already contain a GUI created to build up sequential strategies. Unfortunately this first approach is not very self-explanatory and the usability for novice is rather low. A new GUI pursuing a different approach by introducing an entirely fresh design overcomes this drawback and eases the usage in case the user is inexperienced. Besides improved user-friendliness, it offers the opportunity to build up the aforementioned strategy graphs. The GUI is contained in SMT-RAT, which can be downloaded at [21].

Before presenting the novel GUI implementation and its main features, this chapter starts by motivating reasons for constructing a new GUI and collects important objectives which must be attained by it.

## 5.1 Motivating a new GUI design for building strategy graphs

The user utilizes SMT-RAT to create tailor-made SMT solvers. It must be emphasized that SMT-RAT is mainly a toolbox to construct SMT solvers. Without an underlying strategy graph instance an SMT solver is not complete. Even though releases may contain suitable default strategies, the user is motivated to assemble the integrated SMT-RAT modules to create SMT solvers. As mentioned before, this user-guided creation and customization of SMT solvers is an essential feature, as stated in [9], and thereby plays an important role.

A composition of SMT-RAT modules, i.e. an instance of the strategy graph component, must be present and compiled together with SMT-RAT to obtain the desired SMT solver. In case the user wants to change a composition subsequently, the source code of the corresponding strategy graph instance has to be adjusted. SMT-RAT then needs to be repeatedly compiled. Maintaining strategies as source code keeps SMT solvers compact and more efficient.

The question that arises is how SMT-RAT should enable the user to build and manage strategy graphs. Obviously, it is not an elegant solution to force the

user to grapple with the source code of SMT-RAT. At least, it should not be the only way to handle strategies. An easy and more user-friendly solution would be to let the user author more abstract strategy graphs than given by plain source code. Strategies could be written down as derivations of Grammar $\mathcal{SG}$ of the previous chapter, for instance. A tool could then translate and integrate them into the source code automatically. This rather primitive approach is not sufficient to provide a good user experience. Although this solution diminishes the effort of the user, because it undertakes the source code integration at least, the user still needs time to understand the grammar and derive strategies from it. As could be seen in Example 4.4.4 of the previous chapter, derivations of even small strategy graphs are getting quite long and complex and lack of readability.

Even though SMT-RAT releases may contain suitable default strategies, they might not be suited for all kind of NRA problems. Different scopes of application need different solving approaches. Hence, the user needs to tailor an own strategy for a given problem to exploit the full power of an SMT solver. Workload by the user can therefore not be avoided completely, but the burden should be minimized. A GUI, which represents strategies in a graph style as in Figure 4.1 and which can thereby be operated intuitively, has the potential to achieve this. It would disencumber the user from the requirement to understand a grammar or to study needed parts of the source code. It could thereby allow a quick start for building and managing strategies.

Guidances and objectives, which influenced the design of the following presented GUI, are introduced in the upcoming section.

## 5.2 Guidances and objectives for the design of SMT-XRAT

When creating a GUI, usability should always be considered. Usability suggests that an application should be easy to use and to a high degree self-explanatory, to allow a fast understanding at the side of the user. This concept is important for designing a GUI and should be applied as much as possible. In case the GUI can be operated intuitively and the user is inexperienced, it helps to catch the attention of the user, who can quickly start by trying to create and manage strategies. This is important to keep SMT-RAT attractive among other competitive tools without the preliminary actions of building strategies. Besides the usability, the overlapping concept of user experience should be kept in mind. Whereas the first concept addresses more the pragmatic aspects, the user experience concentrates on the emotions and responses of the user related to the utilized application.

Besides these general advices, concrete aspects related to SMT-RAT have to be considered as well. As the GUI is utilized to create user-defined strategy graphs, it must of course enable the user to input all required data. It must also consider constraints for the inputted data, for example all priority values must be distinct, as stated in Chapter 4. Furthermore the GUI should give the opportunity to build sequential, parallel and combined strategies as it is also proposed in the preceding chapter.

As in the case of the first GUI approach, the new GUI must be a standalone program, which can be used apart of SMT-RAT. This leaves the toolbox applica-

tion with its actual intended purposes. Furthermore the experienced user might as well want to create strategy graphs by changing the source code directly. Then a GUI application is not required.

## 5.3 Overview of SMT-XRAT

The presented GUI is called SMT-XRAT. The GUI possesses an own name to highlight, that it is a standalone program apart of the actual toolbox. The name simply derives from SMT-RAT and adds the letter 'X' to symbolize, that it is a graphical window application, whereas the toolbox is operated in the console.

The following subsections are used to give an overview of SMT-XRAT and to introduce its functionalities. Besides the required features, additional features are stated, which have been implemented to gain a higher degree of usability and user experience. These features help to prevent user frustrations, enable fail-safe working and support the visual creation and manipulation process of strategy graphs.

### 5.3.1 Concept

The underlying concept of SMT-XRAT is the user-guided, visual modeling of module compositions in form of graphs and their mapping onto their corresponding source code for SMT-RAT. A modeled graph expresses an intended strategy graph of the user. Both can easily be projected on each other, because the data structure of a strategy graph also describes a graph structure, as explained in the previous chapter. A mapping considers not only the modeled hierarchy of the SMT-RAT modules, but also their attributes. Furthermore the GUI complies the constraints of these attributes during the modeling process, for example priority values are required to be unique.

The user benefits from that concept, because strategy graphs can be created and manipulated completely independent of SMT-RAT. No knowledge of the inner data structure of strategy graphs and no knowledge about their corresponding source code is required by the user. The GUI does not only support the visual creation of strategy graphs and their translation into source code, but also enables the user to integrate the translated source code into SMT-RAT or, if necessary, delete it subsequently. The conclusive work only involves a recompilation of SMT-RAT with the desired strategy graph instance to obtain a customized SMT solver.

SMT-XRAT has been implemented with the programming language Java. It embeds the freely available Java Universal Network/Graph Framework (JUNG) [17], which fulfills the main demands of the underlying concept, as it allows to model and visualize data, which can be represented as a graph. The JUNG library helped to reduce the basic workload of the GUI creation enormously. Even though, a lot of effort still remained in adjusting its classes to fit for SMT-XRAT.

### 5.3.2 Main window structure

The main window structure of the SMT-XRAT application can be seen in Figure 5.1. It principally consists only of one large pane, which is called *strategy graph*


Figure 5.1: The main window of the SMT-XRAT application in its initial state.

*pane*. This pane embodies the workspace of the user and visualizes the composition of SMT-RAT modules, which are currently modeled. Only a comparatively small area is occupied by a compact menu bar, which offers further necessary or practical functionalities, but which need no visualizations, for instance the exportation of a strategy graph into SMT-RAT.

### 5.3.3 Strategy graph pane

The strategy graph pane is the focus point of the main window and occupies nearly all of its area. It can hereby offer enough workspace to the user to model strategy graphs without distractions. Modeled graphs are acyclic, directed and weakly connected, right as they are needed for Grammar $\mathcal{SG}$. Nodes represent SMT-RAT modules and edges represent the call hierarchy of them. Both of the element types are labeled to display all necessary and editable module attributes within the visualization. Thereby the user is always able to keep a full overview of the modeled strategy graph and its attributes. Moreover, the user can continuously be aware of the presented execution flow of the module composition. Thus, SMT-XRAT nicely supports the user to visualize an imagined strategy while mapping it onto a data structure in the same time.

Modeling strategy graphs on the pane implies the interactive operations of adding, editing and deleting modules and also aligning elements, if desired by the user.

#### Adding backends

An initial visualization of the pane contains the inevitable *Start* module of an SMT solver, which displays no attributes, but marks the starting point for

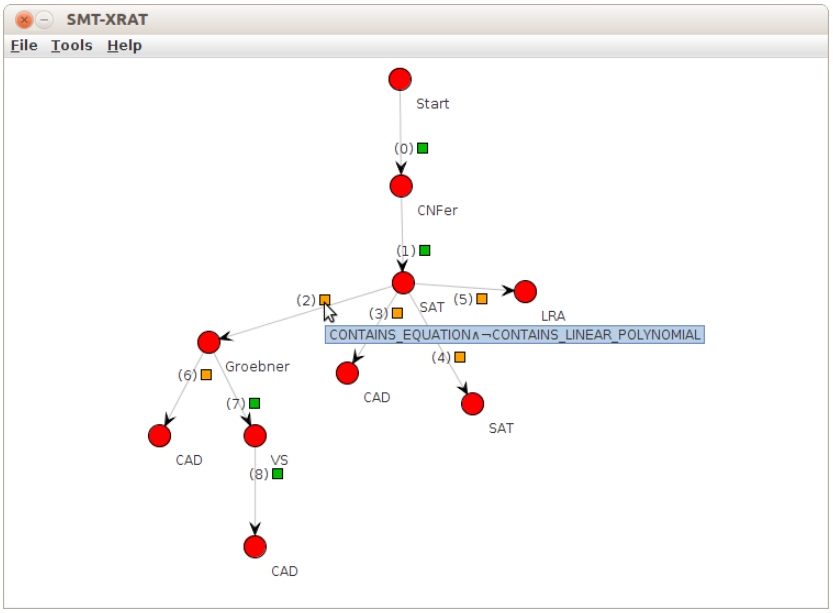



Figure 5.2: The small rectangles alongside the edges reveal the hidden condition of a backend.

the user to create the desired strategy graph. The user can simply consider the *Start* module as the frontend of an SMT solver where NRA problems are passed to. Building up a composition of modules occurs by appending backends to the *Start* module and then to the newly appended backends and so forth. When appending a backend to a selected module, a dialog window requests the operating user to input a condition and to choose the type of SMT-RAT module for the new backend. The GUI provides a special input interface to enter conditions, which is explained later. For each appended backend a new node as well as a directed edge from the originating module to this new node is drawn in the visualization. In this way, the graph gradually arises on the pane. The added graph elements display the inputted data of a backend. A node is labeled with its type of SMT-RAT module whereas an edge holds its condition and an automatically assigned priority value. Initially this priority value is always the total number of currently existing modules decreased by 1, as the *Start* module is not counted. As inputted conditions might get quite long, they cannot be directly seen on the strategy graph pane. Instead, a small rectangle alongside the edge reveals them quickly on request. The user needs to point the mouse cursor over a rectangle to obtain its corresponding tool tip text, which shows the hidden condition, as can be seen in Figure 5.2. This leaves the graph compact and helps to concentrate on the more essential aspect of modeling an execution hierarchy. The user can choose to input an own condition or leave it by the default value of '`TRUE`', which, as described in the last chapter, means that a condition is always satisfied. To point out better which modules contain default conditions and which do not, the color of a rectangle containing a default condition is green and otherwise orange. When looking at the strategy graph

pane, the user is always capable to recognize the execution flow of the modules by following the directed edges in the graph. The colored rectangles signalize at which points of the hierarchy a passed formula must be checked against the condition for an intended backend. This is a quite handy feature, when the user wants to re-enact the solving of an SMT problem.

### Grammar for conditions

When adding a module to the strategy graph pane, the user has to input a valid condition for its intended use as backend. A valid condition is a derivation of the formal Grammar $\mathcal{C}$, which completes Grammar $\mathcal{SG}$ of Definition 4.3.1 of the previous chapter. The grammar is concretely utilized by the GUI. A recursive descent parser [1] has been implemented in the GUI, which applies exactly this grammar.

**Definition 5.3.1** (Grammar $\mathcal{C}$ for conditions)
*Conditions are derived from the formal Grammar*

$$\mathcal{C} = (N, \Sigma, R, S).$$

*The set of nonterminals is given by*

$$N = \{S, T, B, C, D, P\},$$

*whereas the set of terminal symbols*

$$\Sigma = \{\,(\,,\,)\,, \neg, \leftrightarrow, \oplus, \rightarrow, \wedge, \vee\} \cup \{\,\mathit{TRUE}, p_1, \ldots, p_n\}$$

*consists of the union of logical operators and propositions. $S \in N$ is the start symbol of the production rules denoted by the set $R$, which covers the following:*

$$\begin{array}{lcllllllll}
S & \rightarrow & \mathit{TRUE} & | & T & | & C & | & D & | & TBT \\
T & \rightarrow & P & | & \neg T & | & (C) & | & (D) & | & (TBT) \\
B & \rightarrow & \leftrightarrow & | & \oplus & | & \rightarrow & & & & \\
C & \rightarrow & C \wedge C & | & T & & & & & & \\
D & \rightarrow & D \vee D & | & T & & & & & & \\
P & \rightarrow & p_1 & | & \ldots & | & p_n & & & &
\end{array}$$

The nonterminal symbols stay for the following: As said, $S$ for the start symbol of the production rules, $T$ for term, $B$ for binary operator, $C$ for conjunction, $D$ for disjunction and $P$ for proposition.

