Electronic Communications of the EASST
Volume 46 (2011)

Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)

Integrated Model Checking of Static Structure and Dynamic Behavior using Temporal Description Logics

Franz Weitl and Shin Nakajima

16 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Integrated Model Checking of Static Structure and Dynamic Behavior using Temporal Description Logics

Franz Weitl and Shin Nakajima

National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan

Abstract: This paper presents a new notation for the formal representation of the static structure and dynamic behavior of software, based on description logics and temporal logics. The static structure as described by UML class diagrams is represented formally by description logics while the dynamic behavior is represented by linear temporal logic and state transition systems. We integrate these descriptions of static and dynamic aspects into a single formalism called LTL_{DL}. LTL_{DL} enables a concise and natural yet precise definition of the behavior of software w.r.t. UML class diagrams and state transition diagrams. We demonstrate our approach on the sake warehouse problem. Further, we describe how properties of finite LTL_{DL} models can be analyzed based on bounded model checking and SMT (satisfiability modulo theory) solving. We implemented a restricted SMT solver for finite sets and relations. This SMT solver helped to reduce the model checking runtime significantly as compared to bounded model checking with existing tools.

Keywords: Bounded Model Checking, Temporal Description Logics, SMT

1 Introduction

UML class diagrams and state transition diagrams are widely adopted for modeling software. It is desirable to detect flaws in these models as early as possible prior to implementation. We propose a new integrated approach on representing and checking consistency criteria for system models consisting of class diagrams and state transition diagrams. We base our approach on description logic, temporal logic, bounded model checking, and satisfiability modulo theory (SMT) solving.

Description logics are expressive for representing the static structure of some application domain. Their expressiveness is closely related to UML class diagrams [BCG05]. Temporal logics are well-suited to describe the behavior of processes in a formal yet abstract way. We propose to combine these formalisms in a family of temporal description logics called LTL_{DL}, to be able to address both the static and dynamic aspects of modeled systems. This goes beyond existing approaches such as Alloy [Jac02] or Spin [Hol04] which focus either on the static structure or on the dynamic behavior of the modeled system.

For the formal verification of LTL_{DL} properties, we propose a new approach based on bounded model checking and SMT solving. In a first step, LTL_{DL} models and formulae are transformed for a certain bound k into a non-temporal SMT(DL) formula which is a Boolean formula over a restricted theory of finite sets and relations. We implemented a solver for this theory based on OpenSMT [Bru09]. Experimental results show a higher performance as compared to Boolean encodings of relational models and SAT solving.

The contributions of the paper are:

1. Definition of the family of temporal description logics LTL_{DL} as a generalization of $ALC-$LTL proposed in [BGL08].
2. Demonstration of the usefulness of LTL_{DL} for representing static and dynamic properties of software models w.r.t. UML class and state transition diagrams.
3. Approach on model checking LTL_{DL}, based on bounded model checking and SMT solving.

The rest of the paper is organized as follows: first, we introduce the sake warehouse problem as a demonstration case, and model its static structure and dynamic behavior. Next, we define LTL_{DL} and discuss its application to the sake warehouse scenario. In the sequel, we present our approach on bounded model checking LTL_{DL} using SMT solving. Finally, we compare our approach with existing work and conclude the paper.

2 Sake Warehouse Scenario

We demonstrate our approach using the sake (Japanese liquor) warehouse scenario which has been published in 1984 [Yam84] as a shared example case for comparing different modeling and programming methods. In Japan, it has been used extensively to evaluate modeling and analysis methods [NF97]. We summarize the scenario as follows: A sake shop has a warehouse in which containers are stored. A container contains bottles of one or more brands of sake. Customers place orders to the shop. Each order may include one or more brands of sake. If all ordered brands are on stock, the order is delivered immediately to the customer. Otherwise, the customer is notified and the order is put on a list of pending orders. Whenever new containers enter the warehouse, pending orders are checked and delivered in case of sufficient stock.

We use this scenario to illustrate the following steps of our approach:

1. Modeling the static structure in terms of a UML class diagram.
2. Modeling the dynamic behavior in terms of a state transition diagram.
3. Representing target properties w.r.t. the models of step 1) and 2).
4. Checking target properties, using SMT-based bounded model checking.

2.1 Sake Warehouse – Static Structure

Figure 1 depicts a UML model of the static structure of the sake shop scenario.

A *sake shop* keeps a *stock* and maintains a *list of pending orders* (Figure 1 top). The stock consists of a number of *containers* each of which may contain bottles of several *sake brands* (Figure 1 lhs). The sake shop receives new containers at regular intervals (Figure 1 lhs top).

The sake shop handles *orders* which are placed by *customers* (Figure 1 center). Each order contains one or more requested *sake brands* (Figure 1 lhs). During the order handling process, an order may become *delivered*, or *pending* if it cannot be delivered immediately because of insufficient stock (Figure 1 bottom). Pending orders are put on the *pending list* (i.e., list of

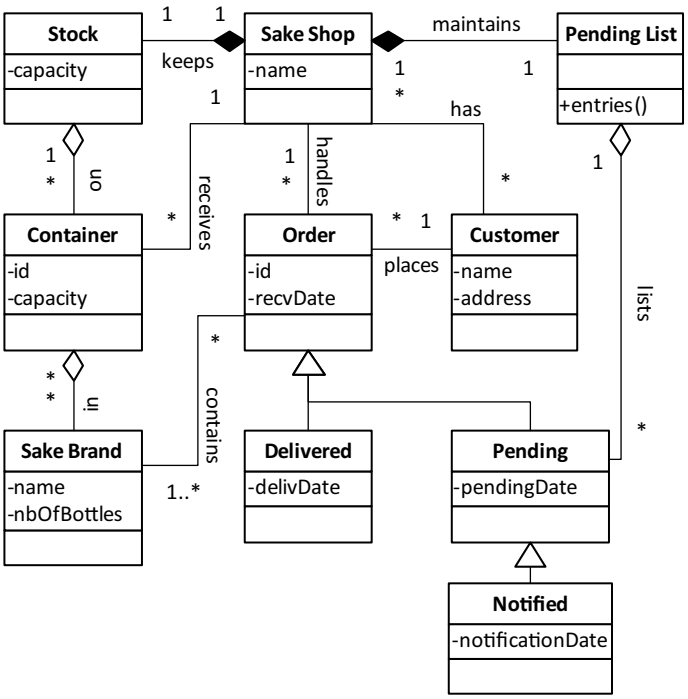

Figure 1: class diagram modeling the static structure of the sake shop.

pending orders) (Figure 1 rhs) and become *notified* (Figure 1 bottom) as soon as the shop keeper issues a notification about the delayed order to the customer.

