JANI: Quantitative Model and Tool Interaction

Carlos E. Budde[1], Christian Dehnert[2], Ernst Moritz Hahn[3], Arnd Hartmanns[4], Sebastian Junges[2], and Andrea Turrini[3]

[1] Universidad Nacional de Córdoba, Córdoba, Argentina
[2] RWTH Aachen University, Aachen, Germany
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4] University of Twente, Enschede, The Netherlands

Abstract The formal analysis of critical systems is supported by a vast space of modelling formalisms and tools. The variety of incompatible formats and tools however poses a significant challenge to practical adoption as well as continued research. In this paper, we propose the Jani model format and tool interaction protocol. The format is a metamodel based on networks of communicating automata and has been designed for ease of implementation without sacrificing readability. The purpose of the protocol is to provide a stable and uniform interface between tools such as model checkers, transformers, and user interfaces. Jani uses the Json data format, inheriting its ease of use and inherent extensibility. Jani initially targets, but is not limited to, quantitative model checking. Several existing tools now support the verification of Jani models, and automatic converters from a diverse set of higher-level modelling languages have been implemented. The ultimate purpose of Jani is to simplify tool development, encourage research cooperation, and pave the way towards a future competition in quantitative model checking.

1 Introduction

Significant progress has been made in the area of formal verification to allow the analysis of ever more realistic, mathematically precise models of performance-, safety- or economically-critical systems. Such models can be automatically derived from the program or machine code of an existing implementation, or they can be constructed in a suitable *modelling language* during the system design phase. Many such languages, including process algebras like CCS [50] and CSP [36], lower-level formalisms like reactive modules [2], and high-level imperative-style languages like Promela [37], have been developed. However, the variety of languages, most of them supported by a single dedicated tool, is a major obstacle for new users seeking to apply formal methods in their field of work. Several efforts have been made to standardise modelling languages for broader use (notably Lotos [10], an ISO standard), or to develop overarching formalisms that offer a union of the features of many different specialised languages (a recent example being the CIF language and format [1]). Yet none of these efforts appears to have had a lasting impact on practice; of our examples,

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-662-54580-5-9

effectively the only implementation of Lotos is in the Cadp toolset [26], and active CIF support appears restricted to the CIF 3 tool [6].

We argue that the adoption of any standard formalism is hindered by a combination of the proposed standard (a) being complex and difficult to implement, (b) appearing at a time when there are already a number of well-established tools with their own modelling formalisms, and (c) existing in a conflict between supporting many different modelling purposes versus being a succinct way to support a particular technique or type of systems. As most new verification tools are still developed in an academic context, problem a creates work that is at best tangential to the actual research, and problem b means that there is little incentive to implement a new parser in an existing tool since such an effort is unlikely to lead to a publication. We observe that new tools continue to define their own new input language or a new dialect of an existing one as a result.

A new format. In this paper, we propose jani-model: *another* format for formal models aimed at becoming a common input language for existing and future tools. However, jani-model was created with problems a-c in mind: First of all, it is targeted to the specific field of *quantitative verification* using (extensions of) automata-like probabilistic models such as Markov decision processes (MDP [52]), probabilistic timed automata (PTA [45]), or continuous-time Markov chains (CTMC). This field is much younger than formal methods in general. Consequently, the tooling landscape is at an earlier stage in its evolution. We believe that problem b yet has little relevance there, and that *now* is actually the time where a push for commonality in quantitative verification tools is still possible as well as maximally beneficial. Several tools already support subsets or extensions of the Prism model checker's [43] language, so a good basis to avoid problem c appears to already exist in this field.

Consequently, the semantic model of the Prism language—networks of discrete- or continuous-time Markov chains (DTMC or CTMC), MDP or PTA with variables—forms the conceptual basis of jani-model. We have conservatively extended this model to also support Markov automata (MA, [21]) as well as stochastic timed and hybrid automata (STA [9] and SHA [24]). We have also replaced or generalised some concepts to allow more concise and flexible modelling. Notably, we took inspiration from the use of synchronisation vectors in Cadp and related tools to compactly-yet-flexibly specify how automata interact; we have added transient variables as seen in RDDL [54] to e.g. allow value passing without having to add state variables; and we have revised the specification of rewards and removed restrictions on global variables.

We could have made these changes and extensions to the textual syntax of the Prism language, creating a new dialect. However, in our experience, implementing a Prism language parser is non-trivial and time-consuming. To avoid problem a, jani-model is thus designed to be easy to generate and parse programmatically (while remaining "human-debuggable") without library dependencies. It defines an intentionally small set of core constructs, but its structure allows for easy extensibility. Several advanced features—like support for complex datatypes or recursive functions—are already specified as separate extensions. We do not expect users to create jani-model files manually. Instead, they will be automatically generated from higher-level and domain-specific languages.

A tool interaction protocol. jani-model helps the users as well as the developers of quantitative verification tools. Yet the latter face another obstacle: New techniques often require combining existing approaches implemented by others, or using existing tools for parts of the new analysis. In an academic setting, re-implementation is usually work for little reward, but also squanders the testing and performance tuning effort that went into the original tool. The alternative is to reuse the existing tool through whatever interface it provides: either a command-line interface—usually unstable, changing between tool versions—or an API tied or tailored to one programming language. The same problems apply to benchmarking and verification competitions. To help with interfacing verification tools, we propose the jani-interaction protocol. It defines a clean, flexible, programming language-independent interface to query a tool's capabilities, configure its parameters, perform model transformations, launch verification tasks, and obtain results. Again, we focused on ease of implementation, so jani-interaction is simple to support without dependencies on external libraries or frameworks, and only prescribes a small set of messages with clearly defined extension points.

Tool support. Jani has been designed in a collaborative effort, and a number of quantitative verification tools implement jani-model and jani-interaction today. They provide connections to existing modelling languages designed for humans as well as a number of analysis techniques with very different capabilities and specialisations based on traditional and statistical model checking. We summarise the current tool support in Section 5. We expect the number of Jani implementations to further grow as more input languages are connected and future new verification techniques are implemented for jani-model right from the start.

