Model Checking Normative Systems

Florine Willemijn de Geus
florine@fwdg.nl

September 29, 2022, 69 pages

Academic supervisor: Dr. L. Thomas van Binsbergen
l.t.vanbinsbergen@uva.nl
Daily supervisor: Christopher Esterhuyse MSc
c.a.esterhuyse@uva.nl
Research group: Complex Cyber Infrastructure (CCI)
Informatics Institute
University of Amsterdam
https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering

Abstract

Software systems that handle personal or otherwise sensitive data are often subject to laws, policies, regulations or other kinds of norms. The formalisation of these norms can enable automatic compliance checking of these systems. One method of doing so is through the domain-specific language eFLINT [1], which allows for the specification and assessment of normative systems. The current development tools that are available to interact with eFLINT specifications can reason over concrete scenarios. However, there are purposes for which it might be beneficial to reason over abstract properties instead. In this thesis, we present eLTL, a temporal logic for specifying these properties and eFLINT-CHECK, a model checker to check these properties against their corresponding norm specifications. By extending the eFLINT language with the possibility to express properties in eLTL and by compiling eFLINT specifications and these properties into models that can be checked by the symbolic model checker NUXMV, it becomes possible to exhaustively check the correctness specifications, and provide comprehensive counterexamples in case a given property is violated. We show that eFLINT-CHECK can check realistic specifications in a reasonable amount of time, but also identify cases in which the performance of eFLINT-CHECK is insufficient. We evaluate and demonstrate the use of eFLINT-CHECK for the different purposes where abstract property specification and model checking might be warranted. Finally, we make recommendations for improving the ease in which eLTL properties could be specified, identify different ways in which we can improve the performance of eFLINT-CHECK and propose how eFLINT-CHECK may be integrated into the tools that already exist to interact with eFLINT specifications.

Contents

Chapter 1

Introduction

Software systems, and the data they collect and process have become unavoidable and indispensable to our society. In less than a decade, data has become the most valuable resource in the world [2]. However, due to a lack of clear policies and practices, the barrier to combining and exchanging different data sources (for example, between different academic and medical institutions) may be high [3], therefore stifling opportunities for collaboration and innovation.

To address these concerns, policies and regulations are being created and used to establish agreements between providing and receiving parties. A notable example of such a regulation is the General Data Protection Regulation (GPDR) [4], but they may also be established on a consortium level. An active area of research is on the formalization of such policies, regulations and other norms in a way that checking of compliance with these norms can be done automatically and as part of the software systems that interact with them.

Currently, there exist several projects that allow for the formalization of these norms by writing them down as specifications using a domain-specific language (DSL). These specifications can be used in conjunction with different tools that allow for interacting with these specifications and to check and report on whether these interactions are compliant with the underlying norms [1, 5–7]. Checking compliance with (the formalization of) a norm can happen offline, by checking complete (hypothetical) interaction scenarios, or online, by checking interactions as they happen. The first way of checking compliance is useful for the development of and implementation of these formalizations, whereas the second one enables the integration of these formalizations in running software systems. Both ways, however, share the fact that they rely on concrete interactions. This means that in order to be able to gain absolute confidence in the correctness of a norm specification, every possible combination of interactions should be explored. As these specifications grow in size (and with that, the number of possible interactions as well), this becomes increasingly harder to do manually. Additionally, norms are subject to change over time, which in turn may lead to inaccurate specifications.

Both of these issues could benefit from the idea of using properties that capture the norms that are formalized in these specifications on a more abstract level. These properties can be used to help guide the design and implementation of specifications, especially when they are meant to be integrated into a larger software system. More importantly, however, by expressing these properties in a formal temporal logic, it becomes possible to apply existing model-checking techniques to verify the specifications that implement these properties [8]. Doing so could give confidence in the correctness of these specifications with regard to the norms they formalize, both as they are implemented as well as when these underlying norms change.

1.1 Research questions

In this thesis, we set out to explore how using abstract properties together with existing model-checking techniques might benefit the formalization, specification and compliance assessment of norms. To be able to do so, we have identified the following research questions we want to answer in this thesis:

RQ 1. What kinds of properties are necessary for abstract reasoning about norm specications and being able to model-check them?

(a) What are the purposes for which model checking of norm specifications would be required?
(b) How can these purposes guide the formalization of properties?

RQ 2. How can norm specifications be represented formally, such that counterexamples can be generated for the properties identified by RQ 1?

RQ 3. How can the results that follow from model checking a norm specification be presented to the user in an actionable and intuitive way?

RQ 4. What is the practical viability of adopting model checking of norm specifications for the purposes we identified in RQ 1(a)?

1.2 Research method

We will answer the research questions we identified in the previous section using eFLINT [1], which is a DSL for the specification and automatic assessment of norms. We will design and implement an extension of this language for the declaration of abstract properties and eFLINT-CHECK, which will serve as a tool for model-checking specifications using these abstract properties. We will evaluate and demonstrate the use and expressiveness of this property language and eFLINT-CHECK through a case study using a subset of the GPDR, and we will evaluate the viability of our tool in terms of performance and scalability.

1.3 Contributions

With this thesis, we aim to make the following contributions:

1. **A theoretic approach to property specification for normative systems.** This theoretic approach will answer RQ 1(a), provide us with the background necessary for answering RQ 1(b), and serve as the basis for the design, implementation and evaluation of eFLINT-CHECK.
2. **A proof-of-concept extension to eFLINT for the specification of properties.** We will design and implement a language extension to eFLINT based on our theoretic approach to property specification for normative systems. This will complete our answer to RQ 1(b).
3. **A proof-of-concept tool to model-check eFLINT specifications using abstract properties specified with our language extension.** With the design and implementation of eFLINT-CHECK, we will be able to answer both RQ 2 and RQ 3.
4. **A case study to evaluate and demonstrate the eFLINT language extension for specifying properties together with eFLINT-check.** With this case study, we apply our theoretic approach to a practical example, thereby completing the answer to RQ 1. Moreover, we use this case study to reflect and make recommendations on the design of eFLINT-CHECK.
5. **An evaluation of the performance and scalability of eFLINT-check.** With this evaluation, we will answer RQ 4.

1.4 Outline

In Chapter 2, we describe the background of this thesis and explain the concepts that we will use and refer to throughout the subsequent chapters. In Chapter 3, we will discuss the different purposes for which we might specify and check abstract properties against eFLINT specifications. We also use the theory from the previous chapter to define a formalization for properties over normative systems. Chapter 4 describes the design of the eFLINT-CHECK. Using the design layed out in this chapter, we discuss the implementation of our language extension and eFLINT-CHECK in Chapter 5. In Chapter 6, we perform a case study based on the GPDR to evaluate the use of abstract properties for the formalization of norms and demonstrate how eFLINT-CHECK can be used to check these properties. Additionally, we will evaluate the performance and scalability of eFLINT-CHECK. We discuss and reflect on the results of the GPDR case study and performance evaluation of eFLINT-CHECK in Chapter 7. Chapter 8 discusses the work related to this thesis. Finally, we give our concluding remarks in Chapter 9.

Chapter 2

Background

In this chapter, we will give an introduction to normative systems and how they can be specified using the domain-specific language eFLINT [1]. Moreover, we will introduce the concept of and theory behind model checking.

2.1 Normative systems and their specification

To be able to understand *normative systems*, we first have to define what *norms* are. Although this term has different interpretations across different disciplines, norms can generally be defined as regulations of interactions between individuals or institutions [9]. While norms can be used to regulate interactions between physical systems, such as access control policies or firewalls in computer systems, most often norms are part of the *social reality*. Social reality, a term introduced by Searle [10], denotes the reality that only exists through the agreement between institutions and individuals. Norms that exist in the social reality are often bound by contracts, policies or laws. Because it is not always possible to represent and enforce these norms in the physical world, it becomes possible to violate them. For example, it is prohibited to drive a car without having a driver's license, but there is no physical barrier that prevents someone without a license from doing so. Using this definition of norms, we can define a normative system as a collection of individuals or institutions (or in general, *actors*) and the normative relations and their consequences between these actors.

We can theoretically reason about norms using the legal framework introduced by Hohfeld [11]. In this framework, it is observed that the duty or power an actor has with respect to a certain action is always related to another actor. Here, we make the distinction between *power-liability* and *duty-claim* relations between two actors with respect to some action. In a power-liability relation, one actor has the power to take a certain action A, while the other actor is liable in the sense that they are bound to the effects of A. In the case of a duty-claim relation, one actor has a duty to perform A, while the other actor has a claim to A being done.

2.1.1 Normative systems by example

To illustrate these relations, we use the act of receiving and providing tutoring on a course assignment. A person ('Alice') can ask another person ('Bob') to help them with their course assignment (we will call this action A), thus creating a power-liability relation between Alice (who is in power) and Bob (who is liable) with respect to A. Provided that Bob accepts this request, this action creates a duty-claim relation between Alice and Bob, where Bob has the duty to provide the promised tutoring, with Alice having a claim to this action. Because tutoring is only effective before the assignment is due, this duty will be violated once the submission deadline of the assignment has passed. The duty for Bob to provide the requested tutoring to Alice can only be resolved by providing said tutoring (an action which we will call P). This introduces yet another power-liability relation between Bob and Alice with respect to P, but this time with Bob being in power. These relations are visualized in Figure 2.1. Once Bob has provided the requested tutoring before the assignment is due, the normative relation between Alice and Bob is terminated.

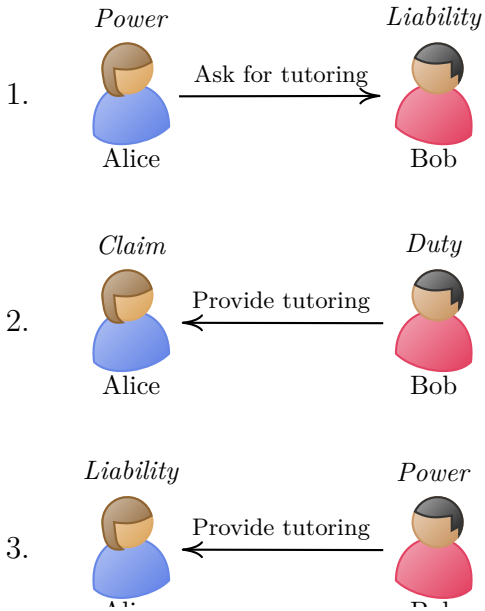


Figure 2.1: Normative relations between actors in the example of asking for and providing tutoring.

2.1.2 Normative systems in eFLINT

The powers and duties illustrated in the example above can be formally expressed in eFLINT. An eFLINT *specification* is a collection of type declarations, which are used to describe the normative system it represents. Such specifications can be built using four types of declarations: *facts*, *acts*, *events* and *duties*.

Facts *Facts* are used to represent knowledge about the system that is specified. Facts can be declared as *atomic* string or integer types, and can be unbounded or bounded to a finite domain:

```
Fact person Identified by String            // Unbounded string fact.
Fact grade  Identified by 1..10             // Bounded integer fact.
```

Additionally, *record-type* facts are facts that are composed of one or more named fields. These fields, in turn, are (aliases for) other fact types.

```
Placeholder student For person
Placeholder tutor   For person

Fact tutor-of         Identified by tutor * student
Fact deadline-passed Identified by student
```

As mentioned before, facts can be used for knowledge representation. This is done in eFLINT's knowledge base, which records the knowledge on instances of the declared types in the specification. This pertains not only to fact types but to acts, events and duties – which we will introduce in subsequent paragraphs – as well. Valid instances of the fact types declared in the examples shown here are `person(Alice)`, `grade(8)` (`grade(11)` would not be a valid instance, since 11 falls outside the specified domain), `tutor-of(person(Alice), person(Bob))`, or `deadline-passed(person(Chloe))`. Note that for record-type facts, the type of the field can be inferred by the eFLINT reasoner, so it is also allowed to write `tutor-of(Alice, Bob)`, for example. The eFLINT knowledge base can be modified by *creating*, *terminating* or *obfuscating* fact instances. When a fact instance is created, it is added to the knowledge base and evaluated to *True*. When a fact is terminated, it is still part of the knowledge base, but it will evaluate to *False*. When a fact is obfuscated, it is removed from the knowledge base altogether, making it impossible to know whether the fact is true or false. A fact is stated to *hold* if it is present in the knowledge base and evaluates to *True*. Facts may also be added to the knowledge base through

derivation from other facts:

```
Fact is-student Identified by student
                Holds when (Exists tutor: tutor-of(tutor, student))
```

Furthermore, the domains of all type declarations including facts may be constrained by a Boolean expression. For example, we could add a constraint to the `tutor-of` fact declaration that specifies that a person cannot be their own tutor:

```
Fact tutor-of Identified by tutor * student
              Where tutor != student
```

Lastly, a special type of fact declaration is the `Bool` fact declaration. As the name suggests, this fact type denotes atomic boolean propositions. `Bool` declarations only have one instance:

```
Bool course-active
```

Acts *Acts* can be used to describe *power-liability* relation between a performing actor (the one in power) and a receiving actor (the one liable). Similar to facts, acts can be derived. Additionally, they may be conditioned by one or more clauses. An act is enabled when at least one of the derivation rules and all of its conditions hold. Note that for the derivation clause, we use the same condition as for the domain constraint for the `tutor-of` fact declaration. This goes to show that there is a high amount of flexibility in the way norms can be formalized using eFLINT, and different specifications can be bisimilar. Facts may be created, terminated or obfuscated as an effect of acts:

```
Act request-tutoring
    Actor          student
    Recipient      tutor
    Holds when     tutor != student                // A student cannot tutor themselves.
    Conditioned by Not(tutor-of(tutor, student)), // A student cannot ask their current
                                                  // tutor to tutor them.
                   course-active                  // A student can only ask for tutoring
                                                  // during an active course.
    Creates        tutor-of(tutor, student)       // A new tutoring relation is created.
```

Events *Events* are similar to acts, with the only difference being that they do not have a performing and receiving actor. Events are used to describe actions that do change the institutional view (*i.e.*, the eFLINT knowledge base), but do not have an institutional relation (*i.e.*, no power-liability or duty-claim relation between two actors):

```
Event assignment-due
    Holds when is-student   // This event can only take place when there is at least
                            // one student.
    Creates    (Foreach student: deadline-passed(student)
                                 When tutor-of(tutor, student))
```

Duties As the name suggests, *duties* describe the *duty-claim* relation between a claiming actor and an actor that holds the duty. They may describe under which conditions the duties hold, and when they are violated:

```
Duty tutoring-duty
  Holder         tutor
  Claimant       student
  Holds when     tutor-of(tutor, student)
  Violated when  deadline-passed(student),
                 Not(course-active)
```

Like facts, duties can be created, terminated and obfuscated as an effect of acts or events:

```
Act provide-tutoring
  Actor          tutor
  Recipient      student
  Holds when     tutor-of(tutor, student)
  Terminates     tutoring-duty(tutor, student).
```

With the examples used above to illustrate the core types[1] of eFLINT, we have created the full specification for the normative system of asking for and providing tutoring, as introduced in Section 2.1.1. The complete specification is shown in Listing 2.1. Using an eFLINT specification, it is possible to check the compliance of *scenarios*. Because the execution of an act or event can modify the knowledge base (and with that, the normative state of a system), they can be seen as transitions in the normative system described by the given specification. Additionally, individual facts and duties may be created, terminated or obfuscated through *postulation*. A scenario is a sequence of acts, events and postulation statements, and can be either *action-compliant*, *duty-compliant*, or both [1]. A scenario is action-compliant when every transition was enabled in the knowledge base at the time it was taken. A scenario is duty-compliant when no actions or events have been executed that lead to a duty being violated. A scenario for the specification from Listing 2.1 could be as follows:

```
+course-active.                          // Postulate the fact that the course is active
+person(Alice).                          // Postulate the creation of person(Alice)
+person(Bob).                            // Postulate the creation of person(Bob)
request-tutoring(Alice, Bob).
provide-tutoring(Bob, Alice).
+person(Chloe).
~person(Bob).                            // Postulate the obfuscation of person(Bob)
request-tutoring(Chloe, Alice).
assignment-due.
provide-tutoring(Alice, Chloe).
```

This scenario is not *duty-compliant*, because the `assignment-due` event happens before the `provide-tutoring(Alice, Chloe)` act takes place, causing the `tutoring-duty(Alice, Chloe)` duty instance to be violated. Contrary to this, it is action-compliant because no disabled actions or events were taken.

The eFLINT reasoner for checking compliance of scenarios is implemented as a generic backend [12] that allows for executing and exploring specifications through these scenarios. Based on this backend, multiple interfaces to eFLINT have been developed. For runtime compliance checking within external software systems, an interface to the eFLINT backend is made available as a TCP server[2]. This server can be used directly, or as a *normative actor* in an actor-oriented framework such as the Scala-based Akka[3]. Additionally, a read-eval-print-loop (REPL)[2] and a Jupyter Notebook implementation[4] are

[1] Note that this overview of the language is not exhaustive. A full overview, together with examples, is provided in the eFLINT source repository, in the form of Jupyter notebooks. However, the language constructs discussed here can capture most (essential) normative constructs to understand the essence of eFLINT. Where necessary, we will introduce other eFLINT constructs throughout this thesis.

[2] https://gitlab.com/eflint/haskell-implementation

[3] https://akka.io

[4] https://github.com/leegbestand/eflint-jupyter

```
Fact person  Identified by String

Placeholder student For person
Placeholder tutor   For person

Fact tutor-of        Identified by tutor * student
Fact deadline-passed Identified by student

Fact is-student Identified by student
                Holds when (Exists tutor: tutor-of(tutor, student))

Bool course-active

Act request-tutoring
    Actor          student
    Recipient      tutor
    Holds when     student != tutor
    Conditioned by Not(tutor-of(tutor, student)),
                   course-active
    Creates        tutor-of(tutor, student)

Act provide-tutoring
    Actor          tutor
    Recipient      student
    Holds when     tutor-of(tutor, student)
    Terminates     tutor-of(tutor, student)

Duty tutoring-duty
    Holder         tutor
    Claimant       student
    Holds when     tutor-of(tutor, student)
    Violated when  deadline-passed(student),
                   Not(course-active)

Event assignment-due
    Holds when is-student
    Creates    (Foreach student: deadline-passed(student)
                                 When tutor-of(tutor, student))
```

Listing 2.1: Example eFLINT specification that describes the normative relation between a tutor and a student.

available which allows for manually exploring scenarios – like the one provided in this section – as well as debugging and prototyping specifications [1]. Checking the correctness of specifications, however, becomes problematic with these tools. As the number of type declarations in a specification grows, checking every possible scenario manually becomes difficult, if not impossible. Model checking could be a solution to this problem, as it would allow users to come up with abstract properties about the normative system they are specifying in eFLINT, rather than concrete scenarios. A model checker would then be able to use these properties to algorithmically determine whether the provided eFLINT specification is valid, and come up with counterexamples if this is not the case.

2.2 Model checking

Model checking is a technique for automatically verifying that a given system satisfies one or multiple desired properties. To be able to do this, both the system and its properties need to be formally represented. We will discuss these formalisms in detail, but in brief, a general model checking procedure consists of the following steps [8]:

1. *System description* The system that needs to be checked needs to be described in a formal representation (a 'model', usually some variation of a transition system). We will discuss this representation in Section 2.2.1. Depending on the system that needs to be checked, the model might be automatically inferred from the original representation of the system, or it needs to be expressed in a separate dedicated modelling language.

2. Property specification In addition to the description of the system, a set of properties about this system. These properties specify the desired behaviour of the possible paths (or 'traces') in the model, and with that, the system itself. This is done by expressing these properties in a (temporal) logic compatible with the system description. Section 2.2.2 discusses linear temporal logic (LTL), a popular temporal logic for model checking.

3. Model checking Using a system description and one or more property specifications, the model checker then algorithmically checks whether the description satisfies the given properties. If this is not the case, a counterexample may be produced that shows the trace of the model that violated one of the properties. Section 2.2.3 gives an overview of the model checking techniques we will consider in this thesis.

2.2.1 System description

As the name suggests, model checking requires the system to be described and represented as a formally defined model. A typical approach to do this is with transition systems, which describe the behaviour of a program or system as a set of states and transitions between states. In model checking, *Kripke structures* are among some of the most widely used types of transition systems [13].

Definition 1. *A* Kripke structure *is a tuple* $(\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{L})$ *[14], where*

- $\mathcal{S}$ *is a set of* states*;*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is a set of* initial states*;*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is a* transition relation*. This relation must be* total*,* i.e.*, for every state* $s \in \mathcal{S}$*, there must exist a state* $s' \in \mathcal{S}$ *such that* $\mathcal{R}(s, s')$*.*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}}$ *is a* labelling function*, where* $\mathcal{P}$ *is a finite, nonempty set of* atomic propositions*.*

In a Kripke structure, each possible behaviour of the represented system corresponds to a finite or infinite trace $\pi = \langle s_0, s_1, \ldots, s_n \rangle$ such that $(s_i, s_{i+1}) \in \mathcal{R}$ for all $i \geq 0$. An example Kripke structure with $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$, $\mathcal{S}_0 = \{s_1\}$, $\mathcal{R} = \{(s_1, s_2), (s_1, s_3), (s_2, s_3), (s_3, s_4), (s_4, s_4)\}$, $\mathcal{P} = \{p, q\}$ and $\mathcal{L} = \{s_1 \mapsto \{p, q\}, s_2 \mapsto \{p\}, s_3 \mapsto \{q\}, s_4 \mapsto \varnothing\}$ is visualized in Figure 2.2. A valid trace through this structure, for example, is $\langle s_1, s_3, s_4, s_4, \ldots \rangle$.