2.2 Sake Warehouse – Behavior

Figure 2 models the basic behavior of the sake shopkeeper.

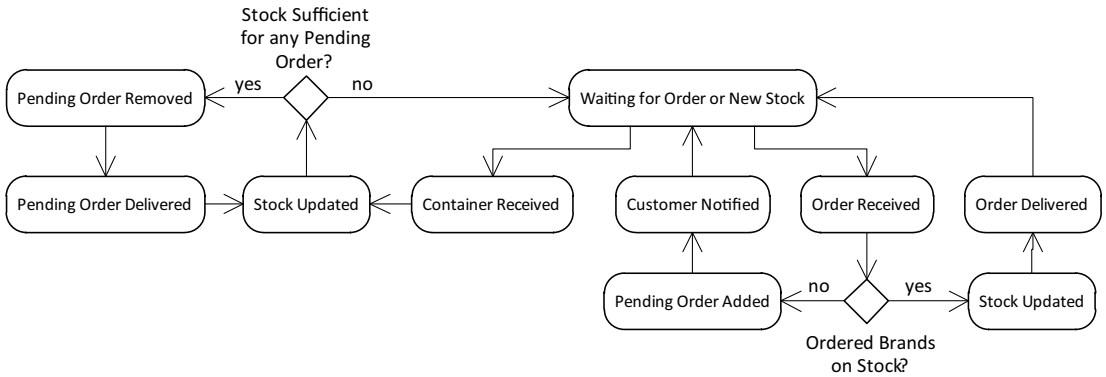

Figure 2: state transition diagram modeling the behavior of the sake shopkeeper.

Initially, the shopkeeper *waits for an order or new incoming stock* (Figure 2 rhs top). When an *order is received*, it is checked, whether all *ordered brands are on stock* (Figure 2 rhs bottom). If this is the case, the *stock is updated* and the *order is delivered* (Figure 2 rhs). Otherwise, the order is *added to the list of pending orders* and the *customer is notified* (Figure 2 center).

If the sake shop *receives a container*, it is put on the stock and the *stock is updated* (Figure 2 lhs center). Next, it is checked, if there are any pending orders and if the updated *stock is sufficient for delivering any of them* (Figure 2 lhs top). If this is the case, an appropriate order will be picked, *removed from the list of pending orders* and *delivered* (Figure 2 lhs). Further pending orders may be delivered as long as there is sufficient stock (Figure 2 lhs).

3 Sake Warehouse – Representation of Target Properties

We aim at representing properties w.r.t. both the static model and the behavior model of some application domain. In the case of our sample scenario, the following properties may be important to meet:

P1 Whenever a customer places an order, the customer will receive some response which may either be the delivery of the order or a notification that the order is pending because of insufficient stock (cf. [Nak08]).

P2 Orders may not be pending forever, i.e., orders delayed due to insufficient stock will be delivered eventually.

P3 If orders are pending then repeatedly incoming stock will eventually cause an order to be delivered.

P4 Pending orders will be handled with higher priority, i.e., a pending order of some brand X will be delivered before new orders of brand X (cf. [Nak08]).

We propose LTL_{DL} for the formal representation of such criteria. LTL_{DL} is a modular composition of linear temporal logic LTL [Eme90] and description logic (*DL*) [BCM+03]. This allows for the representation of properties that address both the static structure and dynamic behavior since the semantics of UML class diagrams can be represented well by *DL*, and properties of state transition diagrams can be expressed by LTL.

LTL_{DL} is similiar to $ALC-$LTL as introduced in [BGL08]. Section 5 contains a detailed comparison of LTL_{DL} with $ALC-$LTL and other temporal description logics.

Definition 1 (LTL_{DL} syntax)

Let P be a set of symbols representing atomic propositions and **DL** be the set of formulae of some decidable description logic *DL*. Let $a \in A \cup \mathbf{DL}$ be an atomic proposition or *DL* formula. Then LTL_{DL} *formulae* p, q are built according to the following rules:

$$p,q \rightarrow a \text{ (atomic prop. or } DL \text{ formula)} \mid \neg p \text{ (not)} \mid p \wedge q \text{ (and)} \mid p \vee q \text{ (or)} \mid p \rightarrow q \text{ (implies)} \mid$$
$$\mathsf{X}p \text{ (next)} \mid \mathsf{F}p \text{ (future/eventually)} \mid \mathsf{G}p \text{ (globally/always)} \mid p \,\mathsf{U}\, q \text{ (until)} \qquad \square$$

Remark 1 (LTL_{DL} syntax)
LTL_{DL} extends LTL by allowing *DL* formulae *in addition to* atomic propositions at locations where only atomic propositions are allowed in LTL. Hence both LTL and *DL* are contained in LTL_{DL}. □

Example 1 (LTL_{DL} syntax)
Consider the logic LTL_{ALC}, i.e., let *DL* in Definition 1 refer to the basic description logic *ALC* [BN03]. The following are *ALC* formulae expressing properties related to the class diagram of Figure 1.