Related work. We already mentioned Lotos as an early standardisation effort, as well as CIF, which covers quantitative aspects such as timed and hybrid, but not probabilistic, behaviour. CIF is a complex specification consisting of a textual and graphical syntax for human use plus an XML representation. It had connections to a variety of tools including those based on Modelica [25], which itself is also an open specification intended to be supported by tools focusing on continuous system and controller simulation. The HOA format [4] is a tool-independent exchange format for ω-automata designed to represent linear-time properties for or during model checking. Atlantif [55] is an intermediate model for real-time systems with data that can be translated to timed automata or Petri nets. In the area of satisfiability-modulo-theories (SMT) solvers, the SMT-LIB standard [5] defines a widely-used data format and tool interface protocol analogous to the pair of jani-model/jani-interaction that we propose for quantitative verification. Boogie 2 [47] is an intermediate language used by static program verification tools. The formats mentioned so far provide concise high-level descriptions of potentially vast state spaces. An alternative is to exchange low-level

```
var ReplyAnalysisEngines = schema({
  "type": "analysis-engines",
  "id": Number.min(1).step(1),
  "engines": Array.of({
    "id": Identifier,
    "metadata": Metadata,
    "?params": Array.of(ParamDef)
  })
});
```

Listing 1. js-schema message specification

```
{ "type": "analysis-engines",
  "id": 123456,
  "engines": [
    "id": "simengine2"
    "metadata": {
       "name": "FIG",
       "version": {
         "major": 1, "minor": 13
    } } ] }
```

Listing 2. JSON message instance

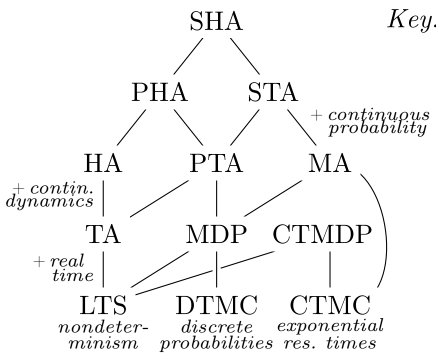
representations of actual state spaces, representing all the concrete states of the semantics of some high-level model. Examples of such state space-level encodings include CADP's BCG format and MRMC's [41] `.tra` files. Disadvantages are that the file size explodes with the state space, and all structural information necessary for symbolic (e.g. BDD-based) verification or static analysis is lost.

A number of tools take a reversed approach by providing an interface to plug in different input languages. In the non-quantitative setting, one example is LTSMIN [39] and its PINS interface. However, this is a C/C++ API on the state space level, so every input language needs to provide a complete implementation of its semantics for this tool-specific interface. A prominent tool with a similar approach that uses quantitative models is Möbius [13]. Notably, a command-line interface has recently been added to Möbius' existing graphical and low-level interfaces to improve interoperability [42]. The MODEST TOOLSET [33] also used an internal semantic model similar to that of jani-model that allows it to translate and connect to various external tools, albeit over their command-line interfaces.

The JANI specification can be seen as a *metamodel*. The Eclipse EMF/Ecore platform [19] is popular for building and working with metamodels. We chose to create a standalone specification instead in order to avoid the heavy dependency on Eclipse and to not force a preferred programming language on implementers.

2 JSON and js-schema

jani-model and jani-interaction use the JSON [11] data format to encode their models and messages, respectively. JSON is a textual, language independent format for representing data based on objects, arrays, and a small number of primitives. In contrast to alternatives like XML, it is extremely simple: its entire grammar can be given in just five small syntax diagrams. A generic JSON parser is easy to write, plus native parser libraries are available for many programming languages. The json.org website shows the syntax diagrams and maintains a list of libraries. In contrast to binary encodings, JSON remains human-readable, aiding in debugging. We show an example of the JSON code of an (abbreviated) jani-interaction message in Listing 2. Many of the advantages of JANI can be directly derived from the use of a JSON encoding. We already mentioned the simplicity of implementing a parser, but another important aspect is that a JSON format is



Key:

SHA	stochastic hybrid automata
PHA	probabilistic hybrid automata
STA	stochastic timed automata
HA	hybrid automata
PTA	probabilistic timed automata
MA	Markov automata
TA	timed automata
MDP	Markov decision processes
CTMDP	continuous-time MDP
LTS	labelled transition systems
DTMC	discrete-time Markov chains
CTMC	continuous-time Markov chains

Figure 1. Model types supported by the jani-model format

inherently extensible as new attributes can be added to objects without breaking an implementation that only reads a previously defined, smaller set of attributes. In addition, both jani-model and jani-interaction contain dedicated versioning and extension mechanisms to cleanly handle situations where future additions may change the semantics of previously defined constructs.

To formally specify what a valid JANI model is, as well as how the messages of the interaction protocol are encoded, we use the js-schema language [51]. js-schema is a lightweight syntax to define object schemas as well as a schema validation library. Compared to the popular alternative of JSON SCHEMA, js-schema specifications are syntactically more similar to the data they describe and thus easier to write and understand. By using an executable schema definition language, we directly obtain a procedure to unambiguously determine whether a given piece of JSON data can represent a JANI object. Some more complex requirements cannot be expressed within js-schema, e.g. when the presence of one attribute is required if and only if another attribute is not present. These additional checks are documented as comments in our js-schema specification for JANI, and they are checked by the reference parser implementation in the MODEST TOOLSET. In Listing 1, we show (part of) the js-schema specification for the `ReplyAnalysisEngines` message type of jani-interaction. The JSON object of Listing 2 conforms to this schema. An attribute name starting with `?` indicates an optional attribute, and in our example, `Identifier`, `Metadata` and `ParamDef` are references to other schemas defined elsewhere within the JANI specification while everything else refers to built-in components of js-schema.