The terminals '$\neg$', '$\leftrightarrow$', '$\oplus$', '$\rightarrow$', '$\wedge$' and '$\vee$' represent their related logical operators, which, in the context of conditions, are negation, equivalence, exclusive or, implication, conjunction and disjunction respectively. Their semantics is defined as usual. The terminal symbols '`(`' and '`)`' are used, in case several different types of logical operators are utilized within one term. They point out the precedences of the operators in the same way as it is known from mathematical contexts. For example, for the term $p_1 \vee p_2 \wedge p_3$ it is unknown, which of the logical operators has the higher precedence. Writing the same term with parenthesis as $p_1 \vee (p_2 \wedge p_3)$ clarifies, that the conjunction operator is of higher precedence.

The propositions $P = \{p_1, \ldots, p_n\}$ are not concretely mentioned in the grammar, as they have already exemplary been listed in Chapter 3 and numerous

are available. Furthermore, the set of available propositions can vary among releases of the toolbox, as well as the user can also define own propositions. For this reason, the set of propositions is dynamically loaded from the SMT-RAT source code each time the GUI is started. As mentioned in the previous chapter, the set of SMT-RAT modules can vary as well. Therefore the list of available SMT-RAT modules is also dynamically loaded. This allows to edit SMT-RAT without editing the GUI additionally.

The grammar is constructed in such a manner, that the user working in the context of SMT solving is naturally enabled to input conditions effortlessly, although they must be derivable from the grammar. Therefore it can be assumed that no additional workload arises.

#### Improved interface for inputting conditions

When adding backends to existing modules on the strategy graph pane, a dialog window requests the user to input a desired condition, which must be derivable from the above defined Grammar $\mathcal{C}$. This dialog window is equipped with additional features to ease the input process for the user and to improve the usability. A specialized text area is used for inputting conditions. Initially, it contains the default proposition value '`TRUE`'. The window also contains a combo box, where the user has the possibility to choose a proposition value from. A chosen proposition value can then be copied to the current caret position of the text area. Should the occasion arise that the user selects a part of an entered condition beforehand, it is simply overwritten by the chosen proposition value. The user can only input proposition values by using this combo box. Proposition values cannot be typed into the text area directly. On the one side, this simply prevents mistyping and, on the other side, the list of propositions might be changed between releases of SMT-RAT, as stated above. The combo box informs the user about all currently available propositions of the toolbox. In many cases it will not be sufficient to use conditions, which contain just one single proposition. When requiring a Boolean combination of conditions, the above stated logical operators are needed. Although, the characters of the operators are generally not present on a keyboard, they can just be typed into the specialized text area of the dialog window. To input the conjunction operator '$\wedge$', for instance, the user simply needs to hit the key 'c' on the keyboard. Instead of the character 'c', the character '$\wedge$' will then appear in the text area. The mapping between the symbols, which express the given logical operators, and characters, which are present on general keyboards, supports the readability and comprehension of inputted terms and therefore achieves a better user experience.

In order to increase the user experience even further, the text area treats the single characters of an inputted proposition value as a block, which cannot be entered by the caret of the text area. This means, that if the caret is positioned directly left of an inputted proposition value and the user navigates the caret to the right, it will jump to the position directly right of the proposition. The caret will never appear between the characters of a single proposition value. This is the analogous case for selecting and deleting proposition values. All characters of a proposition value are always selected, deselected or deleted at once. This allows a faster and more comfortable navigation and editing.

The text area allows to copy and paste conditions or parts of it. Text, which should be pasted into the text area, is checked to guarantee, that it only contains

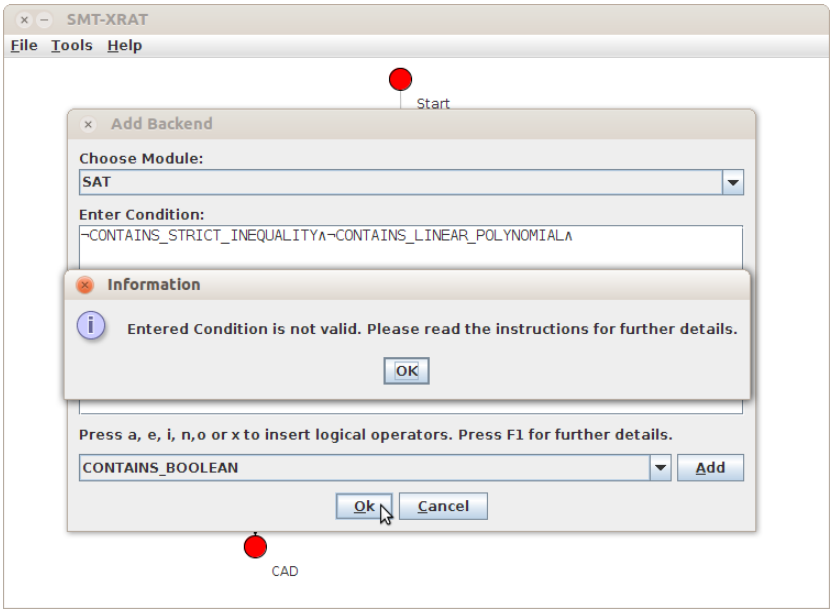



Figure 5.3: A wrong condition has been inputted by the user.

allowed values. Otherwise it will be refused. Allowed values cover proposition values, the characters used to express logical operators and parenthesis.

When the user confirms the dialog window, the implemented recursive descent parser of SMT-XRAT checks, whether the inputted condition is a valid derivation of Grammar $\mathcal{C}$ or not. In case it is not, the user will be returned to the dialog window to re-edit the condition, as can be seen in Figure 5.3. Otherwise the inputted condition is adopted for the backend. This fail-safe method of inputting conditions saves frustrations on the side of the user. A mistake is better pointed out at this stage than at the time of exporting the strategy graph or even when compiling the resulting SMT solver. The user is promptly informed about an error and is directly enabled to correct it.

### Manipulating the strategy graph

Besides the capability of adding modules, the strategy graph pane gives the user also the possibility to remove and edit them subsequently.

The deletion of a single module implicates that all of its succeeding modules in the composition hierarchy will be removed as well. The strategy graph pane is only allowed to contain one single weakly connected graph. Furthermore, when deleting one or implicitly more modules, the priority values of all remaining modules might automatically be adjusted to comply the constraints of the priority values. However, the logical priority order remains untouched.

When editing modules, the same dialog window is displayed as for adding modules. The window components are already filled in with the attributes of the corresponding module. However, priority values are not manipulated via this dialog window. As said before, priority values are automatically assigned, when a module is created, and they are displayed alongside the edges. The

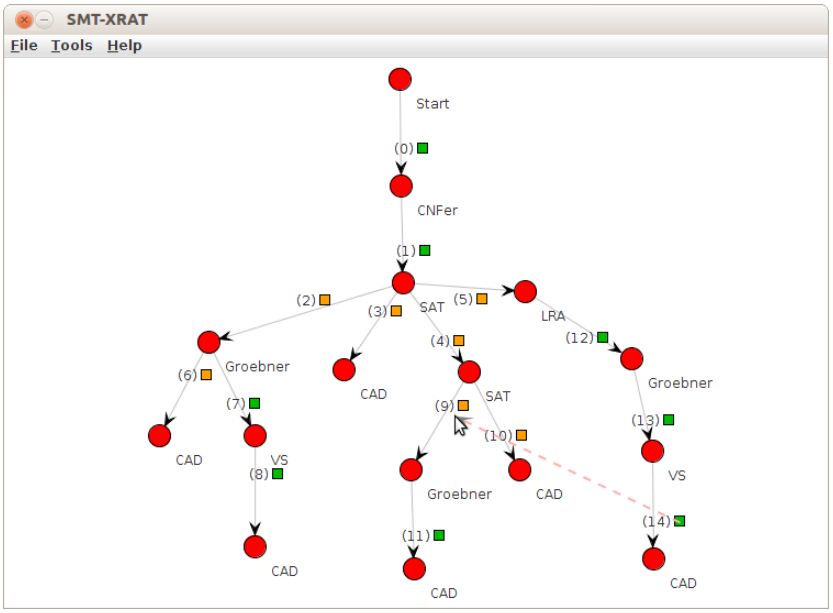



Figure 5.4: Priority values before changes are set.

user can manually change the priority order by pushing the priority value of a lower prioritized module in front of the priority value of a higher prioritized one. The user achieves this by using the mouse pointer to draw a dashed arrow from the edge label of that lower prioritized module to the edge label of the higher prioritized module, as it is illustrated by Figure 5.4. Afterwards the lower prioritized module will have a higher priority than the other one. The priority values of the modules might just be swapped. If this is not possible, the priority values of the modules and of their preceding modules are adjusted automatically, so that as a result, the newly prioritized module will be ordered logically before the other one. The adaptation of the priority values is emphasized by Figure 5.5. This mechanism has been implemented not only to offer a fast way of subsequently changing priority values, but also to picture the actions of the user even more.

### Moving graph elements

As mentioned before, the user can utilize the mouse to position graph elements freely on the strategy graph pane. Either a single element or a whole group of elements can be moved at once. This feature allows to arrange graph elements after the imagination of the user. For instance, a tree graph can be modeled, where the root node is in the middle of the top and the leaves are distributed at the bottom, as it is done in the graphic illustration of Figure 4.1 of the last chapter. The user could also move the *Start* module to the center of the pane and position its subgraphs into the corners.

Moving graph elements makes the process of the strategy building more tangibly. Additionally, it is playful and invites a novice user to dwell on the strategy graph creation and keeps the attention.

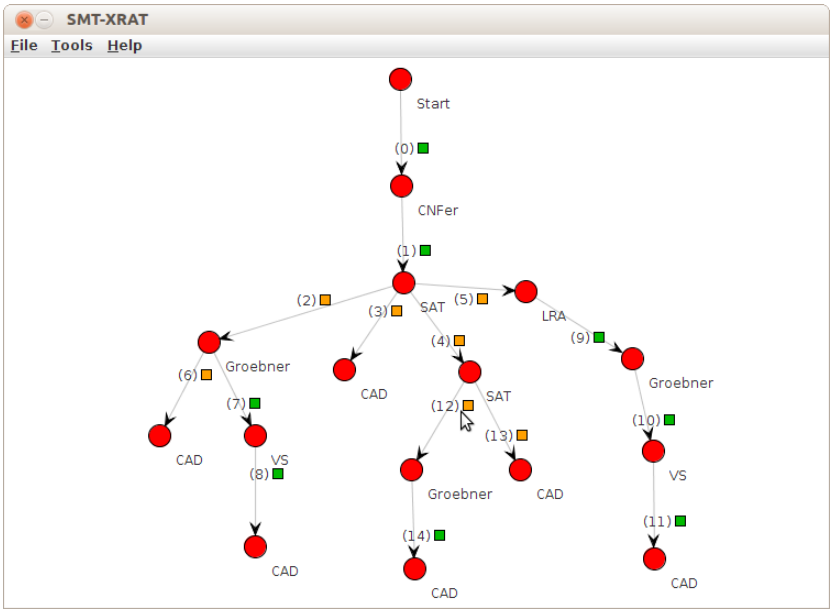



Figure 5.5: Intentionally changed and automatically adapted priority values.

### 5.3.4 Further functionalities

Further features of the GUI are reached through the menu bar. These features need no visualizations and can be performed via simple dialog windows.

