IPASIR-2: Re-entrant Incremental SAT Solver API 2.0

Abstract

This paper describes the second version of the incremental SAT solver interface IPASIR, which is a major revision of the first version. IPASIR-2 introduces error codes, a solver configuration interface, and methods to support clause sharing and proof logging.

KEYWORDS: *incremental, satisfiability, solver, interface*

1. Introduction

IPASIR [1] is a C API for incremental SAT solvers. This paper describes the second version of the IPASIR interface, which is a major revision of the first version. The following major requirements have been addressed in IPASIR-2.

Error Codes. The introduction of error codes makes it possible to distinguish between different types of error. All functions in IPASIR-2 return an error code and data is returned via output parameters.

Solver Configuration Interface. Many SAT solvers have a large number of configuration options. IPASIR-2 introduces a solver configuration interface to allow these options to be set. Some configuration options are universal and therefore specified in the standard, others are solver specific. The interface is designed to be generic and extensible to support a wide range of solver configuration options.

Clause Sharing and Proof Logging. While IPASIR already supported clause export callbacks, IPASIR-2 introduces clause import and clause delete callbacks. This allows IPASIR-2 solvers to be used in clause sharing portfolios and to log drat proofs. For more advanced proof logging, e.g. LRAT, IPASIR-2 allows proof metadata to be passed with each clause. This metadata is method specific and can be configured by the user.

Fact Sharing. IPASIR-2 introduces a callback to notify about fixed assignments. This is useful for applications that require the solver to notify about literals that are implied by the formula.

2. Interface Description

This section describes the IPASIR-2 user interface. We divide the interface into the following functional categories: core functions, solver configuration, clause sharing and proof logging, and fact sharing. Each category is described in a separate subsection. In addition to a functional specification, we provide a brief rationale for the design choices made.

2.1. General Definitions

2.1.1. Thread Safety

The IPASIR-2 specification allows multiple solvers to be initialised and used in parallel. The intention is to cover use cases where the client starts an arbitrary number of solver instances

in parallel, but interacts with each solver instance only sequentially. All IPASIR-2 functions that take a solver instance S as argument must not be called on the same solver instance S concurrently. If S calls a callback function, the client may only access S from the callback of the calling thread. Solver instances are only allowed to call callback functions during `ipasir2_solve`, and only from the same thread as the client calling `ipasir2_solve`. This is to keep the client code simple and to avoid compatibility issues, for example when IPASIR-2 is used in scripting languages. IPASIR-2 implementations may provide custom options to replace these thread safety requirements.

2.1.2. Error Codes

IPASIR-2 functions return error codes to indicate the success or failure of a function call. The error codes are defined in the `ipasir2_errorcode` enum type. The following error codes are defined:

- `IPASIR2_E_OK`: Success.
  The function call was successful.
- `IPASIR2_E_UNKNOWN`: Generic error.
  The function call failed for an unknown reason.
- `IPASIR2_E_UNSUPPORTED`: Unsupported function.
  The function is not implemented by the solver.
- `IPASIR2_E_UNSUPPORTED_ARGUMENT`: Unsupported argument.
  The function is not implemented for handling the given argument value.
- `IPASIR2_E_UNSUPPORTED_OPTION`: Unknown option.
  The configuration option is not implemented by the solver.
- `IPASIR2_E_INVALID_STATE`: Invalid state.
  The function call is not allowed in the current state of the solver.
- `IPASIR2_E_INVALID_ARGUMENT`: Invalid argument.
  The given argument value is invalid.
- `IPASIR2_E_INVALID_OPTION_VALUE`: Invalid option value.
  The option value is outside the allowed range.

2.1.3. Solver States

The state of the IPASIR-2 solver is defined by the state of the underlying state machine. State transitions are triggered by IPASIR-2 function calls. The state machine is initialized in the CONFIG state. This is new in IPASIR-2 and allows for setting configuration options before adding clauses. Functions are only allowed to be called in the states specified in the documentation. If a function is called in the wrong state, the function returns `IPASIR2_E_INVALID_STATE`. The following states are defined:

- `IPASIR2_S_CONFIG`: Configuration state.
- `IPASIR2_S_INPUT`: Input state.
- `IPASIR2_S_SAT`: Satisfiable state.
- `IPASIR2_S_UNSAT`: Unsatisfiable state.
- `IPASIR2_S_SOLVING`: Solving state.

