Tableaux for Verification of Data-Centric Processes

Andreas Bauer, Peter Baumgartner, Martin Diller and Michael Norrish

NICTA[⋆] and Australian National University, Canberra, Australia

Abstract. Current approaches to analyzing dynamic systems are mostly grounded in propositional (temporal) logics. As a consequence, they often lack expressivity for modelling rich data structures and reasoning about them in the course of a computation. To address this problem, we propose a rich modelling framework based on first-order logic over background theories (arithmetics, lists, records, etc) and state transition systems over corresponding interpretations. On the reasoning side, we introduce a tableau calculus for bounded model checking of properties expressed in a certain fragment of CTL* over that first-order logic. We also describe a k-induction scheme on top of that calculus for proving safety properties, and we report on first experiments with a prototypical implementation.

1 Introduction

Current approaches to analyzing dynamic systems are mostly grounded in propositional (temporal) logics. As a consequence, they often lack expressivity for modelling rich data structures and reasoning about them in the course of a computation. To address this problem, we propose an expressive, sorted first-order logic to describe states, a fragment of CTL* to describe systems' evolution, and we introduce a tableau calculus for model checking in that logic. Our approach is based on

– *process fragments* that describe specific tasks of a larger process, inspired by what is known as *declarative business process modelling* [17]. As a result, users do not have to specify a single, large transition system with all possible task interleavings.
– *constraints* for limiting the interactions between the fragments. In this way, users can create many small process fragments whose interconnections are governed by rules that determine which executions are permitted.
– *first-order temporal logic*. Unlike [8], we choose to extend CTL*, *i.e.*, a branching time logic, rather than LTL, since process fragments are essentially annotated graphs and CTL* is, arguably, appropriate to express its properties (*cf.* [6]).
– *sorts* for JSON objects [7], where sorts are governed by a custom, static type system which models and preserves the type information of any input data. JSON objects allow for richly structured data types such as lists and records.

Tableau calculi have been long considered (*e.g.*, [10]) an appropriate and natural reasoning procedure for temporal logics. There is even a tableau procedure for propositional CTL* [18]. However, we are not aware of a first-order logic tableaux calculus

[⋆] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

that accommodates our requirements, hence we devise one (Section 4). We note that we circumvent the difficult problem of loop detection by working in a *bounded* model checking setting, where runs are artificially terminated when they become too long.

The high expressivity of our approach comes at the price of high undecidability, and so practical feasibility is an issue. Ultimately, all our reasoning problems reduce to first-order logic proof obligations, and hence automated reasoning in first-order logic becomes a crucial component. Although we focus on the core logic of our framework, we also report on first experiments with a prototypical implementation that integrates our tableau procedure with the state of the art SMT solver Z3 [14].

Related Work. In the area of business process modelling, the so-called "business artifact" approach pioneered the idea of making data a "first-class citizen" (Nigam and Caswell [16]). The *artifacts* of this approach are records of data values that can change over time due to the modifications performed by *services*, which are formalized using first-order logic. Process analysis answers the following question: given some artifact model, a database providing initial values, and a correctness property in terms of a first-order LTL formula, do all possible artifact changes over time satisfy the correctness property? For the constraints given in Damaggio *et al.* [8], this problem is decidable. We refer to this problem as "concrete model checking" since an initial state has to be given. We are also interested in the generalization thereof, where the set of possible initial states is unconstrained, the "general model checking" problem.

Schuele and Schneider [20] give a categorisation of temporal model checking problems. They differentiate between global model checking techniques, which are basically fix-point iterations, and local techniques, which are inference based and analyse a formula in a top-down fashion by inspecting its syntax tree. As such, both our concrete and general model checking problems fall under the local techniques category.

Bersani *et al.* [4] describe linking SMT-solvers to decide bounded model checking problems over temporal logic extensions. There, LTL with integer constraints is considered, which results in an undecidable satisfiability problem. However, by constraining the number of variables and length of paths, a decidable satisfiability and model checking problem is obtained.

Another example combining data and dynamics is Vianu [23]. This work uses an LTL in which the propositions can be replaced by a background theory statement, in particular FOL, to verify systems whose behaviour is expressible as sequence of database updates. Since the latter results in an infinite-state system, Vianu imposes several restrictions on the database properties, and obtains a PSpace model checking algorithm. In the area of description logics, Hariri *et al.* [12] and Chang *et al.* [5] both present systems that allow for rich queries over dynamically evolving systems. Entities within the systems can be related to one another and characterized in a first-order style, but there is no scope for the use of types such as numbers and lists as in our work.

In Ghilardi *et al.* [9], fragments of first-order linear-time temporal logic with background theories are considered. While the general satisfiability problem of such a logic is necessarily undecidable, the authors identify the quantifier-free fragment, which can be decided in PSpace, given that the background constraints can. Work on temporalizing description logics heads in a similar direction. The challenge there is to determine fragments of, *e.g.*, LTL over description logics so that the desired reasoning services become decidable. See, *e.g.*, Baader *et al.* [1] for recent work.

2 Preliminaries

We work with sorted signatures Σ consisting of a non-empty set *sorts* and function and predicate symbols of fixed arities over these sorts. We assume infinite supplies of variables, one for each sort. A *constant* is a 0-ary function symbol. The (well-sorted Σ-) terms and atoms are defined as usual. We assume Σ contains a predicate symbol $=_s$ (equality) of arity $s \times s$, for every sort $s \in \mathit{sorts}$. Equational atoms, or just *equations*, are written infix, usually without the subscript s, as in $1 + 1 = 2$. We write $\theta[\boldsymbol{x}]$ to indicate that every free variable in the formula θ is among the list $\boldsymbol{x}$ of variables, and we write $\theta[\boldsymbol{t}]$ for the formula obtained from $\theta[\boldsymbol{x}]$ by replacing all its free variables $\boldsymbol{x}$ by the corresponding terms in the list $\boldsymbol{t}$.

We assume a sufficiently rich set of Boolean connectives (such as $\{\neg, \wedge\}$) and the quantifiers $\forall$ and $\exists$. The *well-sorted Σ-formulas*, or just *(FO) formulas* are defined as usual. We are particularly interested in signatures containing (linear) integer arithmetic. For that, we assume $\mathbb{Z} \in \mathit{sorts}$, the $\mathbb{Z}$-sorted constants $0, \pm1, \pm2, \ldots$, the function symbols $+$ and $-$, and the predicate symbol $>$, each of the expected arity over $\mathbb{Z}$.

The semantics of our logic is the usual one: a *Σ-interpretation $\mathcal{I}$* consists of non-empty, disjoint sets, called *domains*, one for each sort. We require that the domain for $\mathbb{Z}$ is the set of integers, and that every arithmetic function and predicate symbol (including $=_\mathbb{Z}$) is mapped to its obvious function or relation, respectively, over the integers. Indeed, we will later see that we treat other sorts, such as lists and other JSON types as "built-in" (see Section 3). In brief, our modelling framework supports the use of (finite) lists, arrays and records in a monomorphically sorted setting. Thus, we further require that Σ-interpretations interpret the function and predicate symbols associated with these sorts in a way consistent with the intended semantics, which can be given axiomatically. This is consistent with the de-facto TPTP standard [21], so that compliant theorem provers can be applied.

