A Modular Approach for Reusing Formalisms in Verification Tools of Concurrent Systems*

Étienne André[1], Benoît Barbot[2], Clément Démoulins[3], Lom Messan Hillah[4], Francis Hulin-Hubard[2], Fabrice Kordon[4], Alban Linard[2], and Laure Petrucci[1]

[1] Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
[2] LSV, CNRS, INRIA & ENS Cachan, France
[3] EPITA Research and Development Laboratory (LRDE), France
[4] LIP6, CNRS UMR 7606, Université P. & M. Curie and Université P. Ouest, France

Abstract. Over the past two decades, numerous verification tools have been successfully used for verifying complex concurrent systems, modelled using various formalisms. However, it is still hard to coordinate these tools since they rely on such a large number of formalisms. Having a proper syntactical mechanism to interrelate them through variability would increase the capability of effective integrated formal methods. In this paper, we propose a modular approach for defining new formalisms by reusing existing ones and adding new features and/or constraints. Our approach relies on standard XML technologies; their use provides the capability of rapidly and automatically obtaining tools for representing and validating models. It thus enables fast iterations in developing and testing complex formalisms. As a case study, we applied our modular definition approach on families of Petri nets and timed automata.

Keywords: Formal methods, Model Driven Engineering, Interoperability, Reusability, Concurrent Systems, Model Checking

1 Introduction

Research teams have built over the past two decades numerous verification tools that have successfully been applied to case studies. Formal models used by these tools are described by formalisms. We call *formalism* a metamodel for a formal notation. A formalism describes the entities to be found in the notation but it also associates these with a behavioural semantics.

In this work, we focus on formal notations that are all dedicated to the description of distributed and concurrent systems behaviour. Hence, their operational semantics (i.e. how they can be executed) must be mathematically

* This paper is the author version of the paper with the same name published in the proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM'13). The final version is available at http://www.springer.com/.

founded to enable automated reasoning techniques such as model checking. Another characteristic of these notations is to be graph-based. This encompasses (but is not limited to) automata, Petri nets and their variants.

Each of these numerous tools handles its own set of formalisms, in its own set of syntactic formats, which makes it hard to harness their verification power into integrated platforms. Some attempts have however been successful in integrating various model checking tools into a single platform. One notable instance is the CPN-AMI platform [9], used worldwide since the 1990's.

As part of the MeFoSyLoMa[5] community, we have had a long tradition of maintaining this platform. The continuous maintenance has become very costly and questionable recently, as new tools requirements in terms of new formalisms and interoperability dramatically increased the time and effort required to build adapters and wrappers for their integration. Moreover, the local syntax of CPN-AMI could no longer cope with the constructs in the new formalisms handled by the new tools. This is especially the case for the compositional and hierarchical aspects. Therefore, its extension in its current form turned out not to be a viable option.

Contribution This context led us to start the development of a new, flexible and extensible syntactic framework for the integration of new formalisms. The supporting format is now XML-based, more specifically on the RELAX-NG standard [12]. We designed the new open format with extensibility in mind, to allow quick and easy definitions of new formalisms, by reusing existing constructs.

A major benefit of this approach is to provide the capability to rapidly and automatically obtain tools for representing and validating models. It thus also reduces the engineering effort to integrate new tools, as libraries to handle models of their formalisms are automatically generated.

We describe in this paper a modular approach for reusing syntactic definitions of formalisms, such as Petri nets and automata, in verification tools. We also report its successful implementation using XML technologies. This approach is implemented in a distributed and fully open platform, *CosyVerif* [3], making it possible for any research team to set up local tools in a server on their premises, and automatically register the provided services in the cloud of *CosyVerif*. The maintenance effort of several days required for CPN-AMI has now decreased to less than half a day for integrating formalisms and tools in *CosyVerif*. From the user point of the view, the use of any tool is greatly eased thanks to a user-intuitive graphical client.

Outline Section 2 presents an overview of current techniques in modelling the abstract syntax of formal notations. With lessons learned from previous experiences, Section 3 describes our solution and details its implementation using standard XML-based technologies. An application to Petri nets and automata shows in Section 4 how we leveraged the combined use of these technologies

[5] "Méthodes formelles pour les systèmes logiciels et matériels" (formal methods for software and hardware systems), see http://www.mefosyloma.fr

to build an extensible and incremental architecture of interrelated formalisms. We also identify good practices for the definition and the reuse of formalisms. Section 5 presents the integration of our approach into the distributed platform *CosyVerif*. We identify future directions of research in Section 6.