$a_1 : Delivered \sqcup Pending \sqsubseteq Order$	Every *delivered* or *pending* thing is an *order*.
$a_2 : Order \doteq \exists contains.SakeBrand$	*Orders contain* at least one *sake brand*.
$a_3 : PendingList \sqsubseteq \forall lists.Pending$	Each *pending list contains pending* orders, only.
$a_4 : Order \sqcap \forall contains.\exists in.Container \sqsubseteq Delivered$	*Orders*, which *contain* sake brands, only, that are... ...available *in* some *container*, are *delivered*.

ALC formulae are build upon *atomic concepts* and *roles*, representing sets of objects and binary relations, respectively. In the formulae above, $Delivered, Pending, Order, SakeBrand, PendingList$, $Container$ are atomic concepts, and $contains, lists, in$ are roles, corresponding to classes and associations in the UML diagram of Figure 1. Since LTL_{ALC} subsumes *ALC*, formulae a_1 through a_4 are also LTL_{ALC} formulae. However, the following LTL_{ALC} formulae are neither in LTL nor in *ALC*.

$la_0 : \mathsf{F}(PendingList \sqsubseteq \neg\exists lists.Pending)$	The list of pending orders will eventually be empty.
$la_1 : \mathsf{G}(\neg(\exists places.Order \sqsubseteq \bot) \rightarrow \mathsf{F}(Order \sqsubseteq Delivered \sqcup Notified))$	Always if somebody places an order then... ...eventually any order will be delivered or notified.
$la_2 : \mathsf{GF}(Pending \sqsubseteq Delivered)$	Always, eventually pending orders are delivered.
$la_3 : \mathsf{G}(\neg(Pending \sqsubseteq \bot) \rightarrow (\mathsf{GF}(SakeShop \sqsubseteq \exists receives.Container) \rightarrow \mathsf{F}\neg(Delivered \sqsubseteq \bot)))$	Always, if there is some pending order then... ...if the sake shop receives some container infinitely ...often then eventually there will be a delivered order.
$la_4 : \mathsf{G}((Order \sqcap \exists contains.BrandX \sqcap \neg Pending \sqsubseteq \neg Delivered) \mathsf{U}(Pending \sqcap \forall contains.BrandX \sqsubseteq Delivered))$	Always, non-pending orders of brand X... ...will not be delivered... ...until all pending orders, which contain nothing... ...but BrandX, are delivered.

la_1 through la_4 are formal representations of properties **P1** through **P4** listed in the introduction of section 3. □

LTL_{DL} formulae are interpreted w.r.t. *finite relational state transition systems* $M = (S, R, L, \Delta, I)$ where S is a non-empty, finite set of states, $R \subseteq S \times S$ is a left-total transition relation, $L : S \rightarrow \mathscr{P}(A)$ is a labeling of states $s \in S$ with sets of atomic propositions $L(s) \subseteq A$ that hold at s, Δ is a finite set representing some domain of objects, and $I : S \rightarrow \{\cdot^{I(s)}\}$ is a state-dependent interpretation function such that $A^{I(s)} \subseteq \Delta$ and $R^{I(s)} \subseteq \Delta \times \Delta$ for each state $s \in S$, atomic concept A and atomic role R, respectively. We denote $I(s_0) \models d$ iff some DL formula d holds at a state $s_0 \in S$ according to the semantics of DL connectives and the interpretation $I(s_0)$ of atomic concepts and roles.

Definition 2 (LTL_{DL} semantics)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system and $x = (s_0, s_1, \ldots)$ an infinite path in M. Let d be a DL formula. Then d *holds* on a path x, in symbols $x \models d$, iff $I(s_0) \models d$. The semantics of all other cases (atomic proposition a, Boolean connectives $\neg, \wedge, \vee, \rightarrow$, and temporal connectives $\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$) is identical to the semantics of LTL [Eme90]. □

Example 2 (LTL_{DL} semantics)
Consider the formula $\mathsf{GF}(\mathit{Order} \sqsubseteq \mathit{Delivered})$, i.e., "always it holds eventually that any order is delivered". Consider the path $x = (s_0, s_1, s_2, s_0, s_1, s_2, s_0, \ldots)$ where

$$\begin{array}{ll} \mathit{Order}^{I(s_0)} = \{o1\} & \mathit{Delivered}^{I(s_0)} = \emptyset \\ \mathit{Order}^{I(s_1)} = \{o1, o2\} & \mathit{Delivered}^{I(s_1)} = \{o1\} \\ \mathit{Order}^{I(s_2)} = \{o1, o2\} & \mathit{Delivered}^{I(s_2)} = \{o1, o2\} \end{array}$$

i.e., there are two orders $o1$ and $o2$ which appear in state s_0 and s_1, respectively, and which will be delivered in state s_1 and s_2, respectively. Then $x \not\models \mathsf{G}(\mathit{Order} \sqsubseteq \mathit{Delivered})$ because, for instance, $\mathit{Order}^{I(s_0)} \not\subseteq \mathit{Delivered}^{I(s_0)}$. However, $x \models \mathsf{GF}(\mathit{Order} \sqsubseteq \mathit{Delivered})$ because in each state s_i of x eventually s_2 will be reached and $\mathit{Order}^{I(s_2)} \subseteq \mathit{Delivered}^{I(s_2)}$.

□

4 Model Checking LTL_{DL}

Definition 3 (LTL_{DL} model checking)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system, $s \in S$ a state, and f a LTL_{DL} formula. Then the LTL_{DL} model checking problem for M, s, and f, is to decide if $x \models f$ for all infinite paths $(s, s_1, s_2, \ldots)$ in (S, R) starting from s (denoted as $M, s \models f$). □

Theorem 1 (LTL reduction)
Let $M = (S, R, L, \Delta, I)$ be a finite relational state transition system and f be a LTL_{DL} formula. Let $D = \{d_1, \ldots, d_n\}$, $n \in \mathbb{N}$, be the set of *DL* formulae in f. Let $A = \{a_1, \ldots, a_n\}$ be a set of atomic propositions not appearing in f such that there is a bijection $d : A \leftrightarrow D : d(a_i) = d_i$. Let $f' = f[d_1/a_1][d_2/a_2]\ldots[d_n/a_n]$ be the formula derived from f by substituting all description logics formula in f with atomic propositions.