Another innovation in IPASIR-2 is that states are partially ordered. This allows for specifying a maximal state in which a function can be called, or in which a configuration option can be set. The partial order is defined as `CONFIG` < `INPUT` = `SAT` = `UNSAT` < `SOLVING`.

```
ipasir2_signature(char** signature)

ipasir2_init(void** S)
ipasir2_release(void* S)

ipasir2_add(void* S, int32_t* clause, int len, int forgetable, void*
proofmeta)
ipasir2_solve(void* S, int32_t* assumps, int len, int* result)
ipasir2_value(void* S, int32_t lit, int* result)
ipasir2_failed(void* S, int32_t lit, int* result)

ipasir2_set_terminate(void* S, void* data, int (*callback)(void* data))
```

Figure 1. IPASIR-2 Core Functions

2.2. Core Functions

The core functions of IPASIR-2 are essentially the same as those of IPASIR. A major difference is that all functions return an error code which is `IPASIR2_E_OK` if the fThe function `ipasir2_init` constructs a new solver instance and returns a pointer to it in the output parameter. The state of the returned solver is initialized to `CONFIG`. The function `ipasir2_signature` returns the name and version of the IPASIR-2 solver in the output parameter. It can be called at any time and also simultaneously from any thread.

The `ipasir2_init` function constructs a new solver instance and returns a pointer to it in the output parameter. The state of the returned solver is initialised to `CONFIG`. Multiple solver instances can be created in parallel. The `ipasir2_release` function destroys the given solver instance, freeing all solver resources and allocated memory.

The `ipasir2_add` function adds a `clause` to the formula. The clause is a pointer to an array of literals of the given `length`. If `forgettable` is set to zero, the solver guarantees that the clause will be satisfied in any model it finds, otherwise the solver is allowed to remove the clause from the formula. The `proofmeta` parameter points to a struct containing additional proof metadata. The struct type and its semantics are specific to the selected proof method and are specified in the configuration options.

The function `ipasir2_solve` solves the formula under the given `assumption` literals. As long as the function is running, the solver is in the `SOLVING` state. If the solver calls one of the callback functions during the execution of `ipasir2_solve`, the state of the solver is also `SOLVING`. Callbacks are allowed to execute solver functions that are allowed in the `SOLVING` state. The function returns the result of the search in the output parameter `result`. If the formula is satisfiable, the output parameter result is set to 10 and the solver state is changed to `SAT`. If the formula is unsatisfiable, the output parameter result is set to 20 and the solver state is changed to `UNSAT`. If the search is aborted, the output parameter result is set to 0 and the solver state is changed to `INPUT`.

The `ipasir2_value` function can only be used when the solver is in the `SAT` state and returns the truth value of the given literal in the satisfying assignment found. The `result` output parameter is set to `lit` if `lit` is satisfied by the model, and to `-lit` if `lit` is not satisfied by the model. The result output parameter can be set to zero if the assignment found supports both values for the literal.

The function `ipasir2_failed` can only be used when the solver is in the `UNSAT` state and indicates whether the given assumption `literal` was used to prove unsatisfiability. The

```
ipasir2_options(void* S, ipasir2_option** options, int* count)
ipasir2_set_option(void* S, ipasir2_option* handle, int64_t value, int64_t
index)
```

Figure 2. IPASIR-2 Configuration Interface

output parameter `result` is set to 1 if the given assumption literal was used to prove unsatisfiability, otherwise it is set to 0. The set of assumption literals for which the result of this function call is 1 forms an unsatisfiable core of the formula.

The function `ipasir2_set_terminate` sets a callback function that is used to indicate a termination requirement to the solver. The solver calls this function periodically while in the `SOLVING` state. When the callback function is called, the given opaque `data` pointer is passed to the callback as its first argument. If the callback function returns a non-zero value, the solver terminates the search.

2.3. Configuration Interface

2.3.1. The Option Struct

The `ipasir2_option` struct is used to define solver configuration options.

NAME. In IPASIR-2, options are identified by a string using dot-separated namespaces for structuring. The top level namespace "ipasir" is reserved for the options specified in the IPASIR-2 standard. If an IPASIR-2 solver supports a standard option, it is guaranteed to have the name specified in the standard. It is recommended, but not mandatory, that IPASIR-2 solvers support all standard options. Other proprietary options may be specified in the solver documentation, but must not be in the "ipasir" namespace.

MIN and MAX values. The 'min' and 'max' fields specify the range of values that the option can take. To simplify the interface, all option values are given as 64-bit integers. Solvers using different option types must map them internally.

MAX_STATE. The 'max_state' field specifies the maximum solver state in which the option can be set. While some options can be set at any time, e.g., from a callback during the `SOLVING` state, others can only be set in the `CONFIG` or `INPUT` states.