2 Related Work

Generally, tools work on models typed after a *formalism*. Tools taking as input the same formalism usually have a different syntax. For example, consider the case of timed automata [1]: among a few examples of tools taking as input (extensions of) timed automata – HyTech [10], Imitator [2], PAT [20] and Uppaal [19] – all have a very different input syntax. Manually translating a model from a given syntax into another one is cumbersome and error-prone; an automated translation can be performed, but must be defined for any pair of tools sharing the same input formalism. Hence unifying formalisms definitions is a necessary condition to an effective integration of heterogeneous tools.

Several approaches have attempted, with various degrees of success, to tackle this challenge, using model-based techniques, and sometimes backed by existing platforms.

2.1 Related Model-Based Approaches

A notable work using model-based techniques is the Petri nets standard, ISO/IEC 15909. Part 2 of this standard [14] defines the Petri Net Markup Language (PNML), a transfer format to foster interoperability among Petri net tools. The standard defines the abstract syntax of PNML using UML class diagram notation. It defines the format concrete representation using RELAX-NG. PNML is supported by PNML Framework [11], a generated Java library thanks to model-driven engineering techniques, relying on the Eclipse Modeling Framework [21].

OMDoc[6] is a markup format and data model for Open Mathematical Documents, defining an ontology language for mathematical knowledge. No platform is associated with this work, but interfaces to existing tools (such as PVS or Coq) are available.

MoWGLI[7] builds on previous standards for the management and publication of mathematical documents (MathML, OpenMath, OMDoc). It relies on XML-based technologies (XSLT, RDF, etc.). However, it seems that there is no associated platform, and it looks like it is not maintained anymore.

2.2 Related Platforms

Several platforms have been designed over the past decade in order to achieve similar goals. CASL (Common Algebraic Specification Language) is a general-

[6] http://www.omdoc.org/
[7] http://mowgli.cs.unibo.it/

purpose specification language. A tool named HetCASL[8] (Heterogeneous Tool Set) has been proposed, that incorporates different theorem provers and different specification languages, hence allowing the designer to handle heterogeneous specifications. This approach is very much theorem prover oriented (including connections with Isabelle, Maude, etc.). In contrast, *CosyVerif* is more general.

Diabelli [23] is a heterogeneous proof system, allowing one to perform theorem proving with both diagrammatic and sentential formulae, and proof steps. It is shipped as a standalone tool combining Isabelle and Speedith. The tool does feature a graphical interface, but models are given in a textual form only. We believe that this tool does not provide a high degree of flexibility (because it requires translations), and apparently it does not work in the cloud, contrarily to *CosyVerif*.

LTSmin [6] is a meta toolkit that supports different input language modules (mCRL2, Promela, etc.) relying on labeled transition systems (LTS). LTSmin allows LTS-based semantic exchanges of state space between different tools (based on a Partitioned Next-State function). Furthermore, it allows the end user to apply alternative verification algorithms to their native tool. However, the tool only works with a LTS-based semantics, whereas we aim at considering a larger set of formalisms.

Rich-model Toolkit[9] is a standardisation of formal languages: it features common formats for systems, formulae, proofs and counterexamples. Contrarily to our approach, it is SAT- and SMT-oriented, and algorithms seem to be built-in, although it is hard to get a precise idea of the features, since this is a very recent initiative.

StarExec[10] is an initiative of the logic community to build a shared logic solving infrastructure (SAT, SMT), to enable researchers to manage libraries, provide solver execution on a large cluster, and facilitate translation between logics. According to their system architecture specification, users interface with the infrastructure via a Web application.

PAT [20] is a multi-formalisms platform based on modules. Each module relies on its own formalism and domain of application (e.g. real-time systems, probabilistic systems, network calculus, etc.), and must provide a semantics in the form of LTS. Then, common algorithms (deadlock-checking, LTL-checking) can be used for any of the modules, in addition to domain-specific algorithms. It also features graphical facilities, a simulator, syntactical checkers, counterexample exhibition, etc. Different from our approach, PAT is mainly LTS-based (with additional integration of Markov Decision Processes and Timed Transition Systems), and formalisms are not related to each other, i.e. the modules are independent.

[8] http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/

[9] http://richmodels.epfl.ch/

[10] http://www.starexec.org/starexec/public/about.jsp

2.3 Discussion

Our approach is similar to the ISO/IEC 15909 standard in terms of formalism definition, and to StarExec, in terms of supporting platform. However there are notable differences.

