Distributed Component Interoperation and Execution for Norm-Based Real-time Compliance

Theodoros Mitsikas[1,2,*], Ralph Schäfermeier[3], Yousef Taheri[5], Kanae Tsushima[6], Hisashi Hayashi[7], Jean-Gabriel Ganascia[5], Gauvain Bourgne[5], Ken Satoh[6] and Adrian Paschke[1,4]

[1]*Institut für Angewandte Informatik, Leipzig, Germany*
[2]*National Technical University of Athens, Zografou, Greece*
[3]*Leipzig University, Leipzig, Germany*
[4]*Freie Universität Berlin and Fraunhofer FOKUS, Berlin, Germany*
[5]*Sorbonne University, Paris, France*
[6]*Center of Juris-Informatics, Research Organization of Information and Systems, Tokyo, Japan*
[7]*Advanced Institute of Industrial Technology, Tokyo, Japan*

Abstract

A framework for integrating and executing components in a distributed norm-based compliance system is presented. The central component, the planning component, relies on legal checking and ethical checking components, which function as services imposing hard and soft constraints, respectively. These constraints enable the planning component to adapt and replan if necessary, ensuring real-time compliance. The system architecture, component communication, and API are detailed, along with performance evaluation and communication overhead assessment.

Keywords

Real-time compliance, Distributed system, Online HTN planning, Legal compliance, Ethical compliance, Prova, Multi-agent system, Interoperation, RuleML

1. Introduction

The increasing use of Artificial Intelligence (AI) in various domains raises significant legal and ethical concerns. Ensuring compliance with legal regulations and ethical principles is crucial for maintaining trust in AI systems, as highlighted in the European Commission's AI Guidelines [1]. Indicative of the arising problem are attempts by governing bodies to set the guidelines and regulate AI, for example in the European Union (EU) the "AI ACT" [2]. Additionally, AI systems rely on data, which can be subject to GDPR [3] (in the EU) or similar legislation in some non-EU countries.

In the legal compliance field, research includes using modal (deontic) logics [4, 5], natural language processing [6] and logic programming [7]. Computational ethics is a field concerned with computational models of ethical principles. Various models of ethical decision processes have been proposed, depending on the ethical principles being modeled and the expressivity of the representation language. Recent examples include [8, 9, 10]. However, this research does not combine legal and ethical compliance checking, nor aims at a real-time execution.

RuleML+RR'24: Companion Proceedings of the 8th International Joint Conference on Rules and Reasoning, September 16–22, 2024, Bucharest, Romania
*Corresponding author.
mitsikas@central.ntua.gr (T. Mitsikas); ralph.schafermeier@gmail.com (R. Schäfermeier); yousef.taheri@lip6.fr (Y. Taheri); k_tsushima@nii.ac.jp (K. Tsushima); hayashi-hisashi@aiit.ac.jp (H. Hayashi); jean-gabriel.ganascia@lip6.fr (J. Ganascia); gauvain.bourgne@lip6.fr (G. Bourgne); ksatoh@nii.ac.jp (K. Satoh); adrian.paschke@fokus.fraunhofer.de (A. Paschke)
https://www.imise.uni-leipzig.de/Mitarbeiter/Ralph_Schaefermeier (R. Schäfermeier); https://research.nii.ac.jp/~ksatoh/ (K. Satoh); https://www.mi.fu-berlin.de/inf/groups/ag-csw/Members/members/paschke.html (A. Paschke)
0000-0002-7570-3603 (T. Mitsikas); 0000-0002-4349-6726 (R. Schäfermeier); 0000-0003-2134-8420 (H. Hayashi); 0000-0003-3156-9040 (A. Paschke)

In the context of the RECOMP (Realtime Compliance Mechanism for AI) project, and focusing on how AI systems obtain the data and delivering the results while ensuring both legal and ethical compliance, our previous work [11] integrated a legal and an ethical checker that impose constraints on a planning component that controls the real-time data transfer for AI systems that process data. Despite the component- and agent-based design, this system is monolithic, as the components are not clearly separated, for example sharing their databases. Such an approach, while being a valuable proof-of-concept, would face problems in a real-world deployment, for example scalability issues. Moreover, in this monolithic approach, the components should be developed using a common formalism, ignoring other potentially more suitable (more efficient or more expressive) languages for each component's respective task.

To this end, we propose a distributed version of the monolithic system presented in [11], and we present the methodology for converting the existing monolithic system to a distributed system by defining the component communication, optionally using RuleML [12] as their communication API. The usage of RuleML as the interchange format also allows for further development of the system, where different agents implementing different aspects of the legal and ethical compliance checking can be implemented using the most suitable formalism and rule language. In such a case, RuleML would provide a standardized interchange format across the different formalism and syntaxes. To complete the system, we present a Prolog $\leftrightarrow$ RuleML translator, targeting a subset of Prolog and RuleML syntaxes that covers the interchange requirements of the system.

The rest of the paper is organized as follows: Section 2 provides an overview of the use case and the system architecture. The communication of the components is described in Section 3, while the RuleML $\leftrightarrow$ Prolog translator is presented in Section 4. Section 5 examines the performance of the distributed system and discusses the query answering and translation results, while Section 6 concludes the paper and proposes future work.

2. Use Case Description and Architecture

This section introduces the use case and describes the components of the system, while the latter are examined from the point of view of their interaction.

