The Many Uses of Dynamic Logic

Wolfgang Ahrendt[1][0000−0002−5671−2555], Bernhard Beckert[2][0000−0002−9672−3291], Richard Bubel[3][0009−0003−4847−4707], Reiner Hähnle[3][0000−0001−8000−7613], and Mattias Ulbrich[2][0000−0002−2350−1831]

[1] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
ahrendt@chalmers.se
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{beckert,ulbrich}@kit.edu
[3] Technische Universität Darmstadt, Darmstadt, Germany
{richard.bubel,reiner.haehnle}@tu-darmstadt.de

Abstract. Dynamic logic is a multi-modal logic for reasoning about programs. In deductive verification systems, it can be used as a versatile alternative to the Floyd-Hoare calculus with uniform syntax and semantics. Dynamic logic has not only been used in functional verification, but one can represent a plethora of verification scenarios in it, including relational and hyperproperties, program equivalence, information flow, incorrectness logic. Dynamic logic is the basis for three deductive verification tools that are highly competitive in their application domain. In this article, we present the foundations of dynamic logic and we review its many uses in state-of-the-art deductive verification.

Keywords: Dynamic logic, deductive verification, symbolic execution, weakest preconditions

This article is dedicated to Wolfgang Reif on the occasion of his 65th birthday

1 Introduction

Dynamic logic [52, 73] is a modal logic for reasoning about programs introduced by V. Pratt [66] in 1976. The term *dynamic logic* was coined in the paper [42] and it was initially investigated by D. Harel [41] and R. Goldblatt [36]. Wolfgang's association with dynamic logic starts in 1984 with his Master's Thesis [67] that laid some of the theoretical foundations of the calculus used in the KIV deductive verification system. Until then, excepting a short-lived attempt [57], dynamic logic had been mainly the object of theoretical investigation and had not been used as a program logic in an actual verification system. Then (and now) the Floyd-Hoare calculus [47] was far more popular, despite clear advantages on the side of dynamic logic including (i) greater expressiveness, (ii) syntactic closure by first-order connectives and quantifiers, and (iii) the proximity to relational program semantics.[4]

[4] For some reasons how this might have come about, see [37, Sect. 6.2.7].

The work of Wolfgang (together with M. Heisel and W. Stephan) was the first serious attempt to apply the benefits of dynamic logic within the context of a deductive verification system [38, 44]. Specifically, Wolfgang presented in his Ph.D. work [68, 69] an elegant solution to a pivotal problem in program verification: how to realise procedure-modular verification. For this he used abstract data types and step-wise refinement of program modules, its correctness being justified by dynamic logic. This approach within a few years yielded impressive practical success [70]. The KIV system is still developed and maintained in Wolfgang's research group [32] and was used for some of the most intricate and comprehensive formal verification efforts to date [72], see also Sect. 4.1 below.

When some of the present authors began considerations [39] on their deductive verification system KeY [4] for the Java programming language, one decision was obvious: To use dynamic logic as its theoretical foundation. Thanks to Wolfgang's work, its usefulness was abundantly clear and we did not regret the choice for a second. In this article we celebrate the versatility of dynamic logic, not only as a language for expressing and verifying the correctness of *real* programs and case studies, but also as a language to express many other problems such as information flow analysis or relational verification.

In Sect. 2 we review the foundations of dynamic logic in terms of modal logic, relational semantics, and the original, "pure" programming language inspired by regular expressions used by Pratt, Harel et al. In Sect. 3 we highlight the expressive power of dynamic logic in *practical terms*: its ability to characterise or incorporate a wide range of specification approaches and program analysis techniques. In Sect. 4 we collect some of the success stories achieved with deductive verification systems based on dynamic logic, before we briefly conclude in Sect. 5.

2 Foundations of Dynamic Logic

2.1 Logics of Change

Dynamic logic belongs to the family of logics reflecting the fact that the *state of affairs* (of the word, of the mind, of a system, etc.) can change. Such logics can be referred to as *logics of change*. For a long time, philosophers had been thinking about the impact of change on reasoning, for example, Aristotle and William of Ockham. In modern times, logic became the subject of mathematical studies in the form of *mathematical logic* (or *meta mathematics*). Within that discipline, there evolved a branch of logic which focuses on the modelling of change, namely *modal logic*, starting with the work of C. I. Lewis [55]. The first, surprisingly little known, approach to semantics of modal logic was proposed by B. Jónsson and A. Tarski [49]. Further attempts on semantics were contributed in the 1950s, by A. Prior, J. Hintikka, and S. Kripke [53], where the latter had the biggest impact by far.[5] In the early years, motivating application areas of modal logic were largely the philosophy of language, epistemology, and metaphysics. But

[5] Kripke was at that point unaware of Tarski's work on modal logic.

over time, *computation* became more and more a prime application of modal logic, and variations thereof.

In modal logic, we assume a non-modal base logic, for instance propositional logic or first-order logic. Moreover, we assume a set of *possible worlds*, and a definition of when a formula of the base logic is valid in any given world. So far, there is no aspect of change. This comes in if we assume, in addition to the above, a relation R between worlds, called *accessibility relation*. When two worlds w and w' are in this relation, i.e., $(w, w') \in R$, it means that we can go from w to w' in *one* R-step (whatever an R-step is supposed to mean intuitively in the application at hand). With these ingredients, we obtain what is called a *Kripke structure*. As an example, let us consider Fig. 1 (where we ignore the formulas containing $\Box$ or $\Diamond$ for now). In two of the possible worlds, p is true, whereas p is false in the others. The arrows depict the accessibility relation R. In our example, R does not enjoy many properties. (In fact, R is neither reflexive, symmetric, anti-symmetric, transitive, total, nor deterministic.)

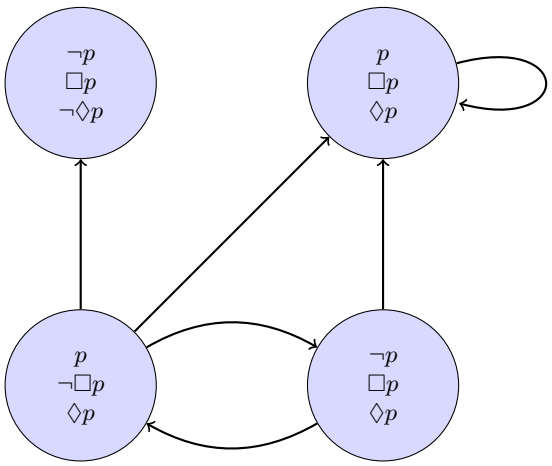

Fig. 1. Kripke Structure: possible worlds with (one-step) accessibility relation

A given base logic, such as propositional or first-order logic, can be extended to a modal logic by adding two logical operators, $\Box$ and $\Diamond$. A formula $\Box\phi$ is valid in a world w iff ϕ is valid in *all* worlds accessible from w via R (in one step). And a formula $\Diamond\phi$ is valid in a world w iff ϕ is valid in *some* world accessible from w via R (in one step). The reader may check that all formulas in Fig. 1 are valid in the worlds they are depicted in. Assume the validity of the propositional literals, the modal formulas follow then from this assumption and the chosen transition relation R. A perhaps unintuitive corner case is the upper left world, from where no world is accessible via R (not even that world itself). Therefore, $\Box p$ and $\neg\Diamond p$ are vacuously true that world.

Modal logic is syntactically closed under all propositional operators, the ones from the base logic as well as $\Box$ and $\Diamond$. For instance, $\Diamond p \wedge \neg\Box p$ and $\Box\Box p$ are formulas of modal logic (both are valid in the lower left world). In general, modal logics discussed in the literature come in many flavours, called K, T, S4, S5, D, among others, which vary in their properties of the accessibility relation.

The reader may have noticed the syntactic similarity of modal logic to temporal logic as introduced by A. Pnueli [65] (even if he wrote G for $\Box$ and F for $\Diamond$). But in temporal logic, $\Box p$ and $\neg\Diamond p$ could not be true at the same time (see Fig. 1 upper left world), neither could $\neg\Box p$ and $\Box\Box p$ be true at the same time (see Fig. 1 lower left world). Moreover, in temporal logic, $\Box$ and $\Diamond$ refer to arbitrarily many steps, not just one as in modal logic. Still, quoting Pnueli, his temporal logic *"is completely isomorphic to the modal logic system S4"* [65].[6] The reason is that, in temporal logic, the accessibility relation is reflexive and transitive, such that arbitrarily many steps forward are at the same time a single step in the modal reading.