In ISO/IEC 15909, only most popular Petri nets types are considered: P/T (places/transitions) nets, Symmetric and High-Level nets. Moreover, the identification of features variability to enable extensibility is an issue not completely solved due to a large family of types in the Petri net community. Many Petri net types are ad-hoc variants of mainstream or exotic types, defined for specific research purposes. So the right level of granularity among such a large family is hard to figure out for defining a feature-proof extensible framework in the standard. We learnt from this case, and now consider the issue from a graph-based approach: any graph-based formalism, including Petri nets and automata (our case study in Section 4) should be adequately integrated in the framework.

The StarExec platform builds upon a grid engine (Oracle Grid Engine), where solvers and benchmarks jobs are scheduled and dispatched over worker nodes, in a typical grid computing fashion. *CosyVerif* is more cloud-based, since each participant can set up his/her own tools server and automatically register the services it provides in the cloud. Moreover *CosyVerif* does not mandate a Web-based user interface for jobs submission. Users can interface with tools via existing front ends like Coloane (available from [22]), or via their own tools using a provided library.

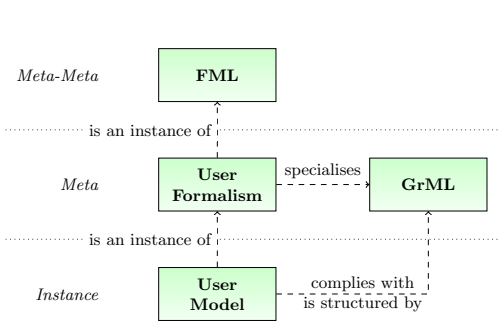
3 A Unified Representation

One of our main goals is to address the high variability in formalisms definitions. For example, since the original definition of timed automata [1], many variants and extensions have been proposed, among which: timed automata with stopwatches, parametric timed automata, interrupt timed automata, hybrid automata, etc. And each of these variants and extensions are themselves subject of variants and extensions. The same applies to other families of formalisms such as Petri nets.

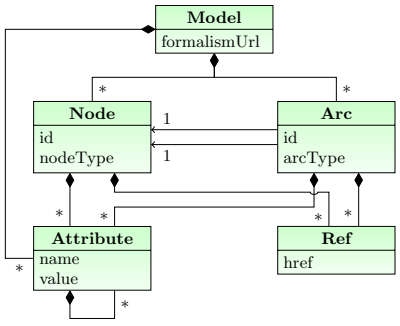
Our model-based solution mainly and directly relies on XML technologies. We use XML as it provides a common, flexible, and very expressive syntax. We decided not to start with a model-based high-level notation such as UML for formalism definition, to keep a lightweight approach, accessible to most tool developers, easy and fast to implement and test. Although verbose, XML can be given a precise semantics and, as XML is both a mature and widely used technology, numerous tools and libraries for manipulating XML files are available.

We defined a two-layered XML-based modelling language, depicted in Fig. 1a:

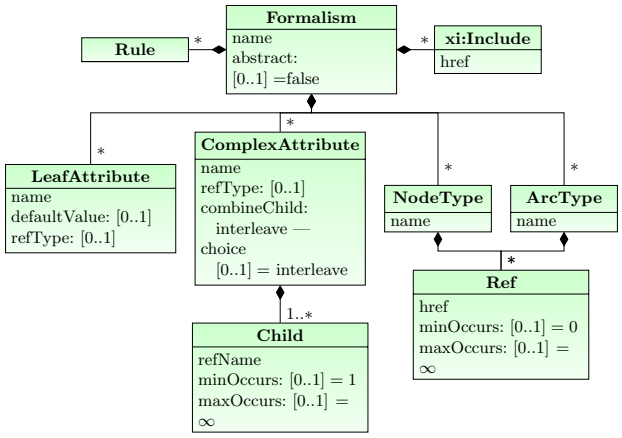
1. FML (Formalism Markup Language), that specifies the concepts of any graph-based formalism and their relationships;
2. GrML (Graph Markup Language), that specifies how a graph-based model, complying with a given FML formalism, is structured.



(a) User formalism and model must comply with FML and GrML



(b) GrML concepts



(c) FML concepts

Fig. 1: FML and GrML concepts

A user-defined formalism must comply with FML constructs, as per the relationship between **User Formalism** and **FML**. A compliance procedure checks the consistency of the user-defined formalism and its structure. Any **User Formalism**, since it is graph-based, is structured by the GrML language.

GrML is a language that describes a specification in the context of a given formalism. In other words, the user domain specific language is defined using FML, and the models of the domain must be structured as graphs in the associated formalism, this being enforced by GrML. By analogy with a model-based specification language such as UML, FML defines the *superstructure* of our framework, while GrML defines its *infrastructure*.

We introduce in more details FML and GrML in the next two subsections.