The use case revolves around data moving and processing, while ensuring the compliance of such operations with legal and ethical norms. In particular, it considers tasks such as data processing, while the processing server-node obtains the data through intermediate server-nodes, and returns the result to the user, again using intermediate sever-nodes, as shown in Fig. 1. The location of server-nodes (e.g., inside the EU), data type (personal or non-personal), and processing purpose determine if a data flow adheres to legislation. For instance, organizations cannot transfer personal data of EU residents to countries without GDPR-compatible legislation (e. g., Japan has such a law). Additionally, while GDPR permits non-personal data transfer to third countries, retaining data in countries with strong data protection laws is preferred from an ethical standpoint.

The proposed system initially generates all possible plans for the movement and processing of data through the nodes. Then, it applies legal and ethical compliance mechanisms to select the optimal one, and finally, executes the optimal plan. It is comprised from the following components:

- The Planner (also referred to as "Planning Agent"), which creates possible plans for data moving and processing. It also performs replanning when the environment changes (for example, one node becomes unavailable).
- The Legal Checker, which evaluates each plan with respect to the compliance to the legal rules, and imposes hard constraints to the Planner.
- The Ethical Checker, which selects the best plan (out of all legal plans) with respect to ethical rules, imposing soft constraints to the Planner.
- The Action Executor, which executes the plan issued by the Planner.

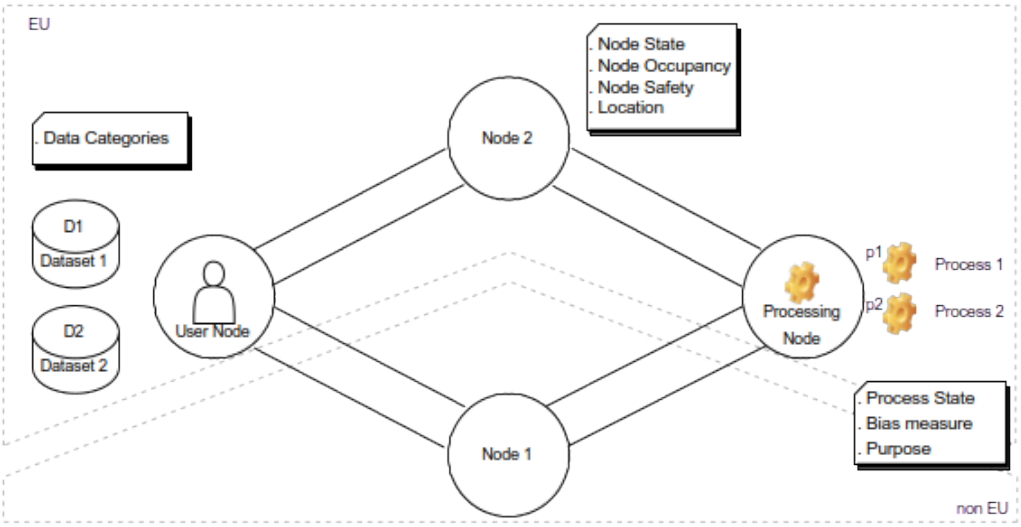


Figure 1: Use case: data moving and processing in a job recommendation plattform.

The Planning Agent runs in SWI Prolog, and relies on the online forward-chaining total-order HTN planning algorithm of Dynagent [13]. For more implementation details, we refer the reader to [11]. In the use case, the Planner aims to deliver job recommendations using personal data stored in the assigned node. The Planning Agent first generates all possible plans, which are then evaluated by the Legal and Ethical Checkers to select the most legal and ethical plan.

As a running example, consider that the following plan is included in the produced plans: 1. user uploads the data `du12`, asking to process them for recommendation purposes, 2. transfer `du12` from the user node to `node_interm2`, 3. transfer `du12` from `node_interm2` to the processing node `node_cloud2`, 4. run the process `p2` using `du12` at the processing node `node_cloud2`, 5. transfer the process output from the processing node to `node_interm2`, 6. transfer the process output from `node_interm2` to the user node. The Prolog representation of the above generated plan is listed below:

```
[load([du12],node_user1,recommendation),
transfer([du12],node_user1,node_interm2,recommendation),
transfer([du12],node_interm2,node_cloud2,recommendation),
run(p2,[du12],node_cloud1,recommendation),
transfer([output(p2,[du12])],node_cloud2,node_interm2, recommendation),
transfer([output(p2,[du12])],node_interm2,node_user1, recommendation)]
```

The list of the generated plans is then conveyed to the Legal Checker. The Legal Checker is implemented in Proleg [14], a language that extends Prolog with exceptions to rules, targeting the easier representation of laws and the exceptions to these laws, and runs in SWI Prolog. It employs rules that can determine if a plan is legal based on e. g., the type of data, the user consent for transfer/processing, the node location, etc. (for more details, we refer the reader to [11]). A rule example is shown below:

```
legal(transfer(_data, _from_country, _to_country, _purpose))
  <= personal(_data),
     gdpr_applicable(_from_country),
     with_consent_of_the_transfer_purpose(_data, _purpose).
```

As the query answering of the Legal Checker is a true/false, the Planner queries the Legal Checker for each created plan.

The list of all legal plans is then communicated to the Ethical Checker, which selects the most ethical plan according to the following criteria:

- **Data minimization (N1)**: Suggests using fewer personal data categories to respect the privacy of the user.