Let $M' = (S, R, L')$ be such a transition system that $L'(s) = L(s) \cup \{a \in A \mid I(s) \models d(a)\}$.

Then f' is a LTL formula and M' a LTL transition system and it holds for each $s \in S$: $M, x \models f$ for all paths x in M' starting from state s iff $M', x \models f'$ for all paths x in M starting from s.

Proof. This is a direct consequence of the syntax and semantics definition of LTL and LTL_{DL}. □

Remark 2 (LTL reduction)
By theorem 1, a model checking algorithm for LTL_{DL} can be constructed by composing a LTL model checker and *DL* model checker as follows: First, using the *DL* model checker to calculate the labeling function L' in Theorem 1, and then check for $M', x \models f'$ using the LTL model checker. This straight forward approach, however, is not efficient in the case of systems with

many states. Hence, we strive for a tighter interaction between the LTL and *DL* model checker to avoid calculating the entire reduced model M'. There are two obvious approaches to achieve this: "on-the-fly model checking" [Hol04] and "bounded model checking" [BCC+03]. In this work, we adopted the latter approach because its modular structure simplified the implementation of a LTL_{DL} model checker based on existing tools for bounded model checking and SMT solving (see section 4.3). However, on-the-fly model checking appears to be equally applicable in our setting and hence should be addressed in future work (see section 6). □

4.1 Bounded LTL_{DL} Model Checking

In bounded model checking [BCC+03], a transition system M, an initial state s and a LTL formula f is transformed for a given bound $k \in \mathbb{N}$ into such a non-temporal formula of the form $T_{M,s,k} \wedge \neg(f_k)$ that the following holds: if $T_{M,s,k} \wedge \neg(f_k)$ is satisfiable then there is a counterexample for $M,s \models f$ the length of which is less or equal to k and hence $M,s \not\models f$. We illustrate the approach of bounded model checking and its application to LTL_{DL} in the following example.

Example 3 (bounded LTL_{DL} model checking)

Consider the following scenario in an order handling process. Initially, there is no order. Next, a new order $o1$ is received and the reception of the order is notified to the customer. Next, another order $o2$ is received and the previously received order $o1$ is delivered. The following state transition system M models this scenario, adopting set type variables $order, notified, delivered$ for representing the set of orders, notified, and delivered orders, respectively:

state s_0	$order = notified = delivered = \emptyset$;	no orders, no deliveries, no notifications.
state s_1	$order \leftarrow order \cup \{o1\}$;	new order $o1$,
	$notified \leftarrow notified \cup \{o1\}$;	reception of $o1$ is notified to the customer.
state s_2	$order \leftarrow order \cup \{o2\}$;	new order $o2$,
	$delivered \leftarrow delivered \cup \{o1\}$;	$o1$ is delivered.
state s_3	$= s_0$	return to state s_0.

Let the *DL* concepts *Order*, *Delivered*, *Notified* represent the set of orders, deliveries, and notifications as used above. Consider the property "At any time, any order, which is not delivered, is notified":

$$f = \mathsf{G}(Order \sqcap \neg Delivered \sqsubseteq Notified)$$

We attempt to find a counterexample for f of a certain maximum length k in the state transition system M starting at s_0. As for the given scenario, a sensible bound is $k = 2$. First, we represent paths in M with maximum length k by a formula $T_{M,s_0,k}$ in which all variables are indexed by state (static single assignment form). For $k = 2$ we get:

$$\begin{aligned} T_{M,s_0,2} \;=\; & (order_0 = \emptyset) \wedge (notified_0 = \emptyset) \wedge (delivered_0 = \emptyset) \wedge \\ & (order_1 = order_0 \cup \{o1\}) \wedge (notified_1 = notified_0 \cup \{o1\}) \wedge (delivered_1 = delivered_0) \wedge \\ & (order_2 = order_1 \cup \{o2\}) \wedge (notified_2 = notified_1) \wedge (delivered_2 = delivered_1 \cup \{o1\}) \end{aligned}$$

Next, f is transformed into a non-temporal formula f_k equivalent to f in the scope k. In the given scenario, if f holds in M then $Order \sqcap \neg Delivered \sqsubseteq Notified$ holds in each state s_0, s_1, and s_2. Adopting the semantics definition of the *ALC* connectives $\sqcap$, $\neg$, and $\sqsubseteq$ we get:

$$\begin{aligned} f_2 \;=\; & (order_0 \backslash delivered_0 \subseteq notified_0) \wedge \\ & (order_1 \backslash delivered_1 \subseteq notified_1) \wedge \\ & (order_2 \backslash delivered_2 \subseteq notified_2) \end{aligned}$$

Finally, we check if $T_{M,s_0,2} \wedge \neg f_2$ is satisfiable. From $T_{M,s_0,2}$, we get:

$$\begin{aligned} order_2 &= order_1 \cup \{o2\} = order_0 \cup \{o1\} \cup \{o2\} &&= \{o1, o2\} \\ delivered_2 &= delivered_1 \cup \{o1\} = delivered_0 \cup \{o1\} &&= \{o1\} \\ notified_2 &= notified_1 = notified_0 \cup \{o1\} &&= \{o1\} \\ order_2 \backslash delivered_2 &= \{o2, o1\} \backslash \{o1\} &&= \{o2\} \end{aligned}$$

and thus $order_2 \backslash delivered_2 \not\subseteq notified_2$ which violates f_2. Hence $T_{M,s_0,2} \wedge \neg f_2$ is satisfied and we conclude $M, s_0 \not\models f$. □