2.2 Dynamic Logic as a Multi-Modal Logic

Dynamic logic was introduced in 1976 by V.R. Pratt in *"Semantical considerations on Floyd-Hoare Logic"* [66]. It extends modal logic by a language of *actions*. Actions can moreover be composed to new actions, such that they are reminiscent of *programs*. Every (elementary or composed) action gives rise to its own accessibility relation. In that sense, dynamic logic is a *multi-modal logic*.

In formulas of dynamic logic, the modalities are "parameterised" by actions. Syntactically, we write the actions inside the modalities. The syntax looks as follows, given here together with its intuitive meaning:

- $\langle\alpha\rangle\phi$ means "ϕ holds in *some* state we can *reach by executing* α"
- $[\alpha]\phi$ means "ϕ holds in *all* states we can *reach by executing* α"

Here, "reach by executing α" refers to one step in the α-accessibility relation. Please note that the worlds of modal logic are called "states" here and in the following.

2.3 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL) [33] is typically based on *non-deterministic* actions. In applications of dynamic logic, the non-determinism serves mainly two purposes: abstraction, and the modelling of an uncontrollable environment. The following notation and definitions are inspired by [52, 64].

Definition 1 (Propositional Dynamic Logic: Formulas and Actions). *We assume a set of atomic formulas and a set of atomic actions. If ϕ, ψ are formulas, and α, β are actions, then*

[6] Temporal logic as introduced by Pnueli [65] did not have a "next" operator.

- $\neg\phi$
- $\phi \vee \psi$
- $\langle\alpha\rangle\phi$ *("some execution of α leads to a state where ϕ holds")*

are also formulas, and

- $\alpha;\beta$ *(sequence)*
- $\alpha \cup \beta$ *(non-deterministic choice)*
- α^* *(execute α a finite, non-deterministic number of times)*
- $?\phi$ *(proceed if ϕ, otherwise fail)*

are also actions.

The above action language is also called *"regular programs"* [33]. Further formulas ($\phi \wedge \psi$, $\phi \rightarrow \psi$,$[\alpha]\phi$) and actions (see below) can be derived from the ones above. In particular, just like modal logic and temporal logic have dual modalities $\Box$ and $\Diamond$, dynamic logic has, for every action α, a modality $[\alpha]$ which is dual to $\langle\alpha\rangle$. The former can be defined as follows:

$$[\alpha]\phi \equiv \neg\langle\alpha\rangle\neg\phi \tag{1}$$

The intuitive meaning of $[\alpha]\phi$ is that *"all executions of α lead to a state where ϕ holds"*. Let us provide some intuition by means of an example.

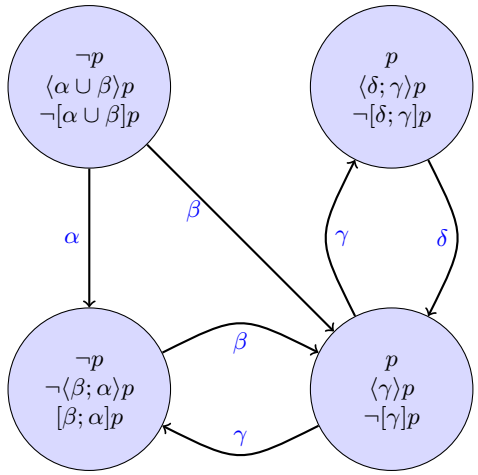

Fig. 2. Kripke structure with multiple atomic action relations

Fig. 2 depicts, for each state, a few dynamic logic formulas that are valid there. The example highlights the difference between box and diamond modalities. For instance, in the upper left state, there exists a state accessible by

non-deterministic choice of α or β where p holds. But in the same state, it is not the case that p is true in *all* states accessible by $\alpha \cup \beta$. A similar situation occurs when executing δ and γ in sequence in the upper right state. The dynamic logic formulas given in the lower left state talk about the action $\beta;\alpha$, which is, however, not feasible from that state (because, after executing β, there is no α that can be executed). We will come back to this phenomenon.

To further explain PDL, we provide semi-formal definitions of its semantics (for a formal account, we refer to [52]). In particular, we build on notions not fully defined in this chapter. In the following, we assume a set of states S, a meaning function $\mathcal{M}$ of atomic formulas $\phi^{\mathcal{M}} \subseteq S$ (assigning to each formula the set of states in which it is true)[7], and the (overloaded) meaning function $\mathcal{M}$ of atomic actions $\alpha^{\mathcal{M}} \subseteq S \times S$ (such that each action denotes a relation between states). This relation does not have to be deterministic, neither does it have to be total. For instance, $\gamma^{\mathcal{M}}$ in Fig. 2 is non-deterministic on the lower right state, and undefined on all other states. The following definition extends the meaning function $\mathcal{M}$ from atomic to non-atomic formulas and actions.

Definition 2 (Relational Semantics of PDL Formulas).
Meaning of formulas $\phi^{\mathcal{M}} \subseteq S$, meaning of actions $\alpha^{\mathcal{M}} \subseteq S \times S$:

- $(\neg\phi)^{\mathcal{M}} = S - \phi^{\mathcal{M}}$
- $(\phi \vee \psi)^{\mathcal{M}} = \phi^{\mathcal{M}} \cup \psi^{\mathcal{M}}$
- $(\langle\alpha\rangle\phi)^{\mathcal{M}} = \{u \mid \exists v.\ (u,v) \in \alpha^{\mathcal{M}}\ \textit{and}\ v \in \phi^{\mathcal{M}}\}$
- $(\alpha;\beta)^{\mathcal{M}} = \{(u,v) \mid \exists w.\ (u,w) \in \alpha^{\mathcal{M}}\ \textit{and}\ (w,v) \in \beta^{\mathcal{M}}\}$
- $(\alpha \cup \beta)^{\mathcal{M}} = \alpha^{\mathcal{M}} \cup \beta^{\mathcal{M}}$
- $(\alpha^*)^{\mathcal{M}} = \bigcup_{n\in\mathbb{N}} (\alpha^n)^{\mathcal{M}}$, *where* $\alpha^{n+1} \equiv \alpha;\alpha^n$ *and* $\alpha^0 \equiv \{(u,u) \mid u \in S\}$
- $(?\phi)^{\mathcal{M}} = \{(u,u) \mid u \in \phi^{\mathcal{M}}\}$

The meaning of non-deterministically choosing between α and β is the union of the behaviours of α and β. The meaning of α^* is all behaviours resulting from choosing some $n \in \mathbb{N}$, non-deterministically, and repeating α n times. This includes zero repetitions of α, which is a *skip* operation in all states. Note that α^* denotes *finite* iterations only. The *test* action $?\phi$ deserves special attention. In states where ϕ holds, it behaves like *skip*, i.e., we stay in the same state. But in states where ϕ does *not* hold, there is *no* state to go to with this action, not even the same state. This is similar to applying a partial function outside the domain, where it is defined. The result would be undefined, not the identity. Intuitively, we can see a computation where this happens as a *failed* computation.[8]

According to Def. 2, $\alpha;\beta$ fails as soon as any of α or β fail. We illustrate this with Fig. 3. The crossed out β arrow from state s_1 illustrates that atomic action β fails on s_1, i.e., there is no state s' such that $(s_1, s') \in \beta^{\mathcal{M}}$. Therefore, by Def. 2, there is also no state s' such that $(s_0, s') \in (\alpha;\beta)^{\mathcal{M}}$, hence $\alpha;\beta$ fails on s_0. Therefore, in state s_0, the choice $\alpha;\beta \cup \gamma;\delta$ collapses to $\gamma;\delta$.

This is an important point in the interplay of choice and failure as defined by the relational semantics. In cases where non-determinism allows numerous

[7] Note that this semantic modelling avoids truth values.
[8] A failed computation is sometimes referred to as *abort* in the DL literature.

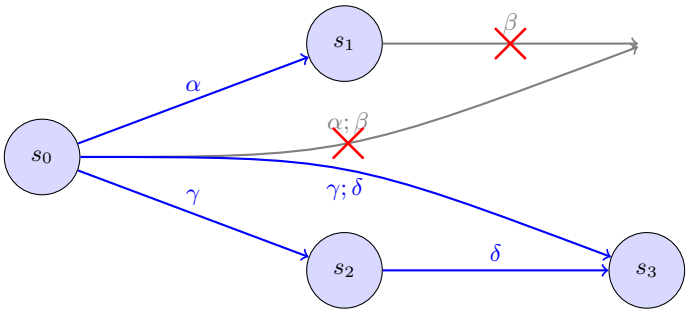

Fig. 3. Propagation of failure over sequence and choice

computation paths, any path p where a failure occurs is taken out from the set of possible behaviours, even if the failure occurs arbitrarily late in p. Accordingly, it is a consequence of Def. 2 that failed computations of α are not included in the set of behaviours which are quantified over in the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$. In other words, we can think of $\langle\alpha\rangle\phi$ as "some non-failing computation of α leads to ϕ", and $[\alpha]\phi$ as "all non-failing computations of α lead to ϕ".