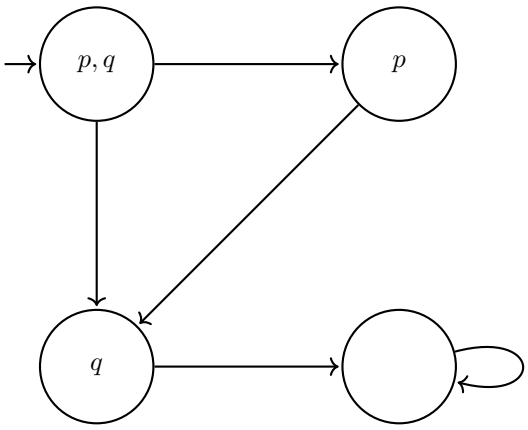


Figure 2.2: Example Kripke structure.

Kripke structures are considered to be state-based, in the sense that the states – as opposed to the transitions – are labelled. Transition-based structures also exist, where the transitions are labelled, rather than the states. These structures are aptly referred to as labelled transition systems (LTSs). Both representations have their advantages and disadvantages in terms of expressivity, state space and applications, and it is possible to translate Kripke structures to labelled transition systems and back [15]. Moreover, formalisms have been proposed that combine both representations to improve expressivity and to reduce the state space of such models [16, 17], which again can be transformed back into the purely state-based Kripke Structures. Because in eFLINT, both the state of the system (in the form of the knowledge base) and the transitions that modify this state (through acts or events) are highly relevant, we will give the definition of labelled Kripke structures (LKSs), as introduced by Chaki, Clarke, Ouaknine, Sharygina, and Sinha [17] and use this representation for the remainder of this thesis.

Definition 2. *A labelled Kripke structure (LKS) is tuple* $(\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{L}, \mathcal{T})$ *[17], where*

- $\mathcal{S}$ *is a set of* states*;*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is a set of* initial states*;*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is a* transition relation*. This relation must be* total*,* i.e.*, for every state* $s \in \mathcal{S}$*, there must exist a state* $s' \in \mathcal{S}$ *such that* $\mathcal{R}(s, s')$*.*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}}$ *is a* state-labelling function*, where* $\mathcal{P}$ *is a finite, nonempty set of* atomic propositions*.*
- $\mathcal{T} : \mathcal{R} \to (2^{\mathcal{A}} \backslash \{\varnothing\})$ *is a* transition-labelling function*, where* $\mathcal{A}$ *is a finite set of* actions*.*

Transitions in an LKS are denoted as $s \xrightarrow{A} s'$, where $(s, s') \in \mathcal{R}$ and $A \subseteq \mathcal{A}$. For transitions that are only labelled with one action $a \in \mathcal{A}$, we write $s \xrightarrow{a} s'$. In LKSs, a trace $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ is defined as a sequence alternating between states and actions, where for each $i \geq 0$, $s_i \in \mathcal{S}$, $a_i \in \mathcal{A}$ and $s_i \xrightarrow{a_i} s_{i+1}$.

2.2.2 Property specification

To be able to check if a model satisfies a given property, this property needs to be expressed using some kind of temporal logic, which allows us to reason about the behaviour of the system. Given a temporal formula φ and a trace π, we can then check if φ *holds at position* i *of* π. This is denoted as $\pi, i \vDash \varphi$. If φ holds for the entire trace (*i.e.*, $\pi, 0 \vDash \varphi$), then we can say that φ *holds on* π, denoted by $\pi \vDash \varphi$. If every trace π of a model $\mathcal{M}$ satisfies φ, we can say that $\mathcal{M}$ *satisfies* φ, denoted by $\mathcal{M} \vDash \varphi$.

A popular temporal logic often used for model checking Kripke structures is linear temporal logic (LTL). In LTL, properties are considered as linear sequences over the system, in the sense that for a given state, only one subsequent state is possible. When the system is non-deterministic, the possible executions are treated as independent sequences [13]. For example, in the example Kripke structure in Figure 2.2, the traces $\langle s_1, s_2, \ldots \rangle$ and $\langle s_1, s_3, \ldots \rangle$ are independent sequences.

LTL formulas consist of atomic Boolean propositions, the standard Boolean connectives for negation and disjunction, together with the temporal operators *next* (denoted as $\bigcirc$ or $\mathcal{X}$) and *until* (denoted as $\mathcal{U}$):

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2.$$

Here, $\bigcirc\varphi$ indicates that φ is true in the immediate successor of the current state. $\varphi_1 \,\mathcal{U}\, \varphi_2$ indicates that φ_1 holds in all future states, until a state is reached where φ_2 holds. Given a trace $\pi = \langle s_0, s_1, \ldots \rangle$, the semantics of these operators are defined inductively as follows [13]:

- $\pi, i \vDash p$ iff $p \in s_i$ for $p \in \mathcal{P}$, where P is a countable set of Boolean propositions.
- $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$.
- $\pi, i \vDash \varphi_1 \vee \pi, i \vDash \varphi_2$ iff $\pi, i \vDash \varphi_1$ or $\pi, i \vDash \varphi_2$.
- $\pi, i \vDash \bigcirc\varphi$ iff $\pi, i + 1 \vDash \varphi$.
- $\pi, i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ iff there exists $k \geq i$ such that $\pi, k \vDash \varphi_2$ and $\pi, j \vDash \varphi_1$ for all j such that $k < j \leq i$.

Similar to how we can define the Boolean connectives for conjunction, implication and equivalence using negation and disjunction, we can define additional temporal operators, namely *future* (denoted as $\Diamond$ or $\mathcal{F}$), *globally* (denoted as $\Box$ or $\mathcal{G}$) and *weak-until* (denoted as $\mathcal{W}$) using *until*. $\Diamond\varphi$ indicates that φ will be true in some future state, while $\Box\varphi$ indicates that φ is true in all future states. $\varphi_1 \,\mathcal{W}\, \varphi_2$ indicates that φ_1 holds in all future states, until a state is reached where φ_2 holds, and if that is not the case, φ_1 must remain true in all future states. These operators are defined as follows:

$$\Diamond\varphi \equiv \top \,\mathcal{U}\, \varphi$$
$$\Box\varphi \equiv \neg\Diamond\neg\varphi$$
$$\varphi_1 \,\mathcal{W}\, \varphi_2 \equiv (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee \Box\varphi_1.$$

The LTL operators introduced here are referred to as *pure future* operators because they can only be used to reason about the future. While LTL also has operators for reasoning about the past (namely $\ominus$ for *previous* and $\mathcal{S}$ for *since*), they are not often used in model checking [13] and for this reason, we will not consider them in this thesis.

Extension to Labelled Kripke Structures The above definition of LTL assumes we are reasoning about traces of states. Because traces of LKSs alternate between states and actions, Chaki, Clarke, Ouaknine, Sharygina, and Sinha [17] introduce state/event linear temporal logic (SE-LTL), which is highly similar to regular LTL but allows action labels to be included in formulae:

$$\varphi ::= p \mid a \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2.$$

The semantics of these operators are extended accordingly:

- $\pi, i \vDash p$ iff $p \in s_i$ for $p \in \mathcal{P}$, where $\mathcal{P}$ is a countable set of Boolean propositions.
- $\pi, i \vDash a$ iff a is the first action of π, starting from position i.
- $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$.
- $\pi, i \vDash \varphi_1 \vee \pi, i \vDash \varphi_2$ iff $\pi, i \vDash \varphi_1$ or $\pi, i \vDash \varphi_2$.
- $\pi, i \vDash \bigcirc\varphi$ iff $\pi, i + 1 \vDash \varphi$.
- $\pi, i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ iff there exists $k \geq i$ such that $\pi, k \vDash \varphi_2$ and $\pi, j \vDash \varphi_1$ for all j such that $k < j \leq i$.

Types of temporal properties

Temporal can be classified into two types: *safety* and *liveness* properties. Simply put, a safety property specifies that 'something bad never happens'. A common type of safety property is the *invariant*, which are formulas of the form $\Box p$, where p is a propositional formula [13]. A safety property of the normative system of providing and receiving tutotoring that we introduced in Section 2.1.1 could be "a person can never be their own tutor", for example. Liveness properties, on the other hand, specify that 'something good will eventually happen' [18]. A liveness property of the same normative system about tutoring could be "the duty for Bob to tutor Alice is lifted as soon as Bob provides said tutoring". Formally, safety and liveness properties are defined as follows:

Definition 3 (Safety)**.** *Let S^ω be the set of infinite traces of a model and σ_i the partial trace up to length i (where $i \in \mathbb{N}$) of some trace $\sigma \in S^\omega$. A property φ is a* safety *property if and only if:*

$$(\forall \sigma \in S^\omega : \sigma \nvDash \varphi \implies (\exists i \in \mathbb{N} : (\forall \tau : \sigma_i \tau \nvDash \varphi))).$$

In other words, for every trace σ that violates the property φ, there exists a partial trace σ_i of length $i \in \mathbb{N}$ such that any arbitrary extension τ of this trace will also violate φ [18].

Definition 4 (Liveness)**.** *Let $S\omega$ be the set of infinite traces of a model and S^* the set of partial traces of a model. A property φ is a* liveness *property if and only if:*

$$\forall \sigma \in S^* : (\exists \tau \in S^\omega : \sigma\tau \vDash \varphi).$$

In other words, for every partial trace σ, there exists an infinite trace τ such that the concatenation $\sigma\tau$ satisfies φ [18].

Alpern and Schneider [18] show that every property φ is the intersection of a safety property and a liveness property, which means that φ can be either a safety property, a liveness property, or both at the same time. A practical example to illustrate this intersection is the property $\varphi_1 \,\mathcal{U}\, \varphi_2$, which can be rewritten as $(\varphi_1 \,\mathcal{W}\, \varphi_2) \wedge \Diamond\varphi_2$ [19]. Here, the safety property is $\varphi_1 \,\mathcal{W}\, \varphi_2$ and the liveness property is $\Diamond\varphi_2$.

2.2.3 Bounded Model Checking

Given a model of a system $\mathcal{M}$ and a property φ over this system, a model checker then algorithmically verifies whether $\mathcal{M} \vDash \varphi$. While this can be done explicitly by storing and traversing the entire model, a generally more efficient and scalable method is by representing the model symbolically, where the initial state and transition relation are represented using logical formulas [8]. A well-known symbolic model checking technique is bounded model checking (BMC), which uses SAT solving [20] and, more recently, satisfiability modulo theories (SMT) solving [21]. In contrast to other common model checking techniques (be it explicit or symbolic), a core principle of BMC is finding errors in systems rather than proving the system is correct [22].

The idea behind BMC is to build traces $\sigma_1, \ldots, \sigma_k$ that incrementally grow in size, up to the length of the maximum bound k. These traces represent negations of the property φ and are encoded into

propositional formulas $f_1, \ldots, f_k$, which are used as input for the selected SAT or SMT solver along with a propositional encoding of the model. If the solver reports a formula f_n ($1 \leq n \leq k$) together with the encoding for $\mathcal{M}$ to be satisfiable, the trace is present in the model, indicating that $\mathcal{M}$ does not satisfy φ and the corresponding trace σ_n is reported back to the user as a counterexample. If f_n together with the encoding for $\mathcal{M}$ is not satisfiable, the procedure is repeated for f_{n+1}. When $n > k$ and no counterexample has been found, φ is considered to hold and the procedure terminates. A schematic overview of the BMC procedure is shown in Figure 2.3.

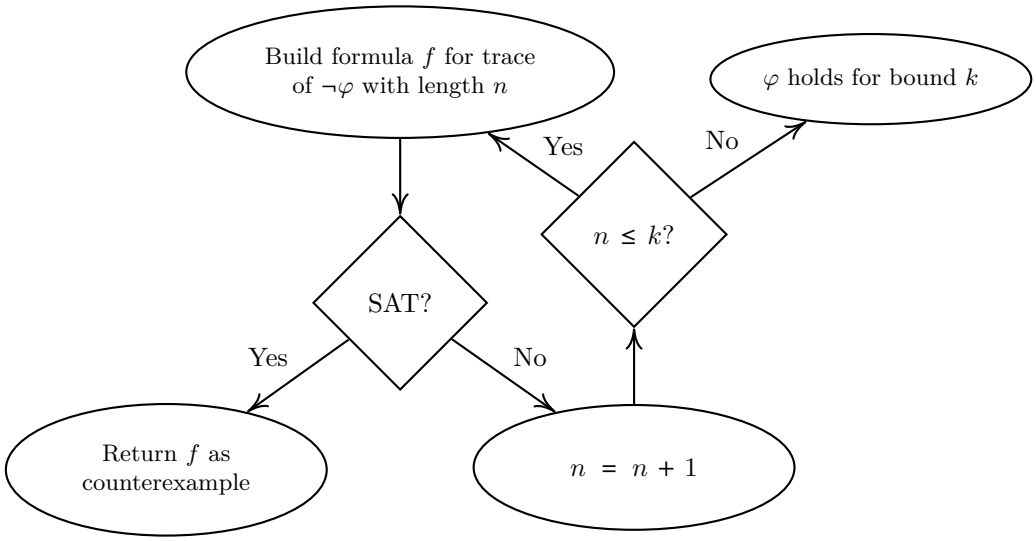


Figure 2.3: The procedure for bounded model checking (BMC), adapted from Clarke [14].

Chapter 3

Property specification for normative systems

In the introduction of this thesis, we briefly touched on some ways in which the formalization and specification of properties over norms could act as a valuable addition to the existing norm specification methods. In this chapter, we will make the case for property specification for normative systems more concrete. Additionally, we will discuss how we can use linear temporal logic (LTL), which we introduced in the previous chapter, to formalize the properties one might express in this context.

3.1 Purposes for property specification

We identify four different conceptual levels in which these properties, and being able to model-check may be used, which we will explain in this section: (1) the classic model-checking purpose of identifying potential mistakes in a system; (2) being able to capture the temporal nature of some norms; (3) being able to give guarantees about the correctness of a specification if the norms on which it is based change; and (4) being able to relate specifications that represent actions in the physical world to those that represent actions in the social reality.

3.1.1 Mistakes in norm specifications

At its origin, model checking is a technique for verifying and identifying potential mistakes in both hardware and software systems. As such, we can apply this same principle to norm specifications. Especially as specifications grow larger and become more and more interdependent, the potential for bugs to appear increases. In eFLINT, mistakes may occur in the preconditions (derivation, condition and violation clauses and domain constraints) of any declaration or postconditions (creating, terminating and obfuscating clauses) of act and event declarations, leading to a mismatch between the specification and the norms they formalize. By expressing the norms as abstract (temporal) properties, we can use model checking to verify whether the specification indeed models the behaviour that is prescribed by the corresponding norms.

3.1.2 Temporal norms

Norms may include notions of temporality that may be difficult to capture by normative relations alone. For example, they might say something about the order in which certain actions may be taken. On the other hand, the notion of temporality is an important aspect of model checking. By formalizing norms as temporal properties, we can use model checking to verify that the actions defined in a specification indeed satisfy the temporal requirements of the norms they represent.

3.1.3 Changing norms

Norms are not static. Laws, regulations and policies might be added, changed or removed over time, which might require the specifications that formalize them to be changed as well. However, the impact that a norm change might have on the implementation of a specification not be immediately apparent. In the best-case scenario, none or only minimal changes to the specification would be required, but it

might also be the case that large portions of the scenario need to be rewritten. By first formalizing norms as abstract properties related to the specification, it becomes possible to check whether this specification complies with these norms. When one of these norms changes, updating the set of properties accordingly and model checking the original specification against these updated properties should provide immediate insight into the changes that would be required to the specification.

3.1.4 The physical world and its relation to social reality

As we discussed in Section 2.1.1, there exists a relation between actions in the physical world and their (normative) consequences in the social reality [10]. To illustrate this, we can use the tutoring example we introduced in the previous chapter. Here, walking up to someone and verbally asking them to become one's tutor is an action in the physical world that relates to the action that exists in the social reality of requesting tutoring which, as a consequence creates the duty to perform this tutoring. In eFLINT, it is possible to relate acts and events to one another with synchronization clauses, which are declared using the `Syncs with` keyword [23]. With the fact in mind that eFLINT is designed for compliance checking running software systems, we can use these clauses to create specifications where these running systems only need to communicate with the eFLINT reasoner through actions pertaining to physical interactions with these systems and have the eFLINT reasoner synchronize them internally with actions that pertain to the social reality. This way, it becomes possible to reuse these last kinds of actions in multiple specifications representing different physical systems. The tutoring specification from Listing 2.1, for example, could be reused in different educational institutions, each implementing its own set of physical actions (for example, the act of submitting an online form on a website) that synchronise with the actions in this specification. To ensure that physical actions synchronise correctly with the norms of the social reality, we could specify properties that relate these physical actions to the consequences of their internal institutional counterparts.

3.2 Formalizing properties

In Section 2.2.2, we presented the distinction between safety and liveness properties. The purposes we identified in this chapter could potentially benefit from both types. Safety properties may be used to, among others, ensure that certain actions are only enabled under specific conditions or that certain (combinations of) facts can never exist. Examples where liveness properties may be used, are to ensure that the correct duties are lifted or violated after certain actions, or that the eFLINT knowledge base is updated correctly after executing certain actions.

As we saw in Section 2.2.2, LTL formulae are expressed solely over Boolean propositions, whereas SE-LTL formulae can also capture actions. For expressing properties over norms, we propose a variation of SE-LTL which we will refer to as eLTL ("eFLINT-LTL") for the remainder of this thesis. The operators for eLTL are defined as follows:

$$\varphi ::= \mathcal{H}p \mid \mathcal{V}d \mid \mathcal{E}a \mid \mathcal{T}a \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2.$$

Here, $\mathcal{H}p$ denotes that a proposition p (which is either a fact or duty) holds, $\mathcal{V}d$ denotes that a duty d is violated, $\mathcal{E}a$ denotes that an action a (which is either an act or event) is enabled, and $\mathcal{T}a$ denotes that an action a (which is either an act or event) has been taken. The semantics are defined as follows:

- $\pi, i \vDash p$ iff $p \in s_i$ for $p \in \mathcal{P}$, where $\mathcal{P}$ is a countable set of Boolean propositions.
- $\pi, i \vDash \mathcal{H}p$ iff $p \in s_i$ for $p \in \mathcal{P}$, where $\mathcal{P}$ is the set of all facts and duties in the given specification.
- $\pi, i \vDash \mathcal{V}d$ iff $d \in s_i$ for $d \in \mathcal{D}$, where $\mathcal{D}$ is the set of all duties in the given specification.
- $\pi, i \vDash \mathcal{E}a$ iff a (which can either be an act or event) is the first action of π, starting from position i.
- $\pi, i \vDash \mathcal{T}a$ iff a (which can either be an act or event) is the action of π, starting from position i, that was most recently taken.
- $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$.
- $\pi, i \vDash \varphi_1 \vee \pi, i \vDash \varphi_2$ iff $\pi, i \vDash \varphi_1$ or $\pi, i \vDash \varphi_2$.
- $\pi, i \vDash \bigcirc\varphi$ iff $\pi, i + 1 \vDash \varphi$.
- $\pi, i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ iff there exists $k \geq i$ such that $\pi, k \vDash \varphi_2$ and $\pi, j \vDash \varphi_1$ for all j such that $k < j \leq i$.

The main difference between these semantics and those of SE-LTL is that while SE-LTL only reasons about outgoing actions (*i.e.*, actions that have yet to be taken), we reason about incoming actions

(*i.e.*, actions that have been taken) as well. We do this with the notion of action compliance in mind. With these semantics, users can specify under which conditions an action *may happen* as well as reason over what the effects are when an action *has happened*. It should be noted that this distinction does lead to a subtle difference in when the *next* operator should be used, which is best illustrated with an example. Consider a property that states that when a certain action a is enabled in the current state, the proposition p should hold in the next state:

$$\Box(\mathcal{E}a \implies \bigcirc\mathcal{H}p).$$

This property warrants the use of the *next* operator. However, a property that states that if an action a has been taken, then the proposition p should be true does not:

$$\Box(\mathcal{T}a \implies \mathcal{H}p).$$

It should also be noted that these properties are not equivalent. The first property is a liveness property that states that the fact that a is enabled must lead to some change in the state (most likely by executing a, although that is not required) whereas the second property is a safety property that states that p always holds as an effect of the execution of a.

Because we want to be able to use eLTL properties in practice for the purposes we discussed in this chapter, we need a way to model-check them. This requires eFLINT specifications to be represented in such a way that they can be encoded into a suitable model checker input language. This also requires an extension to the eFLINT syntax for specifying eLTL properties. We will discuss both in the next chapter.