The most important and also necessary functionality is the management of strategy graphs inside the SMT-RAT source code. It is not sufficient to design a strategy graph on the pane. Its underlying data structure also needs to be translated into source code, which must then be integrated into SMT-RAT. To export a currently modeled strategy graph, the user simply needs to open the corresponding dialog window and choose a name, Figure 5.6 shows an example for exporting the current strategy graph and naming it *SMT_XRAT*. The GUI will then fulfill the translation and integration process. The same dialog window also lists all existing strategy graphs, which are currently integrated in the source code, and gives the opportunity to delete them separately. This can be seen for the existing strategy graph *NRATSolver* of the example.

The remaining features hold by the menu bar are not mandatory, but improve the creation process and usability. For example, the GUI allows the user to save the current strategy graph into an XML file. This file can then be opened again for later editing or it can be exchanged with another user. Another practical feature is the ability to save a screen shot of the strategy graph pane into an image file. Such image files can be used to discuss strategy graphs, when it is not desired to run the GUI. For this purpose, it could be attached into an email or included into a presentation, for instance.

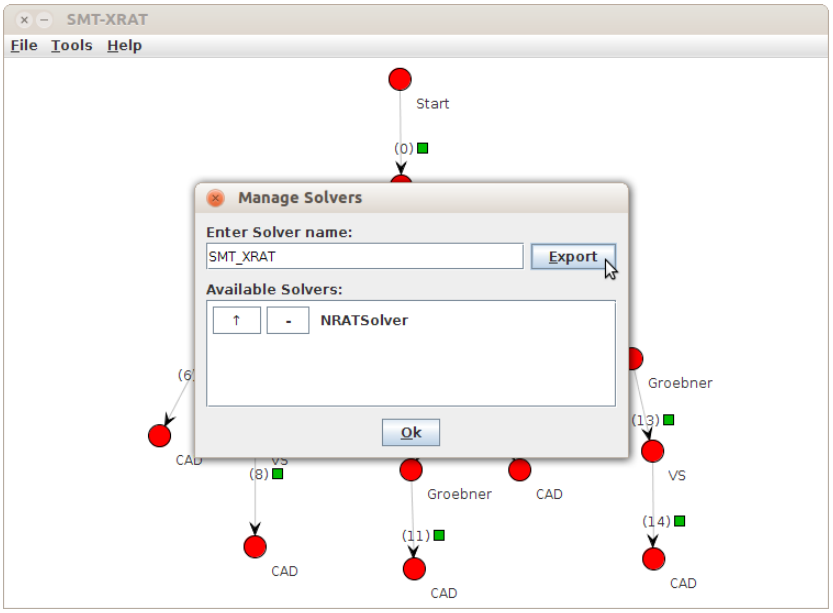



Figure 5.6: Managing SMT solvers in the SMT-RAT source code.

# Chapter 6

# Parallelization of SMT-RAT

The thesis brought forward the new strategy graph component allowing the possibility to store parallel strategies. Furthermore a GUI, which allows the user to create and manage such strategies intuitively has been presented. The fulfillment of the third and last task of this thesis is supposed to complete this work. The task covers the adaptation of SMT-RAT as, at this point, it is only capable of utilizing one single CPU core, which means, that it even processes parallel strategies sequentially. A final step must enable the utilization of multiple CPU cores to process composed SMT-RAT modules concurrently.

When implementing concurrent applications, different kinds of approaches are required which also entail new types of problems. This chapter therefore starts by introducing some specific aspects of parallel programming, compiled from [25], which have been considered before adapting SMT-RAT. Afterwards, the implementation of the parallelization approach is examined.

## 6.1 Aspects of parallel programming

Modern computer architectures offer operating systems the ability of multiprocessing. This means, that multiple CPU cores are available to execute several processes concurrently. One core can serve only one process at once, hence, applications can benefit from multiprocessing in case several are executed simultaneously. Application executions are thereby accelerated, because more total CPU time is available for each process. A lonely executed application itself cannot directly take advantage from multiprocessing, when it is running sequentially. Its program parts must be distributed among available CPU cores. Mostly, it is not worthwhile to split applications into several processes. Maintaining applications, which consist of several processes is complex and interprocess communication is required and that can become very costly. Instead, it is better to apply the concept of multithreading and allow a process the control of a set of threads, which overtake parts of the program procedure. They can then be distributed among available CPU cores in order to fulfill tasks concurrently. As threads share the same process memory, its management is less complex and its communication is less expensive. By introducing this concept, a solely running process can as well benefit from multiprocessing. Furthermore, it can also take advantage of unoccupied CPU cores, even if several applications are

executed simultaneously.

In case an application wants to exploit the concept of multithreading, its algorithms need to be adapted to allow the distribution of problems among threads. Several concerns must be regarded, which are presented in the following.

**Parallelization approaches** Multithreading is generally achieved by dividing a more complex problem into several less complex subproblems, which can then be concurrently processed by threads. After completion their results are joined back together to the solution of the original problem. A different approach is simply achieved by distributing independent tasks on different threads. Both approaches can also be intermixed, in case problems of the independent tasks can as well be split into several subproblems.

**Scalability** When applying a parallelization approach to realize multithreading, a software engineer should implement scalable algorithms, which means, that no fixed number of CPU cores should be assumed at any time. This allows applications to run flexibly on differently equipped computers. The reason is, that it is desirable to utilize all cores in order to exploit a maximum of resources, especially for CPU-intensive applications. If less cores are assumed, less threads might be intended than possible and thereby valuable hardware resources are dismissed. Overhead can then again arise, if more CPU cores are assumed than can actually be utilized simultaneously, as explained in the following.

**Overhead & oversubscription** If more threads are running than CPU cores are available a so called *oversubscription* is present. One core can always issue CPU time to exactly one thread only. An oversubscription forces the process scheduler of the underlying operating system to switch the processing of threads, which causes additional overhead through *context switches*. A context switch of a thread involves saving and loading its state costing process time as well. Overhead can arise the other way around as well. It is not always worthwhile to split a given problem into as many pieces as number of CPU cores are given. The creation and management of threads also causes overhead and if subproblems are getting too small, this overhead might surmount the intended efficiency gain.

**Mutual exclusions** Considerations have to be made about mutual exclusions of data accesses, because threads of the same process are sharing the same memory. It must be prevented, that two or more threads try to manipulate the same data concurrently or *race conditions* might be caused. A race condition is given, when one thread changes data while other threads are still working with then outdated copies of it. Race conditions induce inconsistent data states, which can cause unpredictable program behaviors. Shared resource accesses of threads must therefore be coordinated, for example by introducing locking mechanisms, which are shortly presented in the next section.

**Deadlocks** Ensuring mutual exclusions for accesses on shared resources can cause *deadlock situations*, which must be inhibited. Deadlock situations arise when two or more threads are mutually waiting for each other. This can for example occur, when a first thread is waiting for the release of a resource locked

by another thread, which is already waiting for the release of further resource locked by the first thread. Multithreading approaches harbor the risk of falling into deadlock situations, because they are generally locking accesses to shared memory. Deadlocks do not occur each time when a threaded algorithm is executed, because thread scheduling is arbitrary. Unfortunately the detection of algorithm parts, which might cause deadlock situations, is thus not always an obvious approach.

## 6.2 Applying parallelization in SMT-RAT

As the core of an SMT solver, the manager is able to survey the interaction among all components and is therefore enhanced for the parallelization process of SMT-RAT. The manager requires the application of a parallelization approach under consideration of the above described aspects in order to function efficiently.

### 6.2.1 Parallelization approach

The process of SMT solving must be partitioned into several independent components or into subproblems in order to enable parallelism. As stated throughout this thesis, the target of the parallelization of SMT-RAT is to allow parallel executions of SMT-RAT modules. Thus it is meaningful to treat single modules of a strategy as independent components, whose executions can be running in their own threads. These threads can then simply be distributed among available CPU cores. This approach can be applied, because module instances can work independently on their own received formula. In case a module instance invokes backends on a formula, these backends can calculate a result in their own threads and pass it back to the thread of the invoking module instance. In case one of the threads calculated a *relevant result*, this means different from `unknown`, the overall process can be accelerated by interrupting all corresponding threads, which still have ongoing calculations. Note, that the utilized approach has a disadvantage, because it cannot influence the processing costs. There are chances where the processing costs of the module instances are too small to justify the multithreading induced overhead.

### 6.2.2 Achieving scalability

When utilizing the described parallelization approach, it is deliberately not possible to execute the exact same number of threads as CPU cores are available. The number of backends, which can be invoked by one module instance, is dynamic and depends on the underlying strategy and can furthermore vary through the properties of the different passed formulas. Therefore the overall number of module instances, which are intended to be executed concurrently, is also dynamic and exposed to even stronger variations. It is only possible to provide a maximal possible number of concurrently running module instances. Due to this reason, there might be less threads running than cores could actually be utilized. A possibility to counter this problem and gain full usage of the dismissed resources is proposed in the conclusion at the end of this thesis. The utilized parallelization approach also harbors the risk, that more module

instances are running concurrently than cores are available. This oversubscription must be prevented, as it causes additional overhead. Therefore the only scalability aspect, which must be achieved is to set the number of given CPU cores as an upper bound for the number of concurrently running threads to inhibit oversubscriptions. The full mechanism is explained later in this chapter.

### 6.2.3 Overhead reduction

Executing applications entails overhead, which cannot be prevented completely. Unfortunately, the concept of multithreading increases this overhead further due to the creation, deletion and management of threads as well as their required context switches. It is subject of the manager to lower the overhead caused by multithreading, which is achieved by applying the following introduced aspects.

**Dynamic multithreading utilization** The utilization of multithreading is only meaningful, if the provided computer architecture contains multiple CPU cores. Furthermore, it is not required and cannot even be utilized for the underlying parallelization approach, if the composed strategy does not intend any parallel backend executions. These conditions can dynamically be checked for each SMT solver run and in case not both of them hold, the manager performs the SMT solving completely without multithreading.

**Subsequent use of threads** A combined strategy graph should utilize multithreading, but also contains sequential call hierarchies, which can be processed without it. For cases where a strategy defines only one backend for a module instance, it is not necessary for this backend to be executed in its own thread. Instead, the manager utilizes the same thread for the backend as for the invoking module instance. This is done in the same way as without the adaptation of the manager. The SMT solver transfers this behavior to those cases where indeed several backends are assigned to a module instance, but only a single one is currently available. Therefore, independent of the number of available backends, exactly one backend is always executed in the same thread as of the invoking module instance. This spares context switches and decreases the number of thread creations.

**Interruption of backends** The chosen parallelization approach specifies, that concurrently running backends, which process the same passed formula, can be interrupted once one of them has found a relevant result. This concept does not lower the overhead induced by the usage of threads, but increases the overall efficiency of an SMT solver. Canceling module instances enables other module instances to utilize the freed resources earlier. A thread, which processes a module instance is not simply deleted or canceled on interruption. It is necessary to ensure, that a currently running calculation is terminated in a *safe* fashion, such that a module instance is not left in an inconsistent state. How this is realized is described in Section 6.3. When a thread is not deleted, the opportunity is given to reuse it, what is examined in the following.

**Further reuse of threads** As an SMT solver, which is built with SMT-RAT, works incrementally on NRA formulas, a high number of backend invocations

can be assumed. Instead of creating and deleting threads each time a module instance should be executed, the manager reuses threads once they have been initially created. This prevents the creation and deletion of a considerable high number of threads. For this purpose, the manager takes care about which module instance can be executed by which thread. It thereby prevents module instances in competing for the same thread and inhibits possible deadlock situations as described later in this section. After a thread has been utilized by a module instance, it is locked until it is required again, as this consumes less overhead than a full thread creation and deletion. The approach of reusing threads is explained in detail in Section 6.3.