A *(variable) assignment* α is a mapping from the variables into their corresponding domains. Given a formula θ and a pair $(\mathcal{I}, \alpha)$ we say that $(\mathcal{I}, \alpha)$ *satisfies* θ, and write $(\mathcal{I}, \alpha) \models \theta$, iff θ evaluates to true under $\mathcal{I}$ and α in the usual sense (the component α is needed to evaluate the free variables in θ). If θ is closed then α is irrelevant and we can write $\mathcal{I} \models \theta$ instead of $(\mathcal{I}, \alpha) \models \theta$. We say that a closed sentence θ is *valid* (*satisfiable*) iff $\mathcal{I} \models \theta$ for all (some) interpretations $\mathcal{I}$.

Processes in our framework are modeled as state transition systems. A *state transition system* is a tuple $M = (S, I, R)$ where S is a set of *states*, $I \subseteq S$ are the *initial states*, and $R \subseteq S \times S$ the *transition relation*.[1] Throughout this paper, states are mappings from the variables into their corresponding domains, i.e., every state $s \in S$ is an assignment (but in general not every assignment α is a state in S).

Our query language is a fragment of CTL* over first-logic, which we refer to as CTL*(FO). Its syntax is given by the following grammar:

$$\phi ::= \theta \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{A}\,\psi \mid \mathsf{E}\,\psi \qquad \psi ::= \phi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\,\psi \mid \overline{\mathsf{X}}\,\psi \mid \psi\,\mathsf{U}\,\psi \mid \psi\,\mathsf{R}\,\psi$$

[1] Notice we do not require R to be (left-) total, as runs may be *finite*.

where θ refers to a FO formula, ϕ is called a *state formula* and ψ a *path formula*. The operator $\overline{\mathsf{X}}$ is called "weak next". A CTL*(FO) formula is *pure FO* iff it does not contain any path quantifier and does not contain any temporal operator.

Let $M = (S, I, R)$ be a state transition system as stated above and $s_0 \in S$. A *run r (of M) from s_0* is a possibly infinite sequence $s_0\, s_1\, s_2\, \cdots$ of states such that $(s_i, s_{i+1}) \in R$. Let $r[i] = s_i$, and r^i the *truncated run* $s_i\, s_{i+1}\, \cdots$. By $|r|$ we denote the number of elements in r or ∞, if r is infinite. Obviously, $r^0 = r$.

For any state formula $\phi \in$ CTL*(FO), interpretation $\mathcal{I}$, and state $s_0 \in S$ we define a satisfaction relation $\models$. It differs somewhat from the usual definition (cf. [6]) in that it is implicitly parametric in a set of *admissible runs (of M)*. We identify the set of admissible runs with its closure under truncation of runs.

A finite run $s_0 \cdots s_n$ is called *finished* if there is no $s \in S$ such that $(s_n, s) \in R$. That is, finished runs do not stop prematurely. The set of *standard runs (of M)* consists of all infinite runs and all finished runs. Unless stated otherwise we assume standard runs.

For any state formula $\phi \in$ CTL*(FO), interpretation $\mathcal{I}$, and state $s_0 \in S$, the satisfaction relation $(\mathcal{I}, s_0) \models \phi$ is defined as follows,

$$\begin{array}{ll}
(\mathcal{I}, s_0) \models \theta & \text{iff } (\mathcal{I}, s_0) \models \theta \\
(\mathcal{I}, s_0) \models \neg\phi & \text{iff } (\mathcal{I}, s_0) \not\models \phi \\
(\mathcal{I}, s_0) \models \phi_1 \land \phi_2 & \text{iff } (\mathcal{I}, s_0) \models \phi_1 \text{ and } (\mathcal{I}, s_0) \models \phi_2 \\
(\mathcal{I}, s_0) \models \mathsf{A}\,\psi & \text{iff } (\mathcal{I}, r) \models \psi \text{ for all runs } r \text{ from } s_0 \\
(\mathcal{I}, s_0) \models \mathsf{E}\,\psi & \text{iff } (\mathcal{I}, r) \models \psi \text{ for some run } r \text{ from } s_0,
\end{array}$$

where the satisfaction relation $(\mathcal{I}, r) \models \psi$ for path formulas ψ and admissible r is

$$\begin{array}{ll}
(\mathcal{I}, r) \models \phi & \text{iff } (\mathcal{I}, r[0]) \models \phi \\
(\mathcal{I}, r) \models \neg\psi & \text{iff } (\mathcal{I}, r) \not\models \psi \\
(\mathcal{I}, r) \models \psi_1 \land \psi_2 & \text{iff } (\mathcal{I}, r) \models \psi_1 \text{ and } (\mathcal{I}, r) \models \psi_2 \\
(\mathcal{I}, r) \models \mathsf{X}\,\psi & \text{iff } |r| > 1 \text{ and } (\mathcal{I}, r^1) \models \psi \\
(\mathcal{I}, r) \models \overline{\mathsf{X}}\,\psi & \text{iff } |r| \leq 1, \text{ or } |r| > 1 \text{ and } (\mathcal{I}, r^1) \models \psi \\
(\mathcal{I}, r) \models \psi_1 \,\mathsf{U}\, \psi_2 & \text{iff there exists a } j \geq 0 \text{ such that } |r| > j \text{ and } (\mathcal{I}, r^j) \models \psi_2, \\
 & \quad \text{and } (\mathcal{I}, r^i) \models \psi_1 \text{ for all } 0 \leq i < j \\
(\mathcal{I}, r) \models \psi_1 \,\mathsf{R}\, \psi_2 & \text{iff } (\mathcal{I}, r^i) \models \psi_2 \text{ for all } i < |r|, \text{ or there exists a } j \geq 0 \text{ such that} \\
 & \quad |r| > j,\ (\mathcal{I}, r^j) \models \psi_1 \text{ and } (\mathcal{I}, r^i) \models \psi_2 \text{ for all } 0 \leq i \leq j.
\end{array}$$

We assume the usual "syntactic sugar", which can easily be defined in terms of the above set of operators in the expected way. Note that we distinguish a strong next operator, X, from a weak next operator, $\overline{\mathsf{X}}$, as described in [2]. This gives rise to the following equivalences: $\psi_1 \,\mathsf{R}\, \psi_2 \equiv \psi_2 \land (\psi_1 \lor \overline{\mathsf{X}}\,(\psi_1 \,\mathsf{R}\, \psi_2))$ and $\psi_1 \,\mathsf{U}\, \psi_2 \equiv \psi_2 \lor (\psi_1 \land \mathsf{X}\,(\psi_1 \,\mathsf{U}\, \psi_2))$ as one can easily verify by using the above semantics. This choice is motivated by our bounded model checking algorithm, which has to evaluate CTL*(FO) formulas over finite traces as opposed to infinite ones. For example, when evaluating a safety formula, such as $\mathsf{G}\,\psi$, we want a trace of length n that satisfies ψ in all positions $i \leq n$ to be a model of this formula. On the other hand, if there is no position $i \leq n$, such that ψ is satisfied, we don't want this trace to be a model for $\mathsf{F}\,\psi$. This is achieved in our logic as $\mathsf{G}\,\psi \equiv \psi \land \overline{\mathsf{X}}\,\mathsf{G}\,\psi$ and $\mathsf{F}\,\psi \equiv \psi \lor \mathsf{X}\,\mathsf{F}\,\psi$ hold. Note also that $\neg\mathsf{X}\,\psi \not\equiv \mathsf{X}\,\neg\psi$, but $\neg\mathsf{X}\,\psi \equiv \overline{\mathsf{X}}\,\neg\psi$.

