A Sequent Calculus for a First-order Dynamic Logic with Trace Modalities for Promela$^+$

Florian Rabe[1], Steffen Schlager[2], and Peter H. Schmitt[2]

[1] International University Bremen
[2] Universität Karlsruhe

Abstract. In this paper we introduce the first-order dynamic logic *DLP* for Promela$^+$, a language subsuming the modelling language Promela of the Spin model checker. In *DLP* trace modalities can be used to reason about the temporal properties of programs. The definition of *DLP* includes a formal semantics of the Promela$^+$ language. A sound and relatively complete sequent calculus is given, which allows deductive theorem proving for Promela$^+$. In contrast to the Spin model checker for Promela, this calculus allows to verify infinite state models. To demonstrate the usefulness of our approach we present two examples that cannot be handled with Spin but that can be derived in our calculus.

1 *DLP*—Dynamic Logic for Promela$^+$

Dynamic Logic (DL) [4, 5] is an extension of first-order predicate logic with modalities $[\pi]F$ and $\langle\pi\rangle F$ for each program π of some programming language and DL formula F. DL allows to reason about the input/output behaviour of a program. However, sometimes it is desirable to reason about intermediate states of a program as well. This becomes possible if DL is extended with additional, so-called trace modalities, as shown in [1].

The programming language we consider in this paper is Promela$^+$ whose syntax is essentially the same as of Promela [7], the modelling language of the model checker Spin [6]. Besides the usual constructs like assignments, loops, etc. Promela offers dynamic process creation, synchronous and asynchronous communication through channels, and non-deterministic choice.

In contrast to Promela, Promela$^+$ is not restricted to finite models. E.g., it is possible to create an unbounded number of processes, integer variables are not range restricted, and, most important, the initial state of a system may be (partially) unknown. For an informal syntax and semantics of Promela programs we refer to [7, 6]. A detailed formal syntax and semantics Promela$^+$ and therefore of Promela and can be found in [8].

2 Promela$^+$ and Limited Programs

In this section we give a very brief overview over the semantics of Promela$^+$ as described in [8]. We also introduce limited programs, which are an extension of [8].

Syntax. Promela$^+$ is based on a many-sorted first-order logic over a fixed signature containing all symbols to define syntax and semantics of Promela. It is extended with six temporal operators for each program π: $[\pi]$, $\langle\pi\rangle$, $[[\pi]]$, $[\langle\pi\rangle]$, $\langle[\pi]\rangle$ and $\langle\langle\pi\rangle\rangle$. Promela variables are modelled as function symbols and are different from the logical variables.

The normal programs, which we call unlimited, are essentially the Promela programs. In addition we introduce *limited programs* which are defined by the following clause: If t is a term of sort integer and π is an unlimited program, then π^t is a limited program. Limited or unlimited programs may occur inside temporal operators, e.g., for a formula F both $[\pi]F$ and $[\pi^t]F$ are well-formed formulas.

Semantics. The semantic domain of DL are extended Kripke structures. The states are many-sorted first-order structures such that

- all states share the same universe,
- some function and predicate symbols are *rigid*, i.e. have the same interpretation in all states (e.g. arithmetics and list operation), and
- the interpretation of *non-rigid* function symbols is depending on the state encoding the values of the Promela variables and channels.

For each elementary command there is one transition relation reflecting the semantics of the programming language. A trace of a program π for an initial state s is the (possibly infinite) sequence of states of a possible run of π starting in s. If a trace is finite its last state is labelled with *timeout* or *termination*. Promela$^+$ is non-deterministic, i.e., the semantics of a program is a set of traces for each initial state. Where applicable our semantics of Promela$^+$ agrees with the semantics of Promela used by Spin.

For a trace t and a natural number n let t^n denote the initial segment of length n of t; and for a set of traces T let T^n denote the set of all such segments of traces in T. For a given state s and an assignment α to the free logical variables let val_s^α denote the interpretation function. Then the semantics of the limited program π^t is given by:

$$\mathrm{val}_s^\alpha(\pi^t) = \begin{cases} \mathrm{val}_s^\alpha(\pi)^n & \text{if } \mathrm{val}_s^\alpha(t) = n, n > 0 \\ \emptyset & \text{if } \mathrm{val}_s^\alpha(t) \leq 0 \end{cases}$$

That means that for a limited program only the first $n-1$ computation steps are relevant.