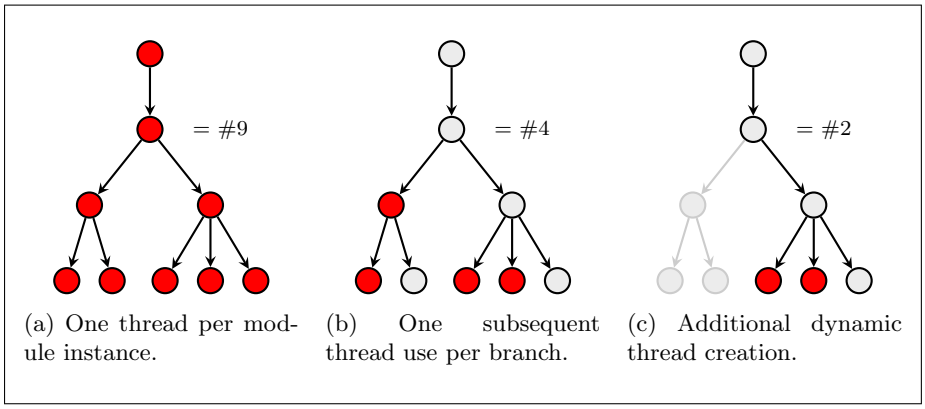
**Dynamic thread creation** The preceding explained approach for creating required threads implies: An SMT solver can start to process a strategy graph with its main thread. When descending in the strategy graph, for each module instance as many threads as the number of its available backends minus 1 are additionally required. This allows to calculate a maximal number of required threads for a strategy graph. There are actually two further ways, which are worthy of remark and stated by Corollary 6.2.1. For a further efficiency gain not all threads should initially be created, but the solving process should instead start with one thread only. Further required threads are then added dynamically. The reason is, that there might be cases where a module instance never needs to utilize any backends or that a formula never satisfies the conditions of certain module instances. This keeps the number of created threads minimal.

Figure 6.1 shows how the number of additionally required thread creations decreases when applying the above mentioned thread savings. Nodes which are colored in red, represent module instances which need to run in an own thread. Module instances displayed by gray nodes are executed in the same threads as their preceding module instance. In Figure 6.1a all module instances need their own thread to run in. This adds 9 additional threads to the preexisting main thread. Without reusing initially created threads, 100 complete passes would require 900 thread creations and deletions. This is an enormous multithreading induced overhead. When reusing threads, the number of 9 threads persists independently of the number of passes. When threads are furthermore subsequently used again for invoked backends, the number decreases to 4 threads, as can be seen in Figure 6.1b. It even decreases further to 2 threads, when for instance the left subgraph is never invoked and threads are dynamically created, as can be seen in Figure 6.1c.

**Corollary 6.2.1** (Number of maximal required threads)
*The number of maximal required threads for a strategy graph can also be calculated by:*

- *the maximal number of intended parallel running module instances, or*
- *the number of leaves (when considering a strategy graph as a tree).*

**Prevention of oversubscriptions** Although the number of created threads is always minimal, it might still surmount the number of available CPU cores. The manager prevents a potential oversubscription by assigning only one thread




(a) One thread per module instance.

(b) One subsequent thread use per branch.

(c) Additional dynamic thread creation.

Figure 6.1: Number of additional required thread creations.

to each CPU core. In case the given strategy prescribes that more module instances should be running concurrently, further executions are blocked by a priority locking mechanism until one CPU core becomes available, as it is described later. The priority values of module instances are utilized to decide on an execution order for all currently module instances intended to run. Even though the number of running threads cannot exceed the number of available CPU cores, it might still be the case that the overall number of existing threads is higher. Note, that this is required to prevent the deletion and later recreation of threads as stated above.

### 6.2.4 Locking mechanisms

For the application of the parallelization approach locking mechanisms are required. On the one hand, they should protect the access on shared resources and, on the other hand, they are used to prevent oversubscriptions. Both kinds of the utilized locking mechanisms are presented in the following.

**General locks** When applying multithreading, accesses on shared variables must be controlled to ensure mutual exclusions. This can be achieved by several methods, but implementations firstly rely on those provided by the programming language. Therefore, when discussing the algorithm of the full procedure of a parallel working SMT solver in Section 6.4, *locks* are simply utilized to give an idea how and where mutual exclusions take place. The mechanism of locks is as follows: The same lock can only be owned by one thread at a time. When a thread wants to access shared resources, it simply tries to acquire its ownership. Once the lock is owned by the thread, it can access the shared resources and then release it afterwards.

**Locking mechanism to prevent oversubscriptions** An own kind of locking mechanism has been developed for the implementation of the parallelization approach to prevent potential oversubscriptions. This locking mechanism is utilized to control the number of simultaneously running threads. In case a maximum number is reached, accesses on the CPU cores are locked for further

threads of the SMT solver. Locked threads are not released before a currently running thread has finished its check. The whole implementation is described in detail in Section 6.3.

### 6.2.5 Deadlock prevention

The introduced parallelization approach of distributing executions of module instances on different threads has been applied under regards of possible deadlock conditions. This means, that situations where threads could be mutually waiting on each other have been prevented. The prevention of two deadlock situations, which are worthy of mention and which are nature of the applied parallelization approach are examined in the following.

**Assignments to available threads** As mentioned, an SMT solver reduces multithreading overhead by reusing threads once they have been initialized. When a module instance should be processed by an already existing thread, a deadlock situation might arise when choosing a wrong thread. The simplest case is given, when a thread pushes the task of executing a module instance to itself. The thread is then waiting for a result, which will never be delivered. In another case, two running threads could be involved. If the first thread sends its task to the second one, and this thread also sends a task to the first one, both end up waiting for each other. Constellations with up to all existing threads are thinkable. For this reason an SMT solver does not only keep track of all existing threads, but must also maintain their availability to process a module instance to prevent this deadlock situation. The full approach is explained in the next section.

**Coordination of thread locking** The preliminary described locking mechanism to prevent oversubscriptions also harbors the risk of running into deadlock conditions, because it implements a mutual exclusion on shared CPU cores. Locking and releasing threads must be coordinated. The reason is, that if the lock of just a single thread is never released, a deadlock arises, because the result of its processed backend can never be returned to the invoking module instance, which is thereby also locked. Thus, a chain of waiting threads is built up to the *Start* module, which is then left waiting forever. In order to prevent such cases, the parallelization approach memorizes all locked threads and releases one of them each time another thread has finished its processing.

## 6.3 Implementing the parallelization approach

This sections introduces the necessary implementations for the parallelization approach of SMT-RAT, whereas the complete algorithmic overview of the manager is presented in Section 6.4.

### 6.3.1 Thread management for module instances

It is the task of the manager to coordinate the execution order of module instances and to maintain a set of threads to allow concurrency. Furthermore, it

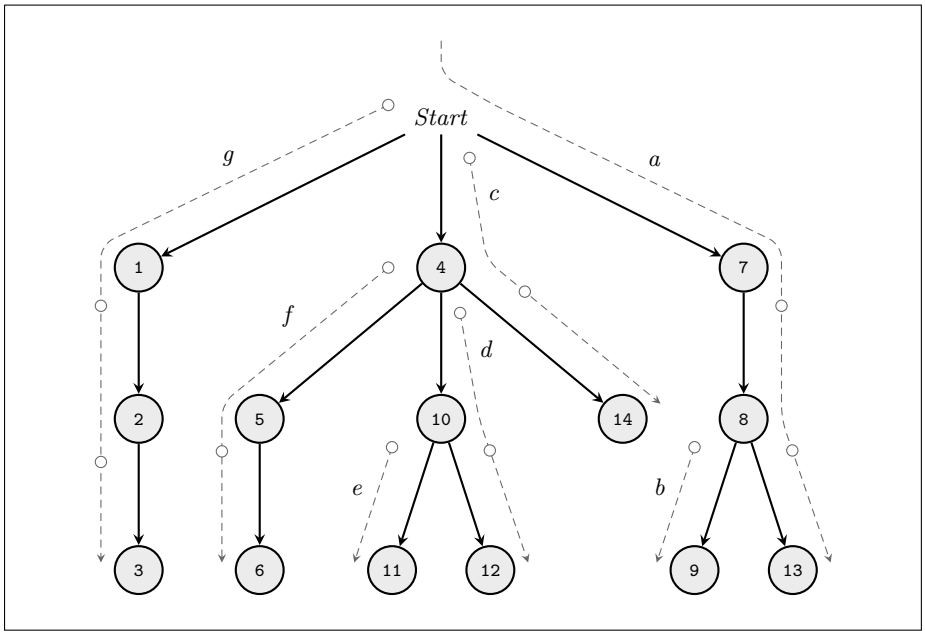



Figure 6.2: Thread management for module instances of a fully available strategy graph, i.e. all backends are always available.

decides which module instance should be processed by which thread. Examining the way how an SMT solver designates module instances for certain threads also provides explanations on how the parallelization approach for SMT-RAT has been implemented. It also states how this implementation already applies some of the above listed aspects for this approach.

**Preliminary illustration** Figure 6.2 shows a graphic illustration of a strategy graph, which helps to present how an SMT solver manages its threads for its execution of module instances. This illustration is simplified and shows neither conditions nor types of module instances, as for this purpose only the priority values are of interest. They are listed alongside the edges of the graph and simultaneously name their corresponding module instances. The thin, dashed arrows next to these edges represent threads, which could be utilized by an SMT solver. In order to be able to distinguish threads, a small letter is assigned to each, which can be read along their related arrows. The graph contains the seven threads $a$ to $g$ and this is also the number of leaves in the graph. As described in the previous section, this number equals the maximal number of required threads for a full processing of the strategy graph and this means that all potentially required threads are depicted in the figure. The beginning of a thread arrow is closely situated to the module instance, which is the driver for the creation of the corresponding thread. In case such a module instance invokes its backends, it is the account of this thread to process exactly one of the backends and return its result to this module instance. In case further backends have been invoked, the remaining threads, whose arrows are also closely placed to the module instance, must process one backend each. For an exam-

ple, module instance `4` is the driver to initiate the creation of thread *d* and *f*. Thread *d* processes module instance `10`, whereas thread *f* takes care of module instance `5`. The main thread of the SMT solver is presented by the thread arrow starting above of the *Start* module instance. All module instances which might potentially be processed by the same thread are presented alongside its related arrow. Thread *g* for example, is created through the *Start* module in order to process module instance `1`. While processing backend `1`, thread *g* might also be required to process module instances `2` and `3` sequentially.

**Decision over utilized threads** The graphic illustration is presented in such a manner, that backends of a module instance are ordered by their priorities, what can be expected without loss of generality. This means, that the backend with the highest priority and therefore the lowest priority value, is placed at the outer left. The lower the priority of a backend the farer right it is placed. The method `runBackends(..)` of the manager uses the same sequence to process the backends of an invoking module instance. Beginning with the highest priority, it invokes each backend on its own thread. For the rightmost backend with the lowest priority the same thread of the invoking module instance will be subsequently used again. Note, that in case only a single backend is available or specified by the strategy, the same thread will as well be subsequently used. In case no backends are specified at all, the thread of the invoking module instance will return to its preceding calculation. This thread management approach can easily be comprehended, when pursuing the different thread arrows of Figure 6.2. Utilizing it guarantees, that threads of higher prioritized backends are invoked earlier and that for each set of backends one thread creation is spared through reuse. The thread, which is reused, must of course initiate all required thread calls before it can start to process its own backend. As the processing of the own backend will then always be started at last, it explains why the backend with the lowest priority is chosen to be that certain backend.

**Ensuring assignments only to available threads** Figure 6.2 presents the underlying thread management for module instances for the case, that all conditions are satisfied in the strategy, whereas the modified Figure 6.3 shows an example where not all conditions are satisfied. In this case, the remaining module instances are not all processed by the same threads as in the first case. On closer observation, however, it can be seen, that this involves only those module instances, which became the lowest prioritized backend of their preceding module instance. Instead they are now processed by the thread of this invoking module instance. The mapping between threads and module instances is therefore always constant except for that particular case. An SMT solver is thereby always enabled to subsequently use one thread for each branch again. When assigning a unique identification number to each thread, all module instances can be mapped onto these numbers and the SMT solver can always be aware about which thread is designated for which set of module instances. In this way, it is ensured, that only available threads are utilized to process invoked module instances and can prevent the previously outlined deadlock condition.

