HSMTLib

Interacting with SMT Solvers in Haskell

Nuno Laranjo and Rogério Pontes

Informatics Department, Minho University, Portugal

Abstract With the constant improving in performance of the SMT solvers [Cok11] and with the initiative SMT-LIB [BST10] that aims to define a common standard, their use is expanding to several areas with new applications emerging. HSMTLib is a Haskell library capable of communicating with several SMT-LIB2 compliant solvers. This library facilitates the creation of applications that need to interact with SMT solvers by providing an API which conforms to the SMT-LIB2 standard. Two modes of operations are available, *online mode* where the solver is kept running externally and *script mode* which launches the solver only when needed. The response from a solver is converted into a Haskell data type making it easy to create an interactive program. Three solvers are currently supported, Z3, MathSAT and CVC4.

1 Introduction

The Satisfiability Modulo Theories (SMT) problem is a generalization of the Boolean satisfiability (SAT) problem which has the focus on proving the satisfiability of logical formulas in regard to specific first-order theories. Theories are expressed in classical many-sorted first-order logic with equality. A theory defines the vocabulary of the logical language (stating its sorts, functions and predicates) and its semantics. A combination of several theories (possibly with some restrictions is called a *logic*. For instance, Ints is the theory of integers and ArrayEx is the theory of applicative arrays with extensionality, while AUFLIA is the logic that combines the Ints and ArrayEx theories restricted to linear integer arithmetic and arrays with integer indices and values.

SMT solvers have gained enormous popularity over the last decade. They have been applied with success in several domains in computer science and are the subject of very active research. Popular SMT solves include CVC4 [Bar+11], Z3 [MB08], Yices [DM06], MathSAT [Boz+05] and Alt-Ergo [Bob+08], among many others.

The SMT-LIB[1] is an initiative aimed at facilitating research and development in SMT. One of its main goals is to provide standard rigorous descriptions of background theories used in SMT systems and to promote a common input and output format for SMT solvers. This standardization allowed the creation of a benchmarking framework for the assessment and comparison of different solvers

[1] http://smt-lib.org

(its performance w.r.t. each logic) and gave rise to the SMT competition, SMT-COMP[2].

A program that may have to deal with several logics can improve its performance if it can connect to different SMT solvers, in order to work with the solver that has better performance with the logic being used at moment. With this in mind the library was developed to facilitate the comunication with multiple solvers.

Usually SMT solvers provide a native API, most commonly, for C, and there are some wrappers for other programming languages. However this raises the problem of being solver specific. The Haskell Z3 package[3] is one of those wrappers that creates a library to interact with the Z3 solver in Haskell by wrapping the native C API. To surpass the limitation of being solver specific HSMTLib was constructed with generic methods and follows a similar approach to the smt-Lib package package[4] which provides tools for parsing SMT-LIB2 files creating an Abstract Syntax Tree (AST) that can be used to create scripts.

Many software need to interact with SMT solvers. An example is the Haskell SBV[5] package that allows to express properties about Haskell programs and automatically prove them correct with the help of an external SMT solver. Another project is LiquidHaskell[6] which is a static verifier for Haskell, based on Liquid Types (refinement types extending the base type system with refinement predicates drawn from decidable logics). Another example is Cryptol[7] is a domain-specific language for specifying cryptographic algorithms. Cryptol has verification features that use SMT solvers to check properties about the programs. All these examples testify the growing usage of SMT solvers and the need for a library which facilitate the interaction with them. This is the main goal of the Haskell package we have developed, to provide easy interaction with multiple solvers.

HSMTLib provides an API based on the functions and AST defined by SMT-LIB, additionally, it offers a high-level API in order to simplify the creation of expressions and to simplify common tasks. We provide a uniform way of interacting with different solvers and two different modes of interaction. The library also has tools to parse the answers returned by the solvers and turn them into a Haskell data type.

The remaining of this document is organized as follows. In the next section we present the inner architecture of HSMTLib and describe in some detail every layer of the library. Section 3 is devoted to a use case example where we show how the library can be used to interact with the solvers. Section 4 describes the tests made to the library and Section 5 conclude.