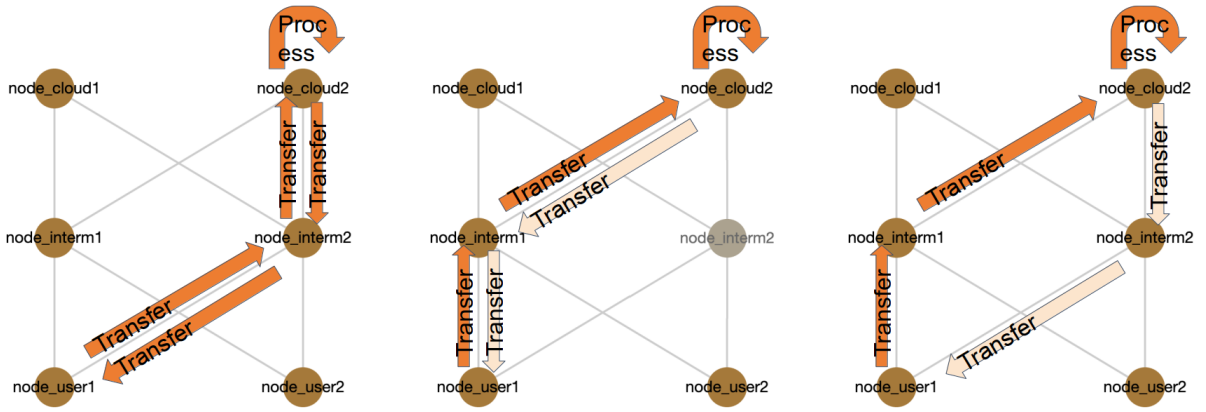
- **Sensitive data (N2)**: Proposes using less sensitive data to respect the privacy and safety of the user.
- **Transfer regions (N3)**: It suggests avoiding transfers outside the legislative zone to protect personal data and the user's safety.
- **Node safety (N4)**: Proposes to avoid less secure storage of personal data to respect the safety of the user.
- **Transfer efficiency (N5)**: Suggests using less busy nodes to increase the efficiency of data transfers and help increase the beneficence of the service.
- **Algorithmic bias (N6)**: Suggests using processing that is not or less biased toward any group in order to respect fairness in providing the service.

The Ethical Checker determines the most ethical plan by pairwise comparisons, where the plan winning the most comparisons for each criterion is selected. It is implemented in Prolog and runs on SWI Prolog; further implementation details can be found in [11]. The Ethical Checker query answer is an integer, representing the index of the most ethical plan in the Planner's proposed list.

After the selection of the legal and most ethical plan, the Planner queries the Action Executor, which is the component that moves or processes data according to commands issued by the Planner. The Action Executor is implemented in the rule language Prova [15, 16], leveraging its reactive agent programming capabilities and its support for reaction rule based workflows. Again, for more implementation details, we refer the reader to [11]. For our running example, assuming the plan presented above is deemed legal and most ethical, and there are no changes in the environment, the execution is shown in Fig. 2a.

The Planner is able to adapt in case of a change in its environment, even when this change happens while executing. For example, if `node_interm2` becomes unavailable at some point during the execution, the Planning Agent replans and produces plans that do not use `node_interm2`. Afterward, it will query again the Legal and Ethical Checkers, in the same way as the initial planning, to obtain the legal plans and the most ethical plan amongst them.

For our running example, we initially assume that the preferred `node_interm2` is unavailable, thus the produced plans do not include it (Fig. 2b). However, during the execution, and after the data were processed, `node_interm2` becomes available. The Planner produces new plans that now include `node_interm2`, and after querying the Legal and Ethical Checkers, the new preferred plan send to the Action Executor includes `node_interm2` (Fig. 2c).



(a) Uninterrupted Execution of Data Moving.

(b) The preferred node `node_interm2` is unavailable, not included in the initial plan.

(c) When the preferred node `node_interm2` becomes available, the Planner produces new plans.

Figure 2: Uninterrupted Execution (a) and Real-time Replanning (b,c).

Expanding the initial monolithic system presented in [11] to a distributed one, the components are decoupled. The Legal and Ethical Checkers (cf. [11]), and the Action Executor are services that accept HTTP requests. They communicate using a text representation of Prolog queries, or exchanging RuleML documents as their API. The second variant of the system, using RuleML as their data and rule interchange format, is depicted in Fig. 3. For the component interaction, multiple instances of the RuleML translator service are shown.