3 The Specification Language

We provide a specification language to define processes and the data they manipulate. A concrete specification consists of the sections `TYPES`, `SIGNATURE`, `DEFINITIONS`, `CONSTRAINTS`, and `DIGRAPH`. We explain each section in turn, including sample extracts in each explanation from a business process modelling domain.

Logic for Individual States. Our specification logic stratifies into two levels. The first level uses non-temporal first-order formulas to describe individual states of a system. Formulas at this level are richly *typed*, and may refer to user-defined logical notions.

JSON Values, `TYPES` and `SIGNATURE`. Users capture the states of their systems with *JSON values*. JSON [7] is an untyped framework for writing structured data, including base types such as strings and integers, as well as structure through records (field names coupled with values) and arrays. The JSON syntax is rich enough to represent complex states while remaining human-readable. We layer a simple type-system over JSON, ultimately providing a connection between these types and the sorts of CTL*(FO).

The atomic types of our specification language are `String`, `Bool` and `Integer`. In addition, users can define new types that are built up from these atomic types, the type operators `Array[_]`, `List[_]`, and a syntax for record types, *i.e.*, a list of field names coupled with types for those fields. Types may occur within other type definitions, as long as there are no recursive loops. This restriction means that users cannot specify their own recursive types (such as trees). This restriction does not seem too onerous in practice and makes the axiomatic characterization of the types straightforward.

The types from the purchase order example are shown on the right. The `DB` type corresponds to the entire system state. The `stock` array holds information about stock items, for each item number 0..`nrStockItems`-1. The `Stock.available` field is the number of items in stock, per item number. The `open` list contains the open order item numbers, those that have not been packed yet. The `gold` bit says whether the customer is a gold customer. The `invoice` filed says whether an invoice has been generated. The `paid` and `shipped` fields control the composition of "process fragments", see below.

```
DB = {
  stock: Array[Stock],
  nrStockItems: Integer,
  open: List[Integer],
  gold: Boolean,
  invoice: Bool,
  paid: Bool,
  shipped: Bool }
Stock = {
  ident: String,
  price: Integer,
  available: Integer }
```

There are naturally various operations over terms of the corresponding JSON types that our logic must support. Thus we support arithmetic function and relation symbols (+, <, *etc.*). Support for JSON record types includes functions for accessing fields of objects (the concrete syntax is the familiar "dot notation"; *e.g.*, `s.value`) and for creating new record values by updating field values of old values ("functional record update"). Depending on the nature of the back-end reasoning tool, elements of the signature may be characterized directly in FOL, as is done for the record functions. By contrast, we expect backend reasoning tools to directly support arithmetic, arrays and lists, with the usual operators on them, freeing us from providing a FOL axiomatization for the latter (this is not a critical limitation). We refer to this extended language as *JSON Logic*, and talk of *JSON sentences* and *JSON terms etc.*

In addition, users can declare and define their own functions, predicates and relations over these types. Those entities without definition will be uninterpreted. All such, whether or not they are later defined, are listed in the `SIGNATURE` section along with their types. For example, the `completed` predicate on `Status` arguments is given in this section with the syntax `completed: [Status] -> Bool`.

`DEFINITIONS` and `CONSTRAINTS`. The `DEFINITIONS` section consists of a set of FO JSON sentences, providing the semantics for (some of) the free predicate and function symbols declared in the `SIGNATURE`. Let *defs* be the image of the `DEFINITIONS` section under translation into CTL*(FO).

The `CONSTRAINTS` sections consists of a set of JSON CTL*(FO) path formulas. Unlike `DEFINITIONS`, the free variable `DB`-sorted variable *db* is permitted. It represents the database at the current time point. The intention is to provide additional constraints on the runs considered in the reasoning problems below. Let *constraints* be the image of the `CONSTRAINTS` section under translation into CTL*(FO).

Examples of definitions and constraints occur in Figure 1. One such is the definition of the mentioned `completed` predicate over system statuses. The sample constraint is a temporal property using the "weak until" `W` operator. It encodes the rule that customers without "gold" status can never have their order shipped before they have paid.

Adding Dynamics. Above the state-based level of the previous section, we allow users to define a "process fragment"-based dynamics for their systems by means of *process graphs*. Formally, a *process graph* G is a directed labeled graph (N, E), where N is a set of *nodes* and $E \subseteq N \times N$ is a set of *edges*. Exactly one node must be labelled as an "init node". Each node can be labelled as an "entry node" or "exit node" (or both). A *guard* is a FO formula with free variables at most $\{db\}$; an *update term* is a FO term with free variables at most $\{db\}$. Entry nodes and edges always have both a guard and an update term attached to them, which are denoted by $guard(n)$ and $upd(n)$ for entry nodes n, respectively, and analogously for edges.

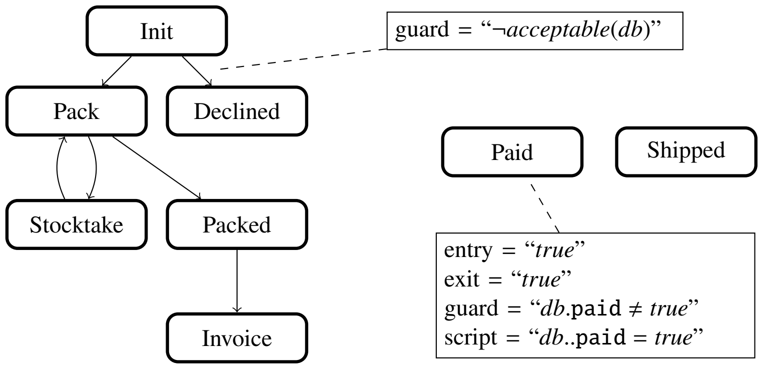
The concrete syntax for process graphs should be obvious from our running example. Every `script`, a sequence of assignments, is taken as an update term. The semantics of entry/exit nodes is defined by implicitly putting an edge between every exit node and every entry node and using the entry node's guard and script for the edge.

We capture this intuition formally and in a uniform way by defining a *labelled edge relation* consisting exactly of the quadruples $m \xrightarrow{\gamma,u} n$ such that $m, n \in N$ and either $(m, n) \in E$, $\gamma = guard(m, n)$ and $u = upd(m, n)$, or m is an exit node, n is an entry node, $\gamma = guard(n)$ and $u = upd(n)$.

The labelled edge relation induces a state transition system $M = (S, I, R)$ as follows. The states S are all assignments s of the form $\{\ell \mapsto n,\ db \mapsto d\}$ where $n \in N$ and d is a domain element of sort `DB`. Notice that ℓ and db are fixed. Then,

$$I \overset{\text{def}}{=} \{s \in S \mid s(\ell) = n_0\}$$

$$R \overset{\text{def}}{=} \{(s, s') \in S \times S \mid s(\ell) \xrightarrow{\gamma,u} s'(\ell),\ (\mathcal{I}, \{db \mapsto s(db)\}) \models \gamma[db], \text{ and } s'(db) = (\mathcal{I}, \{db \mapsto s(db)\})(u[db])$$


Init
Pack
Declined
Stocktake
Packed
Invoice
guard = "¬*acceptable*(*db*)"
Paid
Shipped
entry = "*true*"
exit = "*true*"
guard = "*db*.paid ≠ *true*"
script = "*db*..paid = *true*"

Definitions:
completed: $\forall db{:}\texttt{DB}\ .\ (\mathit{completed}(db) \Leftrightarrow (db.\texttt{paid} = \mathit{true} \wedge db.\texttt{shipped} = \mathit{true}))$
acceptable: $\forall db{:}\texttt{DB}\ .\ (\mathit{acceptable}(db) \Leftrightarrow db.\texttt{open} \neq [|\,|])$
Constraints:
nongold: $(db.\texttt{gold} = \mathit{false} \Rightarrow (db.\texttt{shipped} = \mathit{false}\ \mathsf{W}\ db.\texttt{paid} = \mathit{true}))$

Fig. 1. Model of a purchase order system as process fragments and definitions.