To support a more common notion of "programs", well-known programming constructs are definable in PDL's action language:

- **skip** $\equiv ?\mathit{true}$
- **fail** $\equiv ?\mathit{false}$
- **if** ϕ **then** α **else** β **fi** $\equiv (?\phi;\alpha) \cup (?\neg\phi;\beta)$
- **while** ϕ **do** α **od** $\equiv (?\phi;\alpha)^*\,;\,?\neg\phi$

skip was discussed informally already; **fail** is a computation that always fails: $(?\mathit{false})^{\mathcal{M}} = \{(u,u) \mid u \in \mathit{false}^{\mathcal{M}}\} = \{(u,u) \mid u \in \emptyset\} = \emptyset$. In the definition of the conditional, one of the choices $(?\phi;\alpha)$ and $(?\neg\phi;\beta)$ necessarily fails and is discarded from the possible behaviours. The definition of loops is concise but intricate. The * operator permits an *arbitrary* number of iterations, whereas the α in the **while** loop is supposed to be repeated *exactly* as long as ϕ is true, not more, not less. The resolution of this seeming conundrum is illustrated in Fig. 4, where we depict a scenario where ϕ becomes false after three iterations of α. The key insight is that most choices of the number n of iterations lead to failure and are therefore discarded. If n is chosen too small, we exit the loop at a point where ϕ is still true and the following action $?\neg\phi$ will fail. If, on the other hand, n is chosen too big, we stay in the loop even when ϕ became false, and the following iteration $?\phi;\alpha$ will fail. In our example, all choices of n other than 3 lead to an execution that is discarded, only the choice $n = 3$ leads to an execution which is kept in $((?\phi;\alpha)^*\,;\,?\neg\phi)^{\mathcal{M}}$.

How is non-termination handled, given that the * operator permits only a *finite* number of iterations? To make the point, we consider the extreme case of the program **while** *true* **do** α **od**, which is defined as $(?\mathit{true};\alpha)^*\,;\,?\neg\mathit{true}$.

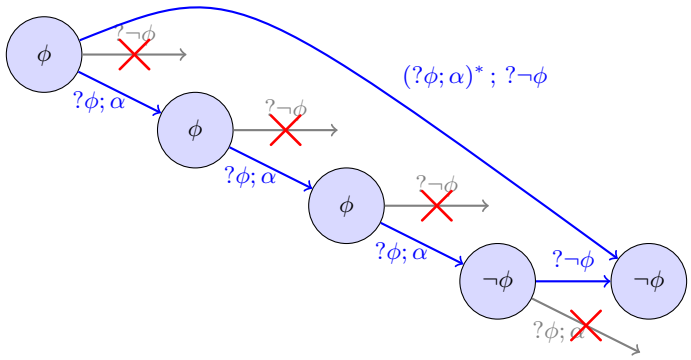

Fig. 4. Illustration of loop definition in dynamic logic

This can simplified to $\alpha^*\,;\,?\mathit{false}$. Regardless of which number n of iterations is chosen (non-deterministically), the execution will fail because of the last action $?\mathit{false}$. Instead of semantically modelling infinite executions, the relational semantics models *infinitely* many attempts on *finite* executions, all of which fail. The resulting relation is empty, i.e., the final state of the program is not defined. (In general, a program's final state may be defined for certain initial states, but not for others.)

2.4 Deterministic PDL

In case the action language has a deterministic semantics, we have the special case of *deterministic PDL*. We assume the atomic actions to be deterministic: For all atomic actions α, if $\{(s,s'),(s,s'')\} \subseteq \alpha^{\mathcal{M}}$, then $s' = s''$. We further assume that the non-deterministic constructs $\cup$ and * appear *only* to abbreviate **if** and **while** (where all but one of the non-deterministic choices fail).

In this setting, we can move from relations to partial functions as meaning of (atomic or composed) actions: $\alpha^{\mathcal{M}} \in S \rightharpoonup S$. Note that the behaviour of an action (program) can still be undefined. But there is at most one defined behaviour for every initial state.

In deterministic PDL (and all other deterministic versions of DL), $[\alpha]\phi$ denotes *partial correctness*, whereas $\langle\alpha\rangle\phi$ denotes *total correctness*. Moreover, partial correctness implies total correctness, i.e., we have $\langle\alpha\rangle\phi \rightarrow [\alpha]\phi$. (This is not the case in general PDL, see $\langle\gamma\rangle\phi$ in Fig. 2.)

2.5 First-Order Dynamic Logic

So far, there is no notion of program variables, which is, however, essential for imperative programs. To address this, D. Harel introduced *first-order dynamic*

logic (FDL) [41], adding variables to programs, and quantification to formulas. In FDL, atomic programs have the following form:

- $v := t$ (deterministic assignment)
- $v := *$ (non-deterministic assignment)

where v is a program variable, and t is an expression in an underlying side effect-free expression language. Atomic formulas are of the forms:

- $p(t_1, \ldots, t_n)$
- $t_1 \doteq t_2$

Regarding composite formulas, we extend the operators from PDL with quantification. If ϕ is an FDL formula, then $\exists x.\phi$ and $\forall x.\phi$ are also FDL formulas. Otherwise, composite programs and formulas are formed exactly as in PDL.[9] FDL is syntactically closed under all its logical operators, the quantifiers as well as the logical operators from PDL. For instance, all of the following are formulas of FDL:

1. $\forall x.\, (\langle t := a; a := b; b := t\rangle b = x \;\leftrightarrow\; \langle a := a + b; b := a - b; a := a - b\rangle b = x)$
2. $\langle\alpha\rangle\exists x.\phi(x)$
3. $\exists x.\langle\alpha\rangle\phi(x)$

We see that operators from propositional and first-order logic can appear inside or outside the scope of modalities. The first formula states equivalence of two programs (relative to the final value of b)—a property most non-dynamic logic program logics cannot express. The other two formulas are equivalent for a simple programming language as the one given here. But in richer programming languages, where resources can be generated by programming constructs (such as object creation in object-oriented languages), these formulas can have a different meaning when α extends the domain we quantify over (see [5] for an in-depth discussion).

The non-deterministic assignment $v := *$ is a tool for abstracting away from irrelevant details, or for receiving input from an unknown environment. Moreover, the combination of non-deterministic assignment and test actions allows for a kind of declarative programming when desired. Concretely, the program fragment $v := *; ?\phi$ expresses the following command: "choose v such that $\phi(v)$ is true". This is used when modelling assumptions on the environment during verification of controllers, as in the KeYmaera X theorem prover, see Sect. 4.3.

3 Expressive Power of Dynamic Logic

As seen in Sect. 2, in dynamic logic, programs as the constituents of modalities are first-class citizens of its syntax, which makes dynamic logic closed under syntactic composition. This property sets dynamic logic apart from most other program logic frameworks that limit the formulation of program properties to

[9] Usually, FDL allows no quantifiers in $?\phi$.

pre-defined patterns (like Hoare calculus), or that see programs as operators on formulas (like Dijkstra's weakest precondition calculus). These formulate *functional* properties corresponding to the canonical pattern $\psi \rightarrow [\alpha]\phi$ in dynamic logic which asserts that a postcondition ϕ holds in some post-state of a program α under the assumption of a precondition ψ in the pre-state. Dynamic logic is not restricted to this pattern, and without the need to define syntactic or semantic extensions, or introducing *ad-hoc* notation, one can use the well-studied and semantically clear composition operations of propositional and first-order logic to obtain formalisations of program properties beyond the ones mentioned. For instance, one may

- quantify over dynamic logic formulas containing modalities,
- nest dynamic logic operators so that the formula in the scope of a modality may again contain modalities, or
- use modalities inside test operators within programs

to name only a few of the possibilities. While the functional property pattern is prominent in functional verification and covers many application scenarios, there are interesting properties that heavily benefit from the expressive power of dynamic logic.

In this section, we show how the core concepts of some important program analysis frameworks can be expressed within dynamic logic. Moreover, we look at a number of interesting program properties beyond the functional pattern. The advantage of the expressive power of dynamic logic is that one can rely on a sound basis in a well-understood program logic when addressing novel verification questions and approaches.