Chapter 4

Design of eFLINT-check

To allow users of eFLINT to model check their specifications, we have designed and implemented eFLINT-CHECK, which takes a specification, a set of properties and optionally a (partial) scenario as input and translates it into a labelled Kripke structure (LKS) where the knowledge base resulting from the provided scenario serves as an initial state. It then encodes this model into a representation accepted by a model-checking backend that is suitable for our purposes. Finally, if any counterexamples are found, they will be reported back to the user in a way that allows them to understand and fix the errors in the specification. The design of eFLINT-CHECK is shown in Figure 4.1. In this chapter, we will explain and discuss the design considerations behind these different steps.

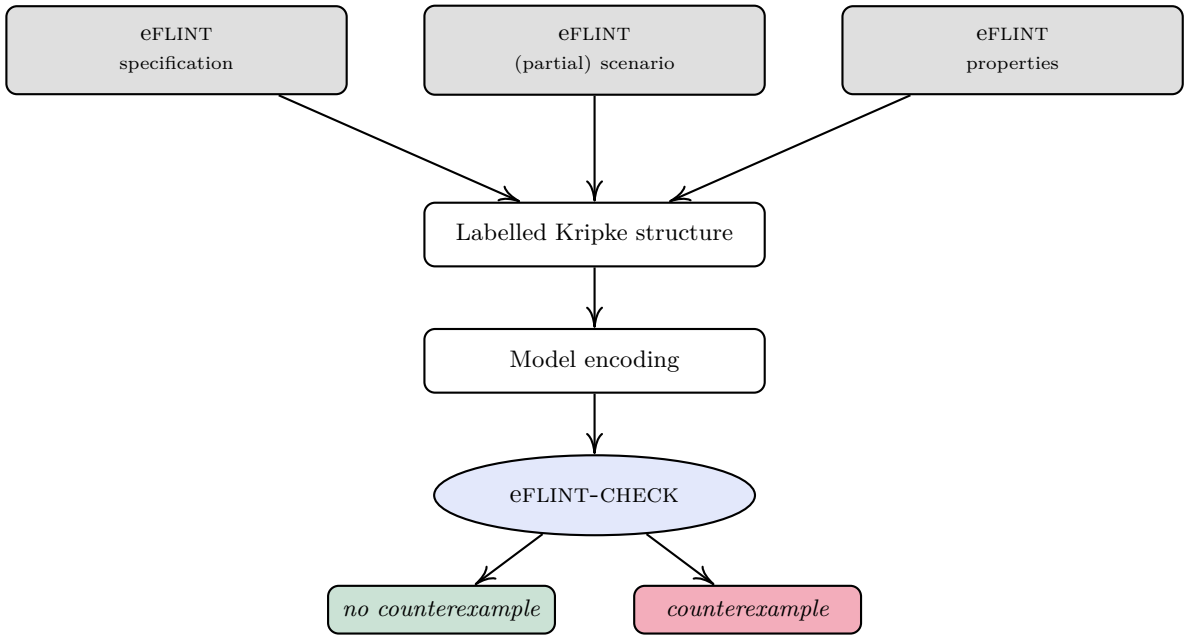


Figure 4.1: Design of eFLINT-CHECK.

4.1 Design scope

Taking into account the scope of this thesis subject and the available time, we will not be able to include every language construct in the design and subsequent implementation of eFLINT-CHECK. Ultimately, we want eFLINT-CHECK to be feature complete, but our main priority right is to create a functional tool that can demonstrate the usefulness of abstract property specification and model checking for normative systems. As such we omit, among others, the `Syncs with` clause. This decision might seem counterintuitive given the fact that it closely relates to the last purpose for model-checking norm specifications we discussed in Section 3.1.4. However, we believe that we could sufficiently demonstrate the potential of eFLINT-CHECK using only the first three purposes we identified.

4.2 Representing eFLINT specifications as labelled Kripke structures

To be able to check properties of eFLINT specifications, the specification first needs to be transformed into a suitable representation. As we discussed in Section 2.2.2, acts and events can be regarded as transitions that modify the state of the normative system represented by the eFLINT specification. Here, the `Holds when` and `Conditioned by` clauses act as preconditions, and the `Creates`, `Terminates` and `Obfuscates` clauses act as postconditions, which in turn can be used to define the transition relation between states. However, the current (internal) representation of these specifications is not yet suitable for model checking. In Section 2.2.1 we introduced labelled Kripke structures (LKSs) as a formal representation for model checking. In the following sections, we will explain how we construct LKSs from eFLINT specifications.

4.2.1 Propositions and actions

In eFLINT, the state of the normative system is represented by the knowledge base. While this is highly similar to the way a state is represented in an LKS in the sense that it represents which fact and duty instances hold, one issue with the design and implementation of eFLINT is that the domains of atomic fact types can be unbounded, which in turn leads to an infinite state space. While different methods have been proposed for model checking infinite systems [24–26], eFLINT-CHECK will only accept specifications in which the facts are bounded, making the state space finite and representable by an LKS. The reason we do this instead of using another way to represent eFLINT specifications is that the majority (if not all) state-of-the-art model checking tools are designed for (mostly) finite models. There do exist BMC-based tools that use SMT solving and are therefore able to handle infinite integer domains which we could potentially leverage for unbounded atomic integer fact declarations, but then the question remains how unbounded atomic string fact declarations, and by extension and record-type fact declarations should be represented. Moreover, we expect that most (if not all) properties over the normative systems expressed by eFLINT specifications can be checked using only bounded domains. Therefore, at least for the proof of concept version of eFLINT-CHECK introduced in this thesis, we will only consider eFLINT specifications that are fully bounded.

In the LKS representing the eFLINT specification, the set of atomic propositions $\mathcal{P}$ will consist of all possible instances for each fact and duty declaration. For the tutoring specification from Listing 2.1, $\mathcal{P}$ is constructed as follows:

$$
\begin{aligned}
\mathcal{P} = \{&\texttt{person(Alice)}, \texttt{person(Bob)},\\
&\texttt{tutor-of(Alice, Alice)}, \texttt{tutor-of(Alice, Bob)},\\
&\texttt{tutor-of(Bob, Alice)}, \texttt{tutor-of(Bob, Bob)},\\
&\texttt{tutoring-duty(Alice, Alice)}, \texttt{tutoring-duty(Alice, Bob)},\\
&\texttt{tutoring-duty(Bob, Alice)}, \texttt{tutoring-duty(Bob, Bob)},\\
&\texttt{is-student(Alice)}, \texttt{is-student(Bob)},\\
&\texttt{deadline-passed(Alice)}, \texttt{deadline-passed(Bob)}\}
\end{aligned}
$$

Each of these atomic propositions denotes whether that fact or duty instance holds in the knowledge base. Additionally, for each duty instance, we need to keep track of whether it has been violated or not. Because this is derived by the `Violated when` clause and is determined by other facts, in theory, it is not necessary to turn this into an atomic proposition. However, depending on the underlying model checker we use, we might need to do this explicitly in order to be able to reason about violated duties in property specifications.

Similar to the construction of the set of atomic propositions $\mathcal{P}$, the set of actions $\mathcal{A}$ of the LKS consists of all possible instances for each act and event. For the tutoring example, this set is constructed as follows:

$$\begin{aligned}\mathcal{A} = \{&\texttt{request-tutoring(Alice, Alice)}, \texttt{request-tutoring(Alice, Bob)},\\ &\texttt{request-tutoring(Bob, Alice)}, \texttt{request-tutoring(Bob, Bob)},\\ &\texttt{provide-tutoring(Alice, Alice)}, \texttt{provide-tutoring(Alice, Bob)},\\ &\texttt{provide-tutoring(Bob, Alice)}, \texttt{provide-tutoring(Bob, Bob)},\\ &\texttt{assignment-due}\}\end{aligned}$$

We should note that due to the derivation clause of `request-tutoring`, the instances `request-tutoring(Alice, Alice)` and `request-tutoring(Bob, Bob)` will never be enabled. This means that technically, they could be removed from the model altogether. However, we will leave this optimization (among others that will be discussed in later chapters) out of scope for now.

Ideally, the domains for each fact declaration should be small enough, such that the number of equivalent traces with different instances is as small as possible. For example, considering the tutoring specification from Listing 2.1, and the following eLTL property:

$$\Box(\mathcal{T}(\texttt{request-tutoring(student, tutor)}) \implies \bigcirc\mathcal{H}(\texttt{tutoring-duty(tutor, student)})),$$

which states that at any point in a scenario, if the act `request-tutoring` is executed for any `student` and `tutor` (which are aliases for the `person` fact type), the duty `tutoring-duty(tutor, student)` holds in the next step. To check this property, it is sufficient to have the domain of the `person` fact type be just two elements. On the other hand, the domains should not be so small that they lead to false positives. For example, if we want to check that it is never possible for a `person` to be the tutor of more than three other people, the domain of `person` should at least contain five elements: one for the `person` instance that serves as the `tutor`, and the other four serving as `student` instances, thus opening up the possibility of having a counterexample be present in the model in case the specification does not sufficiently capture this property. In terms of gaining confidence in the correctness of the specification, it is desirable to have the bounds be larger rather than smaller than necessary. However, the larger the bounds are, the larger the state space of the model will be, which in turn will increase the time and memory required to check the model. Finding the optimal domains for each fact type is difficult, especially because the eFLINT specifications that need to be checked are written with compliance checking of real systems in mind, where it might be desirable to have unbounded fact declarations as much as possible because the domains it might not be known in advance. In the design presented here, we will leave the responsibility of finding sensible bounds to the user. Because eFLINT is designed with modularity in mind [23], it is possible to override fact-type declarations. This allows users to create a mock specification for the specific purposes of model checking.

Terminated and obfuscated facts

As discussed in Section 2.1.2, eFLINT makes a distinction between terminated and obfuscated facts, where terminated facts are present in the knowledge base but evaluate to *False*, whereas obfuscated facts are absent from the knowledge base altogether, technically making it impossible to know whether the fact holds or not. While methods for model checking three-valued Kripke structures [27–29] have been proposed, for now, we will only make a distinction between facts that hold (*i.e.*, that evaluate to *True*), and facts that do not hold (*i.e.*, facts that evaluate to *False* or are absent from the knowledge base).

The closed-world assumption

The eFLINT reasoner supports so-called *open facts* to handle information that is required from an external source not represented by the eFLINT specification. To be able to represent eFLINT specifications as LKSs, they need to be *closed*, meaning that external input is not allowed. This means that open facts cannot be represented in the system. Similar to how mock specifications can be created to add finite domains to otherwise unbounded fact declarations, it is possible for users to temporarily override open fact declarations to make them closed instead.

Types of actions allowed in eFLINT-check

With the eFLINT reasoner, it is possible to *postulate* facts. This means that it is possible to create, terminate or obfuscate a fact or duty manually, without it being the effect of an act or event. Including postulation in the LKS is not feasible as doing so would significantly increase the state space and with that, increase the time and memory required to check the generated model. More importantly, we argue that it would be undesirable to consider the postulation of facts in the generated LKS. By allowing postulation in the LKS it will become possible to reach any state from any other state, effectively nullifying the pre- and postconditions of any specified transition (*i.e.*, acts and events). For this reason, we will not include postulation in the LKS. If there is a reason that warrants the inclusion of postulation for certain facts, this might instead be achieved by (temporarily) adding one or more events that modify the knowledge base as desired.

Similarly, the eFLINT reasoner allows for the execution of disabled actions, therefore creating a scenario that is not action-compliant. For the same reasons regarding the postulation of facts, we will not consider these kinds of actions as transitions in our model. This means that eFLINT-CHECK will only be able to model-check duty-compliance.

Finally, with the eFLINT reasoner it is possible to modify specifications on-the-fly by adding or updating type declarations at runtime [23]. Doing so changes the underlying (normative) transition system of eFLINT, and with that, the LKS representing this system. While being able to do so is desirable in the existing tools based on the eFLINT reasoner because it allows users to create highly dynamic systems that can be modified and extended over time, for eFLINT-CHECK we only want to consider fixed systems. In a practical sense, it is not possible to arbitrarily change the transition relation of the model during a model-checking run, as this would cause the procedure to never halt. Moreover, on a more abstract level, this would defeat the purpose of model checking, because we want to check the specification as it exists now and get feedback on potential mistakes in this specification. Therefore, we will not consider these statements as well. All in all, this means that the only types of actions considered in eFLINT-CHECK are acts and events.

4.2.2 Initial state

The initial state of the LKSs representing eFLINT specifications can be constructed using the knowledge base that follows from the provided partial scenario. Given the tutoring specification from Listing 2.1, we can, for example, specify that the `course-active` fact holds in the initial state in this way. In case no scenario is provided, we assume that all atomic type fact instances hold, and all other facts and duties do not hold (unless they are derived from these atomic facts). We do this because it mirrors the behaviour of the eFLINT REPL. The knowledge base of eFLINT also contains the labels of enabled act and event instances, so to comply with the definition of LKSs, we need to filter them from the knowledge base before we can use it as our initial state.

4.2.3 The transition relation

The transition relation of an eFLINT specification is determined by the pre- and postconditions of acts and events. The precondition of an act or event is composed of its derivation clause, its condition clauses and its domain constraint. Multiple derivation and condition clauses may be defined for one act or event. In that case, they are combined by taking the disjunction of all derivation clauses and the conjunction of all condition clauses. In turn, those are combined using conjunction. For example, consider the following act declaration:

```
Act request-tutoring
  Actor           student
  Recipient       tutor
  Holds when      student != tutor
  Conditioned by  Not(tutor-of(tutor, student)),
                  course-active
  Creates         tutor-of(tutor, student),
                  tutoring-duty(tutor, student)
```

This act has one derivation clause, namely `student != tutor` and two condition clauses, namely `Not(tutor-of(tutor, student))` and `course-active`. The full precondition for this act then becomes

$$\texttt{student} \neq \texttt{tutor} \wedge \texttt{!tutor-of(tutor, student)} \wedge \texttt{!course-active}.$$

The postcondition of an act or event is determined by the creation, termination and obfuscation clauses. Again, multiple clauses may be defined. In this case, they are combined using conjunction. For the above example, the postcondition becomes

$$\texttt{tutor-of(tutor, student)} = \mathit{True} \wedge \texttt{tutoring-duty(tutor, student)} = \mathit{True}.$$

It follows from the discussion in Section 4.2.1 that both termination and obfuscation clauses will cause these facts and duties to evaluate to *False*.

4.2.4 Putting it all together

If we put design considerations discussed here together, we can represent the eFLINT tutoring specification from Listing 2.1 as the LKS visualized in Figure 4.2. This visualization goes to show that even for small specifications with a minimal amount of fact instances, it will be cumbersome to manually check every possible scenario, and the risk of missing a scenario only grows as the specification and number of instances increase.

4.3 Property specification

eFLINT already supports the specification of invariant properties (*i.e.*, Boolean propositions that have to hold in every state) through the `Invariant` keyword. For example, for the tutoring example we can specify the invariant "no one person can be their own tutor" as follows:

```
Invariant not-own-tutor Where Not(Exists person: tutor-of(person, person))
```

The eFLINT reasoner evaluates these invariants every time the knowledge base is modified, either through transitions or postulation, but we can use them in eFLINT-CHECK to verify *a priori* if these invariants will be violated in the given specification. However, as we discussed in Section 3.2, we also want to be able to capture other safety properties as well as liveness properties, which is possible with eLTL. To do so, we will extend eFLINT with the `Property` declaration that allows users to specify eLTL properties. The concrete syntax of this declaration is given in Figure 4.3. Fact, duty and transition labels may contain either concrete or abstract fields, where a concrete field contains an actual instance whereas an abstract field contains the name of the field. This allows users to write properties over both specific instances as well as more abstract scenarios.

Note that while we use the keyword `Until`, this operator will have the semantics of the *weak until* ($\mathcal{W}$) operator. In our initial design, we did use the *until* ($\mathcal{U}$) operator, but during a preliminary evaluation of this language extension and its expressivity concerning (abstract) eFLINT scenarios, we almost exclusively found properties where the use of $\mathcal{W}$ was warranted over $\mathcal{U}$. This leads us to use $\mathcal{W}$ instead of $\mathcal{U}$. In the rare case that the semantics of $\mathcal{U}$ are needed instead of $\mathcal{W}$, it is possible to do so by combining the syntax for the *future* ($\Diamond$) operator with $\mathcal{W}$.

Properties are defined similarly to invariants, where every property consists of an identifier and the formula itself. In contrast to the eFLINT formulae that are used in various clauses, it is not necessary to use quantifiers. Instead, the field names of each fact, act, event or duty can be used directly. Similar to how we create concrete instances for each type declaration to represent atomic propositions and actions in the LKS, we create concrete instances for each abstract proposition or action that is present in the property. For example, the following property over the tutoring specification:

```
Property become-tutor Where
    Always If Taken(request-tutoring(student, tutor))
           Then Holds(tutor-of(tutor, student))
             Until Taken(provide-tutoring(tutor, student))
```

will be expanded into four separate formulae where the fields for `student` and `tutor` are replaced by `Alice` and `Bob`, `Alice` and `Alice`, `Bob` and `Alice`, and `Bob` and `Bob`, respectively.

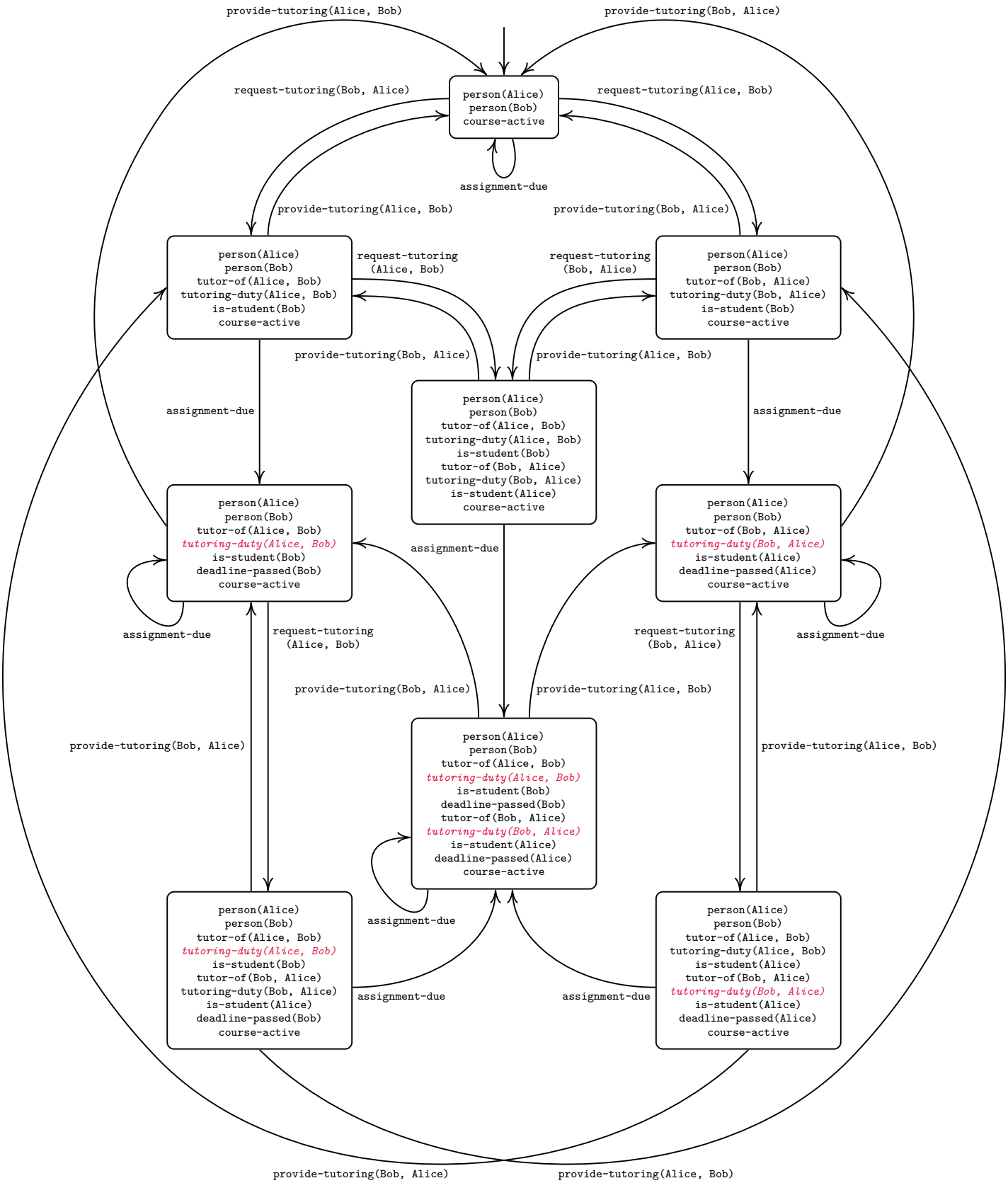


Figure 4.2: Kripke structure based on the tutoring specification from Listing 2.1. Violated duties are shown in *red*.