The formula $[\pi]F$ ($\langle\pi\rangle F$) holds in state s iff *for all* (there *exists* a) possible end state(s) of π labelled with *termination* when started in s the formula F holds. The formula $[[\pi]]F$ ($[\langle\pi\rangle]F$) holds iff *for all* (there *exists* a) state(s) on *all* traces of π the formula F holds. And the formula $\langle[\pi]\rangle F$ ($\langle\langle\pi\rangle\rangle F$) holds iff there exists a trace of π on which *for all* (there *exists* a) state(s) F holds. If the program in the modal operator is a limited program the same semantics applies with respect to the traces in the interpretation of π^t. Formulas not containing modalities are interpreted as usual in first-order logic.

E.g., suppose $\mathrm{val}^{\alpha}_{s_1}(\pi) = \{(s_1, s_2, \mathit{termination}\ s_3),\ (s_1, s_2, s_3', \mathit{timeout}\ s_4')\}$ and F holds precisely in the states s_1, s_2, s_3 and s_3'. Then $\mathrm{val}^{\alpha}_{s_1}(\langle[\pi]\rangle F) = 1$ because F holds in all states of the first trace; $\mathrm{val}^{\alpha}_{s_1}([[\pi]]F) = 0$ because F does not hold in s_4'; $\mathrm{val}^{\alpha}_{s_1}([[\pi^3]]F) = 1$ because now only the first three states of each trace are considered; and $\mathrm{val}^{\alpha}_{s_1}([\pi^2]F) = 1$ because there is no final state within the first two states of a trace, and the universal quantification over the empty set is trivially true.

3 A Sequent Calculus for *DLP*

We use a sequent calculus to axiomatise *DLP*. As usual, the semantics of a rule is that the validity of the premisses above the line implies the validity of the conclusion. A full account (except for limited programs) is given in [8]. The core of the calculus are the rules for symbolic execution of Promela$^+$ code.

The idea of the rules for symbolic execution is that first scheduling rules introduce tags. The semantics of a tagged program $i : \pi$ is the same as of π with the restriction that the next command to be executed is from the i-th process of π. Thus scheduling rules can be seen as case distinctions that have one case for every possible scheduling decision.

Secondly, unwinding rules transform the program $i : \pi$ by replacing composed commands (selections, loops and sequences marked as atomic) with elementary ones. This is necessary since only elementary commands have a well-defined effect on the global state (due to the non-deterministic interleaving of processes). Both scheduling and unwinding lead to branches or alternatives in the proof tree reflecting the indeterminism of Promela.

Thirdly, an elementary command is executed. A typical example rule is:

$$\frac{\mathit{ex}(c) \vdash F \qquad \mathit{ex}(c) \vdash \mathit{eff}(c)[[\mathit{remProg}(\pi, i)]]F}{\vdash [[i : \pi]]F}$$

The purpose of this rule is to execute the first command c of the i-th process of π.

While tags are syntactic entities of *DLP*, *ex*, *eff*, and *remProg* are meta level abbreviations that allow to state several rules in one compact rule scheme: $\mathit{ex}(c)$ is a first-order formula that expresses the executability of c, $\mathit{eff}(c)$ expresses the state transitions caused by the execution of c and modifies the state in which the following formula is to be evaluated, and $\mathit{remProg}(\pi, i)$ is the program that remains to be executed after c has been executed. These abbreviations must be defined for all elementary commands c. E.g., if c spawns a new process, $\mathit{remProg}(\pi, i)$ removes c from π and adds the new process instance to π.

Having the intuitive meaning of these functions in mind it is easy to understand the rule: If c is executable, the formula $[[i : \pi]]F$, which states that F holds in all states on all traces, is reduced to F, which states that F holds in the current state, and $\mathit{eff}(c)[[\mathit{remProg}(\pi, i)]]F$, which states that F holds in all states on all traces that arise if the remaining program is executed in the next

state (characterised by the state updates *eff*(c)). If c is not executable, the proof goal is closed immediately.

Note that the execution rule eliminates the tag and the scheduling rules are applicable again. This cycle is repeated until the program is completely executed and can be discharged.