3.1 Hoare Calculus

The Hoare calculus [47] allows us to specify a program's behaviour in terms of the eponymous Hoare triple

$$\{pre\}\ \alpha\ \{post\}\ ,$$

where $pre, post$ are first-order logic formulas representing the precondition and postcondition, respectively, of program α.[10]

The meaning of the triple is that if program α is executed in a state in which formula *pre* is satisfied *and* program α terminates, then in the[11] reached final state formula *post* holds. In other words, a Hoare triple states the partial correctness of a program. Manna and Pnueli [58] propose an extension for reasoning about total correctness and built-in support to access the pre-state value of variables in the postcondition. For that, they require the provision of a *convergence function* whose value must be shown to decrease by each iteration of the loop. For a general overview of the Hoare calculus and its influence, we refer to [7].

[10] Originally, the syntax was "*pre* $\{\alpha\}$ *post*", but at some point the braces were moved.

[11] The Hoare calculus was formulated for deterministic programs which we assume here as well.

Hoare's objective was to provide a logical framework to prove the behavioural correctness of programs. However, this framework does not define a *logic* as it is not closed under the logical connectives and, hence, not a universal algebra. This omission results in missing compositionality and limitations of properties that are directly expressible within the framework itself. For instance, non-interference is not expressible within Hoare logic without self-composition [26].

Dynamic logic [41, 66] addresses this shortcoming, being a program logic, where programs are first-class citizens (as in the Hoare calculus) of its formulas but which is closed under its operators (logical connectives). As mentioned above, Hoare triples can then be expressed as $pre \rightarrow [\alpha]post$, total correctness is expressible as $pre \rightarrow \neg[\alpha]\neg post$, or simply, $pre \rightarrow \langle\alpha\rangle post$ (for deterministic programs).

3.2 Flexible Verification Patterns

A common scenario where the syntactic flexibility of dynamic logic is useful occurs when a specification contains logic symbols whose definition is given in terms of code (for instance, as a pure function). Consider the proof obligation `valid(x)` $\rightarrow$ [`p`]*post*, where `valid` references a function in the programming language. Dynamic logic allows one to formulate this *as a formula*

$$[\texttt{r=valid(x)}]\texttt{r} \doteq \text{true} \rightarrow [\texttt{p}]post$$

that refers to `valid` as a program function. Consequently, there is more than one piece of code in the proof goals which is perfectly admissible in dynamic logic.

In a further scenario, dynamic logic is used to specify reachability. Consider the singly linked list Java implementation in Listing 1.1, where we navigate from one node of the list to the next via attribute `next`. The final element of the list is reached when its attribute `next` is `null`. Assume we want to specify that method `contains(int)` returns `true` if the list contains a node with the content passed as parameter `e`. To specify the intended behaviour, we use that the precondition (and postcondition) of a dynamic logic specification in turn can contain programs. Consider program `p` which assigns program variable `l` the value of the `idx`-th element in the list (assuming program variable `idx` is initially non-negative). Then the following dynamic logic formula specifies the intended be-

```
class ListNode {
 ListNode next;
 int cnt;

 boolean contains(int e) { ... }
}

// program p for specification
ListNode l = this;
while (l != null && idx >= 0) {
 l = l.next;
 idx--;
}
```

Listing 1.1. Singly linked list and program used for specification

haviour of `contains`:

$$\Big(\exists k.\big(k \geq 0 \ \wedge \ \langle \texttt{idx = k; p}\rangle(\texttt{l} \neq \texttt{null} \wedge \texttt{l.cnt} \doteq \texttt{e})\big)\Big) \\ \leftrightarrow \langle \texttt{res = contains(e);}\rangle(\texttt{res} \doteq \mathit{TRUE})$$

3.3 Dijkstra's Weakest Precondition Calculus

Dijkstra's weakest precondition calculus [29] defines a predicate transformer wp that computes the *weakest precondition* from a given postcondition. To prove correctness of a program, it remains to be shown that the given precondition implies the weakest precondition. As wp is a predicate transformer and thus a meta construct, which transforms formulas, it is not a first-class citizen of the logic.

The original wp-operator includes to prove termination of the given program; one variant, the *weakest liberal precondition* wlp formalizes partial correctness. Although the wp-calculus semantics aligns nicely with the "diamond" semantics of dynamic logic, there is a crucial difference when the target programming language is non-deterministic, as is the guarded command language Dijkstra introduced alongside the wp-calculus. Given a non-deterministic program α that can choose non-deterministically between continuations $\alpha_1, \ldots, \alpha_n$, then $\mathsf{wp}(\alpha, \mathit{post})$ requires that (i) program α can make a non-deterministic choice (i.e., it does not abort or is blocked), and (ii) that for any executable choice α_i, $\mathsf{wp}(\alpha_i, \mathit{post})$ holds. In contrast, the diamond formula $\langle\alpha\rangle \mathit{post}$ holds if (i) program α can make a non-deterministic choice (i.e., it does not abort or is blocked), and (ii) that *there is one* executable choice α_i for which $\langle\alpha_i\rangle \mathit{post}$ holds.

This difference makes a canonical emulation of the wp-calculus in dynamic logic impossible. Of course, extending dynamic logic with a dedicated modality matching wp's semantics is one possibility. This has been realized in KIV [32] by introducing the strong diamond modality $\langle\!|\, \cdot \,|\!\rangle \cdot$. However, wp and wlp share not the duality of the diamond and box operator in dynamic logic but differ only in the termination requirement as such. This means wlp is a natural fit for dynamic logic's box operator and thus can be canonically emulated by inference rules for dynamic logic.

We provide some examples to showcase how wlp-rules can be directly cast into dynamic logic axioms (and dynamic logic inference rules):

- It is easy to see that $\mathsf{wlp}(\alpha, \mathit{post}_1 \wedge \mathit{post}_2) = \mathsf{wlp}(\alpha, \mathit{post}_1) \wedge \mathsf{wlp}(\alpha, \mathit{post}_2)$ corresponds to the valid dynamic logic formula $[\alpha](\mathit{post}_1 \wedge \mathit{post}_2) \leftrightarrow ([\alpha]\mathit{post}_1) \wedge ([\alpha]\mathit{post}_2)$.
- The relation between the two systems becomes even clearer when considering that the equation $\mathsf{wlp}(\alpha_1; \alpha_2, \mathit{post}) = \mathsf{wlp}(\alpha_1, \mathsf{wlp}(\alpha_2, \mathit{post}))$ corresponds to $[\alpha_1\,;\alpha_2]\mathit{post} \leftrightarrow [\alpha_1][\alpha_2]\mathit{post}$ thanks to the possibility of nested modalities.

A further significant difference is that the wp-calculus analyses the program in a backward direction from the end to the start while transforming the postcondition into its weakest precondition. Dynamic logic provides more flexibility

and permits the implementation of both directions of analysis. In particular, dynamic logic allows one to analyse a program in the forward direction (see Sect. 3.5), while still computing the weakest precondition (and not the strongest postcondition).

For a thorough review of Dijkstra's weakest precondition calculus, as well as a comparison to dynamic logic, we refer to [37]. The parallels between the wp-calculus and dynamic logic were already observed by Harel & Pratt in 1978 [43] who reproved formally many of Dijkstra's results [30].

3.4 Relational Properties

Relational and Hyperproperties. Certain relevant program properties concern not only one program but express a relationship between the effects and results of one program relative to the effects and results of another program. Such properties are called *relational*.[12] A relational property may put two (or more) different programs into relation, thus comparing the effects of two different, independent realisations. But relational properties can also be about *the same* program in which case two different independent runs of a program α are compared. This is called a *hyperproperty* of α [21]. Originally, hyperproperties are for traces but can also be formulated for pre- and post-states. The general pattern for formalising a relational property in dynamic logic is as follows:

$$pre \rightarrow \langle\!|\alpha_1|\!\rangle \langle\!|\alpha_2|\!\rangle ... \langle\!|\alpha_n|\!\rangle post$$

Here, $\langle\!|\alpha|\!\rangle$ is either $[\alpha]$ or $\langle\alpha\rangle$ and the α_i have disjoint program variable sets. For a hyperproperty, one can take identical copies of the same program with renamed variables. The relational precondition *pre* and the relational postcondition *post* connect the state spaces of the different programs in the pre- and post-state. Most relational properties relate two programs ($n = 2$), but there are scenarios relating more than two program executions. Functional properties as defined previously are relational properties for the (corner) case of $n = 1$.