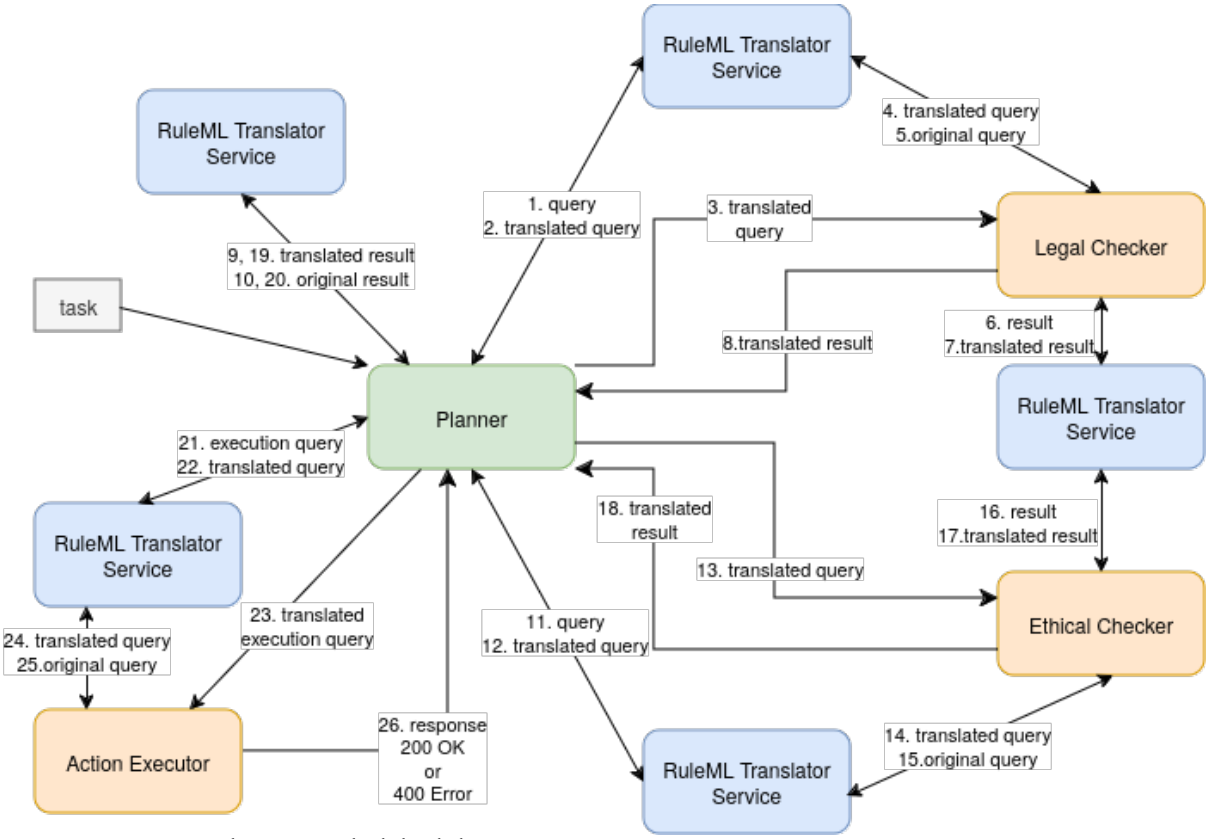


Figure 3: The Architecture and Data Flow.

3. Component Communication and Execution

This section describes the communication between the components, within a distributed architecture.

Both the Legal and Ethical Checkers have a similar Knowledge Base (KB) compared to their non-distributed versions presented in [11]. They are expanded to Prolog servers accepting GET requests containing the query. However, hard-coded facts that are describing "the state of the world" (for example, the location of a server node) are removed, allowing the components to function as global services, not tied to a specific Planner instance.

Moreover, these services do not employ any kind of session management pertaining to asserting facts for a specific Planner instance. On the contrary, the "state of the world" is communicated in each query, and the Checkers answer each query independently of any previous requests from a Planner instance.

The mechanism of reasoning with these temporal and dynamic facts, as well as the parsing and execution of queries that are issued by the Planner, is described after a brief examination of how the Planner constructs and poses queries.

3.1. Planner Queries

After the plan creation, the Planner component poses queries first to the Legal (if a certain plan is legal) and then to the Ethical Checker (select the most ethical plan from the list of legal plans). The query answers are, for the case of the Legal Checker, true or false, while for the case of the Ethical Checker, the index of the preferred plan.

Compared to the monolithic version, the Legal and Ethical Checkers do not have access to the facts (e. g., the state of a node), thus these have to be communicated along with the query. While other options were considered, for example session management where the facts could be communicated and asserted or retracted when needed, we opted to include all necessary facts in each query, while the Legal and Ethical Checkers do not contain any facts that may change during the execution. The reason for this choice lies in additional communication, assertions, and retractions that would be required with each environment change.

Initially, in the monolithic system presented in [11], the evaluation of queries to the Legal Checker subsystem is initiated by calling the `callLegalChecker(+Plan)`. In this case, the sole argument of `callLegalChecker/1` is a Prolog list containing the created plan, for example:

```
[load([du12],node_user1,recommendation),
...
transfer([output(p2,[du12])],node_interm2,node_user1, recommendation)]
```

The facts used by the Legal Checker are, for example, the location of servers (e. g., inside or outside the EU) and the type of data (e. g., personal or non-personal data).

For constructing the query in the distributed version of the system, these facts are gathered in a list, for example:

```
[fact(inEU(node_user1)),fact(inEU(node_interm1)),...]
```

Then, the query to the remote Legal Checker component is constructed, comprised of a single predicate, having two arguments: the first argument is the query that the Legal/Ethical Checker should answer to. The second argument is a list containing the temporal facts (e. g., the location of a node) that are valid for the specific query. The final step converts the query to a string format, to be communicated to the Legal Checker, as seen in the following (indented here for clarity) example:

```
"legal_query(
    [
        load([du12],node_user1,recommendation),
        ...
        transfer([output(p2,[du12])],node_interm2, node_user1,recommendation)
    ],
    [fact(inEU(node_user1)),fact(inEU(node_interm1)),...]
)."
```

The above monolithic to distributed conversion process allowed for a development of a distributed system without a substantial change to the Planner KBs, as invoking the Legal (and similarly, the Ethical) Checker requires only the redefinition of the Planner's calling predicate `callLegalChecker(+Plan)`. Specifically, instead of invoking the Legal Checker directly, the `callLegalChecker/1` is redefined to construct the query, optionally obtain the RuleML translation, issue an HTTP request to the Legal Checker, and subsequently parse the answer, succeeding or failing according to the latter.

For the Ethical Checker, the query construction is similar.