---

[2] http://smtcomp.org
[3] https://hackage.haskell.org/package/z3
[4] https://hackage.haskell.org/package/smt-lib
[5] http://hackage.haskell.org/package/sbv
[6] http://goto.ucsd.edu/ rjhala/liquid/haskell/blog/about/
[7] http://cryptol.net/

2 HSMTLib description

The code in development is hosted at Github[8] and the last stable version can be downloaded from Hackage or installed via Cabal. The library also has haddock documentation available at Hackage[9] and some additional tutorials are available in the Github Wiki[10].

Due to the modular nature of Haskell, the library is divided in four parts as shown in Figure 1. The *API layer* is the module that interacts with the software and therefore is the most interesting to the developer. The API layer is itself splited in two sub-layers, a *lower level* API and a *higher level* API as explained below.Then there is the *solver layer* which implements the functions defined in the lower level API using functions provided by the *communication layer* which handles the interaction with the solvers. There are two modes of interaction with the solvers: the *online mode* where the solver is kept running externally and the *script mode* which launches the solver only when needed. Finally, there is the *response layer* which is responsible for parsing the response from the solver and turn it into a Haskell data type. Let us now explain each of these modules in more detail.

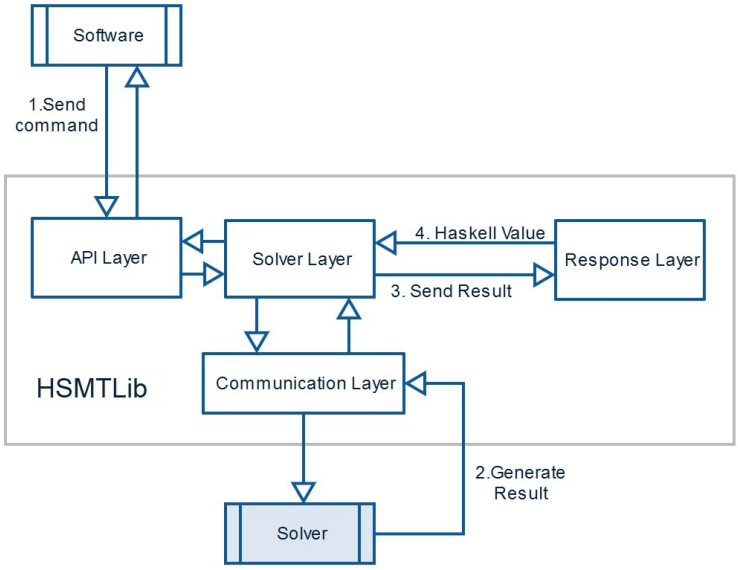

Figure 1: HSMTLib architecture.

[8] https://github.com/MfesGA/Hsmtlib
[9] https://hackage.haskell.org/package/Hsmtlib
[10] https://github.com/MfesGA/Hsmtlib/wiki

2.1 API layer

The datatype *Solver* at Listing 1 aggregates the set of possible commands to send to the solver defined by SMT-LIB in all interaction modes and uses the SMTLib2 package[11] in order to construct the expressions to be sent to the solver, additionally numerous auxiliary functions are provided in order to create the most common expressions, for example addition of integers or selecting a value from an array.

```
data Solver = Solver
    { setLogic      :: Name -> IO Result
    , setOption     :: Option -> IO Result
    , setInfo       :: Attr -> IO Result
    , declareType   :: Name -> Integer -> IO Result
    , defineType    :: Name -> [Name] -> Type -> IO Result
    , declareFun    :: Name -> [Type] -> Type -> IO Result
    , defineFun     :: Name -> [Binder] -> Type -> Expr -> IO Result
    , push          :: Integer -> IO Result
    , pop           :: Integer -> IO Result
    , assert        :: Expr -> IO Result
    , checkSat      :: IO Result
    , getAssertions :: IO Result
    , getValue      :: [Expr] -> IO Result
    , getProof      :: IO Result
    , getUnsatCore  :: IO Result
    , getInfo       :: InfoFlag -> IO Result
    , getOption     :: Name -> IO Result
    , exit          :: IO Result
    }
```

Listing 1: Avaliable commands to send to the solver

Both the defined API and the auxiliary functions provided by the SMTLib2 package are what we call the low level API. This API was the first to be embodied in the library and the aim of it is to be used by software that generates the expressions and uses the library to communicate with the solvers. If the user intends to write source code, it will be easier if he uses what we consider the high level API, which contains functions designed to provide some common functionality, for instance *mapDeclConst* found at Listing 2 declares a list of constants of the same sort.