⟨*ltl_prop*⟩	::=	'Property' ⟨*name*⟩ 'Where' ⟨*ltl_term*⟩	
⟨*ltl_term*⟩	::=	⟨*proposition*⟩	
	|	⟨*action*⟩	
	|	'(' ⟨*ltl_term*⟩ ')'	
	|	'Not' ⟨*ltl_term*⟩	$(\neg\varphi)$
	|	⟨*ltl_term*⟩ 'And' ⟨*ltl_term*⟩	$(\varphi_1 \wedge \varphi_2)$
	|	⟨*ltl_term*⟩ 'Or' ⟨*ltl_term*⟩	$(\varphi_1 \vee \varphi_2)$
	|	'If' ⟨*ltl_term*⟩ 'Then' ⟨*ltl_term*⟩	$(\varphi_1 \implies \varphi_2)$
	|	⟨*ltl_term*⟩ 'Iff' ⟨*ltl_term*⟩	$(\varphi_1 \iff \varphi_2)$
	|	'Always' ⟨*ltl_term*⟩	$(\Box\varphi)$
	|	'Eventually' ⟨*ltl_term*⟩	$(\Diamond\varphi)$
	|	'Next' ⟨*ltl_term*⟩	$(\bigcirc\varphi)$
	|	⟨*ltl_term*⟩ 'Until' ⟨*ltl_term*⟩	$(\varphi_1 \mathcal{W} \varphi_2)$
⟨*proposition*⟩	::=	'Enabled(' ⟨*transition_label*⟩ ')'	$(\mathcal{E}a)$
	|	'Holds(' ⟨*fact_label*⟩ ')'	$(\mathcal{H}p)$
	|	'Holds(' ⟨*duty_label*⟩ ')'	$(\mathcal{H}p)$
	|	'Violated(' ⟨*duty_label*⟩ ')'	$(\mathcal{V}d)$
⟨*action*⟩	::=	'Taken(' ⟨*transition_label*⟩ ')'	$(\mathcal{T}a)$

Figure 4.3: Concrete syntax definition of eLTL

4.4 Reporting counterexamples

An important part of the design is the reporting of counterexamples in case the model constructed from eFLINT specifications does not satisfy the provided properties. These counterexamples should provide users with enough information to be able to debug and fix the provided specification, but at the same time should not be cluttered with unnecessary internal details about the model. To achieve this, we can make use of the tools that already have been implemented to interact with eFLINT scenarios. The eFLINT REPL shows users how the knowledge base is affected by every transition that is taken. Because of this, in principle, it should be sufficient to only report the transitions that are necessary to violate the specified property. Using the REPL, the user would then be able to explore these transitions step-by-step, up to the point where they can identify what changes are necessary to satisfy the property.

4.5 The model-checking backend

The implementation of eFLINT-CHECK is largely dependent on the model-checking backend we use because this will determine how the LKSs based on the eFLINT specifications should be encoded. Currently, a wide range of tools and accompanying model specification languages exist, each with specific purposes and domains in mind. To be able to select an appropriate model checker to use for our backend, we need to create a set of requirements this model checker should satisfy. Seshia, Sharygina, and Tripakis [30] identify the following factors to consider when selecting a model checking tool and language:

1. Type of system;
2. Type of properties;
3. Relevant information about the environment;
4. Level of abstraction;
5. Clarity and modularity;
6. Form of composition;
7. Computational engines;
8. Practical ease of modelling and expressiveness.

For each of these factors, we will give a brief explanation and discuss what requirements and considerations are relevant for model checking eFLINT specifications. Some of these considerations have already been touched upon previously, but we will repeat them here for the sake of completeness.

Type of system The type of system that needs to be modelled and checked largely dictates the tool to use. For example, systems with a lot of concurrent and communicating processes warrant a different formalism (and in turn model checker) than discrete, synchronous systems. eFLINT specifications describe discrete systems, which is why we use LKSs as our modelling formalism. Ideally, the tool we select would be able to handle this formalism directly, but because LKSs can be translated into classic Kripke structures [17], which in turn can be translated into other discrete formalisms such as LTSs [15], our main requirement is that it must be able to handle discrete systems in general.

Type of properties The type of properties that the model checker should be able to check is determined in part by the type of system that is being checked. In Section 2.2.2 and Section 4.3, we established that for checking eFLINT specifications, we want to use SE-LTL and be able to check both safety and liveness properties. Similar to how LKSs can be translated to Kripke structures, SE-LTL formulae can also be translated to corresponding LTL formulae. Therefore, the model checker we select should (at least) support LTL model checking for both safety and liveness properties.

Relevant information about the environment Some systems, especially software systems, may depend on external systems such as other software libraries or user input. Although eFLINT is designed to be integrated with other software systems, under the closed-world assumption (as discussed in Section 4.2.1) and by not including postulation in our model (as discussed in Section 4.2.1), eFLINT specifications can be checked independently from their target environment.

Level of abstraction The level of abstraction of the model that is created to represent a system determines the state space of the model and, with that, the memory and time required to check the provided properties. The goal of eFLINT-CHECK is to automatically model check eFLINT specifications and in order to prevent running into the problem of getting 'lost in translation' (*i.e.*, counterexamples that cannot be retranslated back into eFLINT scenarios), the conversion of these specifications into checkable model representations should have as few hidden abstractions as possible. Specifications that are too large to reasonably check can be turned into more abstract specifications at the eFLINT specification level instead. Therefore, we do not require the model checker and input language we will use to have any particular abstraction features.

Clarity and modularity This factor is mainly relevant for models that are defined and written by humans. Because eFLINT-CHECK is designed to create these models automatically, in principle it is not required for the input language of our model checker to have a high level of clarity and modularity. However, for the development of eFLINT-CHECK, it might be beneficial to use a language with some level of clarity. This will allow us to manually create and verify models based on eFLINT specifications, which we can then use to base the automatic translation on. Moreover, while we want the translation and checking of specifications to be automatic, we do want this process to be transparent by giving users the option to inspect and use the model encodings eFLINT-CHECK generates. For this purpose, it would also be useful if the model-checking input is human-readable. The point of modularity also pertains to the reusability of models, and whether different models can be composed together to form new models. Because we will only consider one eFLINT specification at a time, we are not concerned with this factor.

Form of composition The form of composition of a model is related to the modularity factor discussed in the previous paragraph and depends on whether the components in the system that is being checked are synchronous or asynchronous. Since we only consider one eFLINT specification at a time, we are only concerned with synchronous model checking.

Computational engines In Section 2.2.3, we briefly touched upon the fact that multiple computational approaches to model checking exist, and we introduced the notion of bounded model checking (BMC). One of the key requirements to eFLINT-CHECK is to provide actionable feedback to the users in the form of counterexamples. Because BMC can find counterexamples better than other (symbolic) model checkers and in most cases finds the shortest (and therefore usually the most comprehensible)

counterexample [20], the model checker we use should support this technique. Additionally, because eFLINT supports integers and largely depends on record data types, it would be highly beneficial if the model checker supports SMT solving instead of SAT solving. However because for now, we will only consider bounded domains, this is not a hard requirement.

Practical ease of modelling and expressiveness We partly touched on this factor when we discussed the clarity of the models. In addition to the arguments made previously, an additional requirement of the input language to the selected model checker is that it should support the same logical and arithmetic operators as eFLINT. Additionally, we would like it to offer the ability to define expressions as macros, to allow us to succinctly define derivation clauses, preconditions of acts and events and violation clauses of duties, without having to add them to the state space. This is not a hard requirement, because, in the absence of such functionality, we could express these clauses explicitly. However, this would (unnecessarily) increase the size of the model encoding of the provided specification and could make it harder to understand the generated model encodings in the event a user wants to refer to them. Lastly, in terms of interacting with the actual tool, it should be possible to programmatically interact with the model checker. Ideally, this would be through bindings to an API, but it should at least be configurable and executable through a shell command. That way, we run it from within the eFLINT-CHECK program and interact with it using common shell operations.

4.6 Interaction with eFLINT-check

For this thesis, we will implement eFLINT-CHECK as a standalone executable that takes a `.eflint` file containing the specification, initial state and properties as input, checks it and reports any counterexamples that may be found. This way, users can provide a specification (or mocked version thereof) and get immediate feedback, which they can then use to further develop or refine the provided specification. It also allows us to validate and evaluate our implementation of the tool. Future versions of eFLINT-CHECK may become more closely integrated with the existing eFLINT tooling, such as the REPL. We will discuss these possibilities in more depth in Chapter 7.

Chapter 5

Implementation of eFLINT-check

In this chapter, we will discuss the implementation of eFLINT-CHECK, according to the design we discussed in the previous chapter. The existing eFLINT backend is implemented in Haskell. Because we need to be able to access and manipulate the internal representation of eFLINT specifications, we have developed eFLINT-CHECK in Haskell as well, as a fork[1] of the original repository. We will start this chapter by explaining how we transform eFLINT specifications into labelled Kripke structures (LKSs). Then, we discuss which model checker we selected for our backend, and how we encode the LKSs and the properties provided by the user into valid input for the selected model checker. Finally, we discuss how, in the case counterexamples are found, we report them back to the user.

5.1 From eFLINT specifications to labelled Kripke structures and properties

The first step in the eFLINT-CHECK pipeline is to parse and interpret the provided `.eflint` file, so that we can transform it into an LKS. Because we are implementing eFLINT-CHECK on top of the existing eFLINT source code, we can use the parser and interpreter that are already available and use the resulting internal representation. An interpreted eFLINT specification results in a configuration data structure that stores all type declarations, the current knowledge base, the set of transitions that are allowed according to this knowledge and the set of duties that are active in this knowledge base. For eFLINT-CHECK, we are only interested in the type declarations and the current knowledge base, because this allows us to define the LKS and compute the initial state.

For practical reasons which we will explain throughout this chapter, we do not use the exact representation of LKSs as given in Definition 2. Instead, we define a new data type `Model`[2]:

```
data Model = Model
  { modelSymbolTable  :: MSymbolTable
  , modelPropositions :: MDeclarations
  , modelActions      :: MDeclarations
  , modelInitialState :: MAssignments
  , modelProperties   :: MProperties
  } deriving (Ord, Eq, Show)
```

The `modelSymbolTable` is a utility field that allows us to translate back and forth between eFLINT identifiers and (sanitized) identifiers we use in the model checker encoding of the LKS. We discuss this topic in more detail in Section 5.3.1. The `modelPropositions`, `modelActions` and `modelInitialState` are used to represent the LKS itself. As we discussed in Section 4.2.1, the set of propositions (and by extension, the initial state) in the LKS that follows from an eFLINT specification consists of all fact and duty instances of that specification, and the set of actions consists of the act and event instances. We discuss the implementation of the propositions and transitions in Section 5.1.1 and the implementation of the initial state in Section 5.1.2. Lastly, the invariants and properties that have been declared in the eFLINT specification are stored in the `modelProperties` field, which we will discuss in Section 5.1.3.

[1] https://gitlab.com/enirolf/haskell-implementation/

[2] To avoid name clashes with existing internal eFLINT data types, we prefix every data type related to the eFLINT-CHECK LKS representation with `M`, for 'model'.

5.1.1 Propositions and actions

To represent both the atomic propositions that are present in the states as well as the actions that make up each transition, we define the `MDeclaration` data type:

```
data MDeclaration = MDeclaration
  { declLabel      :: DomId
  , declAttrs      :: MDeclarationAttrs
  } deriving (Ord, Eq, Show, Read)
```

Here, `declLabel` denotes the name of the type declaration. The structure of the `declAttrs` fields depends on the kind of declaration, where we distinguish fact declarations, duty declarations and transition declarations that can be both acts and events. The structure of each of these data types is highly similar but contains subtle differences.

Fact declarations In the LKS, fact declarations are represented as follows:

```
data MFactDecl = MFactDecl
  { factType       :: MFactType
  , factFieldNames :: [DomId]
  , factInstances  :: [MFactInstance]
  } deriving (Ord, Eq, Show, Read)
```

The `factType` field denotes the domain of the fact, and can be `AtomicInt`, `AtomicString`, `Bool` or `RecordType`. The `factFieldNames` field contains a list with the field identifiers. In the case of an `AtomicInt` or `AtomicString` type, due to implementation reasons, this is a singleton list with the name of the fact itself. For `Bool` facts, this list is empty. As discussed in Section 4.2.1, we need to consider all possible instances of each eFLINT type declaration in the LKS. For facts, these are represented by the `MFactInstance` data type:

```
data MFactInstance = MFactInstance
  { factInstanceFields     :: MFields
  , factInstanceDerivation :: ITerm
  } deriving (Ord, Eq, Show, Read)
```

Here `MFields` is a data type that maps the field parameters of a type to an instance. For atomic fact types, we use the name of the type itself instead of any field name. The derivation expression of a fact instance is represented by an `ITerm` ('instantiated term').

Duty declarations Duty declarations are structured similarly to fact declarations:

```
data MDutyDecl = MDutyDecl
  { dutyInstances    :: [MDutyInstance]
  , dutyFieldNames   :: [DomId]
  } deriving (Ord, Eq, Show, Read)
```

Next to the fields from the `MFactInstance` data type, the `MDutyInstance` also contains a field that represents the violation condition:

```
data MDutyInstance = MDutyInstance
  { dutyInstanceFields     :: MFields
  , dutyInstanceDerivation :: ITerm
  , dutyInstanceViolation  :: ITerm
  } deriving (Ord, Eq, Show, Read)
```

Transition declarations Act and event declarations are both represented by the `MTransInstance`, which due to the underlying model checker encoding we will explain in the next sections only stores their instances:

```
newtype MTransDecl = MTransDecl
  { transInstances :: [MTransInstance] } deriving (Ord, Eq, Show, Read)
```

The `MTransInstance` data type stores the fields of the transitions and their pre- and postconditions:

```
data MTransInstance = MTransInstance
  { transInstanceFields    :: MFields
  , transInstancePrecon    :: ITerm
  , transInstancePostcons :: MAssignments
  } deriving (Ord, Eq, Show, Read)
```

Because the postconditions do nothing more than creating, terminating and obfuscating facts and duties, we can store them as a mapping between the identifier of the relevant fact or duty instance and their new assignment. With this representation, we do not need to represent the transition relation separately, which will be helpful during the encoding of the LKS to the model checker input.

5.1.2 Initial state

As mentioned in Section 4.2.2, we can use the eFLINT knowledge base to determine the initial state of the LKS. Following the definition of LKSs, states (including the initial state) are labelled with only the propositions (*i.e.*, all possible fact and duty instances) that hold in that state. However, for practical reasons related to the encoding of the LKS into the model checker input language, we need to represent the propositions that do not hold as well. The knowledge base may already contain some of these propositions, namely the facts and duties that have been terminated and therefore evaluate to *False*. We add the remaining obfuscated facts and duties to the initial state by taking the difference between the set of all propositions and the set of propositions already present in the knowledge base, assigning every property in this resulting set to *False*, and adding it to the initial state.

5.1.3 Properties

Similar to derivation clauses, condition clauses and domain constraints, property declarations need to be instantiated. As we discussed in Section 4.3, abstract fields will be replaced by every possible instance for that field, creating a temporal property for every possible combination between fields.

5.2 The model-checking backend

The model-checking tool that we will use as the backend to eFLINT-CHECK will be selected using the criteria discussed in Section 4.5. We selected the initial set of potential candidates based on the following criteria:

1. It should be possible to directly express the transition relation of the LKS representing the eFLINT specification in the input language of the selected model checker;
2. The selected model checker should support the specification of LTL properties;
3. The selected model checker should be able to check both safety and liveness properties;
4. The selected model checker should be able to provide counterexamples;
5. It should be able to invoke and interact with the selected model checker programmatically.

Based on these criteria, we considered ITS-Tools [31], Apalache [32] and nuXmv [33] as candidates. A systematic evaluation of each of these tools is outside the scope of this thesis. Instead, we manually created a model representation for the tutoring specification from Listing 2.1 in the modelling language of each of the tools, along with a small set of correct and incorrect properties. Using these manually defined models, we were able to evaluate (1) the expressiveness of the language, including the LTL properties; (2) the potential for automatic encoding of eFLINT specifications into this language; (3) (roughly) how long it takes to check a property; and (4) the way counterexamples are reported. Based on these points, we determined that nuXmv would be the best fit for eFLINT-CHECK. The language is extensive enough

to be able to express LKSs based on eFLINT specifications. Moreover, in this limited evaluation, it significantly outperformed the other tools in terms of the time required for checking the provided model, even when we increased the domains (and therefore state space) of the `person` fact type. nuXmv supports a large number of symbolic model checking strategies, including BMC based on both SAT and SMT solving. Lastly, the output format and verbosity of counterexamples are configurable, which will be useful in the interpretation and reporting of counterexamples to eFLINT-CHECK end-users. It can be run as an interactive command line tool, as well as in batch mode where all relevant commands are provided as an input script. This makes it possible to directly integrate it into the eFLINT-CHECK implementation. The only downside of nuXmv is that the expressiveness for composite data structures such as record-types is more limited for our purposes than the other tools. However, we will still be able to capture all language constructs of eFLINT and we argue that the advantages nuXmv has over ITS-Tools and Apalache outweigh this disadvantage.

5.2.1 The SMV language

The modelling language used by nuXmv is based on, and still closely resembles the SMV language introduced by McMillan [34]. The main difference between the nuXmv language and SMV is the fact that the nuXmv language allows for the use of unbounded Integer and Real data types for SMT-based BMC [33]. nuXmv models consist of `MODULE` declarations, where at least a `main` module should be defined for each model. Within each module, variables that make up the state of a system are specified using the `VAR` keyword. Additionally, macros may be defined using the `DEFINE` keyword. These declarations can be used to concisely store common expressions and have no influence on the size of the state space. nuXmv supports the specification and checking of different kinds of properties, including LTL properties using the `LTLSPEC` keyword and invariants using the `INVARSPEC` [35].

There are two ways in which the transition relation of a model can be expressed in the nuXmv language. The first way is through the `ASSIGN` block, where for each `VAR`, the initial value and the value in every next state are expressed. The latter can be expressed using `case` expressions, allowing the user to specify different assignments based on the current state. The other way is by direct specification of the transition relation, using the `INIT` keyword to specify the initial state and `TRANS` keyword to specify the transition relation using Boolean expressions. In general, the use of the `ASSIGN` keyword is preferred over the use over `INIT` and `TRANS`. As McMillan [34] states: "The use of `TRANS` and `INIT` is not recommended, since logical absurdities in these declarations can lead to unimplementable descriptions. [...] However, the flexibility of these mechanisms may be useful for those writing translators from other languages to SMV" ([34]). This applies to our implementation as well. In Section 4.2.3, we discussed how both the preconditions and postconditions for each act and event can be expressed as Boolean formulae, we can (almost) directly encode them into `TRANS` declarations. In the next section, we will describe in more detail how we encode the LKSs generated from eFLINT specifications into nuXmv specifications.

5.3 From LKSs and properties to nuXmv

The internal representation of the LKS that is generated from a given eFLINT specification needs to be encoded into the nuXmv input language to be able to check the properties present in this specification. In this section, we discuss the implementation details of this encoding step.

5.3.1 Identifier sanitation and instance representation

The syntax rules for eFLINT type identifiers are more permissive than those of nuXmv. For example, spaces and square brackets are valid characters for eFLINT identifiers (and are conventially used as well), but not for nuXmv identifiers. Therefore, we substitute each character that is not accepted for nuXmv identifiers with an underscore. Because we want to present counterexamples as executable eFLINT scenarios, we need to be able to translate these sanitized identifiers back into their original representation. To be able to do this, we store a symbol table alongside the LKS (represented by the `MSymbolTable` data type shown in Section 5.1).

Moreover, we need a way to distinguish different type instances in a representation that is both accepted as valid nuXmv syntax, and that enables us to translate them back into eFLINT instance identifiers. To do this, we concatenate the fields of the instance to the sanitized identifier, where the fields are separated by an underscore and a dollar sign character is used as a delimiter. For example, the fact instance `tutor-of(Alice, Bob)` will be encoded as `tutor_of$Alice_Bob`.

5.3.2 Aggregator unfolding

Derivation clauses, conditions and invariants in eFLINT may contain quantifier (`Exists`, `Forall` and `Foreach`) and aggregator expressions (`Sum` and `Count`[3]). These types of operators iterate over possible type instances to produce a single result. An example of an expression using a quantifier based on the tutoring example from Listing 2.1 is `Exists tutor: tutor-of(tutor, Alice)` returns *True* if there is at least one `person` that is the tutor of Alice. An example of an expression using an aggregator is `Count(Foreach student: deadline-passed(student))`, which counts for how many students the assignment deadline has passed. nuXmv does not support direct encoding of these types of operators. However, because we only consider bounded facts, all instances are known upfront. This allows us to unfold these expressions with these operators into quantifier-free expressions. For example, consider the `Exists` example and suppose that the domain of the `person` fact type is {`Alice`, `Bob`, `Chloe`}. Then, we can unfold this quantified expression into the following non-quantified expression:

`tutor-of(Alice, Alice)` ∨ `tutor-of(Bob, Alice)` ∨ `tutor-of(Chloe, Alice)`.