4.2 SMT(DL)

As illustrated by Example 3, we transform LTL_{DL} models and formulae into formulae that contain set-type variables and operations corresponding to the semantics of *DL* connectives. These formulae can be interpreted as SMT formulae with sets and relations as background theory. We define the language SMT(DL) for the representation for such formulae. The concrete (i.e., machine processible) syntax of SMT(DL) is defined by the following rules:

formula	→	NOT formula | formula AND formula | formula OR formula | term
term	→	TRUE | FALSE | *boolvar* | set = set | rel = rel | subset(set,set)
set	→	EMPTYSET | *setvar* | insert(set,*int*) | remove(set,*int*) |
		union(set,set) | intersect(set,set) | minus(set,set) | some(rel,set) | all(rel,set)
rel	→	EMPTYREL | *relvar* | insertrel(rel,*int,int*) | removerel(rel,*int,int*)

Table 1: SMT(DL) syntax definition

The basic symbols are composed by the disjoint sets of Boolean variables *boolvar*, set variables *setvar*, variables for binary relations *relvar*, and integer numbers *int* serving as elements of sets and relations. Formulae are built using Boolean connectives NOT, AND, OR. Basic formulae are terms, which may be either Boolean atoms or set expressions corresponding to the *DL* connectives $\doteq$ and $\sqsubseteq$. Besides the constant "EMPTYSET", set variables *setvar* may be used to represent sets. Further, "insert(set,*int*)" represents a function inserting a single integer value into a set and "remove(set,*int*)" removes an element from a set. Line 4 of Table 1 defines set operators corresponding to the syntax of the *DL* expressions $C \sqcup D$, $C \sqcap D$, $\neg C$, $\exists R.C$, $\forall R.C$. Finally, binary relations may be manipulated by "insertrel", which inserts a pair of integer values into a relation, and "removerel", which removes a pair of integer values from a relation.

Example 4 (SMT(DL) concrete syntax)
Formula $T_{M,s_0,2}$ of Example 3 reads in SMT(DL) syntax as follows:

(order0 = EMPTYSET) AND (notified0 = EMPTYSET) AND (delivered0 = EMPTYSET) AND
(order1 = insert(order0, 1)) AND (notified1 = insert(notified0, 1)) AND (delivered1 = delivered0) AND
(order2 = insert(order1, 2)) AND (notified2 = notified1) AND (delivered2 = insert(delivered1,1))

Note that orders $o1$ and $o2$ in formula $T_{M,s_0,2}$ are represented by integer values 1 and 2, respectively. This is valid in general because we assume a finite interpretation domain (cf. Definition 2) which can be mapped onto integer numbers without loss of information.
Formula f_2 of Example 3 reads in SMT(DL) syntax as follows:

subset(minus(order0,delivered0),notified0) AND
subset(minus(order1,delivered1),notified1) AND
subset(minus(order2,delivered2),notified2)

□

4.3 Prototypical Implementation and Experimental Results

We implemented a partial solver for SMT(DL) based on OpenSMT [Bru09] which is an open source SMT solver implemented in C++. For the representation of SMT(DL) formulae, we use the standard format SMT-LIB 1.2. The current implementation is limited to SMT(DL) formulae, the set and relation expressions of which are bound to finite domains and do not contain cyclic definitions such as "s = insert(s,1)". The latter is not a restriction in our application because, in bounded model checking, LTL_{DL} models are transformed into static single assignment form (cf. Example 3) which do not contain any cyclic definitions by construction.

The aim of the subsequent two experiments is to determine the runtime of model checking LTL_{DL} as compared to existing bounded model checkers. The runtime of bounded model checking is dominated by checking the satisfiability of the generated formula $T_{M,s,k} \wedge \neg f_k$ (cf. Example 3).

4.3.1 Experiment 1

In the first experiment, we use a parameterized scenario similar to that in Example 3 to determine the scaling of runtime w.r.t. the input size:

state s_0	$order = notified = delivered = \emptyset$;
state s_1	$order \leftarrow order \cup \{o_1\}$; $notified \leftarrow notified \cup \{o_1\}$;
state s_2	$order \leftarrow order \cup \{o_2\}$; $delivered \leftarrow delivered \cup \{o_1, o_2\}$;
state s_3	$order \leftarrow order \cup \{o_3\}$; $notified \leftarrow notified \cup \{o_3\}$;
state s_4	$order \leftarrow order \cup \{o_4\}$; $delivered \leftarrow delivered \cup \{o_3, o_4\}$;
...	
state s_{2n-1}	$order \leftarrow order \cup \{o_{2n-1}\}$; $notified \leftarrow notified \cup \{o_{2n-1}\}$;
state s_{2n}	$order \leftarrow order \cup \{o_{2n}\}$; $delivered \leftarrow delivered \cup \{o_{2n-1}, o_{2n}\}$;
state s_{2n+1}	$order \leftarrow order \cup \{o_{2n+1}\}$;

As a property, we check, if each undelivered order is notified at any time (cf. Example 3):

$$f = \mathsf{G}(Order \sqcap \neg Delivered \sqsubseteq Notified)$$

The only state violating f is s_{2n+1}. To detect the error by bounded model checking, the bound k must be chosen greater or equal to $2n+1$, making the case increasingly challenging for larger n. Moreover, the maximum sizes of the sets for representing received, notified, and delivered orders grow linearly in n.

To compare the performance of our approach with existing ones, we chose the SAL tool [MOR+04] since it integrates a variety of state-of-the-art model checking algorithms, including SAT and SMT-based bounded model checking. SAL uses the SMT solver Yices 1.03 [DM06] as a backend engine for bounded model checking. The scenario above can be described compactly in terms of the SAL input language by representing the characteristic function $\mathbf{1}_S : S \to \{false, true\} : \{x \in S \mid \mathbf{1}_S(x) = true\} = S$ of each set S as a Boolean array (cf. [KRW09]). The bounded model checker of SAL translates an input file for a given bound k into a SAT or SMT formula which is then solved by Yices. For our experiment, we chose the transformation into SAT because this yielded higher performance. Alternatively to Yices, we also applied Z3 version 3.2 [MB08] and MiniSat 2.0 [ES04] as backend solvers.