TUNABLE. The 'tunable' field indicates whether the option is eligible for tuning by an automated configuration tuning algorithm.

INDEXED. Many options are global and can only be set once for the solver. However, some options can be set per variable or other types of indices. The 'indexed' field indicates whether the option can be set per variable or other types of indices. In this case the index parameter of `ipasir2_set_option` specifies the index.

HANDLE. The 'handle' is used by the option setter to identify the option.

2.3.2. Option Getters and Setters

The `ipasir2_options` function returns an array of configuration options supported by the solver in the `options` output parameter, and the `count` output parameter contains the number of elements in the array. The array is owned by the solver and must not be freed by the caller.

```
ipasir2_set_export(void* S, void* data, int max_length, void
(*callback)(void* data, int32_t const* clause, int32_t len, void*
proofmeta))
ipasir2_set_import(void* S, void* data, void (*callback)(void* data))
ipasir2_set_delete(void* S, void* data, void (*callback)(void* data, int32_t
const* clause, int32_t len, void* proofmeta))
```

Figure 3. IPASIR-2 Clause Sharing Interface

```
ipasir2_set_fixed(void* S, void* data, void (*callback)(void* data, int32_t
fixed))
```

Figure 4. IPASIR-2 Fact Sharing Interface

The `ipasir2_set_option` function sets the `value` of the option specified by the given `handle`. If the option is indexed, the `index` parameter specifies the variable on which the option is set, otherwise the index parameter is ignored. The value must be in the allowed range as specified in the option handle. The solver must be in a state where the option can be set as specified in the option handle.

2.4. Clause Sharing Interface

The function `ipasir2_set_export` sets a callback function to receive learned clauses from the solver. The solver calls this callback function in the `SOLVING` state for each learned clause whose size is less than `max_length` or if `max_length` is -1. The opaque `data` pointer is passed to the callback function as its first parameter. The `clause` parameter is a pointer to the learned clause of length `len` and is only valid during the execution of the callback function. The `proofmeta` parameter points to a struct containing additional proof metadata. The struct type and its semantics are specific to the selected proof method and are specified in the configuration options.

The function `ipasir2_set_import` sets a callback function for importing a clause into the solver. This callback is called periodically while the solver is in the `SOLVING` state. The opaque `data` pointer is passed to the callback function as its first parameter. The callback function uses `ipasir2_add` to add a clause to the solver. This roundtrip is necessary to resolve any ownership issues with the clause pointer. To import more than one clause, the callback must be called several times. If there are no more clauses to import, the callback returns without calling `ipasir2_add`.

The function `ipasir2_set_delete` sets a callback function for notification of deleted clauses. The solver calls this function in the `SOLVING` state for each clause deleted from the formula. The opaque `data` pointer is passed to the callback function as its first parameter. The `clause` parameter is a pointer to the deleted clause of length `len` and is only valid during the execution of the callback function. The `proofmeta` parameter points to a struct containing additional proof metadata. The struct type and its semantics are specific to the selected proof method and are specified in the configuration options.

2.5. Fact Sharing Interface

The function `ipasir2_set_fixed` sets a callback to notify about fixed assignments. The solver calls this function while being in the `SOLVING` state, whenever it determines that a literal is implied by the formula and thus true in any model. If the option `ipasir.assumptions.fixed` is set to 1, the callback notifies about literals implied by the formala *emph* assumptions (cf. Section 3.1.3). The function does not return the fixed literals in any particular order, nor does it guarantee that all fixed literals are reported.

3. Standard Configuration Options

This section describes the standard configuration options that are part of the IPASIR-2 standard. Solvers are not required to support all options, but if they do, they must use the names specified in the standard. The options are divided into tunable and non-tunable options (Section 3.1). Of the non-tunable options, we treat the variable specific options separately (Section 3.2).

3.1. Non-Tunable Options

Non-tunable options are options which are not intended to be changed by configuration tuning algorithms. Their function is to set the behaviour of the solver to a specific mode or to configure the solver for a specific use case.

3.1.1. Setting of Limits

Use cases of limits configuration include local search for combinatorial optimization [2], the determination of implied facts (cf. Section 2.5), and the setting of deterministic solve limits.

- `ipasir.limits.conflicts = n`
  - `n = -1` no conflict limit (default)
  - otherwise exit when number of conflicts exceeds n
- `ipasir.limits.decisions = n`
  - `n = -1` no decision limit (default)
  - otherwise exit when number of decisions exceeds n

3.1.2. Non-incremental Solving Mode

Even though IPASIR is designed for incremental solving, there are use cases where a non-incremental solving mode is beneficial. This includes the use of one-shot solving in parallel portfolios [4], automatic configuration optimization [6], and the use of SAT solvers as a library in other applications. If the option is activated, only a single call to `ipasir2_solve` is effective while further calls lead to the solver returning `IPASIR2_E_INVALID_STATE`.