3.1 FML: Formalism Markup Language

The characteristic elements of formalisms to be used are described using the FML language. FML caters for interdependent formalisms, allowing for hierarchical definitions.

The metamodel of FML, illustrated in Fig. 1c, defines the concepts of a graph-based formalism. When a **Formalism** is made *abstract*, it is intended to serve as the basis (root or intermediate) for a hierarchy of concrete formalisms of the same family. For instance, one may define the abstract graph of Petri nets (places, transitions and arcs), and then build the concrete type place/transition (P/T) net upon this primary definition. Note that "abstract: [0..1] = *false*" denotes that the attribute "abstract" is optional (its cardinality is 0 or 1); the notation "= *false*" indicates that its default value is false.

A **Formalism** is composed of **NodeType**, **ArcType**, **LeafAttribute** and **ComplexAttribute**. A **LeafAttribute** may have a scalar value (attribute "defaultValue"), and may refer to a concrete **NodeType** or **ArcType** (attribute "refType").

A **ComplexAttribute** is a structured attribute, which also allows for hierarchical composition of formalisms. In a Symmetric net for example, the marking of a place, composed of *tokens*, would be defined as an expression denoting a multiset of tuples where each token in a tuple may be a scalar value of a colour domain, or an expression using built-in Symmetric net functions (e.g. successor, predecessor, broadcast). This concept of tuple also exists in P/T nets, but refers there to a simple integer. This gives an example of reuse of the same notion (i.e. ComplexAttribute) with different definitions in different formalisms.

The hierarchy between formalism descriptions can be achieved by the relationship between **NodeType**, **ArcType** and **Ref**. Typically, in the Petri net model of a hierarchical system, submodels may be attached to a place or a transition.

Finally, reuse between formalisms to allow for compositional and incremental definitions is achieved thanks to the relationship between **Formalism** and the **xi:Include** construct. The latter represents the ability to use the XML XInclude technology[11] to import other formalisms. XInclude enables the inclusion of one or several XML documents into another one. This mechanism allows for defining new formalisms by simply composing one or several previously defined formalisms, and only defining the new features, in order to facilitate modularity. The grammar of FML (in RELAX-NG) is available on the *CosyVerif* Web page[12].

3.2 GrML: Graph Markup Language

The structure of a GrML model is described by the metamodel in Fig. 1b. A **Model** is a graph typed by a user formalism (referred to by the attribute

[11] http://www.w3.org/TR/xinclude/

[12] https://forge.cosyverif.org/projects/formalisms/repository/entry/trunk/formalism.rng

"formalismUrl"). It consists of a set of **Node** and a set of **Arc**. The arcs connect the nodes. A node in a GrML model is typed by its "nodeType", declared in the corresponding formalism. The same principle applies to an arc (typed by its "arcType"). A **Ref** represents the link between two elements or between an element and a model (the reference is provided by its "href" attribute). Any referenced element must be identified by an "id" in the containing model file, and any referenced model is identified by the model file name. The reference ("href") value is an URI.

A user model (Fig. 1a) is thus contained in a GrML document, an XML file describing a model and given in the form of an annotated graph. The model, its arcs and nodes can contain attributes given in the form of a tree and must comply with the associated FML description. The grammar of GrML (in RELAX-NG) is available on the *CosyVerif* Web page[13].

Finally note that verification tools implementing our approach can use GrML as an abstract syntax, and hence implement translation from/to their concrete syntax, or directly use GrML libraries.

3.3 Automated Compliance Checking

Our approach includes a mechanism for automatically checking the conformance of a GrML model with respect to its corresponding user-defined FML formalism and the FML language. It works as follows.

First, GrML and FML syntaxes are validated using RELAX-NG. Then, the content of a GrML file is checked against the corresponding description of its associated formalism. To do so, Schematron [13] rules are generated from the XML description of the formalism. Schematron is a rule-based validation language relying on XPATH[14] idioms to query and validate co-occurrence constraints in an XML file. We chose Schematron since an XML grammar cannot capture some particular constraints: for instance, "in a Petri net formalism, no arc should connect two nodes of the same type" (i.e. only arcs between a place and a transition – or vice versa – are allowed). Schematron is also used to perform consistency checks such as correspondence between the declaration of a variable and its usage.

This automated compliance checking is implemented in a freely available tool: GrML-Check[15].

3.4 Ease of Implementation for Data Structures

Since our approach relies on standard XML technologies that are well developed, we are able to easily generate the data structures representing the models, as well as the read and write operations.