Because nuXmv supports the conversion from Boolean types to Integers, we can unfold the aggregated expression from the example above as follows, using the same domain for the `person` type as before:

int(`deadline-passed(Alice)`) + *int*(`deadline-passed(Bob)`) + *int*(`deadline-passed(Chloe)`).

5.3.3 Encoding propositions

The encoding in nuXmv of a proposition depends on its type. Here, we distinguish between atomic integer- and string-type facts, record-type facts, boolean facts, and duties. We will explain the encoding for each of these propositions below.

Atomic integer- and string-type facts

For each atomic integer- and string-type fact, we define a separate nuXmv module. This module is used to store the instances for these facts and keeps track of the value of each instance and whether it holds or not. Because the value of the instance is not relevant to the state of the LKS itself, we declare it using a `DEFINE` statement rather than a state variable. By storing the value of the fact, it can be used in derivation clauses and conditions for other facts, duties, acts and events. In the `main` module, each fact instance is declared using these modules.

For the derivation clauses of facts, we use `DEFINE` statements as well, because their value can, as the name suggests, be derived from other facts. This way, we can succinctly reference them without increasing the state space. The encoding of the `person` fact in the tutoring specification from Listing 2.1 is shown in Figure 5.1. Because this fact is not derived, the derivation clause for each instance evaluates *False*. This is necessary for the encoding of the transition relation of each act and event, which we will explain in more depth in 5.3.4.

For string-type facts, we must additionally declare the domain values as symbolic constants using the `CONSTANTS` statement. This is not necessary for integer-type facts.

Record-type facts

The encoding of record-type facts is similar to the encoding of atomic integer- and string-type facts. The main difference is that we do not store the value of each field because they have already been instantiated in every expression they appear in. The (abbreviated) encoding of the `tutor-of` fact in the tutoring specification from Listing 2.1 is shown in Figure 5.2.

[3] eFLINT also supports the `Min` and `Max` aggregator operators, but since they are rarely used in real-life cases and are not as trivial to unfold, we will not consider them here.

```
Fact person Identified by Alice, Bob
```

(a) eFLINT declaration.

```
MODULE person(person)
  DEFINE val := person;
  VAR holds : boolean;

MODULE main
  CONSTANTS
    "Alice", "Bob";
  VAR
    person$Alice : person("Alice");
    person$Bob : person("Bob");
  DEFINE person$Alice.derivation := FALSE;
  DEFINE person$Bob.derivation := FALSE;
  INIT person$Alice.holds := TRUE;
  INIT person$Bob.holds := TRUE;
  TRANS
    <...>
```

(b) nuXmv translation.

Figure 5.1: eFLINT declaration and corresponding nuXmv translation of the atomic string-type `person` fact.

```
Fact tutor-of Identified by tutor * student
```

(a) eFLINT declaration.

```
MODULE tutor_of(student, tutor)
  VAR holds : boolean;

MODULE main
  CONSTANTS
    "Alice", "Bob";
  VAR
    tutor_of$Alice_Alice : tutor_of("Alice", "Alice");
    tutor_of$Alice_Bob : tutor_of("Alice", "Bob");
    tutor_of$Bob_Alice : tutor_of("Bob", "Alice");
    tutor_of$Bob_Bob : tutor_of("Bob", "Bob");
  DEFINE tutor_of$Bob_Alice.derivation := FALSE;
  DEFINE tutor_of$Alice_Alice.derivation := FALSE;
  <...>
  INIT tutor_of$Alice_Alice.holds := FALSE;
  INIT tutor_of$Alice_Bob.holds := FALSE;
  <...>
  TRANS
    <...>
```

(b) nuXmv translation.

Figure 5.2: eFLINT declaration and corresponding nuXmv translation of the record-type `is-tutor` fact.

Boolean-type facts

Because Boolean-type facts only ever have one instance, they can directly be encoded as nuXmv state variables, without defining additional modules. To preserve consistency in the way the state of a property is referred to, we add `.holds` as a suffix to the identifier of the fact. The encoding of the Boolean-type `course-active` fact in the tutoring specification from Listing 2.1 is shown in Figure 5.3.

```
Bool course-active
```

(a) eFLINT declaration.

```
MODULE main
  VAR
    course_active.holds := boolean;
  DEFINE course_active.derivation := FALSE;
  INIT course_active.holds := FALSE;
  TRANS
    <...>
```

(b) nuXmv translation.

Figure 5.3: eFLINT declaration and corresponding nuXmv translation of the Boolean-type `course-active` fact.

Duties

Similar to atomic integer- and string-type facts and record-type facts, duties are encoded by defining a separate module for each duty. The violation clause of a duty declaration is derived from the valuation of other declared types. This means we can encode them as `DEFINE` macros instead of state variables. The (abbreviated) encoding of the `tutoring-duty` duty declaration in the tutoring specification from Listing 2.1 is shown in Figure 5.4.

```
Duty tutoring-duty
  Holder         tutor
  Claimant       student
  Holds when     tutor-of(tutor, student)
  Violated when  assignment-deadline-passed(student),
                 Not(course-active)
```

(a) eFLINT declaration.

```
MODULE tutoring_duty(tutor, student)
  VAR holds : boolean;

MODULE main
  CONSTANTS
    "Alice", "Bob";
  VAR
    tutoring_duty$Alice_Bob : tutoring_duty("Alice", "Bob");
    tutoring_duty$Alice_Alice : tutoring_duty("Alice", "Alice");
    <...>
  DEFINE tutoring_duty$Alice_Bob.derivation := tutor_of$Alice_Bob.holds;
  DEFINE tutoring_duty$Alice_Bob.violated := assignment_deadline_passed$Bob.holds
                                           | !(course_active.holds);
  DEFINE tutoring_duty$Alice_Alice.derivation := tutor_of$Alice_Alice.holds;
  DEFINE tutoring_duty$Alice_Alice.violated := assignment_deadline_passed$Alice.holds
                                             | !(course_active.holds);
  <...>
  INIT tutoring_duty$Alice_Bob.holds := FALSE;
  INIT tutoring_duty$Alice_Alice.holds := FALSE;
  <...>
  TRANS
    <...>
```

(b) nuXmv translation.

Figure 5.4: eFLINT declaration and corresponding nuXmv translation of the record-type `tutoring-duty` fact.

5.3.4 Encoding actions

nuXmv is not specifically designed for representing LKSs, or transition-based model representations for that matter. However, the nuXmv language is flexible enough to allow us to encode the LKSs that represent eFLINT specifications. As we discussed in Section 4.2.3, the precondition of an act or event, which determined whether this action is enabled, is composed using its derivation and condition clauses. Similar to the derivation clauses of facts and duties, they can be represented as `DEFINE` statements as they are evaluated using other propositions and do not influence the state of the model.

As discussed in Section 5.2.1, `TRANS` declarations allow for the direct specification of the the transition relation. We encode the transition of an act or event as the conjunction between their pre- and post-conditions, where the preconditions are represented by the `DEFINED` statements we just described. The postconditions in turn are represented by the conjunction of the updated assignments for each proposition, using nuXmv's `next` operator. The updated assignment for each proposition may on the one hand be directly determined by the `Creates`, `Terminates` or `Obfuscates` clauses of the action, but can also indirectly be determined by its derivation clause, which valuation may have changed after a transition. Because of this, we need to not only consider the propositions that are directly affected by an action in the postcondition but all propositions. If a proposition is directly affected by an action, it is assigned the updated value. If not, its updated value is determined by the disjunction between its current assignment and its derivation clause. In Section 5.3.3, we mentioned that in the nuXmv encoding, the derivation clause of facts that are not derived are assigned *False*. Because we take the disjunction between the current assignment and the derivation clause, this means that facts that are not derived will always be assigned to their current assignment. Figure 5.5 shows an (abbreviated version) of the encoding of the

`request_tutoring` act in the tutoring specification of Listing 2.1 and illustrates the difference between the encoding of propositions that are and are not affected by this action. To ensure that our transition relation is total (as is required by both the nuXmv language as well as the definition of LKSs), when no act or event is enabled, each proposition gets updated with its current state, creating a loop to the same state.

```
Act request-tutoring
  Actor           student
  Recipient       tutor
  Holds when      student != tutor
  Conditioned by  Not(tutor-of(tutor, student))
  Creates         tutor-of(tutor, student)
```

(a) eFLINT declaration.

```
MODULE main
  VAR
  <...>
  last_trans : {none, request_tutoring$Alice_Bob, request_tutoring$Alice_Alice,
                request_tutoring$Bob_Bob, request_tutoring$Bob_Alice};
  DEFINE request_tutoring$Alice_Bob.enabled :=
          ((person$Alice.val) != (person$Bob.val)) & (!(tutor_of$Bob_Alice.holds));
  DEFINE request_tutoring$Alice_Alice.enabled :=
          ((person$Alice.val) != (person$Alice.val)) & (!(tutor_of$Alice_Alice.holds));
  <...>
  INIT last_trans := none;
  <...>
  TRANS
    (request_tutoring$Alice_Bob.enabled & (
      next(last_trans) = request_tutoring$Alice_Bob
      & next(tutor_of$Alice_Bob.holds) = TRUE
      & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
      & <...>
      )
    )
    xor (request_tutoring$Alice_Alice.enabled & (
      next(last_trans) = request_tutoring$Alice_Alice
      & next(tutor_of$Alice_Alice.holds) = TRUE
      & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
      & <...>
      )
    )
    <...>
    xor ((!request_tutoring$Alice_Bob.enabled & !request_tutoring$Alice_Alice.enabled
        & !request_tutoring$Bob_Bob.enabled & !request_tutoring$Bob_Alice.enabled) & (
      next(last_trans) = none
      & <...>
      )
    )
```

(b) nuXmv translation.

Figure 5.5: eFLINT declaration and corresponding nuXmv translation of the `request-tutoring` act.

Both for checking properties that use the `Taken` operator, as well as to be able to construct eFLINT scenarios from the counterexamples that nuXmv reports, in each state we need to have access to the transition that was taken last. We do so by defining an auxiliary `last_trans` state variable, which is a nuXmv enumeration type with the identifiers of each action instance in the specification, as well as the symbol `none`. This variable gets updated in the postcondition for each transition with the value of the action taken in that transition, or `none` if no actions were enabled for that transition.

When specifying different transitions through multiple `TRANS` statements, nuXmv will define the transition relation by taking the conjunction between these statements [35]. This means, however, that

when multiple actions are enabled at a time, they will all be taken. This is not desired, because eFLINT only allows for action to be taken at a time[4]. To solve this, we encode the different transitions as one `TRANS` statement that represents the exclusive disjunction (⊕) between each transition. This ensures that each new state will only reflect the effects of one action at a time.

5.3.5 Encoding properties

In Section 4.3, we explained that we will check both invariants using the already implemented `Invariant` declaration as well as eLTL properties using the newly introduced `Property` declaration. nuXmv also distinguishes between these two kinds of properties with the `INVARSPEC` and `LTLSPEC` keywords. The nuXmv identifiers for propositions and actions allow us to convert the eLTL formulae directly into LTL formulae. Similar to expressions in derivation and condition clauses, when a property contains one or more abstract field names, we expand the property into multiple properties for each possible combination of field instances. Figure 5.6 shows an example translation of a property into nuXmv input.

```
Property become-tutor Where
    Always If Taken(request-tutoring(student, tutor))
           Then Holds(tutor-of(tutor, student))
             Until Taken(provide-tutoring(tutor, student))
```

(a) eFLINT declaration.

```
LTLSPEC NAME become_tutor$Bob_Bob :=
    G (last_trans = request_tutoring$Bob_Bob ->
       tutor_of$Bob_Bob.holds U last_trans = provide_tutoring$Bob_Bob)
LTLSPEC NAME become_tutor$Bob_Alice :=
    G (last_trans = request_tutoring$Bob_Alice ->
       tutor_of$Alice_Bob.holds U last_trans = provide_tutoring$Alice_Bob)
LTLSPEC NAME become_tutor$Alice_Bob :=
    G (last_trans = request_tutoring$Alice_Bob ->
       tutor_of$Bob_Alice.holds U last_trans = provide_tutoring$Bob_Alice)
LTLSPEC NAME become_tutor$Alice_Alice :=
    G (last_trans = request_tutoring$Alice_Alice ->
       tutor_of$Alice_Alice.holds U last_trans = provide_tutoring$Alice_Alice)
```

(b) nuXmv translation.

Figure 5.6: eFLINT declaration and corresponding nuXmv translation of the `become-tutor` property.

5.3.6 nuXmv input generation

For the automatic generation of nuXmv input files that declare the necessary modules that makeup facts and duties, state variables, declarations, initial state assignments and the transition relation, we have created a template according to the Mustache template specification [36]. With Mustache, it is possible to render files from a template by providing the content this file should contain in a structured format such as YAML or JSON. For example, the following template snippet enables us to declare a nuXmv module for every record-type fact:

```
{{#recordFacts}}
MODULE {{rfName}}({{rfFields}})
  VAR holds : boolean;
{{/recordFacts}}
```

Template tags are delimited by two curly brackets. The body of that is opened by a tag containing a

[4] Acts and events with `Syncs with` statements are an exception to this rule, but as we discussed in Section 4.2.3, we consider this out of scope for now.

pound symbol and closed by a tag containing a forward slash symbol denotes a section that is repeated for each item in the list assigned to the identifier of the tag. For example, consider the following structure representing the identifiers of all record-type facts of the tutoring specification from Listing 2.1:

```
{
  "recordFacts": [
    {
      "rfName": "tutor_of",
      "rfFields": ["tutor", "student"]
    },
    {
      "rfName": "deadline_passed",
      "rfFields": ["student"]
    },
    {
      "rfName": "is_student",
      "rfFields": ["student"]
    },
  ]
}
```

Using the template snippet and this data structure, we can render a nuXmv module for each record-type fact. The full template is available online and included in Appendix B.

We use the Haskell package `stache` [37] to handle the compilation and rendering of the template. The data structure necessary for this template is created from the `Model` data type we introduced in Section 5.1. Instances of this data type get transformed into `ModelOutput` objects, which is a Haskell data type that closely resembles the structure of a JSON object and is accepted by the rendering engine `stache`. The `stache` package uses Template Haskell [38] to allow for the compilation of Mustache during the compilation of the Haskell code. This makes it possible to render templates without having access to the template at runtime, which is ideal for our purposes.

5.4 nuXmv configuration and invocation

nuXmv can be run both interactively and as a batch command [35]. For eFLINT-CHECK, we will use the latter as that allows us to easily call and get the results from nuXmv within our Haskell implementation. The batch command can be controlled with a number of command-line options (for example, `-bmc` will invoke their default BMC-based model checking engine), but for more fine-grained configuration options, it is also possible to provide a source file. This source file accepts nuXmv commands that are available in the interactive interface, which are more specific and configurable than the command-line options. Both for checking invariants and temporal properties, we use nuXmv's IC3 engine [39]. This BMC engine uses the MathSAT [40] SMT solver and can efficiently check models with a large number of state variables, as well as model check infinite models.

We generate a source file based on the specification that is provided by the user and the following two options that may be specified by the user as well:

Property to check The given specification may contain multiple properties, whereas the user might only be interested in checking one of them (for example, if that specific property was violated in a previous run). If no property is specified, all properties in the specification will be checked.

Bound size The maximum trace length of a counterexample (k in Figure 2.3).

An example source file is shown in Listing 5.1. We save the generated source file in a directory in the working directory called `.eflint-check`. Here, we will also save the generated nuXmv input files, the raw output of nuXmv and the uninterpreted counterexamples provided by nuXmv as well as the interpreted eFLINT counterexamples.

From within our main `Check.hs` file, we spawn a subprocess that executes nuXmv given the source file and the generated input, and we capture both its standard output and standard error. The contents of the standard output get written directly to a file in the `.eflint-check` and may be used to provide the users with extra information regarding the model checker run. Because nuXmv throws an error if traces are requested that are not there (which is the case if no counterexample has been found), we read the contents of the standard error to determine whether we should report a counterexample or not. If this is the case, we need to reinterpret the counterexample reported by nuXmv back into a valid eFLINT

```
set on_failure_script_quits                -- Do not execute the rest of this file if
                                           -- one of the commands fails.
set traces_regexp last_trans               -- Only show the ``last_trans'' state variable
                                           -- in the generated counterexamples.
set default_trace_plugin 6                 -- Generate counterexamples in XML format.
go_msat                                    -- Initialize the system.
build_boolean_model                        -- Compile the provided input to an SMT
                                           -- problem.
check_invar_ic3 -k 10                      -- Check the invariant properties with a
                                           -- bound of 10.
check_ltl_spec_ic3 -k 10                   -- Check the temporal properties with a
                                           -- bound of 10.
show_traces -o .eflint-check/tutoring.xml  -- If present, save the counterexample to
                                           -- an XML file.
quit                                       -- Quit nuXmv.
```

Listing 5.1: Example nuXmv configuration file for the eFLINT specification from Listing 2.1.

scenario, such that the user can execute this scenario themselves and identify the issues that caused the relevant property to be violated. In the next section, we discuss how we implemented the interpretation of nuXmv counterexamples.

5.5 Counterexample interpretation

In the previous section, we briefly touched upon the fact that up to a certain extent, the way in which nuXmv reports counterexamples can be configured. There are two configuration options we use. The first option, configured by setting the `traces_regexp` variable in the source file, allows for stating which state variables are shown in the trace, by providing a regular expression that the identifiers of these variables have to match. For our purposes, we are only interested in the `last_trans` variable, since this provides us with the transitions that make up the scenario. Therefore, we directly provide this identifier. The second option, configured by setting the `default_trace_plugin` variable in the source file, determines the format in which the trace of the counterexample is presented. One of the available formats is XML, which enables us to use the Haskell `tagsoup` [41] package to parse these traces and extract the information we need. Using the symbol table discussed Section 5.3.1, we can translate the identifiers used in the nuXmv input back into identifiers that can be parsed and interpreted by the eFLINT reasoner. This way, we can provide the users with intuitive and executable counterexamples. In the next section, we will show what these counterexamples may look like.

5.6 The eFLINT-check executable

As discussed in Section 4.6, we initially implement eFLINT-CHECK as a standalone executable. This executable can be invoked with the following command:

```
$ eflint-check <SPECIFICATION>
```

where `<SPECIFICATION>` is the path of the eFLINT specification to check. The tool furthermore provides the user with the following options:

`-p, --prop PROPNAME` The invariant or LTL property that needs to be checked. If this option is omitted, all properties and invariants present in the provided specification are checked.

`-s, --max-steps INT` The maximum trace length of each counterexample. If omitted, a default value of 10 is used.

`-P, --print-model` Print the generated nuXmv input to the standard output.

`-x, --dont-check` Generate nuXmv input, but do not check it.

`-d, --debug` Show the internal representation of the eFLINT specification and the LKS based on this specification.

5.6.1 Example use

To illustrate the use of eFLINT-CHECK, we once again consider the tutoring specification from Listing 2.1. Furthermore, we use the `become-tutor` property we introduced earlier in this chapter:

```
Property become-tutor Where
    Always If Taken(request-tutoring(student, tutor))
           Then Holds(tutor-of(tutor, student))
             Until Taken(provide-tutoring(tutor, student))
```

Additionally, suppose that we want to verify that a deadline cannot pass after a course has ended (*i.e.*, is not active anymore). As we discussed, the postulation of facts is not supported by eFLINT-CHECK, so we add two events that toggle the Boolean `course-active` fact to the specification:

```
Event course-starts
    Creates course-active

Event course-ends
    Terminates course-active
```

This way, we can specify the following property:

```
Property no-deadlines-before-course-starts Where
    Always If Holds(is-student(student)) And Taken(course-ends())
           Then Not Enabled(assignment-due())
```

We can check the first property as follows:

```
$ eflint-check tutoring.eflint -p become-tutor
No counterexamples found!
```

Indeed, this property holds in the provided specification. However, running eFLINT-CHECK for the second property results in a counterexample:

```
$ eflint-check tutoring.eflint -p no-deadlines-before-course-starts
Counterexample found! Property or invariant 'no-deadlines-before-course-starts'
was violated.
The following scenario violates the above property or invariant:

  course-starts.
  request-tutoring(Alice, Bob).
  course-starts.
  course-starts.
  course-ends.
  course-starts.

Counterexample written to .eflint-check/tutoring.counterexample.eflint
```

We can execute the actions that are reported by eFLINT-CHECK step-by-step in the eFLINT REPL to identify that indeed, the `assignment-due` event remains enabled after the `course-ends` fact has been terminated. We note that the counterexample that NUXMV (and therefore eFLINT-CHECK) produces is not the shortest possible counterexample to violate this property. This is most likely due to the internal mechanisms of NUXMV's model checking engine. However, it is still short enough to be able to aid the debugging of the specification in this case.

5.7 Validation

It should go without saying that it is important that the transition system that arises from a NUXMV encoding of an eFLINT specification is bisimilar to the transitions system represented by the specification

itself. To gain confidence that this is indeed the case, we have created a set a set of small specifications that capture different behaviours of scenarios. For each of these specifications, we have implemented properties that we are certain should hold, and properties that should not hold. These specifications and their properties are provided in Appendix C. Furthermore, we have created an automation script (available here) to automatically execute eFLINT-CHECK with these specifications and assess whether the specified properties indeed give us the expected results.