Program Equivalence. The arguably most prominent relational property formulates in its purest form that two programs α and β presented with equal input yield equal output (provided they both terminate):

$$\overline{in}_\alpha = \overline{in}_\beta \;\rightarrow\; [\alpha][\beta](res_\alpha = res_\beta) \tag{2}$$

Here, the equality $\overline{in}_\alpha = \overline{in}_\beta$ correlates the corresponding input variables of the two disjoint name spaces of α and β. An important use case of program equivalence checking is *regression verification*, where a more recent program version replaces a less efficient or buggy older version. In case of a bug fix, one evidently does not want the new version to be equivalent to the old one. Hence, it makes

[12] While sometimes one encounters the term *relational verification*, being relational is not a characteristic of the analysis technique, but one of the property itself. Relational properties can sometimes be encoded as functional properties.

sense to amend the premise of the implication with a relational precondition (excluding the buggy cases) under which equivalence has to hold (*conditional regression verification*). In equation (2), we referred to the corresponding inputs in the two variable sets. It is not always and necessarily clear how the state spaces correspond. It may very well be that the value of one variable in α is represented by two variables in β. Hence, equality (2) can take the relaxed form of a one-to-one relationship between state spaces (*relational regression verification*).

Property (2) encodes "partial equivalence", leaving termination aside. But *mutual termination* [35] can also be formulated in dynamic logic (at least for deterministic languages). A program α terminates under a precondition *pre* in precisely the same cases as the program β iff the following formula is valid:

$$pre \rightarrow \big(\langle\alpha\rangle true \leftrightarrow \langle\beta\rangle true\big)$$

Refinement. A very successful concept [1, 2, 48] in formal design is the notion of *refinement*: An abstract, often nondeterministic state transition description is enriched with details about the realisation and formally shown to be compatible with the abstract description. One program γ *refines* another program α if any effects observable under γ can also be observed under α: No new behaviour is introduced by γ, so all invariants of α hold for γ, too.

$$in_\gamma = in_\alpha \;\rightarrow\; [\gamma]\langle\alpha\rangle(res_\gamma = res_\alpha) \tag{3}$$

Formula (3) closely resembles (2) with the difference that opposing modalities are in use. This is to accommodate the situation that for all observable concrete behaviours there must be an abstract action. To deal with the nondeterminism, this alternation of modalities is necessary. As in the case of program equivalence, data representation is not necessarily identical in α and γ. Hence, the equalities in (3) are usually generalised to *coupling predicates* describing the relation between the abstract and concrete state space.

Secure Information Flow. The best-studied hyperproperty of programs is *non-interference*. Non-interference is an information-flow security property ensuring a program does not leak confidential information. To prove this property, one has to show that the value of program variables holding confidential information does not influence the value of a non-confidential program variable. For example, the following two programs do not satisfy the non-interference property (`l` is a non-confidential, "low" program variable, while `h` denotes a confidential, "high" one):

- `l = h;` is insecure, as the secret `h` is directly leaked to `l`
- `if (h > 0) l = 0; else l = 1;` is insecure, as it leaks the sign of `h` to `l`

As shown in [26], it is possible to express non-interference for a program α directly in dynamic logic as

$$\forall\bar{l}.\exists\bar{r}.\forall\bar{h}.\,\big(\langle\alpha\rangle(\bar{l} \doteq \bar{r})\big) \tag{4}$$

which expresses that the final values $\bar{r}$ of non-confidential program variables $\bar{l}$ only depend on the initial values of the non-confidential variables, but not on the value of the confidential variables $\bar{h}$. This formalisation uses alternating quantification over variables (which is possible in dynamic logic, but not in the Hoare calculus).

Other techniques, such as *self-composition* [8, 25], can be used to express non-interference in Hoare calculus as well as dynamic logic, but require additional encoding effort to allow for independent execution of the same program. A compact representation of this approach in dynamic logic is

$$\bar{l} = \overline{l'} \;\rightarrow\; [\alpha][\alpha'](r = r') \;\;,$$

where program α is identical to α, except that all program variables v in α have been renamed to v'.

The formalisation shown in equation (4) requires providing a witness for the result value in contrast to self-composition. This can be mitigated by delaying the provision using free variables. In both cases, dynamic logic permits modelling more complex information-flow security properties, such as declassification, in a natural manner. For example, some information may be intentionally leaked, such as the average salary of all employees, while the salary of a specific employee is confidential.

Relational Properties of Algorithms. Hyperproperties may formally relate *more than two* program runs. We show two examples and their encoding in dynamic logic:

When determining the winner of an election, different vote counting schemes can be applied. One desirable feature of a voting scheme ω is *separability*, requiring that if a candidate *res* wins in two independent elections, then they also win if the ballot boxes *in* and *in′* are joint (*in″*) [9]:

$$in'' = in \cup in' \rightarrow [\omega][\omega'][\omega''](res = res' \rightarrow res = res'')$$

At times it is important to know whether an implemented routine possesses desirable mathematical properties. For example, when implementing distributed routines using the map-reduce paradigm, the independence of the result from the distribution of the input values over computing nodes depends on the fact that the reducers computing intermediate results are commutative and associative, i.e. that reordering and regrouping partial results does not modify the final result. Associativity of α with two inputs a and b and output *res* can be formulated in dynamic logic as a condition relating four program runs:

$$[\alpha][\alpha'][\alpha''][\alpha''']\big(a = a''' \wedge b = res' \wedge a' = b''' \wedge b' = b'' \wedge a'' = res''' \rightarrow res = res''\big)$$

Relational Reasoning For the verification of relational properties, dedicated inference rules tailored to this scenario can be used to increase efficiency. One

elegant inference rule for loop induction in dynamic logic is

$$\frac{\phi \to [\alpha\ ]\phi}{\phi \to [\alpha^*]\phi}$$

formalising that when ϕ is maintained by a single execution of α, then it is maintained by arbitrarily many iterations of α. This induction rule can be generalised to reason about synchronised loops in two related program runs. The formula ϕ serves as *coupling invariant* describing the relationship between the state spaces of the two program runs under verification. In relational refinement proofs, for example, the following rule can be applied

$$\frac{\phi \to [\alpha\ ]\langle\gamma\ \rangle\phi}{\phi \to [\alpha^*]\langle\gamma^*\rangle\phi}\ .$$

If a single iteration of γ refines a single iteration of α (modulo ϕ), then the nondeterministic iteration γ^* refines α^*.

This relational reasoning allows us, in the case of programs with (lock-step) synchronised loops, to reason with local relational knowledge: It does not matter what the program computes; it is merely interesting how the state relates to the state of the concurrent execution of the other programs. We can focus on the differences between the states rather than describing the states explicitly.

3.5 Symbolic Execution

Symbolic execution [50] is a general program analysis that can be used to prove programs correct. In contrast to Dijkstra's wp-calculus, it executes a given program in a forward-directed manner (applying forward substitutions) to compute the weakest precondition. Besides deductive verification [20, 44], symbolic execution has shown to be advantageous for test generation [18, 50, 51] and debugging [45, 51].

Dynamic logic is a suitable candidate to realise symbolic execution for deductive verification, as well as for test generation and debugging. Dynamic logic permits representing the symbolic states generated during symbolic execution naturally, as well as recording path conditions as preconditions. The nature of logic proof systems, such as sequent or tableaux calculi, is to represent proofs as trees. This renders them ideal to represent the unrolled control-flow (and data) graph obtained by symbolic execution.

The semantics of symbolic execution then serves as a design guideline for developing a deduction system. This becomes obvious when looking at the dynamic logic sequent calculus rule for the conditional statement:

$$\frac{\Gamma, e \Longrightarrow \langle\alpha_1\texttt{;}\ \beta\rangle post, \Delta \qquad \Gamma, \neg e \Longrightarrow \langle\alpha_2\texttt{;}\ \beta\rangle post, \Delta}{\Gamma \Longrightarrow \langle\texttt{if (}e\texttt{) then }\alpha_1\texttt{ else }\alpha_2\texttt{;}\ \beta\rangle post, \Delta}$$

Here, the proof goal in the conclusion splits into two subgoals, one for the case where the guard of the conditional statement evaluates to true and one where

it evaluates to false. This maps branching of the control-flow into branching of the proof tree at the current proof goal.

An advantage of using dynamic logic as a framework to implement symbolic execution is that it can use the information provided by the specified precondition and the accumulated path condition to simplify the state representation and exclude unreachable code paths early on. In semi-automated deductive verification [4], the close relation between symbolic execution trees and proof trees is helpful for proof comprehension because it facilitates navigation in a proof and understanding the proof situation.