Notice the transition relation R depends on the interpretation $\mathcal{I}$, which is fixed at the outset. If a guard evaluates to false under $\mathcal{I}$ and the current state, then the edge it is on is just "not there". Otherwise the state must be updated as specified by the update term.

We can now explain the dynamics of our running example, a system for handling purchase orders. The purpose of the modelled system is to accept incoming purchase orders and process them further (packing, shipping, etc.), or to decline them straight away if there are problems. The dynamics of the model is depicted as a graph in Fig. 1. It is comprised of three *(process) fragments*: the biggest fragment on the left, and beside it the two one-node fragments labelled "Paid" and "Shipped".

The depicted model's initial node ("Init") is where it waits for a purchase order to arrive. Subsequently, the system can either start to pack (*i.e.*, enter node "Pack"), or decline the order (*i.e.*, enter node "Declined"). An order can be declined if the depicted guard ($\neg \mathit{acceptable}(db)$) in the annotation of the edge is satisfied. The predicate *acceptable* is defined in the `DEFINITIONS` section of our input specification.

If the order is not declined, an attempt will be made to pack its constituents. As long as the `open` list is not empty, the loop between "Pack" and "Stocktake" packs all items one after the other. Not all guards and scripts are depicted in Figure 1. For example, there is an edge from "Stocktake" to "Pack" labelled with

guard = "$db.\texttt{stock}[\mathit{head}(db.\texttt{open})].\texttt{available} > 0$"
script = "$db.\texttt{stock}[\mathit{head}(db.\texttt{open})].\texttt{available} =$
$db.\texttt{stock}[\mathit{head}(db.\texttt{open})].\texttt{available} - 1;\ db.\texttt{open} = \mathit{tail}(db.\texttt{open})$"

Upon completion, the "Invoice" state is reached, followed by composition with the fragments "Paid" and "Shipped". The "Shipped" fragment has a guard and script analogously to that of "Paid". The `guards` make it impossible to compose these fragments repeatedly, otherwise their composition is subject only to the "nongold" constraint. The intended final states are those that satisfy the "completed" predicate.

Reasoning Problems. Assume as given a specification. Let Σ be the induced signature with sorts *sorts*. Let *defs*, *constraints* and $M = (S, I, R)$ be as defined above. In terms of our specification language we are interested in the following reasoning problems. In each of them, let $\psi[db]$ be a path formula, in this context called the *query*.

Concrete satisfiability problem: Given an initial state $s_0 \in I$.
Is there a Σ-interpretation $\mathcal{I}$ such that $(\mathcal{I}, s_0) \models \mathsf{E}\,(defs \land constraints \land \psi)$ holds?
General satisfiability problem: Is there an initial state $s_0 \in I$ and a Σ-interpretation $\mathcal{I}$ such that $(\mathcal{I}, s_0) \models \mathsf{E}\,(defs \land constraints \land \psi)$ holds?

That is, the concrete *vs.* general dimension distinguishes whether an initial assignment is fixed or not. The concrete problems are interesting for implementing deployed systems and runtime verification. For, if the definitions and constraints are sufficiently restricted (*e.g.*, non-recursive definitions and constraints whose quantifiers range over finite domains) all state transitions can be effectively executed. See Section 6 for examples of reasoning problems.

4 Tableaux for CTL*(FO)

In this section we introduce a tableau calculus for the reasoning problems in Section 3. Without loss of generality it suffices to consider the general satisfiability problem only. (Pragmatics aside, any concrete satisfiability problem can be encoded as a general one as a set of equations in the `CONSTRAINTS` section). With the abbreviation $\psi_0 = defs \land constraints \land \psi$ the reasoning problem, hence, is to ask whether $(\mathcal{I}, s_0) \models \mathsf{E}\,\psi_0$ holds for some $s_0 \in I$ and Σ-interpretation $\mathcal{I}$. In fact, Ψ_0 can be any path formula in the free variable *db*.

It comes in handy to assume the signature Σ contains a `DB`-sorted constant `db`, representing the initial database, and that Σ contains a distinguished sort "Node" and the nodes N from the process graph as constants. We assume $\mathcal{I}(n) = n$ for every $n \in N$.

We formulate the calculus' inference rules as operators on sets of sequents. A *sequent* is an expression of the form $(n, t, l) \vdash_Q \Phi$ where $n \in N$, t is a ground term of sort `DB`, $l \geq 0$ is an integer, $Q \in \{\mathsf{E}, \mathsf{A}\}$ is a path quantifier, and Φ is a (possibly empty) set of CTL*(FO) formulas in negation normal form with free variables at most $\{db\}$. When we write $s \vdash_Q \Phi$ we mean $(n, t, l) \vdash_Q \Phi$ for some $(n, t, l) = s$, and when we write $s \vdash_Q \phi, \Phi$ we mean $s \vdash_Q \{\phi\} \cup \Phi$.

Informally, the sequent $(n, t, d) \vdash_Q \Phi$ means that the computation has reached after l steps ("length") into a run the graph node n with a database represented by t and that database satisfies $Q\,\Phi$. For example, t could be $\mathtt{db}\{\mathtt{open} = [|1, 3, 2|]\}$ (in sugared notation) which stands for an update of the initial database `db` updated on its `open`-field with the list $[|1, 3, 2|]$, and $Q\,\Phi$ could be the formula $\mathsf{A}\,\mathsf{G}\,db.\mathtt{open} \neq [|\,|]$. The calculus analyses a given sequent by decomposing its formula Φ according to its boolean operators, path quantifiers and temporal operators. An additional implicit boolean structure is given by reading the formulas Φ in $s \vdash_{\mathsf{E}} \Phi$ conjunctively, and reading the formulas Φ in $s \vdash_{\mathsf{A}} \Phi$ disjunctively.[2] The purpose is to derive a set of sequents with only classical, i.e., pure FO formulas in Φ, so that a first-order satisfiability check results.

[2] These structures are in general not decomposable, as A does not distribute over "or" and E does not distribute over "and", and so the calculus needs to deal with that explicitly.

We are using notions around tableau calculi in a standard way, and so it suffices to summarize the key points. The nodes in our tableaux are labelled with sets Σ of sequents or the special sign FAIL, which indicates branch closure. Logically, FAIL is taken as an (any) unsatisfiability set of sequents, e.g., $\{(n_0, \mathsf{db}, 0) \vdash_{\mathsf{A}} \emptyset\}$. We often write $\sigma; \Sigma$ instead of $\{\sigma\} \cup \Sigma$, and we often identify the node with its label. A *derivation* $\mathcal{D}$ *(from a path formula* ψ_0*)* is a sequence of tableaux, starting from a root node only tableau labelled with $\{(n_0, \mathsf{db}, 0) \vdash_{\mathsf{E}} \psi_0\}$. A successor tableaux is obtained by applying one of the inference rules below to a non-FAIL leaf of the current tableau and branching out with the conclusions. A *refutation (of* ψ_0*)* is a derivation from ψ_0 of a tableau whose leaves are all FAIL. We suppose a notion of *fair* derivations as commonly used with tableau calculi. Intuitively, a derivation is *fair* iff it is a refutation or no inference rule application is deferred forever.