Chapter 6

Evaluation

In this chapter, we will evaluate our language extension to eFLINT for the specification of abstract properties, together with our implementation of eFLINT-CHECK. First, we will perform a case study using the GPDR to evaluate and demonstrate the eFLINT language extension we designed and implemented and how eFLINT-CHECK can be used for the purposes we identified in Chapter 3. Second, we will evaluate the performance and scalability of eFLINT-CHECK.

6.1 Case study: GDPR

In the paper that introduces eFLINT, van Binsbergen, Liu, van Doesburg, and van Engers [1] perform a case study on the General Data Protection Regulation (GPDR) [4] pertaining to consent. In this case study, the specification shown in Listing 6.1 is created to formalize Article 6(1), point (a) on consent regarding the processing of personal data. As we explained in Section 4.2.1, eFLINT-CHECK requires the domains of each atomic-type fact to be bounded. For the purposes of this case study, we bind the `subject`, `controller`, `processor` and `purpose` fact declarations to a domain with one element, and the `data` fact declaration to a domain with two elements. For eFLINT acts, events and duties, the `Related to` fields are used to add additional parameters to these declarations, which enable the addition of extra domain constraints using the `Where` clause. We will use and, where necessary, extend this specification as a basis to demonstrate the purposes for property specification that we identified in Chapter 3 and the role eFLINT-CHECK can play here. In Section 5.6.1, we already demonstrate how eFLINT-CHECK can be used to find mistakes in eFLINT specifications, so in this section, we will focus on the remaining purposes of (1) being able to capture the temporal nature of some norms; (2) being able to give guarantees about the correctness of a specification if the norms on which it is based change; and (3) being able to relate specifications that represent actions in the physical world to those that represent actions in the social reality.

6.1.1 Lawfulness of processing

Article 6(1) of the GPDR states:

> Processing shall be lawful only if and to the extent that at least one of the following applies:
>
> (a) the data subject has given consent to the processing of his or her personal data for one or more specific purposes;
>
> [...] (Publications Office of the European Union [4])

As stated in the article, it contains multiple conditions of which at least one needs to hold. However, for the sake of this evaluation, we will assume that point (a) always has to hold, regardless of any of the other conditions. Because of the flexibility of eFLINT, this article can be formalized in different ways, of which Listing 6.1 is one example. To ensure that any formalization of the article accurately captures the norm that arises from this, we can express them as an abstract property using the property declaration we introduced in this thesis.

We identify two ways in which properties can be declared: from a state-based perspective or an action-based perspective. If it is purely declared from a state-based perspective, the implementation of the acts and events responsible for achieving this state is left to the responsibility of the eFLINT

```
Fact subject Identified by Subject
Fact data Identified by Data1, Data2

Fact subject-of Identified by subject * data

Fact controller Identified by Controller
Fact processor Identified by Processor
Fact purpose Identified by Purpose

Fact processes Identified by
  processor * data * controller * purpose

Fact accurate-for-purpose Identified by
  data * purpose

Fact consent Identified by
  subject * controller * purpose

Act give-consent
  Actor      subject
  Recipient  controller
  Related to purpose
  Holds when !consent(subject, controller, purpose)
  Creates    consent(subject, controller, purpose)

Act collect-personal-data
  Actor      controller
  Recipient  subject
  Related to data, processor, purpose
        When subject-of(subject, data)
  Holds when consent(subject, controller, purpose)
             && accurate-for-purpose(data, purpose)
  Creates    processes(processor, data, controller, purpose)
```

Listing 6.1: eFLINT specification formalizing Article 6(1)(a) of the GPDR [4], as presented by van Binsbergen, Liu, van Doesburg, and van Engers [1].

developer. On the other hand, properties from an action-based perspective are mostly agnostic of the pre- and postconditions necessary to enable these actions. Of course, it is also possible to declare properties that are both state- and action-based, and the desired way of declaring them will most likely largely be dictated by their intended use.

We can formalize Article 6(1)(a) from a state-based perspective using the fact type declarations from Listing 6.1 as follows:

```
Property lawful-processing Where
  Always If Holds(subject-of(subject, data))
            And Holds(processes(processor, data, controller, purpose))
          Then Holds(consent(subject, controller, purpose))
```

Because the `processes` and `consent` fact types do not explicitly relate a subject to their data, and there might exist a many-to-many relationship between `subject` and `data` instances, we add the expression that states that the data that is being processed must always be that of the subject that gave consent for processing. Using the acts defined in Listing 6.1, we can also write this article as an action-based property. Because the expression of this property already links the subject with their data through the `collect-personal-data` act, we don't have to use `subject-of` anymore:

```
Property lawful-processing Where
  Not Enabled(collect-personal-data(controller, subject, data, processor, purpose))
    Until Taken(give-consent(subject, controller, purpose))
```

These property declarations are bound to the specification that implements the norms by the type identifiers that appear in the property expression. However, apart from their type declaration, they do

not contain any details about how their underlying implementation in the specification. In this sense, these properties specify how the specification should *behave*, without specifically prescribing how the specification should be *implemented*. This is useful because, in this sense, a set of one or more properties can be envisioned as an extra abstraction layer between norms as they are written down in regulatory documents and the specifications that formalize them. This also relates to the purpose we identified in Section 3.1.3, which stated that properties might be used to ensure the correctness of specifications as norms change over time. For example, suppose we have formalized every condition of Article 6(1) as an eFLINT specification and as an eLTL property. Because the article states that *at least one* of the conditions of this article should hold, we can formalize this article as an eLTL property by writing it as the conjunction between each separate condition. Then, in the hypothetical event that the article gets updated to state that not at least one, but *all* of the conditions should hold for the processing of data to be lawful, we can rewrite the property as a disjunction between the conditions and run eFLINT-CHECK to identify which portions of the specification need to be changed to accurately model the article again.

6.1.2 Conditions for consent

Article 7(3) of the GPDR states:

> The data subject shall have the right to withdraw his or her consent at any time. The withdrawal of consent shall not affect the lawfulness of processing based on consent before its withdrawal. Prior to giving consent, the data subject shall be informed thereof. It shall be as easy to withdraw as to give consent. (Publications Office of the European Union [4])

This article states multiple properties that could be captured using eFLINT-CHECK, but we will focus specifically on the second sentence: "*The withdrawal of consent shall not affect the lawfulness of processing based on consent before its withdrawal*". In plain language, this sentence states that if someone (the 'data subject') withdraws their consent on the processing of their personal data, this does not remove the fact that prior to the withdrawal, the act of processing their data was allowed if they gave consent to do so. We can extend the specification from Listing 6.1 to also represent the withdrawal of consent as follows:

```
Act withdraw-consent
  Actor      subject
  Recipient  controller
  Related to purpose
  Holds when consent(subject, controller, purpose)
  Creates    consent-withdrawn(subject, controller, purpose)
  Terminates consent(subject, controller, purpose)

Fact consent-withdrawn Identified by subject * controller * purpose
```

When we only consider the eFLINT specification, we cannot accurately reason about the lawfulness of the processing of personal data before and after withdrawing consent, due to the temporal nature of this article fragment. However, eLTL makes this fairly trivial. We can simply extend the state-based variant of the `lawful-processing` as follows:

```
Property lawful-processing Where
  Always If Holds(subject-of(subject, data))
            And Holds(processes(processor, data, controller, purpose))
          Then Holds(consent(subject, controller, purpose))
            Until Holds(consent-withdrawn(subject, controller, purpose))
```

Or similarly for the event-based variant:

```
Property lawful-processing Where
  Not Enabled(collect-personal-data(controller, subject, data, processor, purpose))
    Until Taken(give-consent(subject, controller, purpose))
  And Enabled(collect-personal-data(controller, subject, data, processor, purpose))
    Until Taken(withdraw-consent(subject, controller, purpose))
```

Running eFLINT-CHECK with one or both of these properties will reveal that the specification indeed faithfully formalizes this norm.

6.2 Relating the GPDR to actions the physical world

The GPDR specification from Listing 6.1 describes actions related to norms in the social reality. van Binsbergen, Kebede, Baugh, van Engers, and van Vuurden [23] show how such specifications may be connected to specifications that describe actions in the physical world by synchronising them, which relates to the fourth and last purpose we identified in Chapter 3. As we discussed in Section 4.1, the design and implementation of synchronising actions in eFLINT-CHECK was out of scope for this thesis. However, we can demonstrate how we can use eLTL for this purpose. In our physical world, consent for the collection of data is commonly given by accepting cookies when visiting a website. We could model this behaviour in eFLINT by declaring an `accept-cookies` act type. To ensure that this action that exists in the physical world indeed leads to the desired normative state in the social reality, we can write the following property:

```
Property accept-cookies-to-give-consent Where
  Always If Taken(accept-cookies(subject, controller, purpose))
         Then Holds(consent(subject, controller, purpose))
```

Using properties in this way further supports the idea of being able to reuse specifications that pertain to the social reality in different physical situations, where they can act as the link that ensures that the consequences of actions in the physical world are accurately reflected in the social reality.

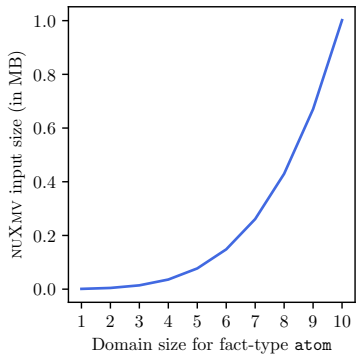
6.3 Scalability

Because the LKSs and corresponding NUXMV encodings require an explicit enumeration of all possible domain instances for each type declaration, the scalability of eFLINT-CHECK both in the dimensions of space and time is a significant concern. The most important issue here is the combinatorial explosion that can occur with record-type facts, acts, events and duties. For example, in the tutoring specification from Listing 2.1, the number of `tutor-of` fact is dictated by the domain size of the `person` fact, which is the type of both fields in this record-type fact. This means that the domain size of `tutor-of` grows quadratically with the domain size of `person`. The number of instances for a fact with three `person`-type fields would grow cubically, and so on. To illustrate this issue, we perform a small set of experiments where we measure the performance of eFLINT-CHECK both as the domain of atomic-type facts increases and as the number of fields for record-type facts increases. We perform these experiments by generating specifications using the Mustache template specification [36] that we also used for the generation of the NUXMV input. Using these templates, we generate eFLINT specifications that grow either in the dimension of domain size or field size and measure the file size of the NUXMV input file that was generated from these specifications, the time it takes for eFLINT-CHECK to generate these NUXMV input files and the time it takes for NUXMV to model check these specifications. The results for the scalability of eFLINT-CHECK concerning the domain size of atomic-type facts are shown in Figure 6.1. The results for the scalability of eFLINT-CHECK concerning the number of fields of record-type facts are shown in Figure 6.2.

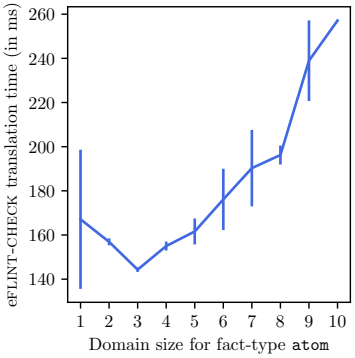
These results indicate that as of now, eFLINT-CHECK is not yet fit to check properties that require the corresponding specification to have a large number of type instances. For example, for the tutoring specification from Listing 2.1, it would currently not be feasible to check the invariant property that states that a person can never be a tutor for more than five other persons at any moment in time, as that would require the domain of the `person` fact type to have at least six elements, which, given the combinatorial nature of the declarations in the specification cannot be checked by eFLINT-CHECK in a reasonable amount of time.

However, we should also note that this is an extreme case and that there still exists a large category of properties and corresponding specifications that could benefit from using eFLINT-CHECK. For example, we were able to check the `lawful-processing` property in the GPDR case study from the previous section in a little over half a second ($6.42 \times 10^{-1} \pm 4.60 \times 10^{-3}$ s, measured over ten runs), which we consider to be a reasonable amount of time.

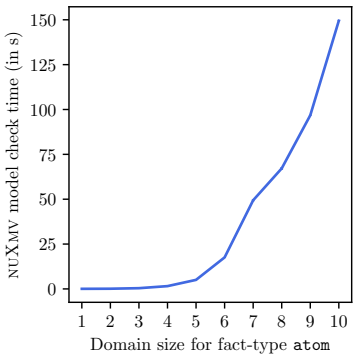
In the next chapter, we will further discuss the implications of the results of this evaluation, as well as make suggestions for optimizations to the compilation of eFLINT specifications into NUXMV input that could potentially significantly reduce the generated output and improve the time it takes for NUXMV to model-check these specifications.



(a) nuXmv input file size

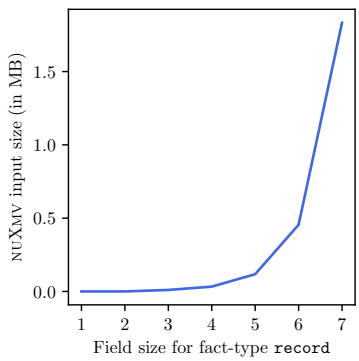


(b) eFLINT-CHECK specification encoding time, average over 10 runs.

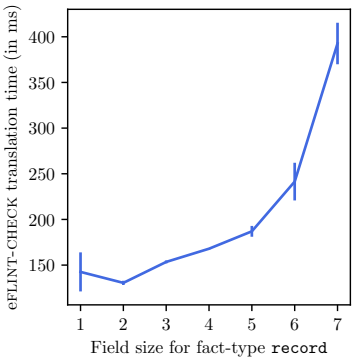


(c) nuXmv model check time, average over 10 runs.

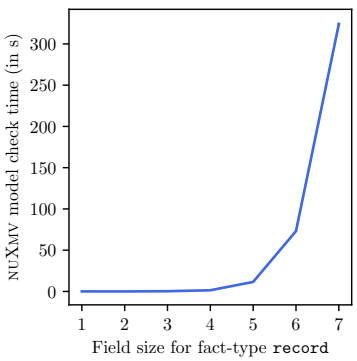
Figure 6.1: Relation between the domain size of an atomic-type fact declaration and the nuXmv input size, eFLINT-CHECK encoding time and nuXmv model check time.



(a) nuXmv input file size



(b) eFLINT-CHECK specification encoding time, averaged over 10 runs.



(c) nuXmv model check time, averaged over 10 runs.

Figure 6.2: Relation between the number of fields of a record-type fact declaration and the nuXmv input size, eFLINT-CHECK encoding time and nuXmv model check time.

Chapter 7

Discussion

In this chapter, we answer the research questions presented in our introduction. Next to that, we will identify possible threats to the validity of our work, and discuss which measures we have taken to mitigate these threats. Lastly, we will reflect on the current shortcomings of our work and make suggestions for future work.

7.1 Research Questions

In the introduction of this thesis, we identified four research questions that were central to this thesis, which we will answer in this section.

7.1.1 Research Question 1

Our first research question was: *What kinds of properties are necessary for abstract reasoning about norm specifications and being able to model-check them?* where we identified the following subquestions:

(a) What are the purposes for which model checking of norm specications would be required?
(b) How can these purposes guide the formalization of properties?

In Chapter 3, we distinguished four purposes where the use of property specification and model checking is warranted, namely for (1) the classic model-checking purpose of identifying potential mistakes in a system; (2) being able to capture the temporal nature of some norms; (3) being able to give guarantees about the correctness of a specification if the norms on which it is based change; and (4) being able to relate specifications that represent actions in the physical world to those that represent actions in the social reality. With these purposes and the underlying theory of model checking in mind, we defined eLTL, which is a variation on state/event linear temporal logic (SE-LTL) with domain-specific operators and semantics for formalizing properties over norms.

7.1.2 Research Question 2

Our second research question was: *How can norm specifications be represented formally, such that counterexamples can be generated for the properties identified by RQ 1?*

With eFLINT-CHECK, we can represent eFLINT specifications as labelled Kripke structures (LKSs) and subsequently encode these representations into the input language of the NUXMV model checker. Similarly, we can represent properties defined in eLTL as LTL formulae and encode them together with eFLINT invariant declarations as NUXMV input. We use the SMT-based bounded model checking (BMC) engine of NUXMV to check these properties and invariants against the encoded eFLINT specifications. The idea behind BMC is to generate counterexamples that incrementally grow in size (up to a certain bound k) from the provided properties and check if they are present in the specification. Because of this approach, the counterexamples that are found to appear in the model are generally small in size. We do, however, observe that NUXMV does not always report the smallest possible counterexample, but together with the other available tools for interacting with eFLINT specifications, they are reasonable enough for identifying which parts of the provided specification caused the violation of a property.

7.1.3 Research Question 3

Our third research question was: *How can the results that follow from model checking a norm specification be presented to the user in an actionable and intuitive way?*

In our nuXmv encoding of eFLINT specifications, we keep track of the last action that was taken in each transition. Moreover, we can configure nuXmv in such a way that it provides us with a trace of these actions. By translating these traces back into executable eFLINT scenarios, users can use the eFLINT read-eval-print-loop (REPL) to explore their specifications based on these scenarios, in order to identify mistakes in their specifications.

7.1.4 Research Question 4

Our fourth research question was: *What is the practical viability of adopting model checking of norm specifications for the purposes we identified in RQ 1(a)?*

The GPDR case study we performed in Chapter 6 demonstrates how eLTL and eFLINT-CHECK can be used for the different purposes we identified in Chapter 3. With this case study, we have shown how norms can be captured by eLTL and how, together with eFLINT-CHECK, these properties can serve a meaningful purpose for the formalization and implementation of norms using eFLINT.

However, we must also acknowledge that our current implementation of eFLINT-CHECK exhibits significant scalability issues, which means that the tool is not (yet) fit to be used for checking large-scale norm specifications. While state space explosion is an inherent problem in model checking, we identify several methods that we could use to minimize this problem as much as possible. We will discuss these methods in Section 7.3.

7.2 Threats to validity

The main threat to validity in the work presented here is the correctness of our encoding of eFLINT specifications into nuXmv input. When this encoding is incorrect, the underlying system represented by the generated nuXmv input will not be equivalent to the one represented by the original eFLINT scenario. In turn, this could lead to false positives, where counterexamples might be found in the nuXmv model that are not present in the eFLINT specification, or false negatives, where nuXmv does not report counterexamples even though it should. While both are undesired, false negatives are especially harmful as they are hidden from the user until a concrete scenario that contains the undesired behaviour is encountered, at which point it might be too late to fix this problem.

As we discussed in Section 5.7, we have created a small framework containing different specifications and properties to validate and gain confidence in the correctness of our implementation. Additionally, during the GPDR case study we did not encounter any suspicious results, which leads us to believe that our implementation is correct. To gain even more confidence in the correctness of the implementation, we can extend our validation framework to include more specifications and properties. This will especially be important if we extend eFLINT-CHECK with the remaining language features of eFLINT and if we implement the optimizations to the compilation process that we will discuss in the next section.

7.3 Future work

Throughout the design, implementation and evaluation of eLTL and eFLINT-CHECK, we encountered possible extensions and improvements that are necessary or interesting to explore next in the context of this work. We discuss these in this section.

7.3.1 eLTL and the specification of properties

The implementation of eLTL in eFLINT directly represents the operators defined by this logic. While this allows for a large amount of freedom in terms of the properties that can be expressed, it might be difficult or even unintuitive for users unfamiliar with temporal logic to accurately specify the properties they have in mind. Therefore, a possible next step could be to redefine the property language to make it more intuitive. One approach to this might be to provide more abstract syntax for common patterns that might occur in these properties. For example, we noticed that many properties we wrote during our

work rely on logical implication, for example in properties where it is expected that the execution of one action leads to a certain fact to hold in the next state. These implication operators must be preceded by a *globally* (□) LTL operator to make sure that they hold in every state onward, but this may not seem obvious to someone not familiar with these operators. By abstracting these operators away into more intuitive syntax, we might be able to avoid unnecessary errors in the specified properties.

Additionally, we could extend the property declarations introduced in this thesis to be able to refer to each other. This could open up the possibility of capturing one norm using multiple properties and thereby improving the readability of these properties, or the reuse of properties of multiple norms. This could also open up the possibility for the specification of conditional properties, where the results of one property could dictate how other properties should be checked.

7.3.2 Selecting the model checking bound

An important consideration in bounded model checking (BMC) is the selection of the length of the maximum bound k. Ideally, k should be large enough such that all relevant behaviour of the model is explored in such a way that a larger k would only explore traces that have already been visited. In the work presented here, we either let the user choose the bound themselves or use a default bound of 10. However, for users that do not have a sufficient understanding of the BMC procedure, this value might not mean anything which introduces the risk of them picking a bound that is too small, therefore potentially missing property violations and counterexamples. Different methods of varying complexity and accuracy have been developed for computing suitable bounds [22], and nuXmv supports some of these methods. A next step might therefore be to investigate the feasibility of automatically determining the bound, without significantly decreasing the performance of eFLINT-CHECK.