3.6 Incorrectness Logic

Recently, P. O'Hearn proposed *incorrectness logic* [60] as a program logic that does *not* formalise that for all states satisfying the precondition, the postcondition holds in the poststate of the execution. Instead, it formalises that *there exists* a state satisfying the precondition such that after execution the postcondition is satisfied. The intended scenario for this specialised logic is to (automatically) reason during bug finding that a potential bug is not spurious, but indeed reachable by at least one program execution.

Incorrectness logic reuses the notation of Hoare triples (with square brackets instead of curly braces), but a triple in it is interpreted as $[\mathit{presumption}]\,\alpha\,[\mathit{result}]$ indicating that the post-relation *result* can be an *under-approximation* of the states reachable via α, starting from states satisfying the *presumption*. In contrast, postconditions in valid Hoare triples are *over-approximations*.

Dynamic logic is expressive enough to model incorrectness logic if one adds the *converse program* α^- inverting a program α to dynamic logic.[13] This operator is well-studied in dynamic logic [74] and has the semantics that pre- and poststates exchange places: $(\alpha^-)^{\mathcal{M}} = \left(\alpha^{\mathcal{M}}\right)^{-1}$. The inverted program α^- can be viewed as executing α backwards. The incorrectness triple $[\phi]\,\alpha\,[\psi]$ holds if the dynamic logic formula $\psi \to \langle\alpha^-\rangle\phi$ is valid.

Interestingly, the *Hoare triple* $\{\phi\}\,\alpha\,\{\psi\}$ corresponds to the validity of the reversed dynamic logic formula $\langle\alpha^-\rangle\phi \to \psi$. This can be read as: The strongest postcondition of α with respect to the precondition ϕ of the Hoare triple implies the postcondition ψ of the Hoare triple.

3.7 Abstract Interpretation

Abstract interpretation [24] is a static analysis technique based on a lattice model for approximation. The rough idea is to use *abstract domains* to approximate the values of (or relations between) program variables for a given program. One can derive an abstract program semantics from these abstract domains for the target programming language. One can then adapt standard analyses like control- or data-flow analyses or deductive reasoning techniques to the abstract semantics. The intention is that with dedicated abstract domains, one can automatically

[13] This is already pointed out in the original publication [60].

derive program properties for a given program with sufficient precision to avoid false positives.

In principle, one can define a dynamic logic calculus variant for each abstract program semantics and thus use deductive reasoning in combination with abstract interpretation. A complementary approach for value abstraction of program variables while reusing the fully precise dynamic logic calculus for the programming language has been developed in [19]. The idea is to use partially interpreted constants that use underspecification to act as representatives for an abstract domain element.

4 Success Story Systems

4.1 The KIV Verification System

Overview. The KIV system[14] [32], in whose design Wolfgang Reif was crucially involved, is a powerful formal verification tool designed to ensure the correctness of software systems.

For specification, KIV combines algebraic data type specifications with different formalisms for describing system behaviour. For the definition of abstract systems, KIV supports Abstract State Machines (ASM) [17]. ASMs consist of abstract programs (rules) that implement the steps of a transition system. Concrete systems to be verified can also be written in Java.

The software development process of KIV is step-wise refinement from abstract specifications to implementations. One can start by verifying an abstract model and then refine it into more concrete implementations, while ensuring that correctness is preserved. KIV supports data refinement as well as system refinement. It offers a graphical user interface for interactive proof development.

Dynamic Logic in KIV. KIV uses higher-order dynamic logic [32], a dynamic logic where the base logic is typed higher-order logic and the modalities can contain abstract imperative programs as well as sequential Java.

The system uses a sequent calculus for its logic that implements wp-style reasoning as well as symbolic execution. It features a powerful simplifier that automatically reduces formulas, and an extensive library of data types with numerous pre-proven theorems.

KIV's Success Stories. KIV was used in various case studies and applications to prove the correctness of real-world systems:

A Prolog compiler that compiles from Prolog to the Warren Abstract Machine was fully verified with KIV using a hierarchy of a dozen refinements [71].

KIV was applied in the area of electronic payment systems to verify a protocol for secure money transfer between Mondex smartcards [40].

[14] www.uni-augsburg.de/de/fakultaet/fai/isse/software/kiv

KIV was used to verify part of Flashix, a file system for flash memory that had been proposed as a case study for Hoares Grand Challenge [31]. The file system was decomposed into a hierarchy of components with eleven levels of refinement.

4.2 The KeY System: Deductive Verification of Java Programs

Overview. The KeY System[15] [3, 4, 10, 11] is a state-of-the-art program verification tool for one of the most widely used programming languages: Java. Its capabilities enable the formal specification and verification of unmodified industrial Java code at source-code level.

In addition to its role as a program verifier, KeY serves as a versatile research platform for implementing various formal methods for Java using the symbolic execution engine of KeY. For instance, KeY has been used to facilitate the generation of test cases with high code coverage [6] and to implement a symbolic-state debugger [45].

The roots of the KeY project trace back to 1999, when the continuous development and refinement of KeY and its verification methodology were started.

Dynamic Logic in KeY. JavaDL is a dynamic logic, where the programs in the modalities are Java code, i.e., executable fragments of Java programs. It is based on a *typed* first-order logic whose type system includes all Java primitive and reference types that are equipped with Java's typing rules. For example, there is a non-rigid function that returns the length of an array `a` in the current execution state as *length*(`a`). JavaDL also permits Java-style syntax like `a.length`.

KeY's deductive verification engine is based on a sequent calculus for JavaDL [13]. The calculus rules perform forward symbolic execution whereby all symbolic paths through a program are explored. Method contracts make verification scalable because one can prove one method at a time to be correct relative to its contract. Contracts do not need to be expressed in dynamic logic, but can be given at the source code level as *Java Modeling Language* annotations [54].

KeY's sequent calculus has rules for the features of Java, including operations on data types, heap operations, object creation, inheritance, polymorphism, method invocation, loops, abrupt termination etc. Since a large number of rules is needed, KeY features a domain-specific textual language (called *taclets*) to add axioms of theories and lemmas, and to define proof rules.

KeY allows both semi-automated (user-guided) and automatic (system-driven) verification of program properties. The tool supports modular reasoning, which means it can verify parts of a program in isolation, making the verification process scalable to larger systems.

KeY's Success Stories. Over the years, a plethora of case studies were conducted, where KeY was used to verify real-world algorithms and data structures; a comprehensive list is on the KeY project website.

[15] www.key-project.org

A verification case study that received much attention is TimSort, an algorithm combining merge and insertion sort. It is prominently used as Java's default for sorting collections of objects. However, that implementation had a bug and crashed for certain large collections. This issue was detected and explained in [28], a fixed version has been presented and verified with KeY in [27].

While the JDK uses TimSort to sort collections of objects, collections of primitive types are sorted using Dual Pivot Quicksort, which is a standard quicksort that partitions into three instead of into two parts. The implementation provided by the JDK has been proven correct in [15], which includes the sortedness property, the permutation property, and the absence of integer overflows.

In [16], the core of the JDK's Identity Hash Map was specified and verified. For that, KeY was used in combination with other JML tools: the bounded model checker JJBMC [12] and OpenJML [23], to exploit the strengths of each of them and jointly verify a large project.

Researchers at CWI showed that Java's LinkedList implementation breaks when lists with more than 2^{31} elements are created [46]. They propose a fixed version and verified it successfully with KeY. This case study shows the capability of KeY to reason about bounded integer data types and handle overflows.

The most recent large case study performed with KeY is the verification of the sorting algorithm in-place super scalar sample sort [14]. This algorithm is efficient on modern machines, as it avoids branch mispredictions, allows high instruction parallelism by reducing data dependencies in the innermost loops, and it is very cache-efficient. This case study shows that with KeY it is possible to verify state-of-the-art sorting algorithms of considerable size (in this case about 900 lines of Java) and complexity *without having to modify the source code.*

4.3 KeYmaera X: A Theorem Prover for Hybrid Systems

Overview. KeYmaera X[16] [34] is a formal verification tool for hybrid systems that combine continuous dynamics (for example, physical processes) and discrete transitions (for example, digital control). Its verification engine is based on differential dynamic logic (d$\mathcal{L}$) [61, 63] to model and verify safety properties of these systems. KeYmaera X is particularly useful for verifying cyber-physical systems, such as autonomous vehicles or medical devices.

The tool is built on top of a small, soundness-critical logic kernel, ensuring the reliability of its proofs, and it is extensible for advanced applications, such as differential game logic [62].