In order to select the solver to use and its interaction mode there is the function *startSolver* defined in Listing 3 which returns an abstraction of a solver that is later passed to the other commands. The function receives as arguments one of the available solvers, the mode of operation (*online* or *script*), the logic

[11] https://hackage.haskell.org/package/SMTLib2

```
mapDeclConst :: Solver -> [String] -> Type -> IO ()
mapDeclConst _ [] _ = return ()
mapDeclConst solver (x:xs) y =
    declConst solver x y >> mapDeclConst solver xs y
```

Listing 2: Definition of the function *mapDeclConst*

to be used in the solver, and two extra optional parameters. The first, *Maybe SolverConfig* allows the user to use a solver with a different configuration (note that by doing this the new configuration may alter the behavior of the solver and the library may not work as expected). The last parameter allows the user define where the script will be created in script mode and is ignored in *online mode*. If *Nothing* is passed to this last two parameters the library uses the predefined configuration for the solvers.

```
startSolver :: Solvers
            -> Mode
            -> Logic
            -> Maybe SolverConfig
            -> Maybe String
            -> IO Solver
```

Listing 3: Type of the function *startSolver*

To illustrate the use of the library the Listing 4 presents a working program, which initiates the Z3 solver in *online mode*, declares the constant x and y of type *Int*, asserts that if x is greater than y then y is less than x, checks for satisfiability of the formulas asserted and finally exits the solver. The function *main* returns a value of type *Result* that will be explained in subsection 2.4. Also the function *ct* used at line 11 is part of the high-level API and is used to declare a constant.

2.2 Solver layer

The solver layer is the core layer since it binds all the other layers, by implementing the API defined in the API layer and by using the functions specific to each solver provided by the communication layer. It also uses the response layer to turn the string that comes from the communication layer into an Haskell data type.

At this level, for each solver supported there is a source file which contains its standard configuration and implements the functions defined in the data type *Solver*, presented at Listing 1, according to the solver specific behavior and supported modes of interaction.

```haskell
import Hsmtlib
import Hsmtlib.Solver
import Hsmtlib.HighLevel
import SMTLib2.Int
import SMTLib2.Core

main :: IO Result
main = do
    solver <- startSolver Z3 Online QF_IDL Nothing Nothing
    mapDeclConst solver ["x", "y"] tInt
    assert solver $ ct "x" `nGt` ct "y" ==> ct "y" `nLt` ct "x"
    checkSat solver
    exit solver
```

Listing 4: Example of the library.

A solver configuration is a simple record, *solverConfig* as shown in Listing 5, which has the path to the solver and the flags to set the desired behavior.

```haskell
data SolverConfig = Config
                  { path :: String
                  , args :: [String]
                  }
```

Listing 5: Solver configuration record

An alternative configuration can be used when a solver is initialized, by giving the function *startSolver* a value of this data type. In Listing 6 it is shown how to create a configuration different from the predefined one. In this case each query is killed after 2 seconds. The configuration can be later used as input in the function *startSolver*.

2.3 Communication layer

This module offers two ways of interacting with the solvers. In online mode the interaction is made by forking a solver as an external process and communicating via pipes. Subsequently each interaction with the solver sends the command as a string by pipe and then blocks until a response is given. The alternative mode of operation is script mode where each time the program needs to communicate with a solver it initiates an external solver, passes the file that contains the input for the solver as an argument and then waits for the solver to end and reads the

```
import HSMTLib2
import HSMTLib2.Solver

z3Timeout :: Maybe SolverConfig
z3Timeout = Just $
    Config { path = "z3"
           , args = ["-smt2","-in", "-t:2"]
           }
```

Listing 6: Add Timeout

output. This last mode simple uses the function *readProcess*[12] from the Haskell Process library.