7.3.3 Performance and scalability optimizations

Staying on the subject of performance, we identify multiple different optimizations that could reduce the size of the generated nuXmv input and the time it takes to model-check the systems they represent. One approach is to perform more static processing upfront. For example during the generation of the LKS, we currently do not do anything with the domain constraint of a declaration except add it as a precondition. This essentially adds state variables to the nuXmv that can never exist in the corresponding eFLINT system, and that will therefore also never change as an effect of any action. However, they still contribute to the size of the generated input. By evaluating the domain constraints before generating the encoding, we could potentially significantly reduce the state space. For example, in the tutoring example that we used throughout this thesis, we could express the fact that a person cannot be their own tutor as a domain constraint for the `tutor-of` fact. By evaluating this expression and the effects it has on the instance domain of this fact during the compilation stage of eFLINT-CHECK, we can omit nuXmv state variables that represent the fact instances that do not satisfy this expression. However, we should also note the fact that some of these domain constraints may require information about the current state of the knowledge base, making it difficult to evaluate them without first explicitly exploring the complete state space (which we want to prevent by using model checking in the first place). A possible way approach is to only do this for a specific subset of domain constraints such as those that only contain operators comparing the value of one field to another.

During our work, two new eFLINT keywords have been introduced to restrict fact types, which we were not able to incorporate in our design and implementation due to time constraints, but that could offer significant improvements to the performance and scalability of eFLINT. The `Var` fact restriction pertains to atomic-type facts, and ensures that such facts can only have one instance that holds in the knowledge base at a time. Essentially, this means they act similar to variables in general-purpose languages. For the encoding to nuXmv, this in turn means that we don't have to create a state variable for each possible domain instance. Instead, we can define one state variable with a type specifier that corresponds to the domain of the fact it represents, and update the value of this state variable when it appears in the postconditions of actions. It follows that this could significantly reduce the size of the generated nuXmv input, and most likely the time it takes to model check this input as well. As an added benefit, this approach would make it possible to represent unbounded integer-type facts, because the SMT-based BMC engine we use in eFLINT-CHECK supports state variables with infinite integer domains.

The `Function` restriction is similar to `Var`, except that it pertains to record-type facts. Here, all fields except for the last one act as the domain to the function, with the last field acting as the codomain. Similar to the `Var` restriction, only one instance for each domain can exist at a time. Again, we can take

this into account in the encoding of these facts in nuXmv. Here, we do still need to create separate state variables for the different possible domains, but we can represent the codomain in the same way we just described for the `Var` restriction.

7.3.4 The model-checking backend

In the version of eFLINT-CHECK presented in this thesis, we used the model checker nuXmv as our backend. We discussed why we chose this tool in Section 5.2. However, there exists a plethora of model-checking tools, where each tool has its intended use and considerations. A systematic review of these tools for use with normative systems was outside the scope of this thesis but would be recommended if the work on eFLINT-CHECK continues. Moreover, we might want to investigate the possibility of directly encoding the model representation and properties that follow from eFLINT specifications into an SMT problem instead of using an existing model checker, as that could decrease the overhead of encoding specifications and invoking an underlying model checker, and it could give more fine-grained control over how eFLINT specifications can be represented and used to check eLTL properties in a robust and scalable way.

7.3.5 Integration of eFLINT-check in the explorer

eFLINT supports the dynamic generation of policies [23]. Essentially, this means that eFLINT specifications can be modified and extended on the fly. In Section 4.2.1, we discussed why this possibility cannot be captured by eFLINT-CHECK. However, it would be interesting to integrate eFLINT-CHECK in the eFLINT reasoner such that it could be invoked as soon as the specification gets updated. By integrating the model representation in the configuration for the explorer backend of eFLINT, we could keep track of the internal LKS representing the specification and update it as declarations are added or updated. We can invoke the parts of eFLINT-CHECK responsible for encoding the LKS into nuXmv input and executing nuXmv to perform the actual model checking, either by default for every change that is made or by introducing a query statement that activates this. However, before we can do this, we must address the performance and scalability issues that we currently run into, for example by implementing the proposed optimizations in the previous subsection.

Chapter 8

Related work

In this chapter, we discuss the work related to our research. We will discuss alternative logics to LTL, and how they can be generalised to be used for normative systems. Then, we will discuss other research related to model checking of normative systems and specifications.

8.1 Linear vs. branching temporal logic

In our work, we solely focused on linear temporal logic (LTL) and derivations thereof. However, depending on the model checking purpose, different temporal logics may be used. Another popular logic is computational tree logic (CTL), which has the same operators as LTL but requires every temporal operator to be preceded by a quantifier operator that indicates that the formula should hold "for all paths" (**A**) or "for some paths" (**E**) [13]. This makes it possible to distinguish between properties that must hold in all paths and properties for which at least one path should hold. Another popular temporal logic is $CTL^\star$, which combines LTL and CTL by allowing path quantification but lifting the restriction that every temporal operator should be preceded by one. Both LTL and CTL have advantages and disadvantages that are discussed in-depth by Vardi [42] but ultimately, the decision for which logic would be best to use depends on the underlying system and the purposes for which to check this system.

8.2 Temporal logics for norms

Multiple temporal logics have been proposed for capturing temporal properties over norms. Ågotnes, van der Hoek, Rodríguez-Aguilar, Sierra, and Wooldridge [43] introduce *Normative Temporal Logic (NTL)*, which is a temporal logic that replaces the path quantifiers of CTL by deontic operators relating to *obligations* and *permissions*.

Another approach is *Temporal Defeasible Logic (TDL)*, presented by [44]. Defeasible logic relates facts (not to be confused with eFLINT fact types), rules (which are made up of facts), and three kinds of relations between rules that pertain to their premises and conclusion. TDL adds the notion of temporality to this logic by adding timestamps to facts.

8.3 Model checking norm systems and specifications

FIEVeL is a specification language for modelling institutions [45]. Its syntax is based on *Ordered Many-Sorted First-Order Temporal Logic (OMSFOTL)* that allows users to both specify the normative system, as well as provide temporal properties to check them. FIEVeL has a built-in model checker in which the specifications and their properties are represented are translated into CTL formulae and subsequently encoded as Ordered Binary Decision Diagrams (OBDDs) [46], and in turn, uses SAT solving to find the solution. This language is solely intended for model-checking institutional policies and does not support on-the-fly compliance checking of the specifications, which is the case with eFLINT.

Norms are related to *multi-agent systems (MAS)* [9]. Astefanoaei, de Boer, Dastani, and Meyer [47] present a language with operational semantics to regulate the behaviour between agents and use LTL to check the norms expressed with this language.

Symboleo [7] is a norm specification language with similar foundations and semantics as eFLINT, but which is designed specifically for the purpose of creating blockchain-based smart contracts from norms.

A model checker for these specifications is in the process of being designed and implemented [48], but at the time of writing, its results have not yet been officially published.

Revani [49] is a norm specification language for analysing and verifying norms of stakeholder requirements regarding privacy. They use CTL for the specification of properties and NuSMV [50], which is the finite-state predecessor of nuXmv to model check these specifications. Revani is meant to be used to guide the design phase of sociotechnical systems and, similar to FIEVeL, cannot check compliance with these specifications on the fly.

Chapter 9

Conclusion

In this thesis, we set out to apply the notion of temporal property specification and model checking to normative systems. We did so in the context of the domain-specific language eFLINT which enables the specification and runtime compliance checking of such normative systems. We identified four different purposes where property specification and model checking could be useful with regard to normative systems, namely to (a) identify mistakes in a specification; (b) more accurately capture the temporal nature some norms might have; (c) give guarantees about the correctness of specifications as norms change; and (d) be able to more closely relate actions in the physical world to the norms that exist in the social reality. With these purposes in mind, we defined eLTL, which is a generalization of state/event linear temporal logic (SE-LTL) with operators that specifically pertain to norms. Furthermore, we designed and implemented eFLINT-CHECK, a tool using the symbolic model checker NUXMV for checking specifications against eLTL properties and providing intuitive counterexamples. We evaluated eLTL and eFLINT-CHECK through a case study on the General Data Protection Regulation (GPDR), where we demonstrated how our work might be used for the purposes we identified. Although our current implementation of eFLINT-CHECK suffers from significant scalability issues, we have identified a number of optimizations to mitigate these issues. Furthermore, we make suggestions to be able to make the specification of eLTL properties more intuitive and identify a possible next step for further integration of eFLINT-CHECK in the eFLINT reasoner.

Acknowledgements

First and foremost, I would like to thank my supervisors, Thomas van Binsbergen and Christopher Esterhuyse, for their guidance and supervision throughout this project. I remember that after one of our weekly meetings shortly after starting with this thesis, Thomas referred to the topic as “high risk, high reward”. There were plenty of moments when I felt like the risk might outweigh the reward, but you were both always able to nudge me in the right direction at exactly the right time with a new idea or suggestion. Also, I want to give an extra special thanks to Christopher for your thorough feedback on my first chapters.

I would also like to thank Ana Oprescu. Even though you were not directly involved with my thesis, ever since I followed your Project Software Engineering course during my bachelor’s, your enthusiasm and encouragement greatly inspire me and have helped me get where I am now in more ways than one.

Next up, I would like to thank all my friends, old and new, for your company during shared study sessions and (maybe more important) after these sometimes long hours of studying. A special shout-out goes to everyone in the *koffiepauze* mailing list for providing me with sometimes much-needed (and on occasion less convenient) distractions during the last months of working on my thesis.

Finally last but not least, I want to thank my parents and my sister for keeping up with me during the moments of stress when I probably wasn’t the most fun person to be around. Thank you for your never-ending support ♥.

Bibliography

[1] L. T. van Binsbergen, L.-C. Liu, R. van Doesburg, and T. van Engers, "eFLINT: A domain-specific language for executable norm specifications", in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, Virtual USA: ACM, Nov. 16, 2020, pp. 124–136, ISBN: 978-1-4503-8174-1. DOI: 10.1145/3425898.3426958.

[2] "The world's most valuable resource is no longer oil, but Data", *The Economist*, May 2017. [Online]. Available: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

[3] R. Vicente-Saez, R. Gustafsson, and L. Van den Brande, "The dawn of an open exploration era: Emergent principles and practices of open science and innovation of university research teams in a digital world", *Technological Forecasting and Social Change*, vol. 156, pp. 120 037–120 037, Jul. 2020. DOI: 10.1016/J.TECHFORE.2020.120037.

[4] Publications Office of the European Union, *General Data Protection Regulation*, Apr. 27, 2016. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[5] H. P. Lam, M. Hashmi, and B. Scofield, "Enabling Reasoning with LegalRuleML", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9718, pp. 241–257, 2016, ISSN: 9783319420189. DOI: 10.1007/978-3-319-42019-6_16.

[6] X. He, B. Qin, Y. Zhu, X. Chen, and Y. Liu, "SPESC: A Specification Language for Smart Contracts", presented at the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), IEEE, Jul. 2018, pp. 132–137, ISBN: 978-1-5386-2666-5. DOI: 10.1109/COMPSAC.2018.00025.

[7] S. Sharifi, A. Parvizimosaed, D. Amyot, L. Logrippo, and J. Mylopoulos, "Symboleo: Towards a Specification Language for Legal Contracts", presented at the Proceedings of the IEEE International Conference on Requirements Engineering, vol. 2020, IEEE Computer Society, Aug. 2020, pp. 364–369, ISBN: 978-1-72817-438-9. DOI: 10.1109/RE48521.2020.00049.

[8] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to Model Checking", in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 1–26, ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1.

[9] T. Balke, C. da Costa Pereira, F. Dignum, E. Lorini, A. Rotolo, W. Vasconcelos, and S. Villata, "Norms in MAS: Definitions and Related Concepts", in *Normative Multi-Agent Systems*, G. Andrighetto, G. Governatori, P. Noriega, and L. W. N. van der Torre, Eds., vol. 4, Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 1–31. DOI: 10.4230/DFU.VOL4.12111.1.

[10] J. R. Searle, *The Construction of Social Reality*, 1st Edition. The Free Press, New York, Mar. 1, 1995, ISBN: 0-02-928045-1.

[11] W. N. Hohfeld, "Some Fundamental Legal Conceptions as Applied in Judicial Reasoning", *The Yale Law Journal*, vol. 23, no. 1, pp. 16–16, Nov. 1913. DOI: 10.2307/785533.

[12] D. Frolich and L. T. van Binsbergen, "A Generic Back-End for Exploratory Programming", in *Trends in Functional Programming*, V. Zsók and J. Hughes, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 24–43, ISBN: 978-3-030-83978-9. DOI: 10.1007/978-3-030-83978-9_2.

[13] N. Piterman and A. Pnueli, "Temporal Logic and Fair Discrete Systems", in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 27–73, ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_2.

[14] E. M. Clarke, *Model Checking*, 2nd ed., in collab. with O. Grumberg and D. A. Peled. Cambridge, Mass: MIT Press, 1999, xiv+314, ISBN: 978-0-262-29274-0.

[15] R. De Nicola and F. Vaandrager, "Action versus state based logics for transition systems", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 469 LNCS, pp. 407–419, Nov. 1990, ISSN: 9783540534792. DOI: 10.1007/3-540-53479-2_17.

[16] R. De Nicola and F. Vaandrager, "Three logics for branching bisimulation", *Journal of the ACM*, vol. 42, no. 2, pp. 458–487, Mar. 1, 1995, ISSN: 0004-5411. DOI: 10.1145/201019.201032.

[17] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, "State/Event-Based Software Model Checking", in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 128–147, ISBN: 978-3-540-24756-2. DOI: 10.1007/978-3-540-24756-2_8.

[18] B. Alpern and F. B. Schneider, "Defining liveness", *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, Oct. 7, 1985, ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90056-0.

[19] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge MA: The MIT Press, 2008, ISBN: 978-0-262-02649-9.

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1579, pp. 193–207, 1999, ISSN: 3540657037. DOI: 10.1007/3-540-49059-0_14.

[21] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers", *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 69–83, Feb. 1, 2009, ISSN: 1433-2787. DOI: 10.1007/s10009-008-0091-0.

[22] A. Biere, "Bounded Model Checking", in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, 2nd ed., vol. 336, Amsterdam: IOS Press, 2021, pp. 739–764, ISBN: 978-1-64369-161-0. DOI: 10.3233/faia201002.

[23] L. T. van Binsbergen, M. G. Kebede, J. Baugh, T. van Engers, and D. G. van Vuurden, "Dynamic generation of access control policies from social policies", *Procedia Computer Science*, 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare, vol. 198, pp. 140–147, Jan. 1, 2022, ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.12.221.

[24] T. Schuele and K. Schneider, "Bounded model checking of infinite state systems", *Formal Methods in System Design*, vol. 30, no. 1, pp. 51–81, Feb. 1, 2007, ISSN: 1572-8102. DOI: 10.1007/s10703-006-0019-9.

[25] S. Escobar and J. Meseguer, "Symbolic Model Checking of Infinite-State Systems Using Narrowing", in *Term Rewriting and Applications*, F. Baader, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 153–168, ISBN: 978-3-540-73449-9. DOI: 10.1007/978-3-540-73449-9_13.

[26] E. Emerson and K. Namjoshi, "On model checking for non-deterministic infinite-state systems", in *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*, Jun. 1998, pp. 70–80. DOI: 10.1109/LICS.1998.705644.

[27] G. Bruns and P. Godefroid, "Model Checking Partial State Spaces with 3-Valued Temporal Logics", in *CAV*, 1999. DOI: 10.1007/3-540-48683-6_25.

[28] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel, "Multi-valued symbolic model-checking", *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 4, pp. 371–408, Oct. 1, 2003, ISSN: 1049-331X. DOI: 10.1145/990010.990011.

[29] H. Wehrheim, "Bounded Model Checking for Partial Kripke Structures", in *Theoretical Aspects of Computing - ICTAC 2008*, ser. Lecture Notes in Computer Science, J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, Eds., vol. 5160, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 380–394, ISBN: 978-3-540-85761-7 978-3-540-85762-4. DOI: 10.1007/978-3-540-85762-4_26.

[30] S. A. Seshia, N. Sharygina, and S. Tripakis, "Modeling for Verification", in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 75–105, ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_3.

[31] Y. Thierry-Mieg, "Symbolic Model-Checking Using ITS-Tools", in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2015, pp. 231–237, ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_20.

[32] I. Konnov, J. Kukovec, and T.-H. Tran, "TLA+ model checking made symbolic", *Proceedings of the ACM on Programming Languages*, vol. 3, 123:1–123:30, OOPSLA Oct. 10, 2019. DOI: 10.1145/3360549.

[33] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker", in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 334–342, ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_22.

[34] K. L. McMillan, "The SMV System", in *Symbolic Model Checking*, K. L. McMillan, Ed., Boston, MA: Springer US, 1993, pp. 61–85, ISBN: 978-1-4615-3190-6. DOI: 10.1007/978-1-4615-3190-6_4.

[35] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, *nuXmv 2.0.0 User Manual*, 2019. [Online]. Available: https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf (visited on 09/18/2022).

[36] C. Wanstrath, *Mustache(5) - Mustache Manual*, 2009. [Online]. Available: https://mustache.github.io/mustache.5.html (visited on 09/19/2022).

[37] Stack Builders, *Stache*, Stack Builders Inc., Sep. 6, 2022. [Online]. Available: https://hackage.haskell.org/package/stache-2.3.3 (visited on 09/19/2022).

[38] HaskellWiki contributors. "Template Haskell", HaskellWIki. (Aug. 10, 2022), [Online]. Available: https://wiki.haskell.org/index.php?title=Template_Haskell&oldid=65274 (visited on 09/19/2022).

[39] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 Modulo Theories via Implicit Predicate Abstraction", in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2014, pp. 46–61, ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_4.

[40] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver", in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. A. Smolka, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 93–107, ISBN: 978-3-642-36742-7. DOI: 10.1007/978-3-642-36742-7_7.

[41] Neil Mitchell, *TagSoup*, version 0.14.8, May 1, 2019. [Online]. Available: https://hackage.haskell.org/package/tagsoup-0.14.8 (visited on 09/19/2022).

[42] M. Y. Vardi, "Branching vs. Linear Time: Final Showdown", in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 1–22, ISBN: 978-3-540-45319-2. DOI: 10.1007/3-540-45319-9_1.

[43] T. Ågotnes, W. van der Hoek, J. A. Rodríguez-Aguilar, C. Sierra, and M. Wooldridge, "A Temporal Logic of Normative Systems", in *Towards Mathematical Philosophy: Papers from the Studia Logica Conference Trends in Logic IV*, D. Makinson, J. Malinowski, and H. Wansing, Eds., Dordrecht: Springer Netherlands, 2009, pp. 69–106. DOI: 10.1007/978-1-4020-9084-4_5.

[44] G. Governatori and A. Rotolo, "Changing legal systems: Legal abrogations and annulments in Defeasible Logic", *Logic Journal of IGPL*, vol. 18, no. 1, pp. 157–194, Feb. 2010. DOI: 10.1093/jigpal/jzp075.

[45] F. Viganò and M. Colombetti, "Symbolic model checking of institutions", presented at the Proceedings of the Ninth International Conference on Electronic Commerce - ICEC '07, New York, New York, USA: ACM Press, Aug. 2007, pp. 35–44, ISBN: 978-1-59593-700-1. DOI: 10.1145/1282100.1282109.

[46] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986. DOI: 10.1109/TC.1986.1676819.

[47] L. Astefanoaei, F. de Boer, M. Dastani, and J.-J. Meyer, "On the Semantics and Verification of Normative Multi-Agent Systems", *Journal of Universal Computer Science*, vol. 15, no. 13, pp. 2629–2652, Jan. 2009. DOI: 10.3217/jucs-015-13-2629.

[48] A. Parvizimosaed, "Towards the Specification and Verification of Legal Contracts", presented at the Proceedings of the IEEE International Conference on Requirements Engineering, vol. 2020-August, IEEE Computer Society, Aug. 2020, pp. 445–450, ISBN: 978-1-72817-438-9. DOI: 10.1109/RE48521.2020.00066.

[49] Ö. Kafalý, N. Ajmeri, and M. P. Singh, "Revani: Revising and Verifying Normative Specifications for Privacy", *IEEE Intelligent Systems*, vol. 31, no. 5, pp. 8–15, Sep. 2016, ISSN: 1941-1294. DOI: 10.1109/MIS.2016.89.

[50] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking", in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 359–364, ISBN: 978-3-540-45657-5. DOI: 10.1007/3-540-45657-0_29.

