All JSON data elements are specified in js-schema.

JANI-based tool interaction always involves a server tool (e.g. a model checker) and a client tool (e.g. a graphical user interface) that initiates the communication with the server tool. Communication can take place in two different ways:

1) **Standard streams**
The client starts the server tool (if necessary with a command-line parameter indicating *jani-interaction* operation, e.g. "--jani") as a new process. The two tools then exchange JANI messages over the server tool's standard input and output streams. Each message is encoded as UTF-8 without byte order mark and written as a single line onto standard input/output (i.e. the message itself must not contain line break characters). This way, a single message can be read from standard input/output by reading one line of input/output.
Note: Implementations must take care not to enter into deadlock situations due to buffers filling up when writing to standard output without being ready to concurrently read from standard input or vice-versa.

2) **Websockets**
The server tool runs independently and accepts incoming WebSocket connections (on a specified port, or port 15291 (unencrypted) / 15292 (secure WebSocket) if not specified). The server tool may limit the maximum number of concurrent active connections to one. The two tools then exchange JANI messages through WebSocket "text" messages. Each JANI message is sent as a single (but possibly fragmented) WebSocket "text" message.

Unique identifiers

Most messages contain a "unique identifier" field named "id". Unique identifiers start at 1, not 0: This allows tools to use "id" 0 for special purposes internally (e.g. the mime GUI uses it to internally refer to the **Authenticate** and **Capabilities** messages). Furthermore, all top-level unique identifiers in messages must be globally unique, i.e. they must not be reused even for different classes of messages (e.g. queries, tasks etc. all share the same unique identifier space).

Protocol state machine

CONNECTED -**Authenticate**-> AUTHENTICATING -**Capabilities**-> INTERACTIVE
| |
Close **Close**
| |
V V
CLOSED CLOSED

When either of the communication partners receives a malformed, unexpected or otherwise invalid message or data, it must reply with a **Close** message[1].

[1] *After sending the Close message, the client or server must behave as described in the comments of that message.*

Basic interaction

Definitions:

```
// Identifiers,
// as defined in the jani-model specification
var Identifier = ...

// jani-model features,
// as defined in the jani-model specification
var JaniModelFeature = ...

// jani-model types,
// as defined in the jani-model specification
var JaniModelType = ...

// Identifiers for non-jani-model modelling formalisms;
// identifiers starting with "x-" will not be reserved and are available for internal use
var ModellingFormalism = schema([
  "iosa", // IOSA
  "modest", // Modest
  "pgcl", // pGCL
  "prism", // the PRISM language
  "xsadf" // xSADF XML
]);

// Versions,
// used for tool and analysis engine versions
var Version = schema({
  "major": Number.min(0).step(1),
  "minor": Number.min(0).step(1),
  "revision": Number.min(0).step(1)
});

// Metadata,
// used for tools and analysis engines
var Metadata = schema({
  "name": String, // the object's name, for display to users
  "version": Version, // the object's version, for display to users or compatibility checks
  "?author": String, // information about the creator(s) of the object or short copyright information
  "?description": String, // a description text for the object
  "?url": String, // a URL pointing to more information about the object
});

// Parameter types,
// used in the ParameterDefinition schema below
var ParameterType = schema([
  "bool",
  "int",
  "real",
  "string",
  Array.of(String), // enum, non-empty array of distinct values
  { // option, i.e. values can be of the specified type or null
    "kind": "option",
    "type": schema.self // must not be an option
  }
]);

// Parameter definition (e.g. for the parameters of the server),
// used in the Capabilities and ReplyAnalysisParameters messages
var ParameterDefinition = schema({
  "id": Identifier, // the identifier of the parameter, unique within the respective set of parameters
  "name": String, // a short text label for the parameter, for display to users
  "?description": String, // a longer description of the parameter, for display to users
  "?category": String, // a category string for the parameter, e.g. for grouping in a user interface
  "is-global": [ true, false ] // indicates whether the parameter is "global", i.e. whether it is expected to
                               // usually not vary between different analyses (e.g. paths to external tools),
                               // to e.g. allow user interfaces to show global and 'local' parameters in
                               // different places
  "type": ParameterType,
  "default-value": [
    null, // if type is an option
    true, false, // if type is "bool" or an option with type "bool"
    Number, // if type is "int" or "real" or an option with type "int" or "real"
    String // if type is "string" or an enum or an option with type "string" or an enum
```

```
  ]
});

// Parameter value (e.g. for the parameters of the server),
// used in the RequestUpdateServerParameters and StartAnalysis messages
var ParameterValue = schema({
  "id": Identifier, // the unique identifier of the parameter
  "value": [
    true, false, // if the parameter's type is "bool"
    Number, // if the parameter's type is "int" or "real"
    String // if the parameter's type is "string" or an enum
  ]
});

// jani-interaction extensions
var Extension = schema([
  "persistent-state" // TODO the server keeps state (e.g. running analysis tasks) for the client between connections
]);

// Models (jani-model or textual),
// used in the "analyse" and "transform" roles
var AnyModel = schema([
  Model, // a jani-model model
  { // a model in another modelling formalism as indicated by modelling-formalism
    "name": String, // the name of the model, for display to users (e.g. to associate messages with the model)
    "parts": Array.of({
      "name": String, // the name of the part of the model (e.g. the name of the underlying model part file)
      "role": String, // the role of this part within the model (e.g. "model", "properties", etc.),
                      // defined by the modelling formalisms (e.g. PRISM-style guarded commands models would
                      // have a "model" and a "properties" part corresponding to the .nm and .pctl files)
      "part": String // the model part itself
    })
  }
]);
```