An alternative SMT(DL)-based representation (cf. Example 4) for different problem sizes n and bounds k has been generated. Generally, we distinguish two cases. 1) $k = 2n+1$: in this case, the generated SAT and SMT(DL) formulae are satisfiable, i.e., the property violation is detected; 2) $k = 2n$: the generated SAT and SMT(DL) formulae are not satisfiable.

Figure 3 (top) shows the runtimes of Yices, Z3, MiniSat, and our SMT(DL) solver for the two cases $k = 2n+1$ and $k = 2n$ and increasing input sizes n, obtained on a desktop computer with and 6 GB RAM and Intel Core i7 processor at 3.8 GHz.

The runtime of **Yices** for $k = 2n$ is slightly lower than in the case of $n = 2n+1$ (Figure 3 top, lhs). Yices takes 17.5 seconds for $n = 100, k = 201$ and 16.2 seconds for $n = 100, k = 200$.

In the case of **Z3**, the runtimes for $k = 2n+1$ and $k = 2n$ are almost identical (Figure 3 top, center). In the given scenario, Z3 is slightly faster than Yices but slower than MiniSat. It takes 15.3 seconds for $n = 160, k = 321$ and 15.1 seconds for $n = 160, k = 320$.

In contrast, **MiniSat** performs significantly faster in the case of $k = 2n$ than in the case of $k = 2n+1$ (Figure 3 top, center). It takes 15.4 seconds for $n = 240, k = 481$ and 7.4 seconds for $n = 240, k = 480$.

The runtime of **SMT(DL)** is almost identical for both cases $k = 2n+1$ and $k = 2n$ (Figure 3 top, rhs). It takes 16.4 seconds for $n = 8000, k = 16001$ and 16.1 seconds for $n = 8000, k = 16000$). As compared to MiniSat, the SMT(DL) solver handles 30 times as large problems in about the same execution time.

4.3.2 Experiment 2

Figure 3 (bottom) shows the runtime of Yices, Z3, MiniSat, and our SMT(DL) solver for checking the formula

$$f' = \mathsf{G}(\neg(\exists places.Order \sqsubseteq \bot) \to \mathsf{XX}(Order \sqcap \neg Delivered \sqsubseteq Notified))$$

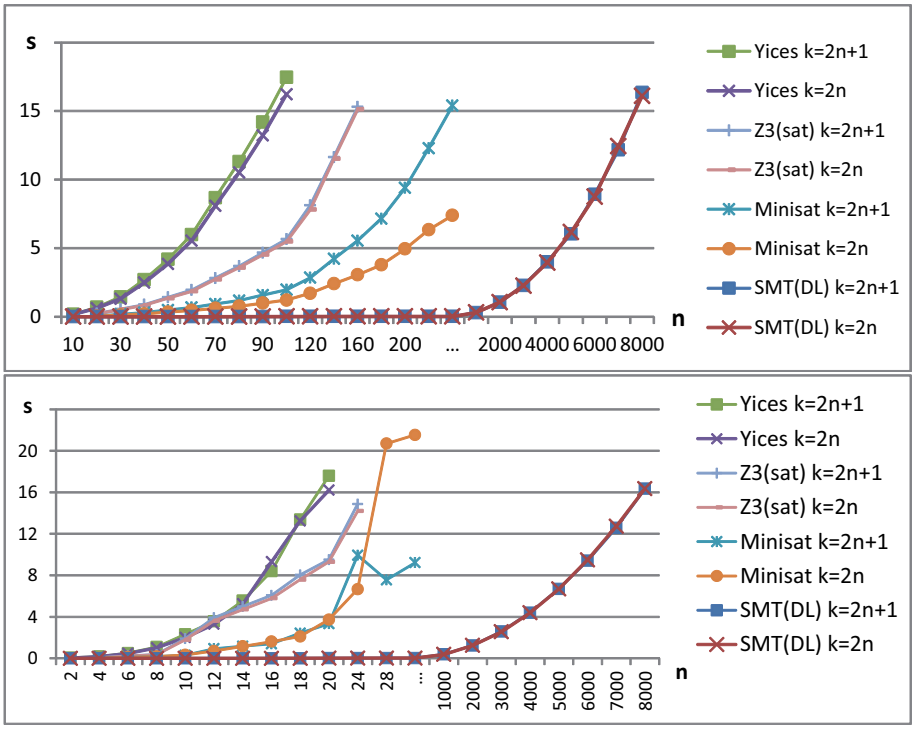

Figure 3: results of **Experiment 1** (top) and **Experiment 2** (bottom). Execution time of SMT(DL) solving as compared to SAT solving with Yices, Z3, and Minisat for different input sizes n.

in a LTL_{DL} model corresponding to the state transition diagram of Figure 2. f' reads: "Always (G), if someone places an order ($\neg(\exists places.Order \sqsubseteq \bot)$) then two states later (XX) each order that has not been delivered is notified ($Order \sqcap \neg Delivered \sqsubseteq Notified$)".

As in Experiment 1, Yices is slower than the more recent solvers Z3 and MiniSat. In this scenario, **MiniSat** takes 9.2 seconds for $n = 32$ if a counterexample is found, but 21.5 seconds if no counterexample is found (Figure 3 bottom, center). Surprisingly, the runtimes of Minisat for the cases $n = 28, k = 57$ and $n = 32, k = 65$ are lower than in the case of $n = 24, k = 49$. This may have been caused by the randomized search strategy applied by MiniSat.

In Experiment 2, the runtimes of Yices, Z3, and MiniSat are significantly lower than in Experiment 1. We suppose that the Boolean encoding of the binary relation *places* in formula f' is the major source of additional complexity. In contrast, the runtime of the SMT(DL) solver is hardly affected by the presence of a binary relation in Experiment 2 and remains almost identical to Experiment 1. In Experiment 2, the SMT(DL) solver handles at least 200 times as large problems than MiniSat in the same execution time.