Since some solvers offer similar ways of interaction we only need a general function for each type of communication. However for those solvers that have an alternative behavior, this module also offers specific functions for the solvers. To be more concrete to use online mode with Z3 or MathSAT only one function is needed, but CVC4 presents different behaviors in Windows and Linux therefore the need for a specific solver functions arises.

2.4 Response layer

According to the AST defined by SMTLib2 we use parsec[13] to construct the AST from the string obtained by communication layer. For most responses no extra work is done, for functions that do not demand feedback such as *setLogic* or *push*, the returned values are either *Success*, *Unsupported* or *Error* with a message. For those commands that do demand an answer the AST is returned expect for the command *getValue*. When a command to check the satisfiability is issued the results are, as expected, *Sat*, *Unsat* or *Unknown*.

For the command *getValue* the response layer attempts to give a special result not defined in the standard. For this particular result the library goes through the AST, and collects the values it can turn to a simpler type, for example if the value of a function is requested the result is a value type *Res*, shown in Listing 7, with the name of the constant and the value retrieved. When the result of an array is detect, it is turned into a map which associates the name of the array with another map that is itself an association between the position in the array and the value. Before delivering the final result, it unites all the maps in a single one. For all the values for which the previous algorithm does not work it simply returns the AST created by parsec. In the case that multiple values are requested a list of results is returned.

[12] http://hackage.haskell.org/package/process-1.2.0.0/docs/System-Process.html#v:readProcess

[13] http://www.haskell.org/haskellwiki/Parsec

All commands return a value of type *Result*, presented in Listing 7, which has a constructor for each command. For instance the command *getUnsatCore* returns a CGUC (Command Get Unsat Core) with the constructed AST.

```
data Result = CGR GenResponse
            | CGI GetInfoResponse
            | CCS CheckSatResponse
            | CGAssert GetAssertion
            | CGP GetProofResponse
            | CGUC GetUnsatCoreResponse
            | CGV [GValResult]
            | CGAssig GetAssignmentResponse
            | CGO GetOptionResponse
            | ComError String -- Error comunicating with the smt or Parsing
```

Listing 7: Result data type.

```
data GValResult = Res String Value
                | VArrays Arrays -- (Array Name,(Position, Value))
                | Results [GValResult]
                | Synt GetValueResponse (Syntax tree)
```

Listing 8: Result of getValue.

If the example presented in Listing 4 was executed the answer would be

```
Main > main
CCS Sat
CGR Success
```

The first *CCS Sat*, means the result from the command *Check Sat(CCS) solver* is Sat. The second result, *CGR Success*, is the result from the command *exit solver* which is show by *GHCi*[14] because it is the result from the function *main* and reports to the user that the solver was successfully terminated.

3 Use case example: VCGen

We will now illustrate the use of HSMTLib with a slightly bigger example. Deductive verification based on Hoare logic, to prove the correctness of imperative programs w.r.t. some specification, is usually done via a Verification conditions

[14] http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html

generator (VCGen) that generates proof obligations to be discharged by a proof tool. Here it is shown how to operate with HSMTLib to communicate with a SMT solver to discharge the verification conditions (VCs) generated by a VC-Gen for a small While language (the full code is available at Github[15]) The VCs are encoded in the data type *LogExp*, presented in Listing 9, which supports all common logic connectors and quantifiers. For the sake of simplicity the only predicates considered are the usual relational operators between integers. The type *AExp* which is not presented here is a simple type that encode arithmetic expressions over integers.

```
data LogExp = BConst Bool
            | Not LogExp
            | Forall String LogExp
            | Exists String LogExp
            | LogBin LogOp LogExp LogExp
            | IneBin IneOp AExp AExp

data LogOp = And | Or | Imp

data IneOp = Lt | Gt | Leq | Geq | Equal | Diff
```

Listing 9: Syntax in Haskell.