**Retrieving thread identification numbers** An SMT solver implements the method `getThreadId(..)` in Algorithm 2 to retrieve the thread identification

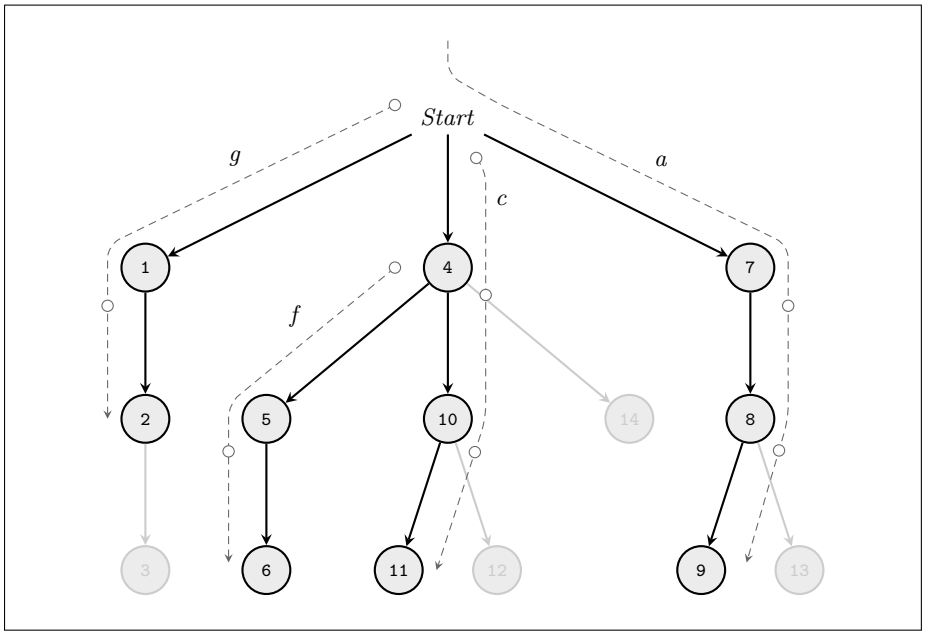



Figure 6.3: Thread management for module instances of a partly available strategy graph.

numbers for its utilized module instances. The procedure of this algorithm is to perform a depth-first search for a module instance in the strategy graph while calculating the thread identification number for it on the fly. The initial thread identification number is 0, as Lines 5 and 11 determine it. At each point where the strategy graph is branching, further threads are required and for each further required thread the number is increased by 1. For this purpose the depth-first search works from right to left and the increase applies for each step to the left, as can be seen in Lines 16 to 20, whereas a further step to the left is only required if the module instance has not been found in the previous branch, what is denoted by a return value of `-1`, as it is outlined in Lines 13 and 17 to 19. This algorithm traverses a derivation of the Grammar $\mathcal{SG}$ and needs to distinguish different top-levels of a strategy graph and its subgraphs as it was done by method `getAvailableBackends(..)` of Algorithm 1, which can be seen in Lines 1, 3, 9 and 15. The method `getThreadId(..)` requires two parameters, a strategy graph $G$ which is traversed in order to search for a module instance passed as its priority value $p$. The way how a run-through of a graph is applied, can also easily be reconstructed when following the thread arrows in Figure 6.2 in their alphabetical order and considering the thread identification numbers for thread $a$ as 0, for $b$ as 1, and so on, until the search module instance is found.

**Priority locking mechanism** In case oversubscriptions must be prevented, the above mentioned priority locking mechanism is used. In the Figures 6.2 and 6.3, small circles can be seen along the thread arrows, which reveal the point of times when it is required to check, whether a priority locking must be

**Algorithm 2** Algorithm to retrieve the thread identification number for a given strategy graph $G$ and a module instance, denoted by its priority value $p$.

```
int getThreadId(graph G, priority p)
begin
    if G = start[0].G_sub then                                          (1)
        return getThreadId(G_sub, p);                                   (2)
    else if G = C ⇒ M[i].G_sub then                                     (3)
        if p = i then                                                   (4)
            return 0;                                                   (5)
        else                                                            (6)
            return getThreadId(G_sub, p);                               (7)
        end if                                                          (8)
    else if G = C ⇒ M[i] then                                           (9)
        if p = i then                                                   (10)
            return 0;                                                   (11)
        else                                                            (12)
            return -1;                                                  (13)
        end if                                                          (14)
    else \\ G = (G_sub1 / G_sub2)                                       (15)
        int id := getThreadId(G_sub2, p)                                (16)
        if id = -1 then                                                 (17)
            id := 1 + getThreadId(G_sub1, p);                           (18)
        end if                                                          (19)
        return id;                                                      (20)
    end if                                                              (21)
end
```

applied. As can be seen, checks are always required just before the processing of a module instance should begin. For this purpose, Algorithm 3 provides the method `checkPriorityLocking(..)` for threads intending to process a module instance. As the manager utilizes threads and its own thread in order to process the backends of a module instance, the method holds two different program flows, whereas the first treats the case for different threads and the second for the own thread, as can be seen in the Lines 2 and 9. The method is explained in the following with help of the method `numberOfCores()`.

`numberOfCores()`: This method must be supplied by the programming language in order to determine the number of available CPU cores of the underlying computer architecture.

**Priority locking for a different thread** A thread $t$, which is intended to process a currently invoked module instance $I_M$, is initially in a locked state and waits to be unlocked to process the method `check(..)` of $I_M$. In case priority lockings are applied, it must be checked, whether the thread is allowed to be unlocked or must be pushed in a queue for being unlocked later. This check consists of testing, whether a maximum number of running threads has already been reached or not, which means, that the algorithm compares the global variable `runningThreads`, initialized by the manager, against `numberOfCores()`, as can be seen in Line 3.

**Algorithm 3** Algorithm to apply priority lockings if required. Invocations for different threads or the same thread are distinguished. Threads are denoted by their thread id *tid* and their priority value passed by *p*.

```
void checkPriorityLocking(thread id tid, priority p, bool ownThread)
begin
    t := retrieve thread in T with thread id tid;                          (1)
    if ownThread = False then                                              (2)
        if runningThreads < numberOfCores() then                           (3)
            runningThreads := runningThreads + 1;                          (4)
            t.unlock();        \\ to run I_M.check(..)                     (5)
        else                                                               (6)
            PQ.push((p, tid));                                             (7)
        end if                                                             (8)
    else                                                                   (9)
        if runningThreads = numberOfCores() then                           (10)
            if PQ has element of higher priority than p then               (11)
                PQ.push((p, tid));                                         (12)
                (p', tid') := PQ.pop();                                    (13)
                t' := retrieve thread in T with thread id tid';            (14)
                t'.unlock();       \\ to run I'_M.check(..)                (15)
                t.lock();                                                  (16)
            end if                                                         (17)
        end if                                                             (18)
    end if                                                                 (19)
end
```

If the maximum is not reached, a counter for counting all currently running threads is increased and the requesting thread can start to process its assigned module instance $I_M$, as it is done from Lines 4 to 5. If otherwise the maximum is reached, the algorithm must store the *thread priority* of the thread in the priority queue $PQ$ and leave it locked, as it is done in Line 7. A thread priority is a pair of the priority value of the current module instance $I_M$, denoted by $p$, and the unique identification number *tid* of the thread. The global priority queue $PQ$, which is initialized in the manager as well, sorts its maintained thread priorities by their priority values.

**Priority locking for the same thread** In case a module instance should be processed by the same thread, the number of running threads must not be increased, but it is checked, whether this number is maximal, as it is done in Line 10, and whether a thread with a higher priority might be waiting, as denoted by Line 11. Are both conditions met, the invoking thread $t$ enqueues its thread priority in $PQ$, Line 12, in order to allow the execution of the higher prioritized module instance $I'_M$ first. The thread priority of it is popped from $PQ$ and the appropriate thread $t'$ is retrieved from the set $T$ of all threads, which is global and also initialized by the manager, as it is outlined by the Lines 13 and 14. Thread $t'$ can then be unlocked to run $I'_M$.check(..) while thread $t$ is locked, as can be seen in

Lines 15 and 16.

**Release of priority locking** Once a running thread, which is not the one of the invoking module instance, has finished the processing of its backend, it can be locked again and the priority lock of another potentially waiting thread can be released. Note, that the locking takes place outside the scope of this algorithm. The method `releasePriorityLocking()` of Algorithm 4 is utilized for releasing priority locks. In Line 1 it tests, whether the priority queue $PQ$ is empty or not. If it is empty, the counter $runningThreads$ is decreased and the thread can be locked, as in line 6. If the priority queue is not empty, the first element with the highest priority is dequeued in order to unlock the corresponding thread $t$ to run the method `check(..)` of its module instance $I_M$, as it was done before.

**Algorithm 4** Algorithm to release a priority locking, if required.

```
void releasePriorityLocking()
begin
    if PQ ≠ ∅ then                                              (1)
        (p, tid) := PQ.pop();                                   (2)
        t := retrieve thread in T with thread id tid;           (3)
        t.unlock();        \\ to run I_M.check(..)              (4)
    else                                                        (5)
        runningThreads := runningThreads - 1;                   (6)
    end if                                                      (7)
end
```

**Global variables:** $runningThreads$, $T$, $PQ$

**Intermediate results** The proposed parallelization approach for SMT-RAT has thus far already been applied under regard of most of the above described aspects. It efficiently decreases the overhead induced by the thread management, as threads are not created before they are not required, module instances reuse the same threads to invoke their backends or reuse its own thread. Moreover, the implemented approach is scalable, because a fixed number of CPU cores is not assumed at any time. The addressed deadlock situations are eliminated, as threads are always reused for the same set of module instances and in case of preventive measures against oversubscriptions, each time a thread finishes its task, it wakes up a suspended thread, if there is one waiting.

### 6.3.2 Interruption of module instances

A module instance invokes several backends with the intention to pursue multiple solving approaches for the same NRA formula. Once one of the backends has found a relevant result, the invoking module instance can continue its own calculation. Currently, a further processing of the remaining backends is not required, because they either return the same relevant result or `unknown`. The parallelism approach therefore specifies, that the remaining backend calculations must be interrupted to accelerate the overall process. Note, that in case of

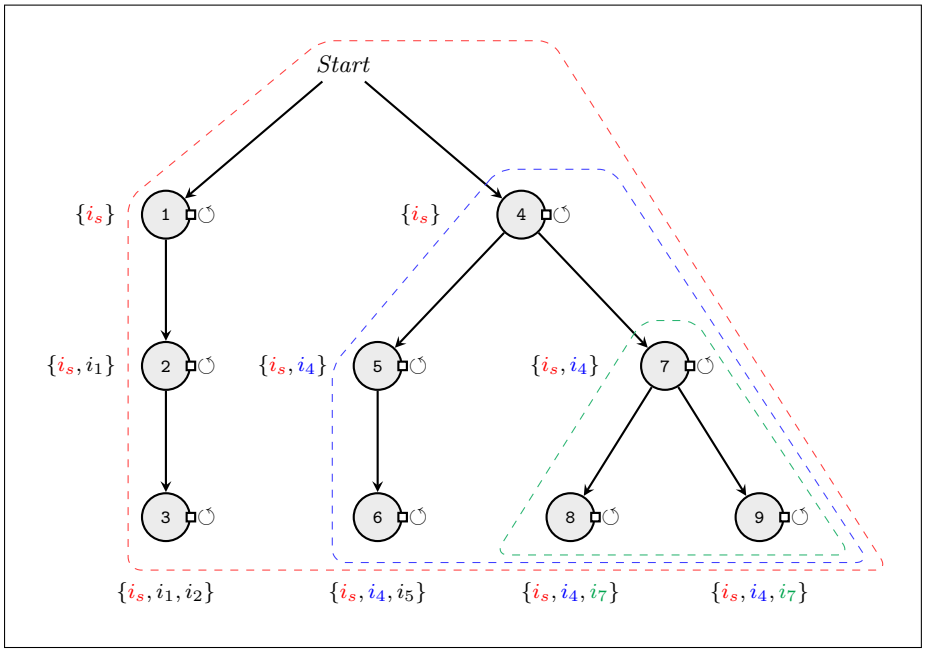



Figure 6.4: Distribution of interruption flags and their sphere of influence.

unsatisfiability, it might be meaningful to leave backends running, if they do not terminate "much" later and if they are thereby enabled to provide further and maybe even better infeasible subsets. This approach is future work and proposed in Chapter 8. All SMT-RAT modules are SMT-compliant, which means that they especially support incrementality and must therefore be enabled to store their intermediate results of a calculation for a later reuse. For this reason, running threads cannot simply be deleted. They must rather be interrupted in a safe fashion without loosing the intermediate calculations of their related backend and without leaving it in an inconsistent state. The `LRAModule` for example, implements the Simplex method, which maintains a tableau, whose elements are updated in each calculation step. Canceling its calculation in an unsafe fashion might not only leave the update unfinished, but also the backend in an unusable state.