3.2. Parsing and Executing the Query

To highlight the mechanism of parsing the query, the Legal Checker is used as an example. The Legal Checker is wrapped in a Prolog server, that accepts HTTP requests, having the single endpoint `/query`.

```
:- http_handler('/query', handle_query, []).
```

```
handle_query(Request) :-
    member(method(get), Request),
    http_parameters(Request, [q(Query, [])]),
    exec_string_query(Query, Response),
    format('Content-type: text/plain~n~n'),
    format('~w', [Response]).
```

The evaluation of the query is realized via the `exec_string_query/2` predicate, implemented as follows:

```
exec_string_query(S,Reply) :-
    term_string(legal_query(Q,TFacts), S),
    snapshot(legal_query(Q,TFacts,Reply)).
```

The first step involves the parsing of the query, initially in a string format (variable `S`), to a Prolog term of the form `legal_query/2`. Notice that only queries of the form `legal_query/2` are matched. In the second step, `legal_query/2` is called through `snapshot(:Goal)`. The latter, runs the `Goal` after freezing the dynamic predicates, while discarding any changes made by `Goal` after the evaluation. In this case, `legal_query/3` has accesses the dynamic predicates at the moment the evaluation is started, performs the modifications, while the changes to the dynamic predicates are not accessible from other threads, while other threads do not see modifications issued. This allows for updating the KB just for the purpose of each query, isolating different calls of `legal_query/3` and the modifications made by it.

```
legal_query(X, TFacts,Reply) :-
    maplist(assertz, TFacts),
    findall(X,callLegalChecker(X),Reply).
```

In particular, `legal_query/3` first applies an assertion over the temporal facts that are relevant to each query. Notice that these facts should be declared as dynamic. Then it evaluates `callLegalChecker/1`, aggregating the answers through `findall/3`. Although the `callLegalChecker` evaluates to true/-false, we use `findall/3` to maintain the general applicability to predicates that also bind variables. If no solution is found, the response is an empty list, otherwise a list of solutions, both in a string format. This string is given as a response to the Planning Agent, which, in turn, parses the list, and depending on the number of solutions either fails (0 solutions), or utilizes the answer.

Notice that for the Legal Checker, the distributed version uses the same invoking predicate, namely the `callLegalChecker/1`, which is not changed compared to the monolithic version, allowing for a distributed version, while keeping the KB almost intact (except from removing the facts). The same parsing mechanism is employed for the distributed version of the Ethical Checker.

In addition to the communication between the Planner and the Action Executor that already utilized such a translation and communication and is presented in [11], the Planner as well as the Legal and Ethical checkers are expanded to communicate using a RuleML serialization. The queries and the subsequent responses, that are both initially given as a string in the NafHornlogEq subset of Prolog syntax, are translated to RuleML format, in the same level of expressiveness. Conversely, the receiving components query again a Prolog ↔ RuleML translator service for obtaining the original message in a Prolog serialization. The translator is described in the section below.

4. Prolog ↔ RuleML Translation

This section describes the Prolog ↔ RuleML translator used by the components to translate to/from RuleML, which is the API through which the components communicate.

The Prolog ↔ RuleML Translator used was developed in house[1], within the context of this system. It is written in Java, utilizing the ANTLR *LL(*)* parser[2], and provides the same expressivity as the bidirectional

[1]The source code is availabe on GitHub: https://github.com/tmitsi/ruleml-prolog-translator and https://github.com/tmitsi/ruleml-prolog-translator-api

[2]https://www.antlr.org/

translator BiMetaTrans [17]. While BiMetaTrans provides the necessary core functionality, it aims to be included in systems running Scryer Prolog[3], where the Prolog code to be translated/generated consists of actual Prolog terms rather than strings. This requires parsing text into Prolog terms or generating textual representations of Prolog terms. Moreover, the metalogical `'$V'` encoding is not suitable for our system. These two restrictions, namely the need to convert from/to the textual representation of Prolog to/from actual Prolog terms, and the extra required step to convert variables from/to the `'$V'` encoding, necessitated the development of a new translator. However, the system's modular design allows for the utilization of a BiMetaTrans-based translation web service, if it becomes available.

The ANTLR grammars of the Prolog ↔ RuleML Translator consists of a Prolog parser/lexer, and the XML parser and lexer. The ANTLR code for both the Prolog and XML parsers/lexers consists of pure grammars without actions (i. e., embedded native Java code), instead using the design patterns of *listeners*, except for the query identification. This is needed to avoid possible confusion between simple queries and facts. An abridged version (omitting, most importantly, the lexer, a more complex rule for the equality definition, and the query identification) of the Prolog grammar that highlights the subset of Prolog that is supported can be found below:

```
Document ::= (Assert | Query)* 'EOF'
Assert ::= (Implies | Fact)+
Implies ::= Conclusion  ':-'  Goal  '.'
Fact ::= Conclusion '.'
Query ::= ('?-')? Goal
Conclusion ::= CompoundTerm

Goal ::= Conjunction
    | Disjunction
    | CompoundTerm
    | Naf
    | '(' Goal ')'
Conjunction ::= ( Naf | CompoundTerm | '(' Goal ')' )
          ( ',' ( Naf | CompoundTerm | '(' Goal ')' ) )+

Disjunction ::= ( Naf | CompoundTerm | Conjunction | '(' Goal ')' )
          ( ';' ( Naf | CompoundTerm | Conjunction | '(' Goal ')' ) )+
Naf ::= '\+' (CompoundTerm | '(' Goal ')')

CompoundTerm ::= Atom '(' Arguments ')'
    | Equality
    | Atom
List ::= '['  Arguments  Repo? ']' | '[]'
Repo ::= RestOp Argument
RestOp ::= '|'

Equality ::= Argument '=' Argument
Function ::= Atom '(' Arguments ')'

Arguments ::= Argument (',' Argument)*
Argument ::= Atom
    | Variable
    | List
    | Number
    | Function
    | Equality
    | '(' Argument ')'
```

Although the majority of the grammar rules are straightforward, conjunctions, disjunctions, and negation as failure (`Naf`) are more complex. This is to avoid left-recursive rules that are not supported

[3] https://www.scryer.pl/

by ANTLR, and to ensure the traditional operator priority in Prolog between conjunction, disjunction, and negation as failure. Thus, a conjunct can be a disjunction enclosed in parentheses (captured by `'(' Goal ')'`), while a disjunct can be a conjunction (captured by `Conjunction`, or, if it's inside parentheses, by `'(' Goal ')'`). Similarly, negation applies either to a term or to a parenthesized goal. In addition (not shown in the above grammar), to avoid left recursion and to disallow chains of equality, `Equality` is composed of the alternatives of `Argument` except for `Equality`, possibly inside parentheses, and the equal sign.