Messages:

```
// Client-to-server, only allowed in CONNECTED state,
// the server must send exactly one Capabilities reply back to the client
// or close the connection with a Close message (e.g. if there is no common jani-version or the login failed)
var Authenticate = schema({
  "jani-versions": Array.of([ Number.min(1).step(1) ]), // the jani-interaction versions supported by the client,
                                                        // at least one
  "?extensions": Array.of(Extension), // the jani-interaction extensions supported by the client
  "?login": String, // optional, can be omitted to indicate anonymous usage
  "?password": String // optional, must be omitted if and only if login is omitted
});

// Server-to-client, only allowed in AUTHENTICATING state
var Capabilities = schema({
  "type": "capabilities",
  "jani-version": Number.min(1).step(1), // the jani-interaction version used by the server,
                                         // must be supported by the client
  "?extensions": Array.of(Extension), // the jani-interaction extensions supported by the server and the client
                                      // that will be used in the communication after this message
  "metadata": Metadata, // metadata for the server tool
  "?parameters": Array.of(ParameterDefinition), // the parameters of the server
  "roles": Array.of([ // the roles supported by the server, at least one;
                      // role names starting with "x-" will not be defined and are available for internal use
    "analyse",
    "transform"
  ])
});

// Server-to-client and client-to-server,
// when standard streams are used, the server process exits after having sent or received this message,
// when WebSockets are used, the WebSocket connection is closed by the receiver of this message
var Close = schema({
  "type": "close",
  "?reason": String // a short description of the reason for closing the connection (e.g. wrong login, ...)
});

// Client-to-server, only allowed in INTERACTIVE state,
// the server must send exactly one ReplyUpdateServerParameters reply with the same id back to the client
var RequestUpdateServerParameters = schema({
  "type": "server-parameters",
  "id": Number.min(1).step(1), // the unique identifier for the request
  "parameters": Array.of(ParameterValue) // the parameter values, at most one element for each
                                         // parameter of the server
});

// Server-to-client, only allowed as a reply to a RequestUpdateServerParameters message
var ReplyUpdateServerParameters = schema({
  "type": "server-parameters",
  "id": Number.min(1).step(1), // the unique identifier of the request that this is a reply for
  "success": [ true, false ], // true if and only if the new parameter values are valid and have been
                              // applied/stored
  "?error": String // the error message, must be present if and only if success is false, for display to users
                   // to describe the error (e.g. parameter values or combinations that are not allowed)
});
```