3 The JANI Model Format

The first part of the JANI specification is the jani-model model format. It defines a direct JSON representation of networks of SHA with variables, or special cases thereof. In Figure 1, we show the automata models supported by jani-model. By providing variables and parallel composition, models with large or infinite state spaces can be represented succinctly. jani-model includes a basic set of variable

```
... "features": [ "derived-operators" ],
    "variables": [ { "name": "i", "initial-value": 0,
                     "type": { "kind": "bounded", "base": "int",
                               "lower-bound": 0, "upper-bound": 7 } } ],
    "edges":
    [ { "location": "loc0",
        "guard": { "op": "∧",
                   "left": { "op": "<", "left": 0, "right": "i" },
                   "right": { "op": "<", "left": "i", "right": 7 } },
        "destinations": [
          { "location": "loc0", "probability": 0.8, "assignments": [
            { "ref": "i",
              "value": { "op": "+", "left": "i", "right": 1 } } ] },
          { "location": "fail", "probability": 0.2 } ] } ], ...
```

Listing 3. Excerpt of a jani-model MDP model

types and expressions with most common operations, and allows the specification of probabilistic and reward-based properties for verification within a model.

The overriding goal of jani-model is simplicity for implementers. The core specification fits on five printed pages. Where expressions over the model's variables are required (such as a guard, the probability of a destination of an edge, or the right-hand side of an assignment), they are represented as expression trees. This is in contrast to other representations of networks of automata, e.g. Uppaal's [7] XML format, where they are stored as expression strings. Using trees makes it entirely unnecessary to write any kind of expression parsing code to process jani-model models. Listing 3 shows a slightly simplified excerpt of an MDP model with two locations `loc0` and `fail`. It has one edge from `loc0` with guard $0 < i \land i < 7$ that loops back to `loc0` with probability 0.8, incrementing i by 1, and goes to `fail` with probability 0.2.

An important aspect of the format is its extensibility, which is based on the mentioned use of Json in combination with an explicit extension mechanism: a model can list a number of *model features* that it makes use of. They are defined separately from the core jani-model specification, and include a `derived-operators` features, which provides for e.g. max and min operations (which could be represented with comparisons and if-then-else in core jani-model), an `arrays` and a `datatypes` feature that specify array types resp. functional-style recursive datatypes (e.g. to define an unbounded linked list type), and a `functions` feature that allows the definition of (mutually) recursive functions for use in expressions. Feature support will vary between tools; for example, BDD-based model checkers will typically not be able to easily handle unbounded recursive datatypes.

While its syntax is completely different, the semantic concepts of jani-model are based on the Prism language. However, it is more general in some aspects:

Locations. Automata in jani-model consist of local *variables* and *locations* connected by edges with action labels, guards, rates, probabilistic branches and assignments over the variables. While being natural for an automaton, having both locations and discrete variables is not strictly necessary as one can be encoded using the other. In fact, PRISM only supports the latter, necessitating the use of "program counter" variables to emulate locations if desired. By supporting both, jani-model provides modelling flexibility; if a tool prefers one extreme, an automatic conversion can easily be implemented. Locations provide structural information for e.g. optimisations and static analysis as well as a natural point to store the time progress conditions ("invariants") of TA-based models.

Synchronisation vectors. A jani-model model consists of a set of automata that execute in parallel. Edges are either performed independently, or two or more automata synchronise on an action label and perform an edge simultaneously. Inspired by CADP's EXP.OPEN tool, jani-model uses *synchronisation vectors* and sets of input-enabled actions as a general specification of synchronisation patterns. As an example, consider three automata. To specify CSP- or PRISM-style multi-way synchronisation on action `a`, we include the one vector $[\mathsf{a}, \mathsf{a}, \mathsf{a}]$. For CCS-style binary synchronisation between `a!` and `a?`, we need the six vectors

$$\{ [\texttt{a!}, \texttt{a?}, -], [\texttt{a?}, \texttt{a!}, -], [\texttt{a!}, -, \texttt{a?}], [\texttt{a?}, -, \texttt{a!}], [-, \texttt{a!}, \texttt{a?}], [-, \texttt{a?}, \texttt{a!}] \}.$$

For UPPAAL-style broadcast synchronisation, we make all automata input-enabled on `a?` and use the three vectors $\{ [\texttt{a!}, \texttt{a?}, \texttt{a?}], [\texttt{a?}, \texttt{a!}, \texttt{a?}], [\texttt{a?}, \texttt{a?}, \texttt{a!}] \}$. Synchronisation vectors can express all common process-algebraic operations like renaming or hiding, too—they are a concise yet extremely powerful mechanism.

As a further difference to PRISM, jani-model allows assignments to global variables on synchronising edges. Inconsistent concurrent assignments are a modelling error. This small extension removes a major modelling annoyance, but also has important implementation consequences (see Section 5 on the STORM tool).

Transient variables and assignments. When edges synchronise in a network of automata, the assignments of all participating automata are typically performed all at once, atomically. In jani-model, we additionally allow each assignment to be annotated with an *index*. Assignments with the same index are executed atomically, but sets of assignments with different indices are performed sequentially in the indexed order. In combination with transient variables, which are not part of the state vectors and get reset before and after taking an edge so they do not blow up the state space, this allows e.g. efficient value passing: If two automata synchronise and want to pass a value v, the first one can "send" v by making an assignment $t := v$ to a global transient variable t with index i on its synchronising edge while the second one can "receive" v by making an assignment $l := t$ to the local variable l with index $i' > i$ on its own synchronising edge.

Rewards. Finally, reward structures in jani-model are simply expressions over global (transient or non-transient) variables. Properties indicate whether they are instantaneous or steady-state rewards, or whether to accumulate when edges are taken (edge/transition rewards) or over time in locations (rate rewards). This is again a very simple but expressive way to specify rewards. As an example,

```
{ "op":"Emax", "exp":"i", "accumulate":["steps"], "step-instant":6 }
```

asks for the maximum expected reward, computed by accumulating the current value of variable `i` whenever a transition is taken, after exactly 6 transitions.