In the inference rules below we use the following notions. A formula is *classical* iff it contains no path quantifier and no temporal operator. A formula is a *modal atom* iff its top-level operator is a path quantifier or a temporal operator. A sequent $s \vdash_Q \Phi$ is *classical* if all formulas in Φ are classical. We define $\mathrm{form}_{\mathsf{A}}(\Phi) \overset{\mathrm{def}}{=} \mathsf{A}\,(\bigvee \Phi)$ and $\mathrm{form}_{\mathsf{E}}(\Phi) \overset{\mathrm{def}}{=} \mathsf{E}\,(\bigwedge \Phi)$ in order to reflect the disjunctive/conjunctive reading of Φ depending on a path quantifier context. If all formulas in Φ are classical then the path quantifier is semantically irrelevant and omitted from $\mathrm{form}_Q(\Phi)$.

Boolean rules.

$$\mathsf{E}\text{-}\wedge\ \frac{s \vdash_{\mathsf{E}} \phi \wedge \psi, \Phi; \Sigma}{s \vdash_{\mathsf{E}} \phi, \psi, \Phi; \Sigma} \qquad \mathsf{E}\text{-}\vee\ \frac{s \vdash_{\mathsf{E}} \phi \vee \psi, \Phi; \Sigma}{s \vdash_{\mathsf{E}} \phi, \Phi; \Sigma \quad s \vdash_{\mathsf{E}} \psi, \Phi; \Sigma}$$

$$\mathsf{A}\text{-}\vee\ \frac{s \vdash_{\mathsf{A}} \phi \vee \psi, \Phi; \Sigma}{s \vdash_{\mathsf{A}} \phi, \psi, \Phi; \Sigma} \qquad \mathsf{A}\text{-}\wedge\ \frac{s \vdash_{\mathsf{A}} \phi \wedge \psi, \Phi; \Sigma}{s \vdash_{\mathsf{A}} \phi, \Phi; s \vdash_{\mathsf{A}} \psi, \Phi; \Sigma}$$

if ϕ is not classical or ψ is not classical (no need to break classical formulas apart).

Rules to separate classical sequents.

$$\mathsf{E}\text{-Split}\ \frac{s \vdash_{\mathsf{E}} \Phi; \Sigma}{s \vdash_{\mathsf{E}} \Gamma; s \vdash_{\mathsf{E}} \Phi \backslash \Gamma; \Sigma} \qquad \mathsf{A}\text{-Split}\ \frac{s \vdash_{\mathsf{A}} \Phi; \Sigma}{s \vdash_{\mathsf{A}} \Gamma; \Sigma \qquad s \vdash_{\mathsf{A}} \Phi \backslash \Gamma; \Sigma}$$

if Γ consists of all classical formulas in Φ and Γ is not empty.

Rules to eliminate path quantifiers.

$$\mathsf{E}\text{-Elim}\ \frac{s \vdash_{\mathsf{E}} Q\,\phi, \Phi; \Sigma}{s \vdash_Q \phi; s \vdash_{\mathsf{E}} \Phi; \Sigma} \qquad \mathsf{A}\text{-Elim}\ \frac{s \vdash_{\mathsf{A}} Q\,\phi, \Phi; \Sigma}{s \vdash_Q \phi; \Sigma \qquad s \vdash_{\mathsf{A}} \Phi; \Sigma}$$

The above rules apply also if Φ is empty. In this case Φ represents the empty conjunction in $s \vdash_{\mathsf{E}} \Phi$, a sequent that is satisfied by every $\mathcal{I}$, and the empty disjunction in $s \vdash_{\mathsf{A}} \Phi$, a sequent that is satisfied by no $\mathcal{I}$.

When applied exhaustively, the rules so far lead to sequents that all have the form $s \vdash_Q \Phi$ such that (a) Φ consists of classical formulas only, or (b) Φ consists of modal atoms only with top-level operators from $\{\mathsf{U}, \mathsf{R}, \mathsf{X}, \overline{\mathsf{X}}\}$.

Rules to expand U *and* R *formulas.*

$$\text{U-Exp}\ \frac{s \vdash_Q (\phi\,\mathsf{U}\,\psi), \Phi; \Sigma}{s \vdash_Q \psi \vee (\phi \wedge \mathsf{X}\,(\phi\,\mathsf{U}\,\psi)), \Phi; \Sigma} \qquad \text{R-Exp}\ \frac{s \vdash_Q (\phi\,\mathsf{R}\,\psi), \Phi; \Sigma}{s \vdash_Q (\psi \wedge (\phi \vee \overline{\mathsf{X}}\,(\phi\,\mathsf{R}\,\psi))), \Phi; \Sigma}$$

The above rules perform one-step expansions of modal atoms with U and R operators.

When applied exhaustively, the rules so far lead to sequents that all have the form $s \vdash_Q \Phi$ such that (a) Φ consists of classical formulas only, or Φ consists of modal atoms only with top-level operators from $\{\mathsf{X}, \overline{\mathsf{X}}\}$.

Rules to simplify X *and* $\overline{\mathsf{X}}$ *formulas.* Below we define inference rules for one-step expansions of sequents of the form $s \vdash_Q \mathsf{X}\,\phi$ and $\vdash_Q \overline{\mathsf{X}}\,\phi$. The following inference rules prepare their application.

$$\text{E-X-Simp}\ \frac{s \vdash_{\mathsf{E}} \mathsf{X}\,\phi_1, \ldots, \mathsf{X}\,\phi_n, \overline{\mathsf{X}}\,\psi_1, \ldots, \overline{\mathsf{X}}\,\psi_m; \Sigma}{s \vdash_{\mathsf{E}} Y\,(\phi_1 \wedge \cdots \wedge \phi_n \wedge \psi_1 \wedge \cdots \wedge \psi_m); \Sigma}$$

if $n+m > 1$, where $Y = \overline{\mathsf{X}}$ if $n = 0$ else $Y = \mathsf{X}$. Intuitively, if just one of the modal atoms in the premise is an X-formula then a successor state must exist to satisfy it, hence the X-formula in the conclusion. Similarly:

$$\text{A-X-Simp}\ \frac{s \vdash_{\mathsf{A}} \mathsf{X}\,\phi_1, \ldots, \mathsf{X}\,\phi_n, \overline{\mathsf{X}}\,\psi_1, \ldots, \overline{\mathsf{X}}\,\psi_m; \Sigma}{s \vdash_{\mathsf{A}} Y(\phi_1 \vee \cdots \vee \phi_n \vee \psi_1 \vee \cdots \vee \psi_m); \Sigma}$$

if $n + m > 1$, where $Y = \mathsf{X}$ if $m = 0$ else $Y = \overline{\mathsf{X}}$.

To summarize, with the rules so far, all sequents can be brought into one of the following forms: (a) $s \vdash_Q \Gamma$, where Γ consists of classical formulas only, (b) $s \vdash_Q \mathsf{X}\,\phi$, or (c) $s \vdash_Q \overline{\mathsf{X}}\,\phi$.