Terms and Acronyms

API application program interface. 26

BMC bounded model checking. 13, 14, 19, 25, 30, 37, 46, 48

CTL computational tree logic. 50, 51

DSL domain-specific language. 4, 5, 52

eLTL eFLINT-LTL. 1, 16, 17, 20, 22, 24, 36, 43, 44, 46, 47, 49, 52

GPDR General Data Protection Regulation. 4, 5, 41–44, 47, 52

LKS labelled Kripke structure. 11–13, 18–22, 24, 25, 27–31, 34, 35, 38, 44, 46, 48, 49
LTL linear temporal logic. 11–13, 15, 16, 25, 29, 30, 36, 46, 48, 50
LTS labelled transition system. 11, 25

REPL read-eval-print-loop. 9, 21, 24, 26, 39, 47

SAT boolean satisfiability. 13, 14, 26, 30, 50
SE-LTL state/event linear temporal logic. 13, 16, 25, 46, 52
SMT satisfiability modulo theories. 13, 14, 19, 26, 30, 37, 46, 48, 49

temporal logic A system of rules and symbols for reasoning about propositions that capture the notion of time. 4, 12

Appendix A

Full SMV encoding of the tutoring specification

```
MODULE person(person)
  DEFINE val := person;
  VAR holds : boolean;

MODULE assignment_deadline_passed(student)
  VAR holds : boolean;

MODULE is_student(student)
  VAR holds : boolean;

MODULE tutor_of(student, tutor)
  VAR holds : boolean;

MODULE tutoring_duty(_holder, _claimant)
  VAR holds : boolean;
  DEFINE holder_val := _holder;
  DEFINE claimant_val := _claimant;

MODULE main
  CONSTANTS
    "Alice", "Bob";
  VAR
    course_active.holds : boolean;
    person$Alice : person("Alice");
    person$Bob : person("Bob");
    assignment_deadline_passed$Alice : assignment_deadline_passed("Alice");
    assignment_deadline_passed$Bob : assignment_deadline_passed("Bob");
    is_student$Alice : is_student("Alice");
    is_student$Bob : is_student("Bob");
    tutor_of$Bob_Alice : tutor_of("Bob", "Alice");
    tutor_of$Alice_Alice : tutor_of("Alice", "Alice");
    tutor_of$Bob_Bob : tutor_of("Bob", "Bob");
    tutor_of$Alice_Bob : tutor_of("Alice", "Bob");
    tutoring_duty$Bob_Alice : tutoring_duty("Bob", "Alice");
    tutoring_duty$Alice_Alice : tutoring_duty("Alice", "Alice");
    tutoring_duty$Bob_Bob : tutoring_duty("Bob", "Bob");
    tutoring_duty$Alice_Bob : tutoring_duty("Alice", "Bob");
    last_trans : {none, request_tutoring$Alice_Bob, request_tutoring$Alice_Alice,
                  request_tutoring$Bob_Bob, request_tutoring$Bob_Alice, deadline_passes,
                  provide_tutoring$Bob_Alice, provide_tutoring$Alice_Alice,
                  provide_tutoring$Bob_Bob, provide_tutoring$Alice_Bob};
  DEFINE person$Alice.derivation := FALSE;
  DEFINE person$Bob.derivation := FALSE;
  DEFINE assignment_deadline_passed$Alice.derivation := FALSE;
  DEFINE assignment_deadline_passed$Bob.derivation := FALSE;
  DEFINE is_student$Alice.derivation := (tutor_of$Bob_Alice.holds) | (tutor_of$Alice_Alice.holds);
  DEFINE is_student$Bob.derivation := (tutor_of$Bob_Bob.holds) | (tutor_of$Alice_Bob.holds);
  DEFINE tutor_of$Bob_Alice.derivation := FALSE;
  DEFINE tutor_of$Alice_Alice.derivation := FALSE;
  DEFINE tutor_of$Bob_Bob.derivation := FALSE;
  DEFINE tutor_of$Alice_Bob.derivation := FALSE;
```

```
DEFINE request_tutoring$Alice_Bob.enabled := ((person$Alice.val) != (person$Bob.val))
                                          & ((!(tutor_of$Bob_Alice.holds))
                                          & (course_active.holds));
DEFINE request_tutoring$Alice_Alice.enabled := ((person$Alice.val) != (person$Alice.val))
                                            & ((!(tutor_of$Alice_Alice.holds))
                                            & (course_active.holds));
DEFINE request_tutoring$Bob_Bob.enabled := ((person$Bob.val) != (person$Bob.val))
                                        & ((!(tutor_of$Bob_Bob.holds))
                                        & (course_active.holds));
DEFINE request_tutoring$Bob_Alice.enabled := ((person$Bob.val) != (person$Alice.val))
                                          & ((!(tutor_of$Alice_Bob.holds))
                                          & (course_active.holds));
DEFINE deadline_passes.enabled := ((is_student$Bob.holds) | (is_student$Alice.holds)) & (TRUE);
DEFINE provide_tutoring$Bob_Alice.enabled := (tutor_of$Alice_Bob.holds) & (TRUE);
DEFINE provide_tutoring$Alice_Alice.enabled := (tutor_of$Alice_Alice.holds) & (TRUE);
DEFINE provide_tutoring$Bob_Bob.enabled := (tutor_of$Bob_Bob.holds) & (TRUE);
DEFINE provide_tutoring$Alice_Bob.enabled := (tutor_of$Bob_Alice.holds) & (TRUE);
DEFINE tutoring_duty$Bob_Alice.derivation := tutor_of$Alice_Bob.holds;
DEFINE tutoring_duty$Bob_Alice.violated := (assignment_deadline_passed$Bob.holds)
                                        | (!(course_active.holds));
DEFINE tutoring_duty$Alice_Alice.derivation := tutor_of$Alice_Alice.holds;
DEFINE tutoring_duty$Alice_Alice.violated := (assignment_deadline_passed$Alice.holds)
                                          | (!(course_active.holds));
DEFINE tutoring_duty$Bob_Bob.derivation := tutor_of$Bob_Bob.holds;
DEFINE tutoring_duty$Bob_Bob.violated := (assignment_deadline_passed$Bob.holds)
                                      | (!(course_active.holds));
DEFINE tutoring_duty$Alice_Bob.derivation := tutor_of$Bob_Alice.holds;
DEFINE tutoring_duty$Alice_Bob.violated := (assignment_deadline_passed$Alice.holds)
                                        | (!(course_active.holds));
INIT person$Alice.holds = TRUE;
INIT person$Bob.holds = TRUE;
INIT course_active.holds = TRUE;
INIT assignment_deadline_passed$Alice.holds = FALSE;
INIT is_student$Alice.holds = FALSE;
INIT tutoring_duty$Alice_Alice.holds = FALSE;
INIT tutor_of$Alice_Alice.holds = FALSE;
INIT tutoring_duty$Alice_Bob.holds = FALSE;
INIT tutor_of$Alice_Bob.holds = FALSE;
INIT assignment_deadline_passed$Bob.holds = FALSE;
INIT is_student$Bob.holds = FALSE;
INIT tutoring_duty$Bob_Alice.holds = FALSE;
INIT tutor_of$Bob_Alice.holds = FALSE;
INIT tutoring_duty$Bob_Bob.holds = FALSE;
INIT tutor_of$Bob_Bob.holds = FALSE;
INIT last_trans = none;
TRANS
  (request_tutoring$Alice_Bob.enabled & (
    next(last_trans) = request_tutoring$Alice_Bob
    & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
    & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
    & next(course_active.holds) = (course_active.holds)
    & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                    | assignment_deadline_passed$Alice.derivation)
    & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
    & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                             | tutoring_duty$Alice_Alice.derivation)
    & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                        | tutor_of$Alice_Alice.derivation)
    & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                           | tutoring_duty$Alice_Bob.derivation)
    & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
    & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                  | assignment_deadline_passed$Bob.derivation)
    & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
    & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                           | tutoring_duty$Bob_Alice.derivation)
    & next(tutor_of$Bob_Alice.holds) = (TRUE)
    & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                         | tutoring_duty$Bob_Bob.derivation)
    & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
    )
  )
```

```
xor (request_tutoring$Alice_Alice.enabled & (
  next(last_trans) = request_tutoring$Alice_Alice
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                    | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (TRUE)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                  | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor (request_tutoring$Bob_Bob.enabled & (
  next(last_trans) = request_tutoring$Bob_Bob
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                    | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                  | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (TRUE)
  )
)
xor (request_tutoring$Bob_Alice.enabled & (
  next(last_trans) = request_tutoring$Bob_Alice
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                    | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (TRUE)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                  | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
```

```
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor (deadline_passes.enabled & (
  next(last_trans) = deadline_passes
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                  | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor (provide_tutoring$Bob_Alice.enabled & (
  next(last_trans) = provide_tutoring$Bob_Alice
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                  | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (FALSE)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor (provide_tutoring$Alice_Alice.enabled & (
  next(last_trans) = provide_tutoring$Alice_Alice
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                  | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (FALSE)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
```

```
                                                | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor (provide_tutoring$Bob_Bob.enabled & (
  next(last_trans) = provide_tutoring$Bob_Bob
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                  | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (tutor_of$Alice_Bob.holds | tutor_of$Alice_Bob.derivation)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (FALSE)
  )
)
xor (provide_tutoring$Alice_Bob.enabled & (
  next(last_trans) = provide_tutoring$Alice_Bob
  & next(person$Alice.holds) = (person$Alice.holds | person$Alice.derivation)
  & next(person$Bob.holds) = (person$Bob.holds | person$Bob.derivation)
  & next(course_active.holds) = (course_active.holds)
  & next(assignment_deadline_passed$Alice.holds) = (assignment_deadline_passed$Alice.holds
                                                  | assignment_deadline_passed$Alice.derivation)
  & next(is_student$Alice.holds) = (is_student$Alice.holds | is_student$Alice.derivation)
  & next(tutoring_duty$Alice_Alice.holds) = (tutoring_duty$Alice_Alice.holds
                                          | tutoring_duty$Alice_Alice.derivation)
  & next(tutor_of$Alice_Alice.holds) = (tutor_of$Alice_Alice.holds
                                     | tutor_of$Alice_Alice.derivation)
  & next(tutoring_duty$Alice_Bob.holds) = (tutoring_duty$Alice_Bob.holds
                                        | tutoring_duty$Alice_Bob.derivation)
  & next(tutor_of$Alice_Bob.holds) = (FALSE)
  & next(assignment_deadline_passed$Bob.holds) = (assignment_deadline_passed$Bob.holds
                                                | assignment_deadline_passed$Bob.derivation)
  & next(is_student$Bob.holds) = (is_student$Bob.holds | is_student$Bob.derivation)
  & next(tutoring_duty$Bob_Alice.holds) = (tutoring_duty$Bob_Alice.holds
                                        | tutoring_duty$Bob_Alice.derivation)
  & next(tutor_of$Bob_Alice.holds) = (tutor_of$Bob_Alice.holds | tutor_of$Bob_Alice.derivation)
  & next(tutoring_duty$Bob_Bob.holds) = (tutoring_duty$Bob_Bob.holds
                                      | tutoring_duty$Bob_Bob.derivation)
  & next(tutor_of$Bob_Bob.holds) = (tutor_of$Bob_Bob.holds | tutor_of$Bob_Bob.derivation)
  )
)
xor ((!request_tutoring$Alice_Bob.enabled & !request_tutoring$Alice_Alice.enabled
    & !request_tutoring$Bob_Bob.enabled & !request_tutoring$Bob_Alice.enabled
    & !deadline_passes.enabled & !provide_tutoring$Bob_Alice.enabled
    & !provide_tutoring$Alice_Alice.enabled & !provide_tutoring$Bob_Bob.enabled
    & !provide_tutoring$Alice_Bob.enabled) & (
  next(last_trans) = none
  & next(person$Alice.holds) = person$Alice.holds
  & next(person$Bob.holds) = person$Bob.holds
  & next(assignment_deadline_passed$Alice.holds) = assignment_deadline_passed$Alice.holds
  & next(assignment_deadline_passed$Bob.holds) = assignment_deadline_passed$Bob.holds
  & next(is_student$Alice.holds) = is_student$Alice.holds
```

```
  & next(is_student$Bob.holds) = is_student$Bob.holds
  & next(tutor_of$Bob_Alice.holds) = tutor_of$Bob_Alice.holds
  & next(tutor_of$Alice_Alice.holds) = tutor_of$Alice_Alice.holds
  & next(tutor_of$Bob_Bob.holds) = tutor_of$Bob_Bob.holds
  & next(tutor_of$Alice_Bob.holds) = tutor_of$Alice_Bob.holds
  & next(tutoring_duty$Bob_Alice.holds) = tutoring_duty$Bob_Alice.holds
  & next(tutoring_duty$Alice_Alice.holds) = tutoring_duty$Alice_Alice.holds
  & next(tutoring_duty$Bob_Bob.holds) = tutoring_duty$Bob_Bob.holds
  & next(tutoring_duty$Alice_Bob.holds) = tutoring_duty$Alice_Bob.holds
  )
)
```

Appendix B

Complete nuXmv input Mustache template

N.B. this template is also available online.

```
{{#atomicFacts}}
MODULE {{afName}}({{afName}})
  DEFINE val := {{afName}};
  VAR holds : boolean;

{{/atomicFacts}}
{{#recordFacts}}
MODULE {{rfName}}({{rfFields}})
  VAR holds : boolean;

{{/recordFacts}}
{{#duties}}
MODULE {{dName}}(_holder, _claimant)
  VAR holds : boolean;
  DEFINE holder_val := _holder;
  DEFINE claimant_val := _claimant;

{{/duties}}
MODULE main
  CONSTANTS
  {{#atomicFacts}}{{#afIsString}}{{#afInstances}}
    {{#fiParams}}"{{{fpValue}}}"{{/fiParams}}{{^fiLast}}, {{/fiLast}}
  {{/afInstances}}{{^afLast}}, {{/afLast}}{{/afIsString}}{{/atomicFacts}};
  VAR
    {{#boolFacts}}
    {{bName}}.holds : boolean;
    {{/boolFacts}}
    {{#atomicFacts}}
    {{#afInstances}}
    {{fiIdentifier}} : {{afName}}({{#fiParams}}"{{{fpValue}}}"{{^fpLast}}, {{/fpLast}}{{/fiParams}});
    {{/afInstances}}
    {{/atomicFacts}}
    {{#recordFacts}}
    {{#rfInstances}}
    {{fiIdentifier}} : {{rfName}}({{#fiParams}}"{{{fpValue}}}"{{^fpLast}}, {{/fpLast}}{{/fiParams}});
    {{/rfInstances}}
    {{/recordFacts}}
    {{#duties}}
    {{#dInstances}}
    {{diIdentifier}} : {{dName}}({{#diParams}}"{{{fpValue}}}"{{^fpLast}}, {{/fpLast}}{{/diParams}});
    {{/dInstances}}
    {{/duties}}
    last_trans : {none{{#transitions}},
      {{#tInstances}}{{tiIdentifier}}{{^tiLast}}, {{/tiLast}}{{/tInstances}}
    {{/transitions}}};
  {{#atomicFacts}}
  {{#afInstances}}
  DEFINE {{fiIdentifier}}.derivation := {{fiDerivation}};
  {{/afInstances}}
  {{/atomicFacts}}
```

```
  {{#recordFacts}}
  {{#rfInstances}}
  DEFINE {{fiIdentifier}}.derivation := {{fiDerivation}};
  {{/rfInstances}}
  {{/recordFacts}}
  {{#transitions}}
  {{#tInstances}}
  DEFINE {{tiIdentifier}}.enabled := {{{tiPrecon}}};
  {{/tInstances}}
  {{/transitions}}
  {{#duties}}
  {{#dInstances}}
  DEFINE {{diIdentifier}}.derivation := {{diDerivation}};
  DEFINE {{diIdentifier}}.violated := {{diViolationCon}};
  {{/dInstances}}
  {{/duties}}
  {{#initialState}}
  INIT {{sIdentifier}}.holds = {{sValue}};
  {{/initialState}}
  INIT last_trans = none;
  TRANS
    {{#transitions}}{{#tInstances}}({{tiIdentifier}}.enabled & (
      next(last_trans) = {{tiIdentifier}}
  {{#tiPostcons}}
      & next({{tpIdentifier}}.holds) = ({{tpAssignment}}{{#tpAssignmentToSelf}}.holds{{^tpIsBool}}
      | {{tpIdentifier}}.derivation{{/tpIsBool}}{{/tpAssignmentToSelf}})
  {{/tiPostcons}}
      )
    )
  {{^tiLast}}  xor {{/tiLast}}{{/tInstances}}{{^tLast}}  xor {{/tLast}}{{/transitions}}  xor (({{#transitions}}
    {{#tInstances}}!{{tiIdentifier}}.enabled{{^tiLast}} & {{/tiLast}}{{/tInstances}}{{^tLast}} & {{/tLast}}
  {{/transitions}}) & (
      next(last_trans) = none
      {{#atomicFacts}}
      {{#afInstances}}
      & next({{fiIdentifier}}.holds) = {{fiIdentifier}}.holds
      {{/afInstances}}
      {{/atomicFacts}}
      {{#recordFacts}}
      {{#rfInstances}}
      & next({{fiIdentifier}}.holds) = {{fiIdentifier}}.holds
      {{/rfInstances}}
      {{/recordFacts}}
      {{#duties}}
      {{#dInstances}}
      & next({{diIdentifier}}.holds) = {{diIdentifier}}.holds
      {{/dInstances}}
      {{/duties}}
      )
    )

{{#properties}}
{{#pIsInvariant}}
INVARSPEC NAME {{{pName}}} := {{{pFormula}}}
{{/pIsInvariant}}
{{^pIsInvariant}}
LTLSPEC NAME {{{pName}}} := {{{pFormula}}}
{{/pIsInvariant}}
{{/properties}}
```

Appendix C

Validation specifications

C.1 Single act instance

```
Fact x Identified by X
Fact y Identified by Y

Fact xy Identified by x * y

Act make-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by !xy(x, y)
  Creates xy(x, y).

Property good-a Where Always If Taken(make-record(x, y)) Then Holds(xy(x, y)).
Property good-b Where Not Holds(xy(x, y)) Until Taken(make-record(x, y)).

Property bad-a Where Always If Taken(make-record(x, y)) Then Not Holds(xy(x, y)).
Property bad-b Where Holds(xy(x, y)) Until Taken(make-record(x, y)).

+x(X).
+y(Y).
```

C.2 Multiple act instances

```
Fact x Identified by X
Fact y Identified by Y

Fact xy Identified by x * y

Act make-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by !xy(x, y)
  Creates xy(x, y)

Act remove-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by xy(x, y)
  Creates xy(x, y).

Property good-a Where Always If Taken(make-record(x, y)) Then Holds(xy(x, y)).
Property good-b Where Not Holds(xy(x, y)) Until Taken(make-record(x, y)).

Property bad-a Where Always If Taken(make-record(x, y)) Then Not Holds(xy(x, y)).
Property bad-b Where Holds(xy(x, y)) Until Taken(make-record(x, y)).

+x(X).
+y(Y).
```

C.3 Act loop

```
Fact x Identified by X1, X2
Fact y Identified by Y

Fact xy Identified by x * y

Act make-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by !xy(x, y)
  Creates xy(x, y)

Act remove-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by xy(x, y)
  Creates xy(x, y).

Property good-a Where Always If Taken(make-record(x, y)) Then Holds(xy(x, y)).
Property good-b Where Not Holds(xy(x, y)) Until Taken(make-record(x, y)).

Property bad-a Where Always If Taken(make-record(x, y)) Then Not Holds(xy(x, y)).
Property bad-b Where Holds(xy(x, y)) Until Taken(make-record(x, y)).

+x(X1).
+x(X2).
+y(Y).
```

C.4 Duties

```
Fact x Identified by X1
Fact y Identified by Y1
Fact z Identified by x * y

Duty duty
  Holder x
  Claimant y
  Violated when z(x, y)

Act create-duty
  Actor x
  Recipient y
  Holds when x && y
  Creates duty(x, y)

Act resolve-duty
  Actor x
  Recipient y
  Holds when x && y
  Terminates duty(x, y)

Act violate-duty
  Actor x
  Recipient y
  Holds when x && y
  Creates z(x, y).

Property good-a Where Always If Not Taken(violate-duty(x, y)) Then Always Not Violated(duty(x, y))

+x(X1).
+y(Y1).
```

Appendix D

Performance evaluation templates

D.1 Variable domain size for atomic-type fact

```
Fact atom Identified by {{#literals}}{{ident}}{{^last}}, {{/last}}{{/literals}}

Placeholder x For atom
Placeholder y For atom

Fact record Identified by x * y

Act make-record
  Actor x
  Recipient y
  Holds when x && y
  Conditioned by !record(x, y)
  Creates record(x, y)

Property prop Where Always If Taken(make-record(x, y)) Then Holds(record(x, y)).
```

D.2 Variable field size for record-type fact

```
Fact atom Identified by A, B

Placeholder actor For atom
Placeholder recipient For atom
{{#fields}}
Placeholder {{ident}} For atom
{{/fields}}

Fact record
  Identified by actor * recipient * {{#fields}}{{ident}}{{^last}} * {{/last}}{{/fields}}

Act make-record
  Actor actor
  Recipient recipient
  Related to {{#fields}}{{ident}}{{^last}}, {{/last}}{{/fields}}
  Holds when {{#fields}}{{ident}}{{^last}} && {{/last}}{{/fields}}
  Creates record(actor, recipient, {{#fields}}{{ident}}{{^last}}, {{/last}}{{/fields}})

Property prop Where
  Always If Taken(make-record(actor, recipient,
                              {{#fields}}{{ident}}{{^last}}, {{/last}}{{/fields}}))
         Then Holds(record(actor, recipient,
                           {{#fields}}{{ident}}{{^last}}, {{/last}}{{/fields}})).
```