**Illustration of the interruption approach** For the above mentioned reasons, it is not possible to cancel a backend processing from outside. It is rather required, that the respective SMT-RAT module implementations regularly control, whether their processing should be interrupted or not. For this purpose *interruption flags* are introduced, which must be distributed from an invoking module instance to its set of backends. In case a backend finishes its check of an NRA formula with a relevant result, it can change its interruption flag to inform the remaining backends of its set. As backends might be enabled to invoke further sets of backends, which must additionally be enabled to be interrupted independently, it is not enough to utilize just a single interruption flag. For this reason sets of interruption flags are required, whose contained flags signalize, which subgraph of backends must be interrupted. The procedure is outlined

by Figure 6.4. In the illustrated strategy graph, sets of interruption flags are positioned next to their related module instances. They are built by utilizing all elements of the set of the preceding module instance and by adding an own interruption flag. Beginning with the *Start* module, which creates the initial interruption flag, sets grow from top to bottom. Building sets in this way guarantees, that all backends of all subgraphs of one module instance contain its interruption flag. It can for example easily be seen, that the initial interruption flag of the *Start* module, which is named $i_s$ and colored in red, is contained in all sets in the strategy graph. The interruption flag $i_4$ of module instance `4`, which is colored in blue in the figure, is then again contained in the sets of the module instances `5` to `9`. If for example any of the backends of module instance `4` calculates a relevant result, the interruption flag $i_4$ is changed and signalizes the interruption for all remaining backends. This means, that all backends in both subgraphs of module instance `4` are subject of this interruption, but no preceding module instances are involved. By applying this method any subgraph can be interrupted independently from each other. While backends are processed for module instance `4`, module instance `7`, for example, can interrupt its own subgraphs independently. Its interruption flag $i_7$, which is colored green, independently interrupts the processing of backend `8` and `9`, but does not influence the processing of backend `5` or `6`. The three dashed fringes show the different spheres of influence of the corresponding interruption flags $i_s$, $i_4$ and $i_7$.

**Interrupting a module instance** When checking a set of interruption flags, all flags are checked and in case any signalizes an interruption, a backend must stop its satisfiability check in order to store its intermediate results for later executions and to return the result of the check to the invoking module instance. An interruption process might be continued into upper levels of the strategy graph, depending on which flag is signalizing the interruption. Once all required interruptions have been fulfilled, the module instance of the reached level can continue its processing.

Note, that in case of utilizing preventive measures against oversubscriptions, a relevant result might be calculated for an NRA formula before all locked threads are released for their calculations. In this case, the backends of the released threads do not start any satisfiability check, but directly return `unknown` to the invoking module instance.

Almost all of the aspects of the proposed parallelization approach have been considered at this point, only the aspect of dynamically utilizing multithreading is missing, which is explained in the next section.

## 6.4 Full procedure of a parallel SMT solver utilizing strategy graphs

Chapter 3 described the different modular components of SMT-RAT, the newly developed strategy graph component has been introduced in Chapter 4 and the application of a parallelization approach for SMT-RAT has been presented in this chapter. All required components are provided to give a detailed description of the implementation of the concurrently working manager.

Before the details of the final and consolidated implementation are examined with the help of Algorithms 5 and 6, a required method is explained.

`numberOfLeaves(graph` $G$`):` This method calculates the number of leaves for a given strategy graph $G$, which can be used as the maximum required number of threads. If the returned value is greater than 1, it shows, that the passed strategy graph contains branches and that multithreading can be utilized.

The implementation of the manager is partitioned into two algorithms. The first algorithm covers the method `solver(..)`, which is initially invoked for preliminary initializations of global variables. The second implements an enhanced version of the method `runBackends(..)`, which has initially been presented in Chapter 3. This method controls the process of SMT solving by applying the newly introduced capabilities.

**Initialization of global variables** The method `runBackends(..)` is recursively invoked and requires a set of global variables, whose preparatory initializations are outsourced into the first Algorithm 5. This algorithm is only called once at the beginning of each SMT solving process and requires a strategy graph $G$ and an NRA formula $\varphi$ as parameters. Depending on this strategy graph and the underlying computer architecture the global variables are dynamically initialized. First of all, this algorithm finds out whether multithreading can be constituted or not, as proposed in Section 6.2. It can be constituted in case the used computer architecture offers multiple CPU cores and the passed strategy graph has more than 1 leave, which is checked with the method `numberOfLeaves(..)`. In this case, the global flag $runsParallel$ is set to $True$, otherwise to $False$. This flag is exploited by the second algorithm, which is thereby enabled to guide its program flow either for a sequential or parallel execution. In case multithreading is utilized, threads are needed and they are maintained in the global set $T$ to allow the manager accesses at any time. Furthermore, the algorithm checks the possibility of oversubscriptions, which are potentially introduced together with the multithreading utilization. A simple check tests, whether the number of leaves exceeds the number of CPU cores. The same method `numberOfCores()` as of Section 6.3 is utilized for this purpose. The result of this testing is stored in the flag $potentialOversubscription$, which signalizes the following algorithm, whether to initiate the preventive measure of utilizing priority lockings or not. In case the risk is given the counter $runningThreads$ is introduced and initially set to 1, as the *Start* module already itself runs in a thread. Moreover, the priority queue $PQ$ is created to store the thread identification numbers of locked threads in a prioritized way. Besides the aspect of dynamic multithreading utilization and oversubscription, a set of interruption flags $F$ is created to apply the approach of interrupting the processing of certain module instances. As the program flow of non-concurrently running SMT solvers is not affected by this set and for reasons of simplicity, it is introduced, whether module instances do or do not run concurrently. At the end, the method `runBackends(..)` of Algorithm 6 is invoked on some initial values in order to invoke the initial module instances.

**Core implementation** Including detours via invoked module instances, the method `runBackends(..)` of the manager is recursively called. Therefore,

**Algorithm 5** The first algorithm of the manager implementation for initializing required global variables.

```
result solver(graph G, formula φ)
begin
    if numberOfLeaves(G) > 1 ∧ numberOfCores() > 1 then                (1)
        runsParallel := True;                                          (2)
        set of threads T := ∅;                                         (3)
        if numberOfLeaves(G) > numberOfCores() then                    (4)
            potentialOversubscription := True;                         (5)
            runningThreads := 1;                                       (6)
            thread priority queue PQ := ∅;                             (7)
        else                                                           (8)
            potentialOversubscription := False;                        (9)
        end if                                                         (10)
    else                                                               (11)
        runsParallel := False;                                         (12)
    end if                                                             (13)
    set of interruption flags F := ∅;                                  (14)
    return runBackends(G, 0, φ, F);                                    (15)
end
```

**Global variables:** *runsParallel*, $T$, *potentialOversubscription*, *runningThreads*, $PQ$

the method exploits several parameters to control its execution flow. A priority value $p$ is used to determine the current position in the strategy graph $G$. Initially the priority value is `0` in order to refer to the *Start* module, which invokes the initial module instances of the strategy graph. The method `runBackends(..)` retrieves a set of available backends for an invoking module instance, and calls for each of these backends the method `check(..)` on $\varphi$, which is the passed formula of the invoking module instance. The last parameter is a set of interruption flags $F$. It is not global, as it is individually determined for each set of the backends to be invoked. At the beginning of the method, the set is filled with an interruption flag and thereby grows with each further recursive call.

The goal of the algorithm is to invoke all available backends on the passed formula $\varphi$ and to return the result to the invoking module instance. For this purpose, a variable *result* is initialized with the value `unknown`, as can be seen in Line 3. In order to receive the available backends of the invoking module instance, the method `getAvailableBackends(..)` of Algorithm 1 of Chapter 4 is utilized, which is outlined in Line 4. Depending on its parameters the method returns a, possibly empty, set $B$ of backends. The algorithm then utilizes a loop to invoke all backends in their prioritized order one after the next, whereas it starts with the backend of highest priority, as outlined by Lines 8 and 9. For each backend a module instance $I_M$ of the corresponding module type $M$ of the backend is used and a set of interruption flags $F$ is assigned, which is stated by Line 11. Note, that backends are instances of modules and must be created when they are initially used. The set of interruption flags consists of the passed set $F$ including an additional interruption flag added in Line 2. This set

must regularly be controlled by the corresponding method `check(..)` of module instance $I_M$. Depending on whether multithreading can be exploited or not and whether a backend is the one with the lowest priority of set $B$, the execution of $I_M$ is then applied differently, as can be seen in Lines 12, 28 and 36. The first case, which is described in the following, is the one where multithreading is enabled and all backends, except the one with the lowest priority, are processed. For utilizing multithreading, a locale set $TID$ of thread identification numbers must be initialized, as presented by Line 5. This set allows to memorize all related threads for this pass of the algorithm in order to be able to retrieve their results at the end.

**Processing backends with multithreading** When utilizing multithreading, the processing of the corresponding formula is moved into an own thread for each backend, except for the one with the lowest priority. For this reason, the thread identification number is received through Algorithm 2 for each backend in order to forward the execution to its related thread, as can be seen in Line 13. The utilized thread identification number *tid* is stored in the set $TID$, as constituted by Line 14, in order to fetch its result at the end. All threads, which have once initially been created, are supposed to be reused and are therefore stored in the global set $T$ of threads before forwarding a backend to them, as it is presented in the Lines 16 to 20. In case of the usage of multithreading, several calls of the method `runBackends(..)` might be processed concurrently. This implies that write accesses of global variables, which are shared resources, must be protected by mutual exclusions. The set of threads $T$, for example, is subject to this and the lock *mutex* ensures that no race conditions appear. The usage of mutual exclusions is a dependent detail of the programming language and the variable *mutex* is exemplary used in Lines 15 and 27, 31 and 33 as well as in Lines 50 and 54 of this algorithm. Once a lock is acquired and the specific thread $t$ has been retrieved, a reference of the method `check(`$\varphi$`)` of backend $I_M$ can be pushed into it, as represented by Line 21. Before thread $t$ can then start the execution it must be released, as it is initially locked in order to prevent a potential oversubscription. In case *potentialOversubscription* is set to `True`, the method `checkPriorityLocking(..)` of Algorithm 3 is utilized. As described, it takes care whether the passed thread, denoted by its thread identification number *tid*, can be unlocked directly or if it is required to stay locked first. In the opposite case that *potentialOversubscription* is set to `False`, the number of running threads cannot exceed the number of available CPU cores with the implemented parallelization approach and any thread can always be unlocked immediately. While a given thread $t$ is starting the processing of its corresponding method `check(`$\varphi$`)`, the initiating thread of the algorithm proceeds. Note, that due to implementation details, threads offer a possibility to retrieve their results once their finished.

**Processing the lowest prioritized backend** When processing the last backend, which means the one with the lowest priority or the only one available, the current thread itself is used to invoke the method `check(`$\varphi$`)` of module instance $I_M$, as can be seen in Line 35. As explained, this guarantees, that at least one of the threads is always subsequently reused, even though multithreading is utilized. In case of potential oversubscriptions, the method

`checkPriorityLocking(..)` must be executed beforehand, as the current thread might as well require to be locked until all threads with higher priority have finished their executions, as it is presented by Line 32.