As seen above, the grammar also allows for recursive nesting of terms, allowing for deeply nested terms. As in [17], nested Prolog terms are interpreted as functions, and are translated to an expression that contains the function, while the enclosing terms are interpreted as relations/predicates. For example, the Prolog snippet '`p(q(a))`' will be translated as follows to RuleML:

```
<Atom>
    <Rel>p</Rel>
    <Expr>
        <Fun>q</Fun>
        <Ind>a</Ind>
    </Expr>
</Atom>
```

Notice the difference between the Prolog notion of 'atom' (as in e. g., predicate name), and RuleML's notion of 'atom' (as in a predicate application).

The RuleML translation to Prolog is straightforward, utilizing the direct correspondence of the XML-based RuleML tags to the equivalent Prolog syntactic structures. One technical difficulty is the different handling of tags, for example the `<Rel>` element, which includes only the predicate name as its content and is enclosed in an `<Atom>`, while the `<Atom>` element content consists of multiple elements. This is solved either by utilizing the *visitor* design pattern or by specifying the traversing of the relevant tree nodes when generating the Prolog translation.

The translator is embedded in a server that accepts HTTP requests, where the parameters specify the translation direction and the content to be translated. In addition, one parameter distinguishes between Prolog KBs and queries, allowing queries to be specified without the '`?-`' prefix. Thus, a Prolog request or response can have separate queries, while the RuleML is agnostic in that respect, given that the tag `Query` already distinguishes facts from queries. The frameworks utilized for the server implementation are Spring[4] and OpenAPI[5]. There is also a frontend available, seen in Fig. 4.

Expanding the Legal and Ethical checkers to include the Prolog ↔ RuleML translation is straightforward, utilizing a library that provides the predicate `translate_ruleml/5`, as well as its variant, `translate_ruleml/3`. The former accepts three arguments for the Prolog KB, the Prolog query, and the RuleML serialization, as well as two arguments for the translation server location (`Host` and `Port`), as follows:

```
translate_ruleml(Prolog,Query,RuleML,Host,Port)
```

where their implementation features an HTTP request to the translation server specified by the `Host`.

Its variant, accepting three arguments, aims to be used when the translation server location is known. Both variants have two modes, each corresponding to a single direction of translation:

```
translate_ruleml(+Prolog,+Query,-RuleML)        (Prolog->RuleML)
translate_ruleml(-Prolog,-Query,+RuleML)        (RuleML->Prolog)
```

requesting a specific translation direction, while providing the translation to the output argument (`-RuleML`, or `-Prolog` and `-Query`, respectively).

[4] https://spring.io/
[5] https://www.openapis.org/

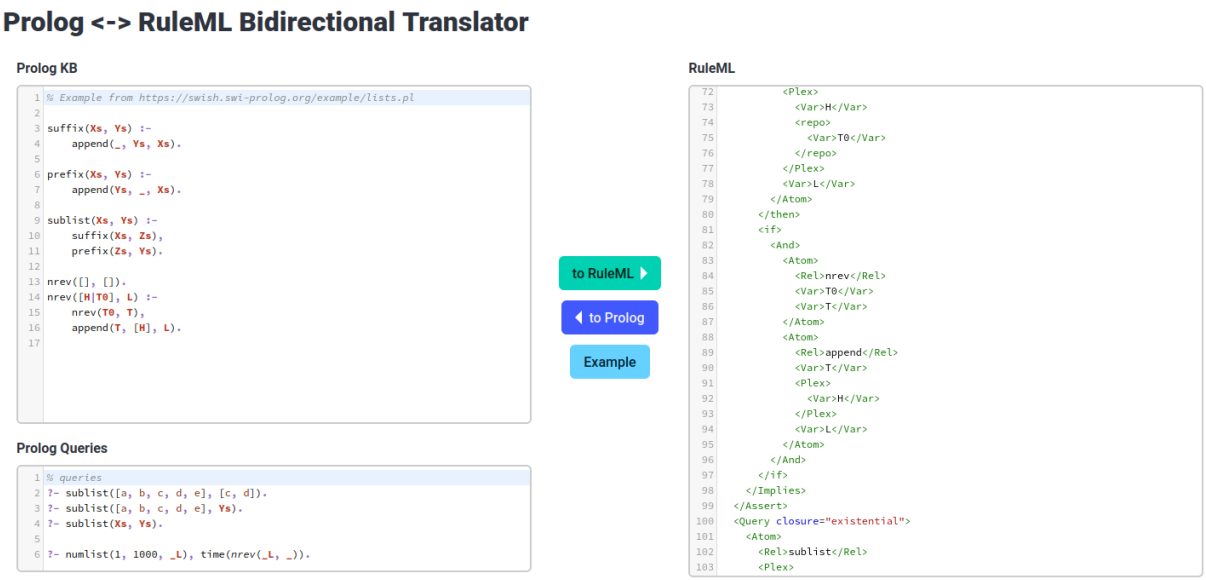


Figure 4: The Prolog ↔ RuleML Translator frontend.