KeYmaera X's proof search is partially automated, it offers tight integration with solvers like Z3 and Wolfram Mathematica for handling real arithmetic. Users can guide proof search interactively by manually applying proof rules, selecting tactics, and providing input such as loop invariants.

Dynamic Logic in KeYmaera X. In differential dynamic logic (d$\mathcal{L}$) the modalities contain hybrid programs. These programs model hybrid systems and

[16] www.keymaerax.org

can describe both continuous changes (using differential equations) and discrete actions (using traditional program constructs like assignments and conditionals). A d$\mathcal{L}$ formula might express that after following the trajectory governed by a system of differential equations, a certain safety condition is guaranteed. For example, consider the d$\mathcal{L}$ formula describing a safety property for a car model [34]:

$$v \geq 0 \wedge A > 0 \;\rightarrow\; [\big((a := A \cup a := 0); \{v' = a\}\big)^*]\; v \geq 0$$

It expresses that a car, when started with non-negative velocity $v \geq 0$ and positive acceleration $A > 0$ (left-hand side of the implication), will always drive forward ($v \geq 0$) after executing $a := A \cup a := 0$ followed by the differential equation $v' = a$ arbitrarily often.

KeYmaera X uses uniform substitution [63] to automatically or interactively prove d$\mathcal{L}$ formulas and, thus, the correctness of hybrid systems. In the uniform substitution framework, variables and formulas can be substituted uniformly within logical rules.

KeYmaera X's Success Stories. Common use cases include autonomous vehicles, aircraft control systems, and robotics.

KeYmaera X has been used to verify collision avoidance systems for autonomous cars. One notable success is the formal verification of the correctness of adaptive cruise control, where KeYmaera X was used to prove that the system would maintain a safe distance from other vehicles under all operating conditions [56]. Another significant application involved automated lane-changing manoeuvres in autonomous cars. The tool verified that under appropriate conditions, the vehicle would change lanes safely without violating safety constraints.

KeYmaera X was involved in the verification of safety properties in robotic systems, especially those operating in dynamic environments. For instance, it has helped ensure that robots navigate safely around obstacles and other moving agents by verifying the correctness of their control algorithms under different scenarios [59].

KeYmaera X has been successfully used in verifying air traffic management systems, particularly the Airborne Collision Avoidance System (ACAS-X), used in airplanes to prevent mid-air collisions [22]. KeYmaera X helped prove that the collision avoidance algorithms work reliably under a wide range of flight scenarios, accounting for the continuous dynamics of aircraft motion and discrete control decisions. Additionally, the tool has been applied in unmanned aerial vehicles (UAVs) for ensuring safe flight path planning.

5 Conclusion

While dynamic logic is well established in theoretical circles as an object of investigation[17], it leads a niche existence in the area of deductive verification:

[17] See, for example, the Dalí workshop series (dblp.org/db/conf/dali/index.html).

The three tools reported in Sect. 4 represent all the major verification system implementations using dynamic logic we are aware of.

This is a pity, because, as we show in the present paper, dynamic logic is extremely versatile:

- It works with different base logics: Propositional, first-order or higher-order, typed or untyped.
- It can be instantiated to a wide range of modelling and real-world programming languages, including hybrid programs.
- It can express proof obligations deriving from different specification paradigms: Refinement, abstract data types, contract-based.
- It can represent different styles of verification: Symbolic execution (unbounded, bounded, concolic), weakest preconditions, abstract interpretation.
- It can express a wide variety of verification scenarios (see Sect. 3): Relational and hyperproperties, program equivalence, information flow, incorrectness logic.

All of this is possible in the same syntactic (multi-modal) and semantic (relational) framework, without the need of *ad hoc* and *meta* constructs like Hoare quadruples, program composition operators, etc. This makes dynamic logic also a good choice to improve interoperability among verification tools. As witnessed by the success stories mentioned in Sect. 4, the generality and versatility of dynamic logic comes without a performance overhead: Deductive verification tools based on dynamic logic are competitive.

For all these reasons, we believe that dynamic logic deserves to be better known and more widely used than it is. We hope that the present article can serve as an inspiration.

Acknowledgements

This work was supported by the DFG projects BE 2334/9-1, BU 2924/3-1, HA 2617/9-1, and UL 433/3-1, the Helmholtz topic Engineering Secure Systems (KASTEL), the Helmholtz pilot program KiKIT, the ATHENE project "Model-centric Deductive Verification of Smart Contracts, and the Swedish Research Council project "Smart Contract Verification".

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139195881
2. Abrial, J., Schuman, S.A., Meyer, B.: Specification language. In: McKeag, R.M., Macnaghten, A.M. (eds.) On the Construction of Programs, pp. 343–410. Cambridge University Press (1980)
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool: Integrating object oriented design and formal verification. Software and System Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x

4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book: From Theory to Practice. No. 10001 in LNCS, Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
5. Ahrendt, W., de Boer, F.S., Grabe, I.: Abstract object creation in dynamic logic – to be or not to be created. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands. LNCS, vol. 5850, pp. 612–627. Springer (2009)
6. Ahrendt, W., Gladisch, C., Herda, M.: Proof-based test case generation. In: Ahrendt et al. [4], chap. 12, pp. 415–451. https://doi.org/10.1007/978-3-319-49812-6_12
7. Apt, K.R., Olderog, E.: Fifty years of hoare's logic. Formal Aspects Comput. **31**(6), 751–807 (2019). https://doi.org/10.1007/S00165-019-00501-3
8. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17,Pacific Grove, CA, USA. pp. 100–114. IEEE Computer Society (2004)
9. Beckert, B., Bormer, T., Kirsten, M., Neuber, T., Ulbrich, M.: Automated verification for functional and relational properties of voting rules. In: Grandi, U., Rosenschein, J.S. (eds.) Sixth International Workshop on Computational Social Choice (COMSOC 2016) (Jun 2016), https://www.irit.fr/COMSOC-2016/proceedings/BeckertEtAlCOMSOC2016.pdf
10. Beckert, B., Bubel, R., Drodt, D., Hähnle, R., Lanzinger, F., Pfeifer, W., Ulbrich, M., Weigl, A.: The Java verification tool KeY: A tutorial. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods, 26th International Symposium, FM 2024. pp. 597–623. Springer (2024)
11. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach – Foreword by K. Rustan M. Leino. No. 4334 in LNCS, Springer (2007)
12. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 60–80. No. 12476 in LNCS, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
13. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Ahrendt et al. [4], chap. 3, pp. 49–106. https://doi.org/10.1007/978-3-319-49812-6_3
14. Beckert, B., Sanders, P., Ulbrich, M.: Formally verifying an efficient sorter. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th Intl. Conf. TACAS, Luxembourg City, Luxembourg. LNCS, Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
15. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 35–48. No. 10712 in LNCS, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
16. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods. pp. 45–62. No. 13274 in LNCS, Springer International Publishing, Cham (2022). https://-doi.org/10.1007/978-3-031-07727-2_4
17. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-level System Design and Analysis. Springer (2003)

18. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—A formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices **10**(6), 234–245 (Jun 1975)
19. Bubel, R., Hähnle, R., Weiss, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F., Bonsangue, M.M., Madelaine, E. (eds.) Post Conf. Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO). LNCS, vol. 5751, pp. 247–277. Springer-Verlag (2009)
20. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing '74, pp. 308–312. Elsevier/North-Holland (1974)
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393, https://-doi.org/10.3233/JCS-2009-0393
22. Cleaveland, R., Mitsch, S., Platzer, A.: Formally verified next-generation airborne collision avoidance games in ACAS X. ACM Trans. Embed. Comput. Syst. **22**(1), 1–30 (2023). https://doi.org/10.1145/3544970
23. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 472–479. No. 6617 in LNCS, Springer, Berlin, Heidelberg (2011). https://-doi.org/10.1007/978-3-642-20398-5_35
24. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth ACM Symposium on Principles of Programming Language, Los Angeles. pp. 238–252. ACM Press, New York (Jan 1977)
25. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Gorrieri, R. (ed.) Workshop on Issues in the Theory of Security, WITS. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS (2003)
26. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) Proc. 2nd International Conference on Security in Pervasive Computing. LNCS, vol. 3450, pp. 193–209. Springer (2005), http://www.springerlink.com/link.asp?id=rdqa8ejctda3yw64
27. De Gouw, S., De Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Automated Reasoning **62**(6) (2019)
28. De Gouw, S., Rot, J., De Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C. (eds.) Proc. 27th Intl. Conf. on Computer Aided Verification (CAV), San Francisco. pp. 273–289. No. 9206 in LNCS, Springer (Jul 2015)
29. Dijkstra, E.W.: Guarded commands, non-determinacy and a calculus for the derivation of programs. In: Shooman, M.L., Yeh, R.T. (eds.) Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975. p. 2. ACM (1975). https://doi.org/10.1145/800027.808417
30. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
31. Ernst, G., Schellhorn G. a nd Haneberg, D., J., P., Reif, W.: Verification of a virtual filesystem switch. In: Proceedings, Verified Software: Theories, Tools, Experiments. LNCS 8164, Springer (2014)
32. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: overview and VerifyThis competition. International Journal on Software Tools for Technology Transfer **17**(6), 677 – 694 (2015). https://doi.org/10.1007/s10009-014-0308-3

33. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences **18**(2), 194–211 (1979). https://doi.org/https://doi.org/10.1016/0022-0000(79)90046-1, https://www.sciencedirect.com/science/article/pii/0022000079900461
34. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE. LNCS, vol. 9195, pp. 527–538. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_36
35. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. In: Manna, Z., Peled, D.A. (eds.) Time for Verification, Essays in Memory of Amir Pnueli. LNCS, vol. 6200, pp. 167–184. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_8
36. Goldblatt, R.: Axiomatising the logic of computer programming, LNCS, vol. 130. Springer (1982)
37. Hähnle, R.: Dijkstra's legacy on program verification. In: Apt, K.R., Hoare, T. (eds.) Edsger Wybe Dijkstra: His Life, Work, and Legacy, pp. 105–140. ACM / Morgan & Claypool (2022). https://doi.org/10.1145/3544585.3544593
38. Hähnle, R., Heisel, M., Reif, W., Stephan, W.: An interactive verification system based on dynamic logic. In: Siekmann, J. (ed.) Proc. 8th Conf. on Automated Deduction CADE, Oxford, UK. LNCS, vol. 230, pp. 306–315. Springer (1986)
39. Hähnle, R., Menzel, W., Schmitt, P.H.: Integrierter Deduktiver Software-Entwurf. Künstliche Intelligenz **12**(4), 40–41 (1998)
40. Haneberg, D., Moebius, N., Reif, W., Schellhorn, G., Stenzel, K.: Mondex: engineering a provable secure electronic purse. Int. J. of Software and Informatics **5**(1), 159184 (2011)
41. Harel, D.: First-Order Dynamic Logic, LNCS, vol. 68. Springer (1979). https://doi.org/10.1007/3-540-09237-4
42. Harel, D., Meyer, A.R., Pratt, V.R.: Computability and completeness in logics of programs (preliminary report). In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proc. 9th Annual ACM Symposium on Theory of Computing, Boulder, Colorado, USA. pp. 261–268. ACM (1977). https://doi.org/10.1145/800105.803416
43. Harel, D., Pratt, V.R.: Nondeterminism in logics of programs. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conf. Record of the Fifth Annual ACM Symp. on Principles of Programming Languages, Tucson, AZ, USA. pp. 203–213. ACM Press, New York, NY (1978). https://doi.org/10.1145/512760.512782
44. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In: Morik, K. (ed.) Proc. 11th German Workshop on Artifical Intelligence. Informatik Fachberichte, vol. 152. Springer (1987)
45. Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (SED): A platform for interactive symbolic execution, debugging, verification and more. STTT **21**(5), 485–513 (Oct 2018)
46. Hiep, H.A., Maathuis, O., Bian, J., Boer, F.S.D., van Eekelen, M.C.J.D., Gouw, S.D.: Verifying OpenJDK's LinkedList using KeY. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 26th Intl. Conf. TACAS, Dublin, Ireland, Part II. pp. 217–234. No. 12079 in LNCS, Springer, Cham (2020)
47. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**(10), 576–580, 583 (Oct 1969)
48. ISO: IEC 13568: 2002: Information technology–Z formal specification notation–Syntax, type system and semantics. Standard, International Organization for Standardization, Geneva, CH (2002)

49. Jónsson, B., Tarski, A.: Boolean algebra with operators I and II. American Journal of Mathematics pp. 73: 891–939 and 74: 129–62 (1951+1952)
50. King, J.C.: A new approach to program testing. In: Shooman, M.L., Yeh, R.T. (eds.) Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975. pp. 228–233. ACM (1975). https://-doi.org/10.1145/800027.808444, https://doi.org/10.1145/800027.808444
51. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (Jul 1976)
52. Kozen, D., Tiuryn, J.: Logics of programs. In: van Leeuwen, J. (ed.) Formal Models and Semantics, pp. 789–840. Handbook of Theoretical Computer Science, Elsevier, Amsterdam (1990). https://doi.org/10.1016/B978-0-444-88074-1.50019-6
53. Kripke, S.: Semantical considerations on modal logic. Acta Philosophica Fennica **16**, 83–94 (1963)
54. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 2013), draft revision 2344
55. Lewis, C.I.: Implication and the algebra of logic. Mind **21**(84), 522–531 (1912), http://www.jstor.org/stable/2249157
56. Lin, Q., Mitsch, S., Platzer, A., Dolan, J.M.: Safe and resilient practical waypoint-following for autonomous vehicles. IEEE Control Syst. Lett. **6**, 1574–1579 (2022). https://doi.org/10.1109/LCSYS.2021.3125717
57. Litvintchouk, S.D., Pratt, V.R.: A proof-checker for dynamic logic. In: Reddy, R. (ed.) Proc. 5th Intl. Joint Conf. on Artificial Intelligence. Cambridge, MA, USA. pp. 552–558. William Kaufmann, San Mateo CA (1977), http://ijcai.org/Proceedings/77-1/Papers/098.pdf
58. Manna, Z., Pnueli, A.: Axiomatic approach to total correctness of programs. Acta Informatica **3**, 243–263 (1974). https://doi.org/10.1007/BF00288637, https://-doi.org/10.1007/BF00288637
59. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. I. J. Robotics Res. **36**(12), 1312–1340 (2017). https://doi.org/10.1177/0278364917733549
60. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020). https://doi.org/10.1145/3371078
61. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
62. Platzer, A.: Differential game logic. ACM Trans. Comput. Log. **17**(1), 1:1–1:51 (2015). https://doi.org/10.1145/2817824
63. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reas. **59**(2) (2017). https://doi.org/10.1007/s10817-016-9385-1
64. Platzer, A.: Logical foundations of cyber-physical systems. Springer (2018). https://doi.org/10.1007/978-3-319-63588-0
65. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (1977). https://doi.org/10.1109/SFCS.1977.32
66. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: 17th Annual Symposium on Foundations of Computer Science (1976). https://doi.org/10.1109/S-FCS.1976.27
67. Reif, W.: Vollständigkeit einer modifizierten Goldblatt-Logik und Approximation der Omegaregel durch Induktion. Master's thesis, Fakultät für Informatik, Universität Karlsruhe (1984)
68. Reif, W.: Korrektheit von Spezifikationen und generischen Moduln. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (1991), https://d-nb.info/931877865

69. Reif, W.: Correctness of generic modules. In: Nerode, A., Taitslin, M.A. (eds.) Logical Foundations of Computer Science, Second Intl. Symp., Tver, Russia. LNCS, vol. 620, pp. 406–417. Springer (1992). https://doi.org/10.1007/BFB0023857
70. Reif, W.: The KIV-Approach to Software Verification. In: Broy, M., Jähnichen, S. (eds.) KORSO: Methods, Languages, and Tools for the Construction of Correct Software, LNCS, vol. 1009, pp. 339–370. Springer (1995). https://-doi.org/10.1007/BFB0015452
71. Schellhorn, G., Ahrendt, W.: The WAM case study: verifying compiler correctness for prolog with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction: A Basis for Applications. Kluwer Academic Publishers (1998)
72. Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., Reif, W.: Development of a verified flash file system. In: Ameur, Y.A., Schewe, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th Intl. Conf., ABZ, Toulouse, France. LNCS, vol. 8477, pp. 9–24. Springer (2014). https://doi.org/10.1007/978-3-662-43652-3
73. van Eijck, J., Stokhof, M.: The gamut of dynamic logic. In: Gabbay, D.M., Woods, J. (eds.) Volume 7: Logic and the Modalities in the Twentieth Century, pp. 499–600. Handbook of the History of Logic, Elsevier, Amsterdam (2006). https://-doi.org/10.1016/S1874-5857(06)80033-6
74. Vardi, M.Y.: The Taming of Converse: Reasoning about Two-way Computations. In: Parikh, R. (ed.) Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings. LNCS, vol. 193, pp. 413–423. Springer (1985). https://doi.org/10.1007/3-540-15648-8_31