Tasks

Several roles use the concept of "tasks", which all use the following set of messages:

```
// Server-to-client, only allowed after a message causing a task with the given task identifier to be started has
// been received and before a TaskEnded message for the task identifier has been sent before
var ProvideTaskStatus = schema({
  "type": "task-status",
  "id": Number.min(1).step(1), // the unique identifier of the task
  "status": String // a short description of the current state of the task
});

// Server-to-client, only allowed after a message causing a task with the given task identifier to be started has
// been received and before a TaskEnded message for the task identifier has been sent before
var ProvideTaskProgress = schema({
  "type": "task-progress",
  "id": Number.min(1).step(1), // the unique identifier of the task
  "progress": Number.min(0.0).max(1.0) // the current progress of the task,
                                       // must be monotonically increasing over successive messages
});

// Server-to-client; only allowed after a message causing a task with the given task identifier to be started has
// been received and before a TaskEnded message for the task identifier has been sent before
var ProvideTaskMessage = schema({
  "type": "task-message",
  "id": Number.min(1).step(1), // the unique identifier of the task
  "severity": [ // the severity of the message
    "info", // an informative message
    "warning", // a warning message
    "error" // an error message
  ],
  "message": String // the message text
});

// Client-to-server; only allowed after a message causing a task with the given task identifier to be started has
// been received and before a TaskEnded message for the task identifier has been received; the server must ignore
// StopTask messages that arrive at a point in time where they are not allowed[2]
var StopTask = schema({
  "type": "task-stop",
  "id": Number.min(1).step(1) // the unique identifier of the task
});

// Server-to-client, only allowed after a message causing a task with the given task identifier to be started has
// been received and before a TaskEnded message for the task identifier has been sent before. Every task must
// eventually end with a TaskEnded message (in particular, the server must also send a TaskEnded message when it
// has received a StopTask message).
var TaskEnded = schema({
  "type": "task-end",
  "id": Number.min(1).step(1) // the unique identifier of the task
});
```

[2] *This is in order to avoid closing the connection unnecessarily due to the race condition between TaskEnded from server to client and StopTask from client to server.*

The analyse role

Definitions:

```
// Analysis results,
// used in the ProvideAnalysisResults message
var AnalysisDataSet = schema({ //exactly one of label, experiment and property must be present
  "id": Number.min(1).step(1), // the unique identifier of the analysis result set
  "?label": String, // for display to users[3]
  "?experiment": Identifier, // the identifier of the experiment this this data set provides the results for
  "?property": Identifier, // the name of the property that this data set provides the results for,
                           // where results is present, non-empty, and its first element is the main result
                           // (e.g. the verdict, probability, etc.) for the property
  "?results": Array.of({ // the result values
    "label": String, // the name of the value (e.g. "Probability", "Variance", etc.), for display to users[4]
    "type": [ "bool", "int", "rational", "decimal", "string" ],
    "value": [
      true, false, // if type is "bool"
      Number.step(1), // if type is "int"
      Number, // if type is "decimal"
      { "num": Number.step(1), "den": Number.min(1).step(1) } // if type is "rational", numerator and denominator
      String // if type is "string"
    ],
    "?unit": [
      "s", "ms", // seconds or milliseconds, only for type "int", "rational" or "decimal"
      "B"  // bytes, only for type "int"
    ],
    "?formatted-value": String // a string representation of value, in order to include information like
                               // units that cannot be derived from type in display to users
  }),
  "?data-sets": Array.of(schema.self) // nested data sets
});
```

[3] *Data sets not associated to a property allow, for example, returning general information that applies to the entire analysis process over multiple properties (such as reporting the total number of states of the model if only one state space exploration is done for all properties).*

[4] *A set of standard labels for common kinds of results (probabilities, variances etc.) should be defined in order to improve consistency between tools.*