**Processing backends without multithreading** In case it is not meaningful to utilize multithreading or it simply cannot be applied, just one thread is utilized to invoke the method `check(`$\varphi$`)` for each backend $I_M$, which executes them one after the next, as it can be seen in Line 37. The loop of Line 8 is iterated until one of the backend returns a result unequal to `unknown` or until all backends have been processed. Even without multithreading the executions of the backends are ordered by their priority for each branch in the strategy graph, whereas the priority orders distributed throughout the whole strategy graph are not observed. Without maintaining threads, it is not sophisticated to stop the processing in one branch of the strategy graph, traverse to another one and then return to the original branch at a later point.

**Determining the result** In case no threads have been utilized, the value of *result* has been determined at this point and can be returned. If threads were included, further measures are required, as they might need to be locked again for further executions and their results must be fetched. As mentioned, the set of thread identification numbers $TID$ has been created for these purposes. It is processed in a loop by removing one thread identification number *tid* in each iteration in order to retrieve the corresponding thread $t$, as can be seen in Lines 42 and 43. Note, that at this point all threads denoted by the thread identification numbers of $TID$ might still be processing their corresponding method `check(`$\varphi$`)`, when no relevant result has been found yet. When fetching their results, due to implementation details, it is ensured, that the thread executing the method `runBackends(..)` is waiting until their results *tmpResult* can be supplied. In case a relevant result has been calculated once, all threads stop their calculations due to their assigned set of interruption flags $F$. In case the fetched result *tmpResult* of a thread $t$ is unequal to `unknown`, it is used to overwrite *result*, as constituted by Lines 45 to 48. Otherwise the result is dismissed, as it might overwrite a relevant result subsequently. Afterwards *potentialOversubscription* constitutes, whether method `releasePriorityLocking(..)` has to be utilized. In this case, a thread $t$ needs to be locked in order to enable the unlocking of a different thread, which might potentially be locked. For this purpose, the loop must be iterated for all threads, even if a relevant result might already be given. At the end of the algorithm, the value of the variable *result* is returned to the invoking module instance. Its value might be `unknown` in case the backends were not capable of checking the formula $\varphi$ or no backends were provided at all. Note, that in the implementations of SMT-RAT it is ensured, that for the case of unsatisfiability the invoking module instance can retrieve the infeasible subsets of the involved backends.

**Algorithm 6** Core algorithm of the parallel working manager to run backends for an invoking module instance. (Part I)

```
result runBackends(graph G, priority p, formula φ, flags F)
begin
    interruption flag i_f := False;                                          (1)
    F := F ∪ {i_f};                                                          (2)
    result := unknown;                                                       (3)
    set of backends B := getAvailableBackends(G, p, φ);      \\Alg. 1        (4)
    if runsParallel = True then                                              (5)
        set of thread ids TID := ∅;                                          (6)
    end if                                                                   (7)
    while result = unknown ∧ B ≠ ∅ do                                        (8)
        M := module in B with highest priority p';                           (9)
        B := B\{M};                                                          (10)
        I_M := instance of module type M with interruption flags F;          (11)
        if runsParallel = True ∧ M not last module of B then                 (12)
            tid := getThreadId(G, p');          \\Alg. 2                     (13)
            TID := TID ∪ {tid};                                              (14)
            mutex.lock();                                                    (15)
            if thread with thread id tid does not exist in T then            (16)
                create locked thread t with id tid;                          (17)
                T := T ∪ {t};                                                (18)
            end if                                                           (19)
            t := retrieve thread in TP with thread id tid;                   (20)
            t.push(I_M.check(φ));                                            (21)
            if potentialOversubscription = True then                         (22)
                checkPriorityLocking(tid, p', False);      \\Alg. 3          (23)
            else                                                             (24)
                t.unlock();        \\ to run I_M.check(φ)                    (25)
            end if                                                           (26)
            mutex.unlock();                                                  (27)
        else if runsParallel = True ∧ M last module of B then                (28)
            if potentialOversubscription = True then                         (29)
                tid := getThreadId(G, p);          \\Alg. 2                  (30)
                lock mutex;                                                  (31)
                checkPriorityLocking(tid, p, True);        \\Alg. 3          (32)
                unlock mutex;                                                (33)
            end if                                                           (34)
            result := I_M.check(φ);                                          (35)
        else                                                                 (36)
            result := I_M.check(φ);                                          (37)
        end if                                                               (38)
    end while                                                                (39)
    ...
    Continued on page 57
```

**Algorithm 6** Core algorithm of the parallel working manager to run backends for an invoking module instance. (Part II)

```
    ...
    if runsParallel = True then                                    (40)
        while TID ≠ ∅ do                                           (41)
            tid := choose element in TID;                          (42)
            TID := TID\{tid};                                      (43)
            t := retrieve thread in TP with thread id tid;         (44)
            tmpResult := t.result();                               (45)
            if tmpResult ≠ unknown then                            (46)
                result := tmpResult;                               (47)
            end if                                                 (48)
            t.lock();                                              (49)
            mutex.lock();                                          (50)
            if potentialOversubscription = True then               (51)
                releasePriorityLocking();      \\Alg. 4            (52)
            end if                                                 (53)
            mutex.unlock();                                        (54)
        end while                                                  (55)
    end if                                                         (56)
    return result;                                                 (57)
end
```

# Chapter 7

# Experimental results

As all demanded approaches of the thesis have been implemented in the preceding chapters, this chapter can present experimental results comparing the efficiency of sequential strategies against parallel strategies, which enable their combined concurrent execution.

## 7.1 Setup

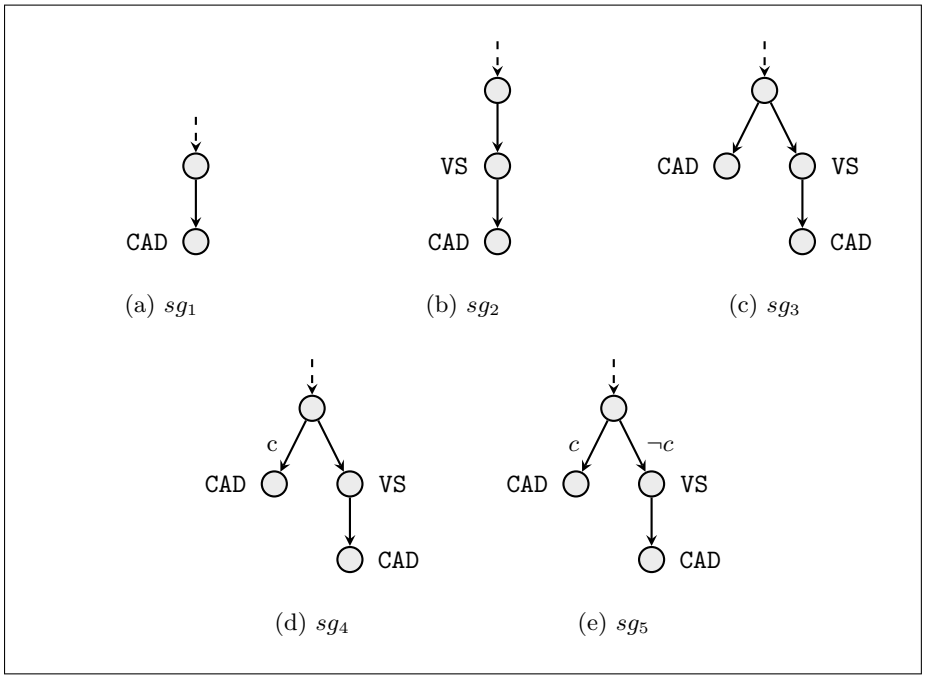
The adapted SMT-RAT application has been utilized to create altogether five SMT solvers, which have been involved in the testing. Although some of them are guided by a sequential strategy, the adapted toolbox could be utilized, as no multithreading induced overhead must be assumed for their solving process. The created SMT solvers differ in their contained strategy graphs, which are presented in the following.

### 7.1.1 Utilized strategy graphs

The strategy graphs have been chosen in order to outline the exemplary situation, where it is uncertain whether to utilize just the single `CADModule` for a given set of NRA problems or to prefix it with the `VSModule`. The `CADModule` contains a complete procedure, which in general works efficiently for constraints containing strict inequalities, but is rather inefficient for equations. The `VSModule` is more efficient but incomplete. It requires a further SMT-RAT module, for instance the `CADModule`, to handle polynomials which have a degree greater than 2. The built strategy graphs apply different approaches to tackle this problem. All of them begin with a sequential composition of the `CNFerModule`, which transforms a given NRA problem into CNF, followed by the `SATModule`, which implements a DPLL-style less lazy SAT solver. Figure 7.1 illustrates the different strategy graphs, but the composition is only pointed out by the dashed arrows above each.

$sg_1$ Strategy graph $sg_1$ is a sequential strategy, which intends the `CADModule` instance for the solving process only.

$sg_2$ As in the preceding case, the strategy graph $sg_2$ also provides a sequential strategy, but contains an additional `VSModule` instance as prefix of the




(a) $sg_1$ (b) $sg_2$ (c) $sg_3$

(d) $sg_4$ (e) $sg_5$

Figure 7.1: The strategy graphs utilized for the testing.

`CADModule` instance. The strategy graphs $sg_1$ and $sg_2$ simulate the case, where only sequential executions of strategies can be applied.

$sg_3$ The parallel strategy graph $sg_3$ combines the alternative strategies $sg_1$ and $sg_2$. This means, that it enables the concurrent execution of the `CADModule` instance and the composition of the `VSModule` instance and `CADModule` instance.

$sg_4$ Strategy graph $sg_4$ bases on strategy $sg_3$, whereas for the invocation of the single `CADModule` instance the condition $c := contains\_strict\_inequalities$ is added. This means, that this instance can only be invoked, if a passed formula contains strict inequalities, where strict inequalities can only contain the operators '$<$', '$>$' or '$\neq$'. The composition of the `VSModule` instance and `CADModule` instance is invoked for all passed formulas, as no condition is applied for it.

$sg_5$ The last strategy graph $sg_5$ further on bases on $sg_4$ by adding a second condition to the composition, which is the inverting of condition $c$. This means, that always either the single `CADModule` instance or the composition can be invoked, but not simultaneously. The strategy graphs $sg_4$ and $sg_5$ present implications for $sg_3$ when introducing dynamic by conditions.

### 7.1.2 Benchmark sets

Two benchmark sets, which contain 50 example files each, have been assembled from a large amount of example files. The first set contains example files selected

from several different categories of the *Meti-Tarski* benchmarks of SMT-LIB [2]. The selected example files of the second set also originate from various Meti-Tarski benchmarks, but have been preprocessed with iSAT [15]. The SMT solver iSat implements the ICP method, which searches for possible candidate solutions, in form of an interval box, for the variables of a given NRA formula. As the method is incomplete, it needs to call a further decision procedure to check the consistency of its proposed candidate solutions. These calls have been intercepted in order to create a collection of preprocessed sets from which the second benchmark set assembles.

### 7.1.3 Results

All example files have been processed on an Intel® Core™ i5-2520M CPU containing four threads (Hyper-Threading) and two cores of 2.5 GHz each. This means, that all described strategy graphs could occupy enough CPU cores to execute all intended approaches in parallel. Moreover, a time-out of 15 seconds and a memory limit of 4 GB have been assigned to each example file. The obtained results are presented in Table 7.1, which shows how many of the example files could successfully be checked by each utilized strategy graph before a time-out occurred and how much time, in milliseconds, has been required.

The experimental results for the strategy graph $sg_1$, which intends only the `CADModule` instance as decision procedure for solving NRA formulas, are the worst. The module instance did not only solve the least number of examples, but also required the longest time in both sets. This could be expected, as the `CADModule` is complete, but works quite inefficiently.

The strategy graph $sg_2$ already provides improved experimental results. The table shows, that in many cases it is useful to prefix a `CADModule` instance with a `VSModule` instance in a strategy graph. For the given benchmark sets, the SMT solver containing strategy graph $sg_2$ was able to check more formulas and additionally required only around 60 percent of the time of the SMT solver maintaining $sg_1$. Note, that with the utilization of strategy graph $sg_1$ formulas could be checked, which could not be checked with $sg_2$, and vice versa.