4 The Jani Interaction Protocol

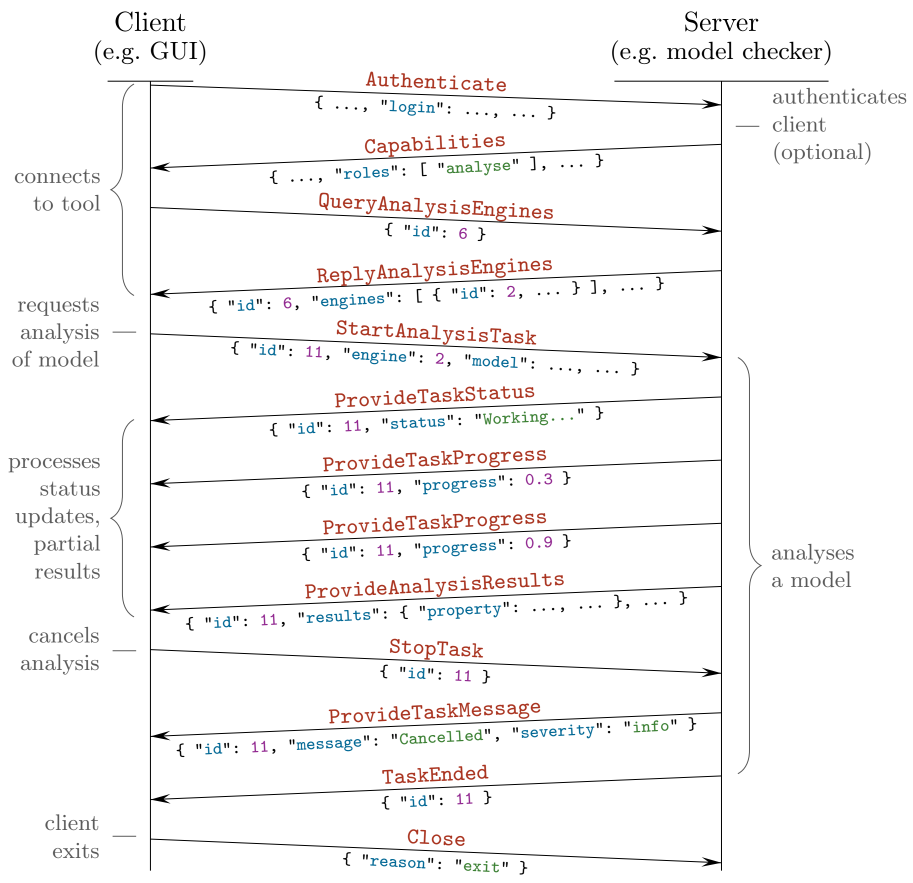
The second part of the Jani specification is the jani-interaction tool interaction and automation protocol. Its purpose is to provide a stable interface that allows the reuse of existing implementations from new tools, reduce setup problems by allowing communication between tools running on different machines, and allow for a common integrated graphical user interface for Jani-based verifiers.

jani-interaction is a client-server protocol. A server can support a number of *roles*. We currently define the `analyse` and `transform` roles, which offer access to verification procedures and model transformations, respectively. Roles are the main extension point, allowing new roles to be added in the future. A tool supporting the `analyse` role provides a number of *analysis engines*, which represent the verification algorithms it implements. The protocol then allows analysis tasks to be started, with the server subsequently sending status updates to the client and the client having the ability to cancel the analysis. The jani-interaction specification defines a total of 18 message types, out of which 4 are specific to the `analyse` and 4 are specific to the `transform` role. 5 message types are for task management and used by both roles. The `ReplyAnalysisEngines` message that we showed (in a slightly shortened form) in Listings 1 and 2, for example, is a server-to-client message of the `analyse` role that is sent when the client has queried for the available analysis engines. It includes an array of self-describing parameter definitions; the client can supply values for these parameters to configure the analysis engine when it starts an analysis task. Within the corresponding `StartAnalysisTask` message, the client also submits the model to be analysed. It can be either a jani-model model, which is Json data and thus included verbatim in the message, or a set of Json strings with the contents of the model files of any other modelling formalism with a textual representation.

A jani-interaction session consists of the exchange of a number of Json messages. This can occur in one of two ways: either remotely over the WebSocket network protocol [23], with each message transmitted in one WebSocket text message, or locally by the client starting the server tool and writing its messages into the server's standard input stream, with the server writing its replies onto its standard output stream, one message per line. Using WebSocket communication allows running a tool remotely on a machine that is configured in exactly the way required for the tool to run, and makes it possible to access tools using JavaScript from websites in a browser. Using standard streams is an easier-to-implement alternative for making an existing tool support jani-interaction. We show an example jani-interaction session in Figure 2.

5 Tool Support

The Jani specification is already supported by a number of quantitative verification tools as outlined in Figure 3. These tools provide translations from several higher-level modelling languages to jani-model and, in some cases, vice-versa, thus implementing the functionality of the `transform` role of jani-interaction.



Client
(e.g. GUI)
Server
(e.g. model checker)
Authenticate
{ ..., "login": ..., ... }
authenticates client (optional)
Capabilities
{ ..., "roles": ["analyse"], ... }
connects to tool
QueryAnalysisEngines
{ "id": 6 }
ReplyAnalysisEngines
{ "id": 6, "engines": [{ "id": 2, ... }], ... }
requests analysis of model
StartAnalysisTask
{ "id": 11, "engine": 2, "model": ..., ... }
ProvideTaskStatus
{ "id": 11, "status": "Working..." }
processes status updates, partial results
ProvideTaskProgress
{ "id": 11, "progress": 0.3 }
ProvideTaskProgress
{ "id": 11, "progress": 0.9 }
analyses a model
ProvideAnalysisResults
{ "id": 11, "results": { "property": ..., ... }, ... }
cancels analysis
StopTask
{ "id": 11 }
ProvideTaskMessage
{ "id": 11, "message": "Cancelled", "severity": "info" }
TaskEnded
{ "id": 11 }
client exits
Close
{ "reason": "exit" }

Figure 2. An example jani-interaction session

Each of them also comes with a set of analysis engines that perform transitional exhaustive or statistical model checking of jani-model models to produce consistent verification results, corresponding to jani-interaction's `analyse` role.