Messages:

```
// Client-to-server, only allowed in INTERACTIVE state when the server supports the "analyse" role
var QueryAnalysisEngines = schema({
  "type": "analysis-engines",
  "id": Number.min(1).step(1) // the unique identifier for the query
});

// Server-to-client, only allowed as a reply to a QueryAnalysisEngines message
var ReplyAnalysisEngines = schema({
  "type": "analysis-engines",
  "id": Number.min(1).step(1), // the unique identifier of the query that this is a reply for
  "engines": Array.of({ // the analysis engines supported by the server, at least one
    "id": Identifier, // the identifier of the analysis engine, unique among the analysis engines of this server
    "metadata": Metadata, // metadata for the analysis engine
    "?parameters": Array.of(ParameterDefinition), // the parameters of the analysis engine
    "?model-features": Array.of(JaniModelFeature) // the jani-model features supported by the analysis engine
    "?model-types": Array.of(JaniModelType) // the model types supported by the analysis engine
    "?modelling-formalisms": Array.of(ModellingFormalism) // the modelling formalisms supported by the
                                                          // analysis engine beyond jani-model
  })
});

// Client-to-server; only allowed in INTERACTIVE state when the server supports the "analyse" role;
// starts a task (see Tasks) that the server must eventually end with a TaskEnded message (or a Close message)
var StartAnalysisTask = schema({
  "type": "analysis-start",
  "id": Number.min(1).step(1), // the unique identifier for the task
  "engine": Identifier, // the identifier of the analysis engine
  "?modelling-formalism": Identifier, // the name of the modelling formalism supported by the server
                                      // that the model is in if and only if it is not jani-model
  "model": AnyModel, // the model to analyse
  "?properties": Array.of(Identifier), // the names of the properties of the model to be analysed only if
                                       // modelling-formalism is not omitted,
                                       // if omitted all properties shall be analysed
  "parameters": Array.of(ParameterValue), // the parameter values, at most one element for each
                                          // parameter of the analysis engine
  "?experiments": Array.of({ // a set of experiments, i.e. of sets of values for the model's open constants
    "experiment": Identifier, // the identifier of the experiment, unique among all experiments for the task
    "values": Array.of({
      "name": Identifier, // the name of the constant, unique per experiment
      "value": [ true, false, Number ] // the value
    })
  })
});

// Server-to-client, only allowed after a StartAnalysisTask message for the task identifier has been received
// and no TaskEnded message for the task identifier has been sent before
var ProvideAnalysisResults = schema({
  "type": "analysis-results",
  "id": Number.min(1).step(1), // the unique identifier of the task
  "results": AnalysisDataSet // the complete or partial results of the analysis; the analysis result set has an id
                             // itself: if a result set with the same top-level id has been sent before, this set
                             // replaces the previous one (e.g. to allow providing imprecise results early and
                             // replacing them with precise/final results later)
});
```

The transform role

Definitions:

(none yet)

Messages:

```
// Client-to-server, only allowed in INTERACTIVE state when the server supports the "transform" role
var QueryModelTransformers = schema({
  "type": "model-transformers",
  "id": Number.min(1).step(1) // the unique identifier for the query
});

// Server-to-client, only allowed as a reply to a QueryModelTransformers message
var ReplyModelTransformers = schema({
  "type": "model-transformers",
  "id": Number.min(1).step(1), // the unique identifier of the query that this is a reply for
  "transformers": Array.of({ // the model transformers supported by the server, at least one
    "id": Identifier, // the identifier of the model transformer, unique among the model transformers
                      // of this server
    "metadata": Metadata, // metadata for the model transformer
    "?parameters": Array.of(ParameterDefinition), // the parameters of the model transformer
    "?model-features": Array.of(JaniModelFeatures) // the jani-model features supported by the model transformer
  })
});

// Client-to-server; only allowed in INTERACTIVE state when the server supports the "transform" role;
// starts a task (see Tasks) that the server must eventually end with a TaskEnded message (or a Close message)
var StartTransformationTask = schema({
  "type": "transformation-start",
  "id": Number.min(1).step(1), // the unique identifier for the task
  "transformer": Identifier, // the identifier of the model transformer
  "?modelling-formalism": Identifier, // the name of the modelling formalism supported by the server
                                      // that the model is in if and only if it is not jani-model
  "model": AnyModel, // the model to transform
  "parameters": Array.of(ParameterValue) // the parameter values, at most one element for each
                                         // parameter of the model transformer
});

// Server-to-client; only allowed after a StartTransformationTask message for the task identifier has been received,
// no TaskEnded message for the task identifier has been sent before, and no ProvideTransformationResults message
// for the task identifier has been sent before
var ProvideTransformationResults = schema({
  "type": "transformation-results",
  "id": Number.min(1).step(1), // the unique identifier of the task
  "?modelling-formalism": Identifier, // the name of the modelling formalism
                                      // that the model is in if and only if it is not jani-model
  "model": AnyModel // the result of the transformation
});
```