The SMT solver guided by strategy graph $sg_3$ presents the best experimental results. For both benchmark sets the most formulas could be checked by $sg_3$ and only one example file has been missed. For the Meti-Tarski benchmark sets a similar amount of time have been required as with strategy graph $sg_2$. In the second benchmark set, which have been preprocessed with iSAT, the required time is significantly reduced. Improved results could be expected, as both approaches of the strategy graphs $sg_1$ and $sg_2$ have been executed in parallel.

The strategy graph $sg_4$ added a condition for the single `CADModule` instance such that it is only invoked for passed formulas, which contain strict inequalities. Conditions can be meaningfully added to strategy graphs which intend more concurrently running backends than CPU cores are available. Thereby the number of concurrently running backends can dynamically be reduced, as it is done with $sg_4$. In case of the applied condition of $sg_4$, the experimental results are very close to those of $sg_3$, as only two less example files could be checked.

Whereas strategy graph $sg_4$ just pointed out, that the utilization of conditions might be helpful, $sg_5$ shows, that this can also be disadvantageous. The SMT solver guided by strategy graph $sg_5$ presents even worse results as the

one guided by the sequential strategy $sg_2$. For the first benchmark set around the same time has been required for less formulas. For the second benchmark set less formulas could be checked whereas even more time has been required. Strategy graph $sg_5$ illustrates, that for the given benchmark sets it is not appropriate to use a `CADModule` instance whenever strict inequalities occur in an NRA formula.

The experimental results illustrate, that the utilization of parallel strategies is a successful approach for achieving more efficiency for SMT solving.

| | Meti-Tarski (50) | | iSAT-Meti-Tarski (50) | | *all* (100) | |
|---|---|---|---|---|---|---|
| | # | *time* | # | *time* | # | *time* |
| $sg_1$ | 32 | 122074.0 | 41 | 110211.0 | 73 | 232285.0 |
| sat | 26 | 96491.0 | 22 | 73609.0 | 48 | 170100.0 |
| unsat | 6 | 25583.0 | 19 | 36602.0 | 25 | 62185.0 |
| $sg_2$ | 39 | 82347.0 | 49 | 67954.0 | 88 | 150301.0 |
| sat | 22 | 47527.0 | 25 | 48600.0 | 47 | 96127.0 |
| unsat | 17 | 34820.0 | 24 | 19354.0 | 41 | 54174.0 |
| $sg_3$ | 44 | 82549.0 | 50 | 47551.0 | 94 | 130100.0 |
| sat | 29 | 65809.0 | 26 | 36009.0 | 55 | 101818.0 |
| unsat | 15 | 16740.0 | 24 | 11542.0 | 39 | 28282.0 |
| $sg_4$ | 43 | 75585.0 | 49 | 39147.0 | 92 | 114732.0 |
| sat | 28 | 56375.0 | 25 | 27836.0 | 53 | 84211.0 |
| unsat | 15 | 19210.0 | 24 | 11311.0 | 39 | 30521.0 |
| $sg_5$ | 36 | 82251.0 | 41 | 87971.0 | 77 | 170222.0 |
| sat | 28 | 56070.0 | 20 | 36129.0 | 48 | 92199.0 |
| unsat | 8 | 26181.0 | 21 | 51842.0 | 29 | 78023.0 |

Table 7.1: The experimental results.

## 7.2 Further benchmarks

As SMT-RAT offers many more module implementations as just the `CADModule` and `VSModule` and the utilization of strategy graphs allows a rich possibility of composing SMT-RAT modules, many more meaningful SMT solvers can be created, evaluated and compared. Unfortunately, this must be proposed as future work, because most of the SMT-RAT modules are not thread-safe yet. Due to constantly occurring race conditions, further created SMT solvers could not be tested properly. The problem actually does not relate to SMT-RAT directly, but originates from the integrated library GiNaC [4], which is used for symbolic computations within the decision procedure implementations. In order to gain thread-safety for all SMT-RAT modules GiNaC must therefore be adjusted, which is out of the scope of the thesis.

# Chapter 8

# Conclusion

The thesis introduced the new strategy graph approach and its complete implementation in the SMT-RAT application. The toolbox has thereby successfully been enriched with the following main features:

- The strategy graph approach enhances the expressiveness of strategies by enabling several alternative backends for a module instance.
- The high self-explanatory and easy to use GUI SMT-XRAT allows even the inexperienced user to compose strategy graphs without burden.
- The applied parallelization approach of the manager supplies the exploitation of further hardware resources for concurrently executed SMT-RAT modules. Hereby general accelerations of solving processes can be assumed due to the confident experimental results of the of the previous chapter.

## 8.1 Future work

Future work is first of all required for achieving thread-safety for GiNaC and thereby also for all SMT-RAT modules. After that, a lot of different strategy graphs can be evaluated in order to gain further knowledge about the strengths and weaknesses of the different decision procedures and in order to allow a further fine-tuning of the applied parallelization approach.

Furthermore, during the implementation of the above stated features, many ideas could be conceived, which have not been applied due to the shortness of time. The most interesting ones are proposed as future work in the following.

### 8.1.1 Backlinks

The currently underlying structure of a strategy graph is an acyclic, directed and weakly connected graph. It is useful to further allow the construction of cyclic graphs in order to introduce the concept of *backlinks*. A backlink is a module-backend-relation from a module instance to a preceding module instance in the strategy graph. This is useful in case a sequential sequence of SMT-RAT modules can logically be compounded to a single backend, which can recursively invoke itself on its calculated passed formulas. The concept

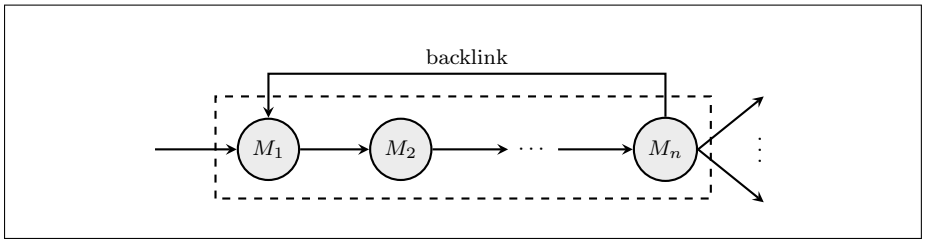



Figure 8.1: The concept of backlink utilization.

of backlinks is illustrated by Figure 8.1, where the depicted module instances $M_1$ to $M_n$ are logically compounded to one backend. The displayed backlink signalizes, that this backend can invoke further module instances and itself on a passed formula. Note, that when a compounded backend invokes itself, a new backend instance is created, which means the same composition with new module instances must be instantiated for each invocation. Compounding module instances to one backend is meaningful, in case the contained decision procedures and simplifying approaches can be iteratively applied on a formula and its thereby arising passed formulas.

### 8.1.2 Favoring infeasible subsets over interruptions

When a module instance invokes several concurrently running backends, and one calculates the unsatisfiability of the passed formula, it might be useful to leave the remaining backends running instead of interrupting them, as mentioned in Chapter 6. When preventing an interruption further and maybe even better infeasible subsets could be obtained from the remaining backends. For this approach an appropriate balance between further required processing time and the expected quality of the obtained infeasible subsets must be found, which could be achieved by machine learning, for instance.

### 8.1.3 Shared pool of infeasible subsets

The utilization of parallel strategies implies, that several module instances are checking different passed formulas of the same initial NRA formula. For this purpose, it is meaningful to propose a shared pool of infeasible subsets, which can be accessed and enhanced by all module instances of a strategy graph in order to provide a common overall knowledge to all of them.

### 8.1.4 Parallel decision procedures

The applied parallelization approach of SMT-RAT is just one step towards concurrently working SMT solvers. Furthermore, parallel working decision procedures are required, which means, that they itself need to scalable divide their task of checking a passed formula into several subproblems, which can be concurrently processed. In this case, even sequential strategies could fully exploit multiprocessing, as a single decision procedure can thereby occupy all available CPU cores for its calculation. Moreover, as stated in Chapter 6, the applied parallelization approach might utilize less CPU cores than available, even for

parallel strategies. The reason is, that the approach only intends the usage of one core for each backend, but in case of applying this proposal, remaining available CPU cores could be utilized by parallel working decision procedures.

### 8.1.5 Message Passing Interface

After applying the possibility to create SMT solvers containing parallel decision procedures, the parallelization approach can be further extended by the utilization of the *Message Passing Interface* (MPI). MPI can enable the possibility to distribute the calculation of the solving process even among multiple computers, for example using grid computing or supercomputer. This is especially useful for hard to solve problems, which require a lot of time, in order to justify the further induced overhead.

# Bibliography


[1] Aho, A., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, & Tools*, 2 ed. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[2] Barrett, C., Stump, A., and Tinelli, C. The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[3] Barrett, C., and Tinelli, C. CVC3. In *Proceedings of the* $19^{th}$ *International Conference on Computer Aided Verification (CAV '07)* (Jul 2007), W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 298–302.

[4] Bauer, C., Frink, A., and Kreckel, R. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. Symb. Comput. 33*, 1 (Jan 2002), 1–12.

[5] Brown, C. W. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *SIGSAM BULLETIN 37* (2003), 97–108.

[6] Collins, G. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, H. Brakhage, Ed., vol. 33 of *Lecture Notes in Computer Science*. Springer, 1975, pp. 134–183.

[7] Corzilius, F., Loup, U., Junges, S., and Ábrahám, E. SMT-RAT: An SMT-Compliant Nonlinear Real Arithmetic Toolbox. In *Theory and Applications of Satisfiability Testing – SAT 2012*, A. Cimatti and R. Sebastiani, Eds., vol. 7317 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 442–448.

[8] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (July 1962), 394–397.

[9] de Moura, L., and Passmore, G. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics*, M. Bonacina and M. Stickel, Eds., vol. 7788 of *Lecture Notes in Computer Science*. Springer, 2013, pp. 15–44.

[10] Dolzmann, A., and Sturm, T. Simplification of Quantifier-free Formulas over Ordered Fields. *Journal of Symbolic Computation 24* (1995), 209–231.

[11] Dolzmann, A., and Sturm, T. REDLOG Computer Algebra Meets Computer Logic. *ACM SIGSAM Bulletin 31* (1996), 2–9.

[12] Dutertre, B., and de Moura, L. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, T. Ball and R. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*. Springer, 2006, pp. 81–94.

[13] Eén, N., and Sörensson, N. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 502–518.

[14] Fränzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *Journal on Satisfiability, Boolean Modeling and Computation 1* (2007), 209–236.

[15] iSAT. `http://isat.gforge.avacs.org/`.

[16] Jovanovi, D., and de Moura, L. Solving Non-linear Arithmetic. In *Automated Reasoning*, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 339–354.

[17] JUNG. `http://jung.sourceforge.net/`.

[18] Kroening, D., and Strichman, O. *Decision Procedures: An Algorithmic Point of View*, 1 ed. Springer, 2008.

[19] Maxima. `http://maxima.sourceforge.net/`.

[20] Passmore, G., and Jackson, P. Combined Decision Techniques for the Existential Theory of the Reals. In *Intelligent Computer Mathematics*, J. Carette, L. Dixon, C. Coen, and S. Watt, Eds., vol. 5625 of *Lecture Notes in Computer Science*. Springer, 2009, pp. 122–137.

[21] SMT-RAT. `http://smtrat.sourceforge.net/`.

[22] Tseitin, G. On the complexity of proofs in propositional logics. In *Automation of Reasoning: Classical Papers in Computation Logic 1967-1970* (1983), vol. 2, Springer.

[23] Weispfenning, V. Quantifier Elimination for Real Algebra – the Quadratic Case and Beyond. *Applicable Algebra in Engineering, Communication and Computing 8*, 2 (1997), 85–101.

[24] Weispfenning, V. A New Approach to Quantifier Elimination for Real Algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, B. Caviness and J. Johnson, Eds., Texts and Monographs in Symbolic Computation. Springer, 1998, pp. 376–392.

[25] Williams, A. *C++ Concurrency In Action: Practical Multithreading*. Manning Pubs Co Series. Manning, 2012.