Consider the following annotated program where the assertions "pre:" "inv:" and "pos:", are the pre-condition, the invariant of the loop and the post-condition respectively.

```
pre: x > 100;
while(x < 1000){
    inv: 100 < x &&  x <= 1000;
    x = x + 1;
}
pos: x > 1000;
```

The VCs generated by the VCGen algorithm based on the weakest precondition [FP11] are the following

1. $x > 100 \Rightarrow 100 < x \wedge x \leq 1000$
2. $100 < x \wedge x \leq 1000 \wedge x < 1000 \Rightarrow 100 < x + 1 \wedge x + 1 \leq 100$
3. $100 < x \wedge x \leq 1000 \wedge \neg(x < 1000) \Rightarrow x = 1000$

These VCs are terms of type *LogExp*.

Taking into account that the function *assert* from the HSMTLib API takes as input values of the type *Expr*, the type *LogExp* needs to be converted so the VCs can be asserted. This conversion can be done by using the functions provided by the SMTLIB package as shown in Listing 10.

[15] https://github.com/MfesGA/Vcgen

```
import SMTLib2 as SL
import SMTLib2.Core as C

createSexpr :: LogExp -> SL.Expr
createSexpr (BConst b) = boolToExpr b
createSexpr (Not a) =  C.not (createSexpr a)
createSexpr (S.Forall s expr) = forall [bind s tInt] (createSexpr expr)
createSexpr (S.Exists s expr) = exists [bind s tInt] (createSexpr expr)
createSexpr (LogBin logop expr1 expr2) = logBinToExpr logop expr1 expr2
createSexpr (IneBin ineop axp1 axp2) = ineBinToExpr ineop axp1 axp2

logBinToExpr :: LogOp -> LogExp -> LogExp -> SL.Expr
logBinToExpr And expr1 expr2 = C.and (createSexpr expr1) (createSexpr expr2)
logBinToExpr Or expr1 expr2 = C.or (createSexpr expr1) (createSexpr expr2)
logBinToExpr Imp expr1 expr2 = createSexpr expr1 ==> createSexpr expr2

boolToExpr :: Bool -> SL.Expr
boolToExpr True = true
boolToExpr False = false

ineBinToExpr :: IneOp -> AExp -> AExp -> SL.Expr
ineBinToExpr Equal aexp1 aexp2 =  aExpToExpr aexp1 ===  aExpToExpr aexp2
ineBinToExpr Diff aexp1 aexp2 = aExpToExpr aexp1 =/=  aExpToExpr aexp2
ineBinToExpr Lt aexp1 aexp2 = nLt (aExpToExpr aexp1) (aExpToExpr aexp2)
ineBinToExpr Gt aexp1 aexp2 = nGt (aExpToExpr aexp1) (aExpToExpr aexp2)
ineBinToExpr Leq aexp1 aexp2 = nLeq (aExpToExpr aexp1) (aExpToExpr aexp2)
ineBinToExpr Geq aexp1 aexp2 = nGeq (aExpToExpr aexp1) (aExpToExpr aexp2)
```

Listing 10: Maping LogExp to Expr.

The Listing 11 contains some functions which can be used to construct the VCGen. Let us now explain each of these functions.

To use a SMT-Solver to check the validity of a logical formula ϕ we need to check the satisfiabillity of $\neg\phi$ since ϕ is valid iff $\neg\phi$ is unsatisfiable. After converting the condition its negation must be asserted in the SMT solver by using the function *assert* as presented in function *assertExpr*. Moreover, the constants used in the formula must be declared. The function *getConstants* collects the constants used in a formula in a list of strings and *MapDeclConst* makes the declaration of the constants as integers in the solver which is exemplified in the function *decConstants*. Function *check* makes the constant declaration the assert of the formula and asks for a satisfiability check to the solver.

If the examples were executed with the negation of conditions 2 and 3 previously presented it would return the result *CCS Unsat* which means that they are valid. The first condition would return *CCS Sat* which means there exists a model in which the condition does not hold. In order to get a model the function *getValue* could be used and it would return

```
assertExpr :: Solver -> LogExp -> IO Result
assertExpr solver expr = assert solver  $ createSexpr expr

decConstants :: Solver -> LogExp -> IO Result
decConstants solver expr = mapDeclConst solver (getConstants expr) tInt

check ::Solver -> LogExp -> IO Result
check solver expr =
    decConstants solver expr >> assertExpr solver expr >> checkSat solver

getValX :: Solver -> Result
getValX solver = getValue Solver [ct "x"]
```

Listing 11: Auxiliary functions

```
CGV [RES "X" (VInt 1001), Synt []]
```

giving the value 1001 to x.