5. Discussion

This section presents the findings with respect to the query answering, the system performance under different configurations and test scenarios, as well as the communication and translation overhead. Notice that for better comparison, the execution part, originally a task performed by the Action Executor, is omitted and has been replaced with a noop component. This is because the Action Executor is distributed since the start of its implementation (see [11]), and its benchmark is out of the scope of this paper.

Three variants of the system were tested; the monolithic system presented in [11], its distributed variant that omits the Prolog ↔ RuleML translation step, and a distributed variant that includes the Prolog ↔ RuleML translation step. The comparison of these three systems provides a comparison of the results with respect to the query answering of each variant, as well as a clear view of the performance overhead.

Regarding the query answering, the three systems provided identical answers, and all tests were successful. This demonstrates a viable methodology to convert a monolithic system to a distributed one, namely through the following steps: 1. all facts describing the state of the world are moved to the coordinating component (here, the Planner), which, 2. constructs a query containing in its positional arguments the original query and the list of the temporal facts, and sends it to the relevant subcomponent 3. the subcomponent parses the query, and using `snapshot/1` asserts the communicated facts and executes the query. In addition, it indirectly verifies the correctness and invertibility of the Prolog ↔ RuleML Translator, at least for the subset of the Prolog/RuleML syntax that was applicable for this system and use cases. Namely, the typical translated queries of the use cases are single-term queries, but contain nested lists and nested terms, and are relatively large. For example, the following is a shortened query that omits seven out of eight plans, and nineteen out of twenty-one facts:

```
ethical_query([[load([du11],node_user1,recommendation),
    ↪transfer([du11],node_user1,node_interm2,recommendation),
    ↪transfer([du11],node_interm2,node_cloud2, recommendation),
    ↪run(p3,[du11],node_cloud2,recommendation),
    ↪transfer([output(p3,[du11])],node_cloud2,node_interm2, recommendation),
    ↪transfer([output(p3,[du11])],node_interm2,node_user1,recommendation)],...],
    ↪[belief_e(dataCategory(du11,c1)),..., belief_e(nodeOccupancy(node_cloud2,normal))])
```

This query, compared to the query posed by the Planning Agent to the Ethical Checker in the monolithic version where the list of eight plans is already a part of the query, is enclosed in the `ethical_query`

predicate, and is further augmented in size only by the list of relevant facts residing in the list of its last argument.

Regarding the performance of the three variants, benchmarking tests were conducted. The tests were performed in SWI-Prolog (threaded, 64 bits, version 9.3.7), on a laptop running OpenSUSE Tumbleweed, Linux kernel version 6.9.3-1, equipped with 2 × Intel® Core™ i5-7200U CPU @ 2.50GHz and 7.5 GiB of RAM, while all servers (Planning Agent, Legal Checker, Ethical Checker, Prolog ↔ RuleML Translator) were deployed locally. The Prolog ↔ RuleML translator service was deployed from IntelliJ IDE and in a Tomcat server.

For the testing purpose, various testing scenarios were obtained from [11], and are shown in Table 1. The tested scenarios include a scenario (4.1) for which replanning is necessary as the state of the nodes changes amid the ongoing execution, and is depicted in a previous section in Fig 2b and 2c.

Scenario	Scenario Description
1	normal execution
2.1	`node_interm2` becomes inactive after getting the data at `node_user1`.
2.2	`node_interm2` becomes inactive after getting the data at `node_user1`. `node_interm2` becomes active after processing the data at `node_cloud2`.
3.1	Non-EU nodes (`node_interm1`, `node_cloud2`) are not allowed, which is checked by the legal checker.
3.2	Processor `p1` is not allowed, which is checked by the legal checker.
4.1	`node_interm2` becomes busy and `node_interm1` becomes available after getting the data at `node_user1`. `node_interm2` becomes available and `node_interm1` becomes busy after processing the data at `node_cloud2`.

Table 1
Testing Scenario Description.

The results are presented in Table 2. Converting the monolithic system to a distributed one without the Prolog ↔ RuleML Translation step, requires aggregating the relevant facts, a series of conversions of Prolog terms to strings, an HTTP request, the parsing of the latter, conversion from string to a Prolog term, and building and parsing the response. These additional tasks induce an overhead, that is, in general, equivalent to the execution time. The distributed system that also includes the Prolog ↔ RuleML Translation step, for each query, induces a further overhead of about one order of magnitude. This is attributed to additional HTTP request, the translation, and the response building and parsing, twice for each query (one for obtaining the RuleML, one for the other direction). In addition, the XML-based RuleML translation has a significantly greater size compared to the Prolog serialization.

Scenario	Monolithic	Distributed w/o RuleML	Distributed w/ RuleML
1	0.1193	0.2807	0.8239
2.1	0.1391	0.2591	0.9338
2.2	0.1972	0.3397	1.5434
3.1	0.0439	0.0966	0.6881
3.2	0.0982	0.1492	0.7410
4.1	0.2540	0.3753	1.7074
Total	0.8519	1.5008	6.4378

Table 2
Execution Time (sec).

Despite the overhead of the Prolog ↔ RuleML Translation, such a feature could be beneficial. On one hand, the communication through a machine-readable oriented format such as the XML-based RuleML allows for the validation of each query against the enclosed schema[6], and for ensuring the data integrity. On the other hand, the usage of a rule interchange standard RuleML [12], possibly expanded

[6] https://github.com/RuleML/deliberation-ruleml/blob/1.03/xsd/nafhologeq.xsd

to include semantic profiles [18], could allow for the implementation of any component of the system in any suitable declarative language or formalism, and not be limited to Prolog.