5.1 Modelling Languages

jani-model is designed to be easily machine-readable and we do not expect users to write jani-model files directly. Instead, we provide automated translations from the Prism language, GSPN, IOSA, Modest, pGCL and xSADF.

Prism language. The Prism language is based on reactive modules [2] and used as input language of the Prism model checker [43]. Variants and subsets are used by other quantitative verification tools, which is why we decided to base jani-model on its core concepts. A model in the Prism language consists of a set

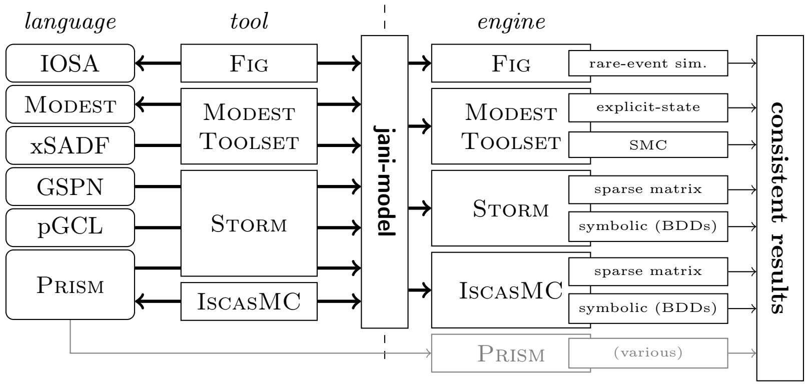


Figure 3. The Jani landscape

of modules executing in parallel. Each has a number of discrete variables and a set of probabilistic commands with a guard and a probability distribution over assignments. There are no control flow constructs like e.g. loops; they have to be manually encoded in variables. The Prism language was originally designed to model DTMC, CTMC and MDP, and has since been extended to support PTA. We show an example of a Prism model in Figure 4.

The official bidirectional conversion between the Prism language and jani-model is implemented in IscasMC. This gives access to the vast collection of Prism case studies and benchmarks [44] to all tools that support jani-model, and allows the use of Prism's model checking engines to analyse jani-model files and models in all input languages for which a conversion to jani-model exists.

GSPN. Petri Nets are a widely-used model for concurrent processes. Generalised stochastic Petri nets (GSPN, [48]) provide *exponentially delayed* transitions in addition to the standard *immediate* transitions. Nondeterminism arising due to the latter has often been resolved by assigning weights, thereby implicitly having discrete probabilistic branching in the model. We show an example GSPN in Figure 5, which contains two exponentially delayed transitions with rates λ_1 and λ_2. A formal semantics for *every* GSPN, including "confused" ones with actual nondeterminism, in terms of MA has been developed recently [20].

Based on an implementation of this semantics, the Storm tool can translate GSPN given either as a greatSPN project [3] or in a variant of the ISO-standard PNML [38] format into a jani-model description. Variables describe the markings, and the encoding of nondeterministic and delayed transitions is straightforward. Only weights require a somewhat more involved encoding as expressions.

IOSA. Stochastic automata (SA, [14]) are decision processes in which the occurrence of events is governed by random variables called *clocks*. These can follow arbitrary continuous probability distributions. Input/output SA (I/O SA, [15]) are a variant of networks of SA that guarantee the absence of nondeterminism:

PRISM language:

```
module Channel
  l: [0..1]; // control loc
  c: clock;  // for delay
  invariant
    l=1 => c<=2
  endinvariant
  [snd] l = 0  -> 0.01:(l'=0)
       + 0.99:(l'=1)&(c'=0)
  [rcv] l=1 & c>=2 -> (l'=0)
endmodule
```

MODEST:

```
process Channel() {
  snd palt {
  :99: delay(2)
       rcv
  : 1: // msg lost
       {==}
  };
  Channel()
}
Channel()
```

Figure 4. A channel PTA model in PRISM and MODEST

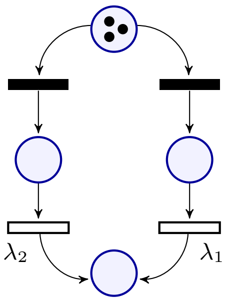


Figure 5. A GSPN

Automata must be input-enabled, each output can only be produced by a single automaton in the network, and clocks can only control the timing of outputs. Networks of input/output SA can be modelled in the IOSA language, which is syntactically a variant of the PRISM language. We show an example in Listing 4, where action `a` is output (`!`) for `M1` and input (`?`) for `M2`. Synchronisation is performed in a *broadcast* fashion, meaning an output will synchronise with all matching inputs. This ensures the input-enabledness requirement.

The FIG tool [12] translates IOSA to and from jani-model. In jani-model, the STA model type is used, since I/O SA are a proper subset of STA. When converting from jani-model to IOSA, STA and CTMC models where the synchronisation vectors correspond to broadcast synchronisation are supported. STA are accepted only if the STA clocks are used in a way that can be mapped to SA.

MODEST. The MODEST language is a modelling formalism with a semantics in terms of STA [9], later extended to SHA [29]. It is an expressive, high-level language with features like recursive process calls, `do` loops, exception handling, and complex datatypes. We show a very small example in Figure 4. The MODEST TOOLSET implements conversions from MODEST to jani-model and back. In terms of supported model types, MODEST is the most expressive language currently connected to jani-model because everything can be seen as a special case of SHA.

pGCL. Probabilistic programming languages extend standard languages with constructs to sample from random distributions and to condition program runs on observations about (random) data. Such constructs are at the heart of algorithms in machine learning, security, and quantum computing [27]. The operational semantics of probabilistic programs are (possibly infinite) MDP.