[13] https://forge.cosyverif.org/projects/formalisms/repository/entry/trunk/model.rng
[14] http://www.w3.org/TR/xpath20/
[15] https://forge.cosyverif.org/projects/grml-check

The XML Schema is generated using the `trang`[16] utility from the RELAX-NG description. Then, tools are run to generate the API for loading, storing, and manipulating GrML models, in different programming languages:

- JAXB[17] for Java;
- Code Synthesis xsd[18] for C++;
- and other languages that could be made available, such as Python.

This approach has one drawback: it only generates APIs for the generic GrML models, not for a particular formalism. We have to write API generators to wrap the GrML API with the notions defined in each formalism. The automation of this work is currently being explored. It can be solved by writing only one generator per language, that takes a formalism as input and generates the wrapping of the GrML API.

The performance of the libraries is usually very good, as they can load big models very fastly within a reasonable amount of memory. For example, a JAXB-generated parser for PNML P/T models can load a 336 MiB PNML file in 7.90 seconds, with 1 GiB of memory allocated to the Java Virtual Machine (using the `-Xmx` option).

The next section instantiates this modular definition approach on a zoo of formalisms made up of the families of (timed) automata and Petri nets.

4 Application to Automata and Petri Nets

We leveraged the flexibility, compositional and incremental reuse characteristics of FML and GrML to build an architecture of interrelated formalisms, for Petri nets and automata. Our goal here is to obtain an architecture structuring these two families of graph-based formalisms.

4.1 Description

Our proposal is given in Fig. 2. In this proposal, formalisms can *reuse* existing formalisms: for example, `Parametric Timed Automaton` reuses the syntactic features of `Timed Automaton`. Other formalisms can be defined as a restricted version of an existing formalism: this is the case of `Linear Hybrid Automaton` that reuses the concepts of `Hybrid Automaton`, but adds *constraints*. Abstract formalisms as explained earlier, are meant to provide the bridge linking concrete formalisms of the same family, structured in a hierarchical architecture.

In Fig. 2, the composition and incremental reuse takes place from top to bottom. So, the core building blocks appear at the top of the figure, and each

[16] http://www.thaiopensource.com/relaxng/trang.html
[17] http://jaxb.java.net/
[18] http://www.codesynthesis.com/products/xsd/

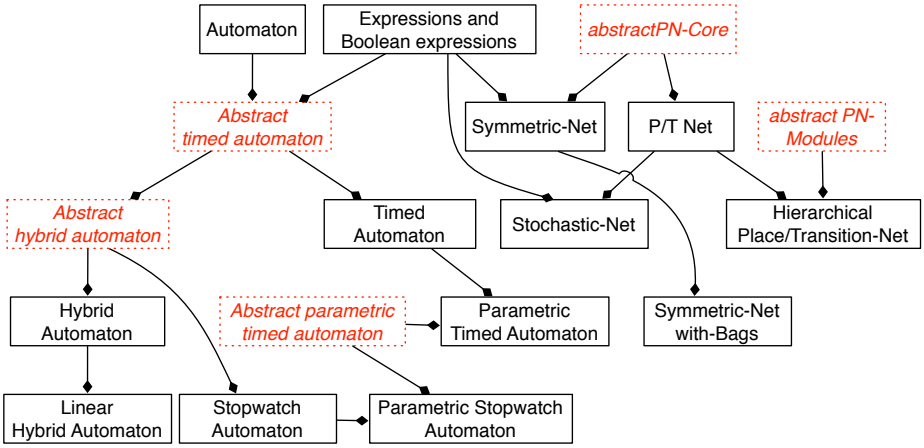


Fig. 2: An architecture of formalisms

formalism (or abstract formalism) in a layer is a potential building block for the formalisms in the layer below[19].

For example, we will create the formalism `Timed Automaton` in this section. It is built upon `Abstract Timed Automaton`, which is itself built upon `Automaton`, described below:

```
<formalism name="Automaton" xmlns="http://cosyverif.org/ns/formalism">
    <leafAttribute name="initialState" />
    <leafAttribute name="finalState" />
    <complexAttribute name="type" refType="state">
        <child refName="initialState" minOccurs="0" maxOccurs="1" />
        <child refName="finalState" minOccurs="0" maxOccurs="1" />
    </complexAttribute>
    <leafAttribute name="name" refType="state" />
    <leafAttribute name="label" refType="transition" />
    <nodeType name="state" />
    <arcType name="transition" />
</formalism>
```

An automaton contains states (defined by the nodeType tag), and transitions (defined by the arcType tag). States have two attributes, "name" and "type" (defined by the leafAttribute and complexAttribute), where "type" is a combination of optional "initialState" and "finalState". Transitions have only a label.