Rules for Limited Programs. In order to derive formulas with non-terminating programs we apply induction rules, usually on the length of the traces. However, this is only possible if the formula contains a term that encodes this length. Therefore, limited programs are introduced by the rules given in Fig. 1. Here the rules in the right column are not necessary for completeness and are only given for symmetry. After introduction of the limited programs induction on t can be applied.

$$\frac{M \in \{[], [[]], [\langle\rangle]\}}{\vdash\ \forall t : int.\ t \geq 1 \rightarrow M(\pi^t)F \over \vdash\ M(\pi)F} \qquad \frac{M \in \{\langle\rangle, \langle[]\rangle, \langle\langle\rangle\rangle\}}{\vdash\ \exists t : int.\ t \geq 1 \wedge M(\pi^t)F \over \vdash\ M(\pi)F}$$

Fig. 1. Introducing limited programs

Formulas that contain limited programs are treated by similar symbolic execution rules as those with unlimited programs. In particular the scheduling and unwinding rules are the same. For command execution one detail is changed: The limit is decremented because one program step has been executed. For example the above execution rule corresponds to

$$\frac{t \geq 2\ ,\ ex(c)\ \vdash\ F \qquad t \geq 2\ ,\ ex(c)\ \vdash\ eff(c)[[remProg(\pi, i)^{t-1}]]F}{t \geq 2\ \vdash\ [[i : \pi^t]]F}$$

The rules for discharging limited programs when the limit is reached, i.e., $t = 1$, and for the degenerate cases where $t \leq 0$ are given in Fig. 2. If $t = 1$ a special case must be distinguished: If the program is non-empty, i.e. $\pi \neq \epsilon$, it has not terminated yet and no final state exists, which means that $[\pi^t]F$ is always true and $\langle\pi^t\rangle F$ is always false. If $t \leq 0$, formulas that contain operators that quantify universally over the traces are always true.

3.1 Soundness and Relative Completeness

Soundness has to be shown separately for each rule. Most of the proofs are technical, but not difficult. Except for the rules for limited programs they can be found in [8].

The soundness of the rules for limited programs either follows directly from the definition of the semantics or from the soundness of the corresponding rule for unlimited programs—except for the introduction rules for limited programs for the operators $[\langle\rangle]$ and $\langle[]\rangle$. For the soundness of these rules the following theorem.

$$\frac{\pi \neq \epsilon,\ M = []}{t = 1\ \vdash\ M(\pi^t)F} \qquad \frac{\pi \neq \epsilon,\ M = \langle\rangle \quad t = 1\ \vdash}{t = 1\ \vdash\ M(\pi^t)F} \qquad \frac{\text{otherwise} \quad t = 1\ \vdash\ F}{t = 1\ \vdash\ M(\pi^t)F}$$

$$\frac{M \in \{[], [[]], [\langle\rangle]\}}{t \leq 0\ \vdash\ M(\pi^t)F} \qquad \frac{M \in \{\langle\rangle, \langle[]\rangle, \langle\langle\rangle\rangle\} \quad t \leq 0\ \vdash}{t \leq 0\ \vdash\ M(\pi^t)F}$$

Fig. 2. Discharging limited programs

Theorem 1. *For all unlimited programs π, all formulas F, all states s, and all assignments α:*

$$\mathrm{val}_s^\alpha(M(\pi)F) = \left\{ \begin{matrix} \inf \\ n \in \mathbb{N}^* \\ \sup \\ n \in \mathbb{N}^* \end{matrix} \right\} \mathrm{val}_s^\alpha(M(\pi^n)F) \left\{ \begin{matrix} \text{if } M = \langle[]\rangle \\ \text{if } M = [\langle\rangle] \end{matrix} \right\}$$

This theorem states that it is sufficient for the evaluation of formulas containing the modalities $\langle[]\rangle$ and $[\langle\rangle]$ to consider only finite prefixes of the traces. The proof is based on quite involved an induction and can be found in [8]. It is essential for this theorem that the number of indeterministic choices in Promela is always finite. That guarantees that the sets $\mathrm{val}_s^\alpha(\pi^n)$ are always finite whereas the set $\mathrm{val}_s^\alpha(\pi)$ may even be uncountable.