Rules to expand X *and* $\overline{\mathsf{X}}$ *formulas.*

$$\text{E-}\overline{\text{X}}\text{-Exp}\ \frac{(m, t, l) \vdash_{\mathsf{E}} \overline{\mathsf{X}}\,\phi; \Sigma}{(n_1, u_1[t], l+1) \vdash_{\mathsf{E}} \gamma_1[t] \wedge \phi; \Sigma \quad \cdots \quad (n_k, u_k[t], l+1) \vdash_{\mathsf{E}} \gamma_k[t] \wedge \phi; \Sigma \quad (m, t, l) \vdash_{\mathsf{E}} \neg\gamma_1[t] \wedge \cdots \wedge \neg\gamma_k[t]; \Sigma}$$

if there is a $k \geq 0$ such that $m \xrightarrow{\gamma_i, u_i} n_i$ are all labelled edges emerging from m, where $1 \leq i \leq k$. Notice that the case $k = 0$ is possible. In this case there is only one conclusion, which is equivalent to Σ.

This rule binds the variable *db* in the guards to the term t, which represents the current database. The variable *db* in $\overline{\mathsf{X}}\,\Phi$ refers to the databases in a later state and hence cannot be bound to t.

There is also a rule E-X-Exp whose premise sequent is made with the X operator instead of $\overline{\mathsf{X}}$. It differs from the E-$\overline{\text{X}}$-Exp rule only by leaving away the rightmost conclusion. We do not display it here for space reasons. Dually,

A-X-Exp
$$\frac{(m,t,l) \vdash_{\mathsf{A}} \mathsf{X}\,\phi; \Sigma}{(n_1, u_1[t], l+1) \vdash_{\mathsf{A}} \neg\gamma_1[t] \vee \phi; \cdots (n_k, u_k[t], l+1) \vdash_{\mathsf{A}} \neg\gamma_k[t] \vee \phi; (m,t,l) \vdash_{\mathsf{E}} \gamma_1[t] \vee \cdots \vee \gamma_k[t]; \Sigma}$$

if there is a $k \geq 0$ such that $m \xrightarrow{\gamma_i, u_i} n_i$ are all labelled edges emerging from m, where $1 \leq i \leq k$.

The conclusion sequent $(m,t,l) \vdash_{\mathsf{E}} \gamma_1[t] \vee \cdots \vee \gamma_k[t]$ forces that at least one guard is true. Analogously to above, there is also a rule A-$\overline{\mathsf{X}}$-Exp for the $\overline{\mathsf{X}}$ case, which does not include this sequent. This reflects that $\overline{\mathsf{X}}$ formulas are true in states without successor.

These rules are the only one that increase the length counter l.

Rule to close branches.

$$\text{Close}\ \frac{(m_1,t_1,l_1) \vdash_{Q_1} \Phi_1; \cdots; (m_n,t_n,l_n) \vdash_{Q_n} \Phi_n}{\text{FAIL}}$$

if every formula in every Φ_i is classical and $F = \bigwedge_{i=1,\ldots,n} \text{form}_{Q_i}(\Phi_i[t_i])$ is unsatisfiable (not satisfied by any interpretation $\mathcal{I}$).

Notice that F is a classical formula, It is meant to be passed to a first-order theorem prover for checking unsatisfiability.

Let us now turn to analyzing the calculus' theoretical properties. To this end, we equip sequents with a formal semantics within the temporal logic framework in Section 2. In that framework, a state is a mapping from variables to domain elements. In our case the (relevant) variables are fixed, which are the Node-sorted variable ℓ and the DB-sorted variable db. Given an interpretation $\mathcal{I}$, we associate to the triple (n,t,l) the state $\text{state}_{\mathcal{I}}(n,t,l) \stackrel{\text{def}}{=} \{\ell \mapsto n,\ db \mapsto I(t)\}$ (the length l has no relevant meaning for that).

Definition 4.1 (Tableau node semantics). *Let $\mathcal{I}$ be an interpretation. We say that $\mathcal{I}$* satisfies *a sequent $s \vdash_Q \Phi$, written as $\mathcal{I} \models s \vdash_Q \Phi$, iff $(\mathcal{I}, \text{state}_{\mathcal{I}}(s)) \models \text{form}_Q(\Phi)$. We say that $\mathcal{I}$* satisfies *a set Σ of sequents, written as $\mathcal{I} \models \Sigma$, iff $\mathcal{I}$ satisfies every sequent in Σ.*

The following lemma expresses the correctness of our inference rules.[3]

Lemma 4.2. *Let $\mathcal{I}$ be an interpretation and Σ a set of sequents. For every tableau rule inference with premise Σ and conclusions $\Sigma_1, \ldots, \Sigma_n$ it holds that $\mathcal{I} \models \Sigma$ if and only if $\mathcal{I} \models \Sigma_j$, for some $1 \leq j \leq n$.*

Theorem 4.3 (Soundness). *Given a state transition system $M = (S, I, R)$ as described in Section 3 and a path formula $\Psi_0[db]$. If there is a refutation of Ψ_0 then for no interpretation $\mathcal{I}$ and no $s_0 \in I$ it holds $(\mathcal{I}, s_0) \models \mathsf{E}\,\Psi_0$.*

We are now turning to completeness. In its simplest form, the completeness statement reads as "if for no interpretation $\mathcal{I}$ and no $s_0 \in I$ it holds $(\mathcal{I}, s_0) \models \mathsf{E}\,\Psi_0$ then there is a refutation". For efficiency in practice, one should exploit confluence properties of

[3] Proofs are in the long version of this paper, see http://www.nicta.com.au/pub?id=6988

the inference rules and work with fair derivations instead. To this end, we demand that the inference rules are applied in the order given above, with decreasing priority. (In the bounded setting described below this is indeed a fair strategy.) Additionally, we would like to get a stronger model-completeness result saying that a non-refutation leads not only to a model of Ψ_0 but also delivers the corresponding run.

However, the infinite-state model checking problems we are dealing with make any general completeness result impossible. Our pragmatic solution is to use a form of bounded model checking by limiting runs to user-given length, as follows.

Let $l_{\max} \geq 0$ be an integer, the *length bound*. We define *bounded versions* of the rules to expand X and $\overline{\mathsf{X}}$ formulas by taking $k = 0$ whenever $l = l_{\max}$, otherwise the rule is applied as stated. That is, after $l_{\max}$ expansions of X or $\overline{\mathsf{X}}$ formulas the bounded versions of the inference rules pretend that the underlying run (of length $l_{\max}$) has stopped. The *bounded version* of the tableau calculus uses that bounded rules.

We need to reflect the bounded version of the calculus at the semantics level. Given a state transition system $M = (S, I, R)$ and $l_{\max} \geq 0$, let the admissible runs of M consist of all runs r from each $s_0 \in I$ such that $|r| \leq l_{\max}$ and if $|r| < l_{\max}$ then r is finished. We qualify the resulting satisfaction relation of Section 2 by "wrt. runs of length $l_{\max}$".

Theorem 4.4 (Bounded tableau calculus completeness). *Given a state transition system $M = (S, I, R)$ as described in Section 3, $l_{\max} \geq 0$ a length bound, and a path formula $\Psi_0[db]$. Let $\mathcal{D}$ be a fair derivation from Ψ_0 in the bounded version of the calculus.*

Then, D is finite, every non-FAIL leaf Σ consists of classical sequents only, and for every model $\mathcal{I}$ of Σ it holds $(\mathcal{I}, s_0) \models \mathsf{E}\,\Psi_0$ wrt. runs of length $l_{\max}$.