`Abstract Timed Automaton` is defined above `Automaton` by:

```
<formalism abstract="true" name="Abstract timed automaton"
           xmlns="http://cosyverif.org/ns/formalism">
    <xi:include href="automaton.fml" />
    <xi:include href="abstract_expression.fml" />
    <complexAttribute name="declaration" refType="Abstract timed
        automaton">
        <child refName="clocks" minOccurs="0" maxOccurs="1" />
    </complexAttribute>
```

[19] For the sake of readability and saving space, we sometimes depict different formalisms at the same level, although one includes another one (see, e.g. `Stopwatch Automaton` and `Parametric Stopwatch Automaton`).

```
    <complexAttribute name="clocks">
        <child refName="clock" minOccurs="0" />
    </complexAttribute>
    <complexAttribute name="clock">
        <child refName="name" maxOccurs="1" />
    </complexAttribute>
    <complexAttribute name="guard" refType="transition">
        <child refName="boolExpr" maxOccurs="1" />
    </complexAttribute>
    <complexAttribute name="updates" refType="transition">
        <child refName="update" minOccurs="0" />
    </complexAttribute>
    <complexAttribute name="update">
        <child refName="name" maxOccurs="1" />
        <child refName="expr" maxOccurs="1" />
    </complexAttribute>
</formalism>
```

This formalism includes the base `Automaton` formalism and another formalism that describes Boolean and Integer expressions. It defines clocks on the automaton and guards and updates of the clocks on the transitions.

This latter formalism instantiates the `Abstract Timed Automaton` as `Timed Automaton`.

```
<formalism name="Timed Automaton"
           xmlns="http://cosyverif.org/ns/formalism">
    <xi:include href="abstract_timed-automaton.fml" />
</formalism>
```

Note that the `Abstract Timed Automaton` is useful to define others formalisms, like `Hybrid Automaton`. `Timed Automaton` itself becomes a building block for `Parametric Timed Automaton`.

Once the formalism has been defined, the developer can write models in GrML and check them using the GrML-Check tool. He/she can also manipulate models using the GrML library available in his programming language. In a near future, developers will also be able to generate an API for their specific formalisms and use it in their tools.

In this incremental definition approach, for two same concepts in two consecutive layers, the most recent one (in the layer below) subsumes (i.e. merges) the previous one. The XInclude technology allows to specify how to combine elements and attributes of the subsumed concept with those of the including one. Multiple composition is allowed.

Note that, although in theory Petri nets and automata do not share much syntax, we do not have two independent connected components in Fig. 2, as one could expect. Instead, the hierarchy of automata (on the left) and the hierarchy of Petri nets (on the right) share a common formalism, i.e. `Expressions and Boolean expressions`. Furthermore, if time(d) Petri nets are to be defined, they will certainly share some attributes with timed automaton (e.g. the definition of clocks), and hence both `Abstract Time Petri Nets` and `Abstract Timed Automaton` may build upon a new abstract formalism, e.g. `Abstract timed systems`. The same holds for the extension to the parametric case. This shows the interest of reusability in our solution, and of the notion of abstract formalisms.

All the formalisms defined in *CosyVerif* can be found on the *CosyVerif* Web page[20].

4.2 Discussion

Our aim is not only to describe formalisms, but also to ease the development of new formalisms possibly using parts of existing ones. Thus formalisms we use in this framework are not built independently of one another, but factor as much as possible their common features, as the above application shows. Most formalisms are extensions of other formalisms. Maintaining relations of hierarchy or dependency is also an important issue both to navigate through the zoo of formalisms and to ensure consistency in the long run.

4.3 Towards Good Practices

While working on this structured architecture of formalisms, we identified a good practice for defining formalisms, based on *abstract* and *concrete* formalisms. Abstract formalisms (depicted in Fig. 2 in dotted red) define the core of our formalisms; they must be as organised (through inclusion) as possible. Each abstract formalism can include other abstract formalisms, and add new features. Concrete formalisms can include several abstract or concrete formalisms, but should not add new features. They can however add constraints. Of course, only concrete formalisms can be instantiated in a GrML model. This separation between concrete and abstract formalisms is inspired by the object-orientation paradigm.

Finally, an important issue to be addressed is to identify whether tools would still be compatible in case the hierarchy of formalisms is subject to modifications. We should find criteria to allow backward compatibility; in particular, modifications could be performed to the hierarchy, as long as the (abstract) syntax of the formalism supported by the tool remains unchanged.