The idea of the proof of relative completeness is to discharge all programs by symbolic execution if possible and to introduce limited programs and use induction for non-terminating programs. Then the remaining first-order formulas can be handled by standard methods. Relative completeness means that all valid formulas could be derived in the calculus if an oracle for arithmetic was available, i.e., a rule scheme providing all valid arithmetic formulas as axioms. Of course, in reality such a rule cannot exist but this is not harmful to "practical completeness". Rule sets for arithmetic are available, which—as experience shows—allow to derive all valid first-order formulas that occur during the verification of actual programs. Moreover, many arithmetic formulas can be automatically discharged by external decision procedures like CVC [9] and the Simplify tool, which is part of ESC/Java [3].

It is possible to give further operators, e.g., $\mathit{Inf}(\pi)F$ stating that F holds infinitely many times along every trace of π, for which the above theorem does not hold. Therefore the described technique cannot be extended to give a sound and relatively complete axiomatisation for these operators.

3.2 Examples

We now present two examples. Although they are extremely simple they cannot be verified using the model checker Spin, whereas their deductive verification can be done in a standard way. This shows the fundamental advantages of the deductive approach.

For these examples, note that in Promela **do**...**od** denotes a guarded non-deterministic choice, that is repeated until a break is encountered. Consider the program π defined as

```
do
  :: skip
  :: x=0; break
od
```

We now want to verify that $x \doteq 0$ holds in all possible final states of π. In *DLP* this property is expressed as $[\pi]x \doteq 0$. Spin cannot handle this because infinitely many and arbitrarily long runs exist for this model. However, using limited programs and induction on t in π^t this model can be easily verified deductively. The induction hypothesis is $\forall t : int.\ t \geq 1 \rightarrow [\pi^t]x \doteq 0$.

For the second example let π be defined as

```
do
  :: x != 0; x=x−1
  :: else; break
od
```

where x is decreased as long as it is non-zero, and if x is zero, the loop terminates. We want to prove validity of the formula $\forall x : int.x \geq 0 \rightarrow [\pi]x \doteq 0$ expressing that for every initial value of x greater than 0 and all terminating runs $x \doteq 0$ holds in the final state. This property cannot be verified with Spin since it does not allow arbitrary initial states. However, Spin can easily verify a similar property with a fixed value for x.

While arbitrary initial states are not provided for in Promela they naturally occur in many realistic scenarios, e.g., if a program is started in the final state of another program or if it depends on some external input. The initial state of a *DLP* verification is always arbitrary. If only certain initial states are to be considered restrictions must be included in the property to be verified.

The formal proof of this example is done by induction on x and can be found in [8]. Note that the induction hypothesis has to be specified interactively which can be very hard to find in practice.

4 Related Work and Conclusions

There are some approaches to define the semantics of Promela formally, most notably [2]. But the semantics that is induced by our semantics for Promela$^+$ is by far the most comprehensive one. Relative to this semantics we introduced a calculus that allows deductive theorem proving for Promela. No previous work in this direction exists for Promela or other non-deterministic multi-process languages. We showed that it is—in principle—possible to approach such programming languages with methods from deductive theorem proving, thus preparing the ground for implementations in automatic provers. The methods we used are very general and can be easily applied to other languages.

The given examples show that *DLP* increases the set of verifiable Promela models significantly. As a minor drawback *DLP* has only six temporal operators to make statements about traces whereas Spin allows to specify arbitrary LTL formulas. E.g., in order to express the property "A holds for a while, and then B holds forever" *DLP* must be extended by a specific modality whereas this can be easily expressed in LTL. We intend to extend our definitions to allow for arbitrary modal operators, which would render our language as expressive as LTL. First approaches have shown that it is indeed possible to give rules for the case where the modal operator is arbitrary.

References

1. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In *International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 626–641, 2001.
2. M. del Mar Gallardo, P. Merino, and E. Pimentel. A generalized semantics of Promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.
3. ESC/Java (Extended Static Checking for Java). `http://research.compaq.com/SRC/esc/`.
4. D. Harel. *First-order Dynamic Logic*, volume 68 of *LNCS*. Springer, 1979.
5. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
6. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
7. Promela Language Reference. Available at `http://spinroot.com/spin/Man/promela.html`.
8. F. Rabe. A dynamic logic with temporal operators for Promela. Master's thesis, Universität Karlsruhe, 2004. Available online at `http://i12www.ira.uka.de/~frabe/DLTP.pdf`.
9. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CA V)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002.