One example of a probabilistic programming language is the probabilistic guarded command language (pGCL, [49]) with observe statements [40]. The STORM tool implements a translation from pGCL via program graphs to jani-model. A noticeable feature of the translation is the detection of rewards: In the example pGCL program given in Listing 5, if we omit the `observe` statement, the variable `x` can be considered a reward, which then makes the MDP finite and thus amenable to probabilistic model checking.

```
module M1
  c: clock;
  [a!] true @ c -> (c' = gamma(0.5, 2 * N));
endmodule
module M2
  i: [0..M];  x: [1..M + 1];
  [a?] i <= M -> (i' = x) & (x' = i + 1);
endmodule
```

Listing 4. An IOSA model of two modules

```
while(c = 0)
{
  { x := x + 1 }
  [1/2]
  { c := 1 }
};
observe "x is odd"
```

Listing 5. pGCL

xSADF. Dataflow formalisms are popular in the study of embedded data processing applications. The recently introduced formalism xSADF [35], an extension of scenario-aware dataflow [56], adds cost annotations (to model, for example, power consumption), nondeterminism, and continuous stochastic execution times. It is equipped with a compositional semantics in terms of STA, which is implemented in the Modest Toolset. Via the latter's support for jani-model, we can now also convert xSADF specifications to jani-model. The resulting models are networks of STA that make use of the `datatypes` and `functions` features to encode the unbounded typed scenario channels of xSADF.

5.2 Analysis Tools

Support for the verification of jani-model models is currently provided by Fig, IscasMC, the Modest Toolset and Storm, as well as Prism via IscasMC's ability to convert jani-model to the Prism language. We summarise the capabilities and restrictions of the various analysis engines in Table 1. ✓denotes current support, while $*$ means that an implementation is planned. (1) indicates that only broadcast-based input/output STA that correspond to stochastic automata are supported. (2) marks planned support of the `arrays` feature that will be restricted to fixed-size arrays. The Modest Toolset's support for SHA is via the prohver tool [29], indicated by (3), and its statistical model checker only supports deterministic models where marked (4). Concerning supported properties, we consider the broad classes of probabilistic reachability (P), probabilistic computation tree logic (PCTL), the probabilities of linear temporal logic formulas (LTL), any type of expected values or rewards (E) and steady-state measures (S).

Fig. Specialised in rare event simulation, Fig [12] implements novel techniques that allow the use of importance splitting [46] in a fully automated way. Importance splitting speeds up the occurrence of some user-defined rare event in order to better estimate its probability of occurrence.

Fig can be used to study transient and long run behaviour. Transient properties are expressed as $\mathsf{P}(\neg \mathit{stop}\ \mathsf{U}\ \mathit{rare})$, where *stop* and *rare* are propositional formulas describing *simulation truncation* and *rare event occurrence*, respectively. Steady state properties correspond to the CSL expression $\mathsf{S}(\mathit{rare})$. Aside from standard Monte Carlo simulation, an engine based on RESTART-like [57]

Table 1. Support for model types, features and property classes in analysis tools

tool	engine	LTS	DTMC	CTMC	MDP	CTMDP	MA	TA	PTA	STA	SHA	arrays	datatypes	functions	P	PCTL	LTL	E	S
FIG	rare	–	*	✓	–	–	–	–	–	(1)	–	(2)	–	*	✓	–	–	*	✓
IscasMC	sparse	✓	✓	✓	✓	*	*	–	–	–	–	–	–	–	✓	✓	✓	*	*
	BDD	✓	✓	✓	✓	*	*	–	–	–	–	–	–	–	✓	✓	✓	*	*
Modest Toolset	explicit	✓	✓	–	✓	–	–	✓	✓	✓	(3)	✓	✓	✓	✓	*	–	✓	–
	SMC	✓	✓	–	✓	–	(4)	(4)	(4)	(4)	–	✓	✓	✓	✓	–	–	✓	–
Storm	sparse	✓	✓	✓	✓	–	✓	–	–	–	–	*	*	*	✓	✓	–	✓	✓
	BDD	✓	✓	✓	✓	–	*	–	–	–	–	(2)	–	–	✓	✓	–	✓	*
Prism	(various)	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	✓	✓	✓	✓	✓

importance splitting can be used. The importance function needed by the latter can be provided *ad hoc* by the user or computed automatically by the tool.

IscasMC. A Java-based model checker for stochastic systems, IscasMC [31] offers an easy-to-use web interface for the evaluation of Markov chains and decision processes against PCTL, PLTL, and PCTL* specifications. It is particularly efficient in evaluating the probabilities of LTL properties, supporting multiple resolution methods that improve the actual runtime on complex LTL properties [30]. IscasMC provides two analysis engines: one based on an explicit sparse matrix encoding of the state space, and a symbolic one using binary decision diagrams (BDD). IscasMC can be extended with plugins. This permits to support the analysis of other formalisms, like quantum Markov chains [22] and stochastic parity games [32], as well as to use different (multi-terminal) BDD libraries [18] to symbolically represent both the model and the automaton for the LTL formula.

The Modest Toolset. A modular collection of model transformation and analysis tools centred around an internal metamodel of networks of stochastic hybrid systems, which greatly influenced the design of jani-model, the Modest Toolset [33] is an implementation of the multiple-formalism, multiple-solution idea. Its core analysis engines today are the explicit-state model checker mcsta and the statistical model checker (SMC) modes. The former handles MDP, PTA and STA with billions of states via a disk-based approach [34] and efficiently checks time- and reward-bounded properties without unnecessarily unfolding the state space [28]. The latter focuses on detecting spurious nondeterminism on-the-fly during simulation in order to be able to handle not just Markov chains.