Conversely, for every interpretation $\mathcal{I}$ such that $(\mathcal{I}, s_0) \models \mathsf{E}\,\Psi_0$ wrt. runs of length $l_{\max}$ there is a non-FAIL leaf Σ such that $\mathcal{I}$ satisfies Σ (model completeness).

Thanks to the tableau calculus maintaining the history of expanding formulas, it is easy to extract from the branches leading to the leaves Σ the corresponding runs. Moreover, the formula Σ represents the weakest condition on $\mathcal{I}$ and this way provides more valuable feedback than, say, a fully specified concrete database.

But notice that in order to exploit Theorem 4.4 in practice, one has to establish satisfiability of the non-FAIL leaf node Σ. In general this is impossible, and the first-order proof problems we are dealing with are highly undecidable (Π^1_1-complete [19]), as the `DEFINITIONS` section may contain arbitrary FO sentences over integer arithmetics with free function symbols [11].

5 Inductive Proofs of Safety Properties

Verifying a safety property $\mathsf{A}\,\mathsf{G}\,\phi$ of a state transition system M (under given *constraints*) is especially problematic when using bounded model checking. The complexity of model checking will in most cases be prohibitive in case ϕ is in fact invariant and the failure to find a counterexample trace of a given length does not entail invariance.

A relatively simple method for verifying safety properties that has been shown to often work well in practice in the context of SAT and SMT based model checking is the *k-induction principle* [22, 15, 13]. It attempts to prove an invariant ϕ by iteratively increasing a parameter $k \geq 1$, the maximal length of considered traces, until a counterexample trace for the base case is found, k-induction succeeds, or some pre-determined bound for k is reached. In our setting the the k-induction principle reads as follows:

Base Case: There does *not* exist a Σ interpretation $\mathcal{I}$ and an assignment α_0 with $\alpha_0(l) = n_0$ such that $(\mathcal{I}, \alpha_0) \models \mathsf{E}((\mathit{constraints} \wedge \mathit{defs}) \wedge \neg(\phi \wedge \overline{\mathsf{X}}\phi \wedge ... \wedge \overline{\mathsf{X}}^{k-1}\phi))$.
Induction Step: There does *not* exist a Σ interpretation $\mathcal{I}$ and an assignment α_0 with $\alpha_0(l) = n_i$ for some $n_i \in N$ such that $(\mathcal{I}, \alpha_0) \models \mathsf{E}\ (\mathit{defs} \wedge \neg((\phi \wedge \mathsf{X}\phi \wedge ... \wedge \mathsf{X}^{k-1}\phi) \rightarrow \mathsf{X}^{k-1}\overline{\mathsf{X}}\phi))$.

Constraints (e.g. of the form $\mathsf{A}\,\mathsf{G}\,\psi$) can be used for traces starting at the initial states, but in general not for the inductive step. An upper bound for k can sometimes be computed from the structure of ϕ and *constraints*. In general, k-induction based model checking is incomplete because a counterexample trace to the inductive step for some k may start at a state which is unreachable from an intitial state. While the objective of increasing k is precisely to avoid such “spurious” counterexamples, some properties are not k-inductive for any k. Strengthening the property to be verified [15, 13] is one means of attempting to avoid that problem. We have adapted the strengthening strategy presented in [15] to our framework, though more work is required to make this approach practical.

6 Implementation and Experiments

We have implemented the modelling language of Section 3, the tableau calculus of Section 4, and the k-induction scheme of Section 5 on top of it.[4] The implementation, in Scala, is in a prototypical stage and is intended as a testbed for rapidly trying out ideas. As the first-order logic theorem prover for the Close rule we coupled Microsoft’s SMT-solver Z3 [14]. Z3 accepts quantified formulas, which are treated by instantiation heuristics. Moreover, Z3 natively supports integers, arrays, and lists. For JSON record types we have to supply axioms explicitly. Non-recursive definitions are passed on as “functions” to Z3, recursive ones as “constraints”. The coupling of Z3 is currently rather inefficient, through a file interface using the SMT2 language.

The lack of further improvements currently limits our implementation to problems that do not require too much combinatorial search induced by a process’ dynamics. But, in fact, we are currently mostly interested in investigating the usefulness and limits of currently available first-order theorem proving technology in an expressive verification framework as ours (recall from Theorems 4.3 and 4.4 and the accompanying discussions how critically our approach depends on that).

A basic query is $\mathsf{F}\,\mathit{completed}(\mathit{db})$, which checks whether or not it is possible to fully execute an (acceptable) order into a completed state. Such “planning” queries are useful, *e.g.*, for flexible process configuration from fragments during runtime, but also for static analysis during design time. Our prover can be instructed to exhaust all branches under

[4] Our implementation supports concrete reasoning problems (Section 3) by evaluation of `scripts` with a Groovy interpreter and a tailored, model elimination based proof procedure for guards, but we do not discuss this here.

inference rule applications and extract all runs represented by non-FAIL leaves. With a length bound $l_{\max} = 8$ it returns the runs

$$\begin{array}{l}
\mathit{Init} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Invoice} \to \mathit{Shipped} \to \mathit{Paid} \\
\mathit{Init} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Invoice} \to \mathit{Shipped} \to \mathit{Paid} \\
\mathit{Init} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Invoice} \to \mathit{Paid} \to \mathit{Shipped} \\
\mathit{Init} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Stocktake} \to \mathit{Pack} \to \mathit{Invoice} \to \mathit{Paid} \to \mathit{Shipped}
\end{array}$$

which are exactly the expected ones. In total, 223 branches have been closed, with 912 inference rule applications, and Z3 was called 529. The total runtime is 30 seconds, the time spent in Z3 was negligible. A variation is the query $(\mathsf{F}\, \mathit{completed}(\mathit{db})) \land (\mathit{db}.\mathtt{shipped} = \mathit{true}\ \mathsf{R}\ \mathit{db}.\mathtt{paid} = \mathit{false})$ ("Is there a completed state that has shipment before payment?") which returns the first two of the above runs. We also experimented with unsatisfiable variants, e.g., by adding the CONSTRAINT $\mathit{db}.\mathtt{gold} = \mathit{false}$ to the latter query. All these queries can be answered in comparable or shorter time.