5 Integration into the *CosyVerif* Platform

This work has been implemented in the *CosyVerif* verification environment [3]. *CosyVerif* aims at gathering within a common interface various existing tools for specification and verification. It has been designed in order to:

1. support different formalisms with the ability to easily create new ones;
2. provide a graphical interface for every formalism;
3. include verification tools called via the interface as Web services; in fact, the provided graphical front end in the *CosyVerif* platform wraps such Web services; any other client (third party product) able to wrap such Web services can be used as well;
4. offer the possibility for a developer to integrate his/her own tool without much effort, also allowing it to interact with the other tools.

[20] https://forge.cosyverif.org/projects/formalisms/repository/entry/trunk/

5.1 Architecture

CosyVerif consists of two components:

- a distributed server (Alligator), which provides an integration framework based on Web Services;
- a client (Coloane), which provides a graphical front-end.

Users may either install the server (containing all verification tools) as a standalone binary, or only install the light client and connect to an existing server.

CosyVerif relies on the use of GrML files describing models that comply with FML describing formalisms. The Coloane client allows to graphically design a model, which is automatically translated into a GrML file.

The *CosyVerif* platform is OS-independent and entirely open source (server, client and verification tools). Alligator is published under the GNU Affero General Public License (AGPL) version 3. Coloane is published under the Eclipse Public License (EPL) version 1.

5.2 Advantages

Among the advantages of *CosyVerif* is the easy use of the platform: the end user can simply install the client, that will automatically connect to an existing server. For the tool developer, integrating a tool into the platform first requires the definition of a FML formalism (if it was not previously available), by reusing portions of formalisms. Then, (s)he only needs to write a parser taking GrML as input. Here again, much reuse can be performed: for formalisms reusing other formalisms, parts of their parser can be reused here as well . The whole operation usually requires less than half a day.

Finally, the platform is client-independent; although we provide Coloane, any home-made client using a provided library or using appropriately the Web service protocol can connect to an existing server as well and benefits from the services provided by the tools integrated in *CosyVerif*.

5.3 Integrated Tools

Up to now, 8 tools (that support GrML input) are available in *CosyVerif*:

- Cosmos [5], a statistical model checker for Petri net with general distribution against specification given as a linear hybrid automaton;
- Crocodile [7], a model checker for Symmetric nets with bags;
- Cunf [4], a toolset for carrying out unfolding-based verification of Petri nets extended with read arcs;
- Imitator [2], a tool for synthesising timing parameters for networks of timed automata augmented with stopwatches;
- LoLA [24], an explicit Petri net state space verification tool;

– ModGraph [18], a tool for the construction and analysis of modular state spaces;
– ObsGraph [15], a BDD-based tool implementing a verification approach for workflows using Symbolic Observation Graphs;
– PNXDD [17], a model checker for place/transition Petri nets based on hierarchically structured decision diagrams.

More details on the tools can be found in the publications related to the tools, as well as in [3]. The integration of other tools is still ongoing.

5.4 Banks of Formalisms and Models

Two major sets of FML formalisms are available so far: Petri nets and timed automata (see Fig. 2). They can be downloaded from the *CosyVerif* repository.

Using the translation facilities offered by some tools to convert models given in their native format to models in the GrML syntax, several lists of benchmarks are now available in the GrML format, and have been grouped on the *CosyVerif*'s Web site [22] (in Downloads → Repository).

A first list of case studies of parametric timed automata comes from IMITATOR [2]. These case studies concern hardware circuits (including original industrial case studies), communication protocols, scheduling problems, as well as some classical case studies from the literature. A second list of benchmarks consists of a large list of Petri nets models and their extensions (including coloured Petri nets), coming from the model checking contests in 2011, 2012 and 2013 at the International Conference on Application and Theory of Petri Nets and Concurrency [17,16].

6 Conclusion and Perspectives

This paper proposes a mechanism to integrate heterogeneous formalisms and associated tools, and enhance interoperability between tools, using a modular formalisms definition approach. Our solution relies on the FML language for describing formalisms, and the GrML language for describing models. An automated compliance check is performed between any GrML model and its corresponding FML formalism. We aim at emphasising the reusability of formalisms as much as possible, so as to ease the engineering of verification tools. Our approach was implemented in the *CosyVerif* platform. A hierarchy of FML formalisms for extensions of Petri nets and automata has been defined and implemented, and several lists of benchmarks are available.

We give below some directions for future research.

6.1 Properties

The approach described in this paper allows the modular definition of formalisms. It can be pushed one step further to cover not only the models, but also their properties and results of tools.