4 Testing the library

HSMTLib was tested by two methods. In the first a collection of tests written by hand were created using the HUnit[16] package in order to test specific parts of the code, mainly the commands that demand a value such as *getValue*. The second consisted on parsing some benchmarks provided by SMT-LIB and converting them into a Haskell program which uses our library.

From the battery of tests several errors were detected, CVC4 has a different behavior both on Linux and Windows. On Linux, when communicating by pipes, the command sent to the input pipe also appears in the output pipe, creating the need to discard the first result read from the pipe. However on Windows such behavior does not happen. Only the most recent version of CVC4 is supported (version 1.3), due to the fact that previous version behave in a different manner. On Z3 in script mode an unwanted behavior is still present in the library, when a value is requested from the command getValue the response is given in separate lines which makes impossible to decide in the current state which output is the correct. As a result of not being able to get the correct value when requesting several values using the command *getValue*, it is advised to request one value at a time. MathSAT does not present problems in online mode on both Windows and Linux, although script mode is not supported in either platform. This feature is not implemented as a result of not having an option to give a file and needing to redirect the file to its input. Even though we initially supported Yices [DM06] and Alt-Ergo [Bob+08], through our testing they behaved very different from what were expected and even being capable of interacting with them by pipes or files, their response was not compliant with SMT-LIB2 standard.

[16] https://hackage.haskell.org/package/HUnit

5 Conclusion

As exposed, HSMTLib provides a set of functions to easily interact with SMT-solvers and two modes of interaction with predictable behavior. The library was tested by different methods and is currently compatible with Z3, MathSat and CVC4. Additional solvers can be included as long as they follow the guidelines of SMT-LIB, otherwise the addition of a new one might need some extra work especially in the communication layer.

The approach taken in splitting the library in several layers allows a lot of flexibility to add additional features. For instance, adding support to the native Z3 API. In order to achieve this goal, first a wrapper needs to be created for all functions provided by Z3 native API which would be added to the communication layer and then map them to the SMT-LIB functions which would be done in the solver layer. Both API and response layer would remain untouched.

A few improvements could be done in the response layer, since the same response from different solvers might create a different syntax tree and all this cases should be covered in order to give a consistent answer across all solvers.

We believe the library is ready to be used by other software and free the user from repeating the task to create software that interacts with the SMT solvers and focus on creating the tool he expected. Software such as Liquid Haskell, Cryptol and the SBV package are examples of applications that could use this library. HSMTLib can also be used as a more friendly way to create expressions for asserting or declaring functions that are much easier to read and think about than the S-expressions that need to be written to send to the solver.

In the next version we plan to run multiple solvers concurrently in order to either issue the same command or a different one to each solver. This means we could take advantage of the individual strength of each solver. For instance, one solver might not be able to answer a query while another solver could.

We also plan to add stable support to Yices, Boolector and Alt-Ergo.

References

[Bar+11] Clark Barrett et al. "CVC4." In: *CAV*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22109-5.

[Bob+08] François Bobot et al. *The Alt-Ergo Automated Theorem Prover*. `http://alt-ergo.lri.fr/`. 2008.

[Boz+05] Marco Bozzano et al. "MathSAT: Tight Integration of SAT and Mathematical Decision Procedures." In: *J. Autom. Reasoning* 35.1-3 (2005), pp. 265–293.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard Version 2.0." In: *Department of Computer Science The University of Iowa Tech Rep* (2010), pp. 1–43. URL: `http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.03.30.pdf`.

[Cok11] David R Cok. "The SMT-LIBv2 Language and Tools : A Tutorial." In: *Language* c (2011), pp. 2010–2011.

[DM06] Bruno Dutertre and Leonardo de Moura. "The YICES SMT Solver." In: *Citeseer* (2006), pp. 1–5. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7567&rep=rep1&type=pdf`.

[FP11] Maria João Frade and Jorge Sousa Pinto. "Verification conditions for source-level imperative programs." In: *Computer Science Review* 5.3 (2011), pp. 252–277.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0.