The results of Experiment 1 and 2 indicate that supporting sets and relations in SMT solving can significantly speed up bounded model checking of relational models as compared to SAT-based bounded model checking.

5 Related Work

Description logics are well-known to be appropriate for the formal representation of conceptual data models such ER diagrams and UML class diagrams. For instance, [CLN98] proposes a unifying description logics for the logical representation of class-based data models such as ER and object-oriented data models. [BCG05] presents an encoding of UML class diagrams in the description logic *ALCQI* to discover inconsistencies in models by means of description logic reasoning. We extend these approaches by combining a description logic with a temporal logic to support the representation of properties related to both state transition diagrams and class diagrams.

In the past, several combinations of description logics and temporal logic have been suggested [AF01, LWZ08]. A first temporal extension of the description logic *ALC* called *ALCT* was suggested by Schild [Sch93]. In *ALCT*, the temporal connectives G, F, and U can be applied to concepts but not to axioms. A similar combination of LTL and *ALC* is called LTL_{ALC} in [LWZ08]. In contrast, *ALC*−LTL, as introduced in [BGL08], supports the application of temporal connectives to *ALC* axioms but not to *ALC* concepts.

LTL_{DL}, as proposed in this paper, follows the latter approach because, this way, a higher degree of modularity between the temporal and non-temporal part of the logic is achieved. This simplifies the formalization of properties in close correspondence with UML class diagrams (*DL* component) and state transition diagrams (LTL component), as well as the implementation of a model checker. However, LTL_{DL} is different from *ALC*−LTL in the following aspects:

- LTL_{DL} is a family of logics, obtained by a modular combination of some *DL* with LTL, rather than a single logic.
- While in *ALC*−LTL, atomic propositions are replaced by *ALC* axioms, LTL_{DL} supports *DL* formulae *in addition* to atomic propositions. This ensures compatibility with propositional LTL widely adopted in model checking.
- In contrast to *ALC*−LTL, we do not consider ABox assertions in LTL_{DL} since they seem to be dispensable for formalizing general domain models represented by UML class diagrams.
- As opposed to *ALC*−LTL, we do not consider *rigid* symbols, i.e., concepts and roles the interpretations of which do not depend on states. Incorporating rigid symbols to LTL_{DL} may be an interesting topic of future research.

[BGL08] focusses on the satisfiability problem of *ALC*−LTL and the impact of rigid symbols on the complexity of solving the satisfiability problem. In this paper, we do not consider the satisfiability problem but the model checking problem of LTL_{DL}. A thorough investigation of complexity properties will be an issue of future work.

[BBL09] proposes runtime verification based on *ALC*−LTL. In runtime verification, a monitor constantly observes the behavior of a system in execution and determines if the observed prefix of an execution trace conforms to a temporal formula. Each state of the execution trace is represented in a potentially incomplete way by a set of *ALC* ABox assertions (open world assumption). In our work, we adopt a model checking approach, i.e., all possible behaviors of

a system described by a state transition system are considered. However, the information about each single state is assumed to be complete (closed world assumption).

An algorithm for model checking the temporal description logics *ALC*CTL has been proposed in [Wei08]. In this paper, we consider the bounded model checking problem of LTL_{DL} and reduce it to SMT solving which we believe is a new approach that simplifies the integration of LTL_{DL} model checking into an existing model checking environment such as SAL and helps to increase the performance of model checking for bounded sets and relations.

State-of-the-art model checkers supporting linear temporal logic are Spin [Hol04], SAL [MOR+04], and NuSMV [CCG+02]. However, the input languages of these model checkers do not support set and relation data types and hence are inefficient for representing properties w.r.t. relational models.

Alloy is a declarative object-oriented modeling notation, the semantics of which is based on sets and relations [Jac02]. The notation supports the formulation of assertions. Dynamic aspects may be addressed in terms of pre- and post-conditions or by explicitly representing time as a linearly ordered set of states. However, temporal logic for the representation of behavioral properties is not supported. A tool based on SAT solving automatically analyzes whether assertions hold in models where the sizes of all sets and relations are bounded by some user chosen constants [JSS00]. In [GT11], an alternative approach is presented which is not limited to bounded sets: Alloy relational specifications are translated into first order quantified SMT formulae which are passed on to the SMT solver Z3 [MB08]. However, since the Alloy specification language is undecidable, the SMT solver may fail to prove assertions.

Event B [Abr10] is a formal specification language for the required behavior of a system, based on set theory and logic. A central concept is the refinement-based modeling for system requirements. Consistency and refinement checking of specifications, based on theorem proving, is supported by the Rodin tool [ABH+10] which generates and manages the necessary proofs. However, user interaction may be required for certain types of proofs. ProB [LB08], an animation and model checking tool for (Event) B specifications, supports model checking of properties expressed in LTL. Similar to Alloy, data types such as sets and relations must be restricted to small sizes for exhaustive analysis. LTL_{DL} is less expressive than the temporal logic supported by ProB but the supported constructs are closely related to UML class and state transition diagrams. We believe that this simplifies the identification and formalization of relevant consistency properties which is usually considered as a rather difficult task.

The syntax definition for SMT(DL) (Table 1) is inspired by [KRW09] which suggests a format for representing finite lists, sets, and maps as part of the SMT-Lib 2.0 format. As for solving formulae over finite sets, a mapping onto Boolean arrays is suggested. We have adopted this approach in our experiments with SAL and Yices (see section 4.3). To the best of our knowledge, none of the currently available SMT solvers implements dedicated decision procedures for sets and relations.

6 Conclusion

We have presented a new integrated approach on representing both static and dynamic aspects of software models. We defined LTL_{DL} as a modular composition of linear temporal logic LTL and

a description logic *DL*. LTL_{DL} supports representing properties w.r.t. both UML class diagrams and state transition diagrams. We believe that the close correspondence of LTL_{DL} formulae to these commonly used diagram notations facilitates the identification and formalization of important consistency requirements at an early development stage. Further, we have demonstrated how LTL_{DL} formulae can be checked by SMT-based bounded model checking. We have implemented a prototypical SMT solver for formulae containing set-type expressions corresponding to the semantics of LTL_{DL} connectives. As compared to reducing set-type expressions to Boolean arrays, 30 to 200 times as large problems could be solved in the same execution time.