Let us now turn to safety properties. They typically occur during design time, and are clearly general (as opposed to concrete) problems. Here are some examples, stated in non-negated form:

$\mathsf{A}\,\mathsf{G}\,(\forall i{:}\mathtt{Integer}.((0 \leq i \land i < \mathit{db}.\mathtt{nrStockItems}) \Rightarrow \mathit{db}.\mathtt{stock}[i].\mathtt{available} \geq 0))$
("The number of available stock items is non-negative")

$\mathsf{A}\,\mathsf{G}\,((\mathit{db}.\mathtt{paid} = \mathit{true} \land \mathit{db}.\mathtt{shipped} = \mathit{false}) \Rightarrow \mathsf{F}\, \mathit{db}.\mathtt{shipped} = \mathit{true})$
("Orders that have been paid for but not yet shipped will be shipped eventually")

$\mathsf{A}\,\mathsf{G}\,((\mathit{db}.\mathtt{gold} = \mathit{false} \land \mathit{db}.\mathtt{shipped} = \mathit{true}) \Rightarrow \mathit{db}.\mathtt{paid} = \mathit{true})$
(Follows from non-gold CONSTRAINT)

$\mathsf{A}\,\mathsf{G}\ \mathtt{inRange}(\mathit{db}.\mathtt{open}, \mathit{db}.\mathtt{nrStockItems})$
where inRange is defined as $\forall l{:}\mathtt{List[Integer]}.\forall n{:}\mathtt{Integer}.(\mathtt{inRange}(l, n) \Leftrightarrow$
$(l = \mathtt{[|\ \ |]} \lor (0 \leq \mathtt{head}(l) \land \mathtt{head}(l) < n \land \mathtt{inRange}(\mathtt{tail}(l), n))))$
("The open list contains valid item numbers only, in the range $0 \ldots \mathit{db}.\mathtt{nrStockItems}$")

The first property requires the additional CONSTRAINT $\mathit{db}.\mathtt{nrStockItems} \geq 0 \land (\forall i{:}\mathtt{Integer}.((0 \leq i \land i < \mathit{db}.\mathtt{nrStockItems}) \Rightarrow \mathit{db}.\mathtt{stock}[i].\mathtt{available} \geq 0))$, which asserts that that property holds true initially. The proof of that is found with $k = 1$. The second property needs $k = 3$, both proven within seconds.

The third property is problematic. Although valid, it cannot be proven by k-induction because it admits spurious counterexamples due to ignoring *constraints* in the induction step. The fourth property is again valid after adding $\mathit{db}.\mathtt{nrStockItems} \geq 0 \land \mathtt{inRange}(\mathit{db}.\mathtt{open}, \mathit{db}.\mathtt{nrStockItems})$ to CONSTRAINTS, which asserts the property holds initially. It is provable by k-induction for $k = 2$. There is a caveat, though: as said, our prover tries $k = 1, 2, \ldots$ in search for a proof by k-induction. Here, for $k = 1$ an unprovable (satisfiable) proof obligation in the induction step turned up on which Z3 did not terminate. Z3, like other SMT-solvers, does not reliable detect countersatisfiability for non-quantifier free problems. This is a general problem and can be expected to show up as soon as datatypes with certain properties (like inRange) are present. Our workaround for now is to use time limits and pretend countersatisfiability in case of inconclusive results. Notice that this preserves the soundness of our k-induction procedure.

7 Conclusions and Future Work

In this paper we proposed an expressive modelling framework based on first-order logic over background theories (arithmetics, lists, records, etc) and state transition systems over corresponding interpretations. The framework is meant to smoothly support a wide range of practical applications, in particular those that require rich data structures and declarative process modelling by fragments and constraints governing their composition. On the reasoning side, we introduced a tableau calculus for bounded model checking of properties expressed in a certain fragment of CTL* over that first-order logic. To our knowledge, the tableau calculus as such and our soundness and completeness results are novel. First experiments with our implementation suggests that bounded model checking is already quite useful in the business domain we considered, in particular in combination with k-induction.

From another point of view, this paper is meant as an initial exploration into using general first-order logic theorem provers as back-ends for dynamic system verification. Developing such systems that natively support quantified formulas over built-in theories has been become an active area of research. Improvements here directly carry over to a stronger system on our side. For instance, we plan to integrate the prover described in [3].

We also plan to work on some conceptual improvements. Among them are blocking mechanisms to detect recurring nodes, partial-order reduction to break symmetries among fragment compositions, and cone of influence reduction. Each of these reduces, ultimately, to first-order logic proof problems, which again emphasizes the role of first-order logic theorem proving in our context.

References

1. F. Baader, H. Liu, and A. ul Mehdi. Verifying properties of infinite sequences of description logic actions. In *ECAI*, 2010, pp. 53–58.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Logic and Computation*, 20(3):651–674, 2010.
3. P. Baumgartner and U. Waldmann. Hierarchic superposition with weak abstraction. In M. P. Bonacina, ed., *CADE-24*, 2013, LNAI. Springer.
4. M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi. SMT-based verification of LTL specification with integer constraints and its application to runtime checking of service substitutability. In J. L. Fiadeiro, S. Gnesi, and A. Maggiolo-Schettini, eds., *SEFM*, 2010, pp. 244–254. IEEE Computer Society.
5. L. Chang, Z. Shi, T. Gu, and L. Zhao. A family of dynamic description logics for representing and reasoning about actions. *J. Autom. Reasoning*, 49(1):1–52, 2012.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
7. D. Crockford. RFC 4627—The application/json media type for JavaScript Object Notation (JSON). Technical report, IETF, 2006.
8. E. Damaggio, A. Deutsch, R. Hull, and V. Vianu. Automatic verification of data-centric business processes. In S. Rinderle-Ma, F. Toumani, and K. Wolf, eds., *BPM*, 2011, *LNCS 6896*, pp. 3–16. Springer.

9. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Combination methods for satisfiability and model-checking of infinite-state systems. In *CADE-21*, Berlin, Heidelberg, 2007, LNAI, pp. 362–378. Springer-Verlag.
10. R. Goré. Chapter 6: Tableau methods for modal and temporal logics. In M D'Agostino, D Gabbay, R Hähnle, J Posegga, ed., *Handbook of Tableau Methods*, pp. 297–396. Kluwer Academic Publishers, 1999.
11. J. Halpern. Presburger Arithmetic With Unary Predicates is Π^1_1-Complete. *Journal of Symbolic Logic*, 56(2):637–642, 1991.
12. B. B. Hariri, D. Calvanese, G. D. Giacomo, R. D. Masellis, P. Felli, and M. Montali. Verification of description logic knowledge and action bases. In L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, eds., *ECAI*, 2012, *Frontiers in Artificial Intelligence and Applications*, vol. 242, pp. 103–108. IOS Press.
13. T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. In J. Barnat and K. Heljanko, eds., *PDMC*, 2011, *EPTCS*, vol. 72, pp. 55–62.
14. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, eds., *TACAS*, 2008, *LNCS 4963*, pp. 337–340. Springer.
15. L. M. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. H. Jr. and F. Somenzi, eds., *CAV*, 2003, *LNCS 2725*, pp. 14–26. Springer.
16. A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
17. M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In J. Eder and S. Dustdar, eds., *Business Process Management Workshops*, 2006, *LNCS 4103*, pp. 169–180. Springer.
18. M. Reynolds. A tableau for CTL*. In A. Cavalcanti and D. Dams, eds., *FM*, 2009, *LNCS 5850*, pp. 403–418. Springer.
19. H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. The MIT Press, Cambridge, Massachusetts, 1987.
20. T. Schuele and K. Schneider. Global vs. local model checking: A comparison of verification techniques for infinite state systems. In *SEFM*, Washington, DC, USA, 2004, pp. 67–76. IEEE Computer Society.
21. G. S. S. Schulz, K. Claessen, and P. Baumgartner. The TPTP typed first-order form with arithmetic. In N. Bjoerner and A. Voronkov, eds., *LPAR-18*, 2012, *LNAI 7180*. Springer.
22. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. H. Jr. and S. D. Johnson, eds., *FMCAD*, 2000, *LNCS 1954*, pp. 108–125. Springer.
23. V. Vianu. Automatic verification of database-driven systems: a new frontier. In R. Fagin, ed., *ICDT*, 2009, *ACM International Conference Proceeding Series*, vol. 361, pp. 1–13. ACM.