Storm. Newly developed as the successor of the probabilistic model checker MRMC [41], Storm [17] works with DTMC, CTMC, MDP and MA models. In addition to its support for jani-model and the Prism language, it can also read files in an explicit state space-level format similar to MRMC's. The analysis of models is backed by different engines that use different representations for

Table 2. Comparison of Prism- and jani-model-based state space generation

		sparse/explicit engines				*symbolic engines (BDD)*			
			Storm		Prism		Storm		Prism
model	type	params	Jani	Prism		params	Jani	Prism	
crowds	DTMC	$\langle 20, 5\rangle$	8.9 s	8.4 s	26.2 s	$\langle 20, 25\rangle$	9.1 s	9.6 s	9.6 s
cluster	CTMC	250	20.2 s	18.4 s	26.5 s	3000	32.1 s	31.1 s	96.7 s
consensus	MDP	$\langle 6, 4\rangle$	15.3 s	14.6 s	48.3 s	$\langle 10, 100\rangle$	24.1 s	25.4 s	27.5 s
CSMA	MDP	$\langle 3, 4\rangle$	13.8 s	13.1 s	15.4 s	$\langle 4, 4\rangle$	10.2 s	10.2 s	27.8 s

the model structure and reachable states, including sparse matrices and BDD. Storm's first aim is to achieve good performance, but special attention is also given to a modular design that enables coherent and easy access to a variety of solvers used by the analysis processes such as linear equation, mixed-integer linear programming, and SMT solvers. Storm also supports parametric DTMC and MDP. As the backend for Prophesy [16] and using a parameter lifting approach [53], it significantly outperforms other parametric discrete-time verification tools.

The Prism language is known for its ability to compactly represent gigantic models which can be very efficiently handled by BDD-based engines. In Storm, jani-model and Prism models are currently handled by separate code paths. This provided the opportunity to investigate whether the changes in state space generation code caused by the new concepts of jani-model (in particular to support synchronising assignments to global variables in the BDD-based engine) impact performance. Experiments were run on a quad-core 3.5 GHz Intel Core i7 system with Mac OS X 10.12, using four Prism benchmark models [44] and their conversions to jani-model. We tested both explicit-state and symbolic engines. Table 2 lists the model construction time of Storm with the jani-model and Prism files and, for comparison, of Prism with the Prism file. The results indicate that allowing for the extra language features in jani-model does not significantly influence the model construction performance; the comparison with Prism furthermore shows that this is not just due to a naïve implementation of the Prism code path within Storm.

6 Conclusion

We have proposed the Jani specification for model exchange and tool interaction. The complete specification and a library of models are available at jani-spec.org. The goal of Jani is to reduce the effort required to develop verification tools, especially in an academic setting, and to foster tool interoperation and comparison. Supporting the jani-model format gives access to a large number of existing models (in the format itself and in the various connected languages) for testing and benchmarking at little effort compared to writing a full parser for one of the existing modelling languages, which prioritise being easily human-writeable over being easily machine-readable. While Jani is currently focused on quantitative verification (cf. problem b of Section 1), standard labelled transition systems or Kripke structures as used in traditional verification approaches can be represented in jani-model, too, and the jani-interaction protocol can be used with any modelling formalism with a textual representation.

Outlook. As Jani is an ongoing effort, we use the jani-spec.org website to track the growing list of implementing tools and their status (akin to Table 1). Ultimately, we hope that Jani can lead the way towards a more coordinated tool development process in quantitative verification that, together with the previous definition of the Prism benchmark suite [44], will eventually enable a quantitative model checking competition. Such competitions have been shown to have a strong positive impact on the tooling landscape in affected fields [8].

Acknowledgements. This work is supported by the 3TU project "Big Software on the Run", ANPCyT grant PICT-2012-1823, BMBF-IKT 2020 project 16KIS0138 HODRIAN, CDZ project GZ 1023 (CAP), the CAS Fellowship for International Young Scientists, the CAS/SAFEA International Fellowship Program for Creative Research Teams, the National Natural Science Foundation of China (grants no. 61550110506 and 61650410658), and SeCyT-UNC grant 05/BP12.

References

1. Agut, D.E.N., van Beek, D.A., Rooda, J.E.: Syntax and semantics of the compositional interchange format for hybrid systems. J. Log. Algebr. Program. 82(1), 1–52 (2013)
2. Alur, R., Henzinger, T.A.: Reactive modules. FMSD 15(1), 7–48 (1999)
3. Amparore, E.G.: A new GreatSPN GUI for GSPN editing and CSLTA model checking. In: QEST. LNCS, vol. 8657, pp. 170–173 (2014)
4. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi omega-automata format. In: CAV. LNCS, vol. 9206, pp. 479–486 (2015)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., Dep. of Computer Science, The University of Iowa (2015), www.smt-lib.org
6. van Beek, D.A., Fokkink, W., Hendriks, D., Hofkamp, A., Markovski, J., van de Mortel-Fronczak, J.M., Reniers, M.A.: CIF 3: Model-based engineering of supervisory controllers. In: TACAS. LNCS, vol. 8413, pp. 575–580 (2014)
7. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST. pp. 125–126. IEEE CS (2006)
8. Beyer, D.: Software verification and verifiable witnesses (report on SV-COMP 2015). In: TACAS. LNCS, vol. 9035, pp. 401–416 (2015)
9. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.: MODEST: A compositional modeling formalism for hard and softly timed systems. IEEE TSE 32(10), 812–830 (2006)
10. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks 14, 25–59 (1987)
11. Bray, T.: The JavaScript Object Notation (JSON) data interchange format. RFC 7159, RFC Editor (March 2014), rfc-editor.org/rfc/rfc7159.txt