- `ipasir.yolo = n`
  - `n = 0` incremental mode (default)
  - `n = 1` “you only live once” mode

3.1.3. Treat Assumptions as Fixed

Normally the `fixed` callback only notifies about literals implied by the formula (cf. Section 2.5). With this option enabled, it notifies about literals implied by the formula *and* the assumptions. Applications include combinatorial optimization [2].

- `ipasir.assumptions.fixed = n`
  - `n=0` do not treat assumptions as fixed (default)
  - `n=1` treat assumptions as fixed

3.2. Variable-Specific Options

Some options need to be set per variable, e.g., to set variable scores, variable phases, or to freeze variables. For such use cases, the `index` parameter of `ipasir2_set_option` can be used to specify the variable id. If `index` is set to 0, the option is set globally for all variables.

3.2.1. Initializing Phases

Use cases for setting initial phases include the activation of zero-first branching (as in Minisat [3]), application-specific phase initialization heuristics, or search diversification in parallel portfolios [4].

- `ipasir.variables.phase.initial = n`
  - `n = 0` use default initial phase (default)
  - `n = -1` set initial phase to false
  - `n = 1` set initial phase to true

3.2.2. Fixing Phases

Fixing phases means to force a solver to always make a certain decision for a variable. Use cases for fixing phases include combinatorial optimization [2].

- `ipasir.variables.phase.fixed = n`
  - `n = 0` do not fix phases (default)
  - `n = -1` fix phase to false
  - `n = 1` fix phase to true

3.2.3. Initializing Branching Order

Use cases for setting initial branching order includes the setting of application specific branching priorities [7]. The solver guarantees that the initial branching order aligns with the ordering induced by the variable scores set by the user.

- `ipasir.variables.score.initial = n`
  - set the initial variable score to `n`

3.2.4. Freezing Variables

Some simplification methods, such as variable elimination, require additional care to be taken if the solver is used incrementally. For example, eliminated variables also have to be restored when they are used as assumptions or when they appear in clauses added between two calls to `ipasir2_solve`. In some applications it is forseable which variables will be used as assumptions or in added clauses. As a performance optimization, such applications can

set variables to a frozen state to entirely prevent the solver from eliminating them, thus preventing forseable on-demand restoring of clauses from the elimination stack.

- `ipasir.variables.frozen = n`
  - `n=0` disable frozen state (default)
  - `n=1` enable frozen state

3.3. Tunable Options

3.3.1. Propagation of Assumptions

Specifies if assumptions are propagated one by one or all at once as described in [5].

- `ipasir.assumptions.propagate = n`
  - `n=0` propagate one assumption per decision level (default)
  - `n=1` propagate all assumptions *at once*

References

[1] T. Balyo, A. Biere, M. Iser and C. Sinz, SAT Race 2015, *Artif. Intell.* **241** (2016), 45–65. https://doi.org/10.1016/j.artint.2016.08.007.

[2] A. Cohen, A. Nadel and V. Ryvchin, Local Search with a SAT Oracle for Combinatorial Optimization, in: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, 2021. https://doi.org/10.1007/978-3-030-72013-1_5.

[3] N. Eén and N. Sörensson, An Extensible SAT-solver, in: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT*, 2003. https://doi.org/10.1007/978-3-540-24605-3_37.

[4] L. Guo, Y. Hamadi, S. Jabbour and L. Sais, Diversification and Intensification in Parallel SAT Solving, in: *Principles and Practice of Constraint Programming, CP*, 2010. https://doi.org/10.1007/978-3-642-15396-9_22.

[5] R. Hickey and F. Bacchus, Speeding Up Assumption-Based SAT, in: *Theory and Applications of Satisfiability Testing, SAT*, M. Janota and I. Lynce, eds, 2019. https://doi.org/10.1007/978-3-030-24258-9_11.

[6] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H.H. Hoos and K. Leyton-Brown, The Configurable SAT Solver Challenge (CSSC), *Artif. Intell.* **243** (2017), 1–25. https://doi.org/10.1016/j.artint.2016.09.006.

[7] M. Iser, M. Taghdiri and C. Sinz, Optimizing MiniSAT Variable Orderings for the Relational Model Finder Kodkod, in: *Theory and Applications of Satisfiability Testing, SAT*, A. Cimatti and R. Sebastiani, eds, 2012. https://doi.org/10.1007/978-3-642-31612-8_46.