Researchers have defined properties for the models, for instance place bounds or invariants for Petri nets. These properties can be either filled by the modeler or computed by tools. Currently, the tools usually display the result but do not make it easily available to other tools. Thus, it is difficult to reuse the results of one tool into the computations of another tool. Such communication between tools is interesting though. For instance, the place bounds of a Petri net can be used by a model checker (especially those based on decision diagrams) to improve their efficiency.

As we propose a way to define modular formalisms, we propose to provide properties as formalism extensions in the future. It would benefit from an easy integration with the current approach, while allowing better communication and interaction between the tools. This important and much asked-for feature in the model checking community was totally missing in most platforms.

6.2 Semantics

An additional challenge is the ability of composing models defined using different formalisms (e.g. a Petri net with a finite state automaton). This requires semantic information (in order to define, e.g. what kind of variable of a first formalism corresponds to what kind of variable in a second formalism and hence be able to synchronise them). Attaching some semantic information to the FML formalisms is the subject of ongoing work.

Attaching more semantic information to FML formalisms will also make it possible to automatically translate a model described using any FML formalism into a model described using any other FML formalism. Of course, if these formalisms are incompatible (e.g. a timed Petri net can in general not be translated into a finite state automaton), this must be detected. Such a work can be related to the automated composition of logics, with automated feedback for consistency [8].

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, 2012.
3. Étienne André, Lom-Messan Hillah, Francis Hulin-Hubard, Fabrice Kordon, Yousra Lembachar, Alban Linard, and Laure Petrucci. CosyVerif: An open source extensible verification environment. In *ICECCS*. IEEE Computer Society, 2013. To appear.
4. Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient unfolding of contextual Petri nets. *Theoretical Computer Science*, 449:2–22, 2012.
5. Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. HASL: An expressive language for statistical verification of stochastic models. In *VALUETOOLS*, pages 306–315, 2011.

6. Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.
7. Maximilien Colange, Souheib Baarir, Fabrice Kordon, and Yann Thierry-Mieg. Crocodile: A symbolic/symbolic tool for the analysis of symmetric nets with bags. In *ICATPN*, volume 6709 of *Lecture Notes in Computer Science*, pages 338–347. Springer, 2011.
8. Sébastien Ferré and Olivier Ridoux. Logic functors: A toolbox of components for building customized and embeddable logics. Technical report, INRIA, 2006. Available at http://www.irisa.fr/LIS/ferre/logfun/doc/ResearchReportInria0000.pdf.
9. Alexandre Hamez, Lom-Messan Hillah, Fabrice Kordon, Alban Linard, Emmanuel Paviot-Adet, Xavier Renault, and Yann Thierry-Mieg. New features in CPN-AMI 3: Focusing on the analysis of complex distributed systems. In *ACSD*, pages 273–275. IEEE Computer Society, 2006.
10. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
11. Lom Messan Hillah, Fabrice Kordon, Laure Petrucci, and Nicolas Trèves. PNML framework: An extendable reference implementation of the Petri net markup language. In *ICATPN*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2010.
12. ISO/JTC1/SC34. *ISO/IEC 19757-2:2008: Information Technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG*. ISO/IEC, http://relaxng.org.
13. ISO/JTC1/SC34. *ISO/IEC 19757-3:2006: Information Technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*. ISO/IEC, http://schematron.com/.
14. ISO/JTC1/SC7/WG19. ISO/IEC 15909-2:2011. Systems and software engineering – High-level Petri nets – Part 2: Transfer format, 2011.
15. Kais Klai and Hanen Ochi. Modular verification of inter-enterprise business processes. In *eKNOW*, pages 155–161, 2012.
16. Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Lukasz Fronc, Lom-Messan Hillah, Niels Lohmann, Emmanuel Paviot-Adet, Franck Pommereau, Christian Rohr, Yann Thierry-Mieg, Harro Wimmel, and Karsten Wolf. Raw report on the model checking contest at Petri nets 2012, 2012. Technical report, CoRR.
17. Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel. Report on the model checking contest at Petri Nets 2011. *Transactions on Petri Nets and Other Models of Concurrency*, V:121–140, 2012.
18. Charles Lakos and Laure Petrucci. Modular analysis of systems composed of semi-autonomous subsystems. In *ACSD*, pages 185–196. IEEE Computer Society, 2004.
19. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
20. Yang Liu, Jun Sun, and Jin Song Dong. PAT 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199. IEEE, 2011.
21. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley Professional, second edition, 2008.

22. The CosyVerif group. CosyVerif Web page. http://www.cosyverif.org.
23. Matej Urbas and Mateja Jamnik. Diabelli: A heterogeneous proof system. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 559–566. Springer, 2012.
24. Karsten Wolf. Generating Petri net state spaces. In *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2007.