12. Budde, C.E., D'Argenio, P.R., Monti, R.E.: Compositional construction of importance functions in fully automated importance splitting. In: VALUETOOLS. ICST (2016)
13. Courtney, T., Gaonkar, S., Keefe, K., Rozier, E., Sanders, W.H.: Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In: DSN. pp. 353–358. IEEE CS (2009)
14. D'Argenio, P.R., Katoen, J.: A theory of stochastic systems part I: stochastic automata. Inf. Comput. 203(1), 1–38 (2005)
15. D'Argenio, P.R., Lee, M.D., Monti, R.E.: Input/output stochastic automata – compositionality and determinism. In: FORMATS. LNCS, vol. 9884, pp. 53–68 (2016)
16. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J., Ábrahám, E.: PROPhESY: A PRObabilistic Parameter SYnthesis tool. In: CAV. LNCS, vol. 9206, pp. 214–231 (2015)
17. Dehnert, C., Junges, S., Katoen, J., Volk, M.: The probabilistic model checker Storm (extended abstract). CoRR abs/1610.08713 (2016)
18. van Dijk, T., Hahn, E.M., Jansen, D.N., Li, Y., Neele, T., Stoelinga, M., Turrini, A., Zhang, L.: A comparative study of BDD packages for probabilistic symbolic model checking. In: SETTA. LNCS, vol. 9409, pp. 35–54 (2015)
19. Eclipse Foundation: Eclipse Modeling Framework (EMF) website. eclipse.org/modeling/emf, accessed: 2016-01-27
20. Eisentraut, C., Hermanns, H., Katoen, J., Zhang, L.: A semantics for every GSPN. In: Petri Nets. LNCS, vol. 7927, pp. 90–109 (2013)
21. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS. pp. 342–351. IEEE CS (2010)
22. Feng, Y., Hahn, E.M., Turrini, A., Zhang, L.: QPMC: A model checker for quantum programs and protocols. In: FM. LNCS, vol. 9109, pp. 265–272 (2015)
23. Fette, I., Melnikov, A.: The WebSocket protocol. RFC 6455, RFC Editor (December 2011), rfc-editor.org/rfc/rfc6455.txt
24. Fränzle, M., Hahn, E.M., Hermanns, H., Wolovick, N., Zhang, L.: Measurability and safety verification for stochastic hybrid systems. In: HSCC. pp. 43–52. ACM (2011)
25. Fritzson, P.: Modelica – A cyber-physical modeling language and the OpenModelica environment. In: IWCMC. pp. 1648–1653. IEEE (2011)
26. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2), 89–107 (2013)
27. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE. pp. 167–181. ACM (2014)
28. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. In: SETTA. LNCS, vol. 9984, pp. 85–100 (2016)
29. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.: A compositional modelling and analysis framework for stochastic hybrid systems. FMSD 43(2), 191–232 (2013)
30. Hahn, E.M., Li, G., Schewe, S., Turrini, A., Zhang, L.: Lazy probabilistic model checking without determinisation. In: CONCUR. LIPIcs, vol. 42, pp. 354–367. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
31. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: IscasMc: A web-based probabilistic model checker. In: FM. LNCS, vol. 8442, pp. 312–317 (2014)
32. Hahn, E.M., Schewe, S., Turrini, A., Zhang, L.: A simple algorithm for solving qualitative probabilistic parity games. In: CAV. LNCS, vol. 9780, pp. 291–311 (2016)

33. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS. LNCS, vol. 8413, pp. 593–598 (2014)
34. Hartmanns, A., Hermanns, H.: Explicit model checking of very large MDP using partitioning and secondary storage. In: ATVA. LNCS, vol. 9364, pp. 131–147 (2015)
35. Hartmanns, A., Hermanns, H., Bungert, M.: Flexible support for time and costs in scenario-aware dataflow. In: EMSOFT. pp. 3:1–3:10. ACM (2016)
36. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
37. Holzmann, G.J.: The model checker SPIN. IEEE TSE 23(5), 279–295 (1997)
38. ISO 15909-2:2011. High-level Petri nets – Part 2: Transfer format (2011)
39. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-performance language-independent model checking. In: TACAS. LNCS, vol. 9035, pp. 692–707 (2015)
40. Katoen, J., Gretz, F., Jansen, N., Kaminski, B.L., Olmedo, F.: Understanding probabilistic programs. In: CSD. LNCS, vol. 9360, pp. 15–32 (2015)
41. Katoen, J., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. 68(2), 90–104 (2011)
42. Keefe, K., Sanders, W.H.: Möbius shell: A command-line interface for Möbius. In: QEST. LNCS, vol. 8054, pp. 282–285 (2013)
43. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806, pp. 585–591 (2011)
44. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST. pp. 203–204. IEEE CS (2012)
45. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. TCS 282(1), 101–150 (2002)
46. L'Ecuyer, P., Le Gland, F., Lezaud, P., Tuffin, B.: Splitting techniques. In: Rare Event Simulation using Monte Carlo Methods, pp. 39–61. John Wiley & Sons, Ltd. (2009)
47. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: TACAS. LNCS, vol. 6015, pp. 312–327 (2010)
48. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley & Sons, Inc., 1st edn. (1994)
49. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005)
50. Milner, R.: A Calculus of Communicating Systems, LNCS, vol. 92. Springer (1980)
51. Molnár, G.: js-schema website. molnarg.github.io/js-schema, accessed: 2016-01-28
52. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc. (1994)
53. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Parameter synthesis for Markov models: Faster than ever. In: ATVA. LNCS, vol. 9938, pp. 50–67 (2016)
54. Sanner, S.: Relational dynamic influence diagram language (RDDL): Language description (2010), users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf
55. Stöcker, J., Lang, F., Garavel, H.: Parallel processes with real-time and data: The Atlantif intermediate format. In: iFM. LNCS, vol. 5423, pp. 88–102 (2009)
56. Theelen, B.D., Geilen, M., Basten, T., Voeten, J., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: MEMOCODE. pp. 185–194. IEEE CS (2006)
57. Villén-Altamirano, M., Villén-Altamirano, J.: The rare event simulation method RESTART: efficiency analysis and guidelines for its application. In: Network Performance Engineering, LNCS, vol. 5233, pp. 509–547 (2011)