6. Conclusions and Future Work

We presented a distributed system targeting real-time compliance to legal and ethical rules. Specifically, we focus on the component communication and interaction of the central component, the Planner, with the Legal Checker and Ethical Checker components. Utilizing a use case concerning data moving between nodes, we described the steps of converting the initial, monolithic system to a distributed one, while performing minimal changes to the Knowledge Bases (KBs) of the components. In addition, we proposed RuleML as the component communication API, and we described the bidirectional translation from/to the component language, Prolog, to/from RuleML.

Compared to the initial monolithic system, the developed distributed system provides the same answers and the same functionality, in both cases of direct communication, or using the RuleML interchange format for the component communication. This demonstrates the viability of such a methodology for converting monolithic systems to distributed, accounting for future scalability, fault-tolerance, and formalism-independence of components.

Based on the evaluation, possible future work could focus on the performance of the Prolog $\leftrightarrow$ RuleML translator, for example by leveraging the visitor design pattern to generate the Prolog translation from the RuleML content. The translator can also be extended to include semantic profiles, allowing for other implementations of the Planner, and the Legal and Ethical Checker components, possibly in other formalisms and rule languages. One possible direction could be also the usage of BiMetaTrans [17] as the backend of the translator component, keeping the same API calls, and a subsequent comparison of the two translators. If such an implementation is possible, a reimplementation of (some) components in Scryer Prolog would allow for embedding the translator and eliminating one bidirectional translation step, thus potentially improving the performance.

Acknowledgments

This work has been partially funded by the Agence Nationale de la Recherche (ANR, French Research Agency) project RECOMP (ANR-20-IADJ-0004), Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project RECOMP (DFG – GZ: PA 1820/5-1), JST AIP Trilateral AI Research Grant No. JPMJCR20G4, JST Mirai Program Grant No. JPMJMI23B1, and JSPS KAKENHI Grant No. 22H00543 and 21K12144.

References

[1] the High-Level Expert Group on AI, Ethics guidelines for trustworthy AI, https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai (2019).

[2] European Commission, Proposal for a regulation of the european parliament and of the council laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts, https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52021PC0206 (2018).

[3] European Commission, Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[4] G. Governatori, et al., Designing for compliance: Norms and goals, in: Proc. of RuleML2010, 2011, p. 282–297.

[5] M. B. van Riemsdijk, et al., Agent reasoning for norm compliance: a semantic approach, in: Proc. of AAMAS 2013, 2013, pp. 499–506.

[6] G. Contissa, et al., Claudette meets GDPR: Automating the evaluation of privacy policies using artificial intelligence, https://ssrn.com/abstract=3208596 (2018).

[7] F. Chesani, et al., Compliance in business processes with incomplete information and time constraints: a general framework based on abductive reasoning, Fundamenta Informaticae 161 (2018) 75–111.
[8] S. B. Naveen Sundar Govindarajulu, On automating the doctrine of double effect, in: Proc. of IJCAI 2017, 2107, pp. 4722–4730.
[9] A. Saptawijaya, L. M. Pereira, Logic programming for modeling morality, Logic Journal of the IGPL 24 (2016) 510–525.
[10] F. Lindner, R. Mattmueller, B. Nebel, Moral permissibility of action plans, in: Proc. of AAAI 2019, 2019, pp. 7635–7642.
[11] H. Hayashi, T. Mitsikas, Y. Taheri, K. Tsushima, R. Schäfermeier, G. Bourgne, J.-G. Ganascia, A. Paschke, K. Satoh, Multi-agent online planning architecture for real-time compliance, in: Proceedings of the 17th International Rule Challenge and 7th Doctoral Consortium RuleML+RR 2023, volume 3485, CEUR, 2023. URL: https://ceur-ws.org/Vol-3485/.
[12] H. Boley, The RuleML knowledge-interoperation hub, in: J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, D. Roman (Eds.), Rule Technologies. Research, Tools, and Applications, Springer International Publishing, Cham, 2016, pp. 19–33.
[13] H. Hayashi, S. Tokura, T. Hasegawa, F. Ozaki, Dynagent: An incremental forward-chaining HTN planning agent in dynamic domains, in: M. Baldoni, U. Endriss, A. Omicini, P. Torroni (Eds.), Declarative Agent Languages and Technologies III, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 171–187.
[14] K. Satoh, et. al., PROLEG: An implementation of the presupposed ultimate fact theory of Japanese civil code by Prolog technology, in: New Frontiers in Artificial Intelligence, Springer Berlin Heidelberg, 2011, pp. 153–164.
[15] A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, M. Schroeder, Prova: Rule-based Java scripting for distributed Web applications: A case study in bioinformatics, in: T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, J. Wijsen (Eds.), Current Trends in Database Technology – EDBT 2006, Springer, Berlin, Heidelberg, 2006, pp. 899–908.
[16] A. Kozlenkov, Prova Rule Language version 3.0 User's Guide, 2010. URL: https://github.com/prova/prova/tree/master/doc.
[17] M. Thom, H. Boley, T. Mitsikas, Invertible bidirectional metalogical translation between Prolog and RuleML for knowledge representation and querying, in: V. Gutiérrez-Basulto, T. Kliegr, A. Soylu, M. Giese, D. Roman (Eds.), Rules and Reasoning, Springer International Publishing, Cham, 2020, pp. 112–128.
[18] A. Paschke, Reaction RuleML 1.0 for rules, events and actions in semantic complex event processing, in: A. Bikakis, P. Fodor, D. Roman (Eds.), Rules on the Web. From Theory to Applications, Springer International Publishing, Cham, 2014, pp. 1–21.