In this paper, we discussed LTL_{DL} from an application-oriented perspective and demonstrated its usefulness and performance by a case study. Fundamental properties of LTL_{DL} such as expressiveness and runtime complexity of model checking and deciding satisfiability are left to be studied in future work.

In our current experiments, we use the input language of SAL for representing LTL_{DL} models, adopting a Boolean encoding for sets and relations. A more adequate representation language for LTL_{DL} models offering explicit support for sets and relations is a major issue of ongoing work. Ongoing is also the improvement of the implemented SMT solver in terms of supported types of formulae and performance. Issues are, for instance, the support of cyclic expressions and negation in unbounded domains (cf. section 4.3). To this end, a mapping of SMT(DL) formulae onto either first order quantified SMT formulae or description logic knowledge bases seems to be promising and calls for further examination. Finally, further case studies to compare our approach with existing approaches such as Event-B and Alloy are necessary. In addition, the comparison with existing state-of-the-art model checkers such as CBMC [CKL04] and Spin [Hol04], as well as SMT solvers, which support quantified formulae such as Z3 [MB08], is an important issue of future work. Finally, the applicability and efficiency of on-the-fly model checking as compared to bounded model checking needs to be addressed in future work.

Acknowledgements: This work is funded by the program "Research at International Science and Technology Centers" of the German Academic Exchange Service (DAAD). We thank the reviewers and workshop attendees for their detailed comments which helped to improve the paper significantly and gave directions for future work.

Bibliography

[ABH+10] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* 12(6):447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[AF01] A. Artale, E. Franconi. A Survey of Temporal Extensions of Description Logics. *Annals of Mathematics and Artificial Intelligence (AMAI)* 30(1-4):171–210, 2001.

[BBL09] F. Baader, A. Bauer, M. Lippmann. Runtime Verification Using a Temporal Description Logic. In Ghilardi and Sebastiani (eds.), *Frontiers of Combining Systems*. LNCS 5749, pp. 149–164. Springer-Verlag, 2009.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded Model Checking. In Zelkowitz (ed.), *Highly Dependable Software*. Advances in Computers 58, pp. 118–149. Academic Press, 2003.

[BCG05] D. Berardi, D. Calvanese, G. D. Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence* 168(1-2):70–118, 2005.

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (eds.). *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BGL08] F. Baader, S. Ghilardi, C. Lutz. LTL over description logic axioms. In *Proceedings of the 11th Inernational Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*. Pp. 684–694. Morgan Kaufmann, Sydney, Australia, 2008.

[BN03] F. Baader, W. Nutt. Basic description logics. In *[BCM+03]*. Chapter 2. 2003.

[Bru09] R. Bruttomesso. An Extension of the Davis-Putnam Procedure and its Application to Preprocessing in SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT2009)*. Montreal, Canada, 2009.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*. LNCS 2404. Springer, 2002.

[CKL04] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Jensen and Podelski (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. LNCS 2988, pp. 168–176. Springer-Verlag, 2004.

[CLN98] D. Calvanese, M. Lenzerini, D. Nardi. Logics for databases and information systems. In Chomicki and Saake (eds.). Chapter 8 Description logics for conceptual data modeling, pp. 229–263. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[DM06] B. Dutertre, L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification Conference (CAV'06)*. LNCS 4144, pp. 81–94. Springer-Verlag, 2006.

[Eme90] E. Emerson. Temporal and Modal Logic. In Leeuwen (ed.), *Handbook of Theoretical Computer Science: Formal Models and Semantics*. Pp. 996–1072. Elsevier, 1990.

[ES04] N. Een, N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. LNCS 2919, pp. 333–336. Springer-Verlag, 2004.

[GT11] A. A. E. Ghazi, M. Taghdiri. Relational Reasoning via SMT Solving. In *17th International Symposium on Formal Methods (FM)*. Limerick, Ireland, 2011.

[Hol04] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)* 11(2):256–290, 2002.

[JSS00] D. Jackson, I. Schechter, I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. Pp. 730–733. ACM Press, 2000.

[KRW09] D. Kröning, P. Rümmer, G. Weissenbacher. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard. Published on http://www.cprover.org/SMT-LIB-LSM/, 2009. Visited 9 Jan 2010.

[LB08] M. Leuschel, M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer* 10(2):185–203, 2008.

[LWZ08] C. Lutz, F. Wolter, M. Zakharyaschev. Temporal Description Logics: A Survey. In *Proceedings of the 15th International Symposium on Temporal Representation and Reasoning (TIME '08)*. Pp. 3–14. IEEE Computer Society, Washington, DC, USA, 2008.

[MB08] L. de Moura, N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. LNCS 4963, pp. 337–340. Springer-Verlag, 2008.

[MOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, A. Tiwari. SAL 2. Tool description presented at CAV 2004. LNCS 3114, pp. 496–500. Springer-Verlag, 2004.

[Nak08] S. Nakajima. *Model Checking with SPIN*. Chapter 9: Case Study(4). Kindaikagakusha, Tokyo, Japan, 2008.

[NF97] S. Nakajima, K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings of the 19th international conference on Software engineering (ICSE '97)*. Pp. 34–44. Boston, Massachusetts, United States, 1997.

[Sch93] K. Schild. Combining terminological logics with tense logic. In *Proceedings of the 6th Portuguese Conference on Artificial Intelligence*. Pp. 105–120. Porto, 1993.

[Wei08] F. Weitl. *Document Verification with Temporal Description Logics*. PhD thesis, University of Passau, 2008.

[Yam84] T. Yamasaki. Surveys of Program Design Methods Using a Common Example Problem. *Journal of IPS Japan* 25(9):934, 1984. In Japanese.