Extending Dynamic Logics with First-Class Relational Reasoning

Jian Xiang[1] and Stephen Chong[2]

[1] UNC Charlotte, 8723 Cameron Blvd, Charlotte, NC 28262, USA
[2] Harvard University, 150 Western Ave, Boston, MA 02134, USA

Abstract. Many important properties of computer systems are relational properties, which are often difficult to express and verify. Dynamic logics are well-known formalisms for program verification. We present a general extension, called the REL extension, of dynamic logics to support first-class relational reasoning. The extension provides intuitive syntax to express relational properties, which may be difficult or impossible to express in the host dynamic logic. The REL extension can be instantiated for different host logics. Verifying relational properties expressed by the REL extension can benefit from techniques developed for general relational reasoning and domain-specific relational reasoning, and existing tools developed for the host logic.

We validate the applicability of the REL extension by instantiating it for a well-known dynamic logic: differential dynamic logic ($\mathsf{d}\mathcal{L}$). As a result, the instantiation can express key relational properties that cannot be easily expressed with $\mathsf{d}\mathcal{L}$. We develop an encoding for the instantiation to leverage existing verification tools for $\mathsf{d}\mathcal{L}$. We conduct an experiment on a set of benchmarks, and successfully verify a set of non-trivial relational properties either fully or semi automatically.

Keywords: relational reasoning · dynamic logic · formal verification.

1 Introduction

Many important properties, such as noninterference [39], are relational properties [18], i.e., predicates over sets of pairs of executions. They are fundamental properties but often harder to express and reason about than single-trace properties, as they require reasoning simultaneously about multiple executions. Moreover, there is less tool support for verification of relational properties, compared to single-trace properties.

Dynamic logics are multi-modal logics widely used for verifying single-trace properties of imperative programs [40, 1, 34, 28]. They are specified over a set of programs α and a set of formulas ϕ. Program specifications are stated with modality of necessity $[\alpha]\phi$ that reads "after any execution of α, ϕ is true" and modality of existence $\langle\alpha\rangle\phi$ that reads "after some execution of α, ϕ is true". Different variants of dynamic logics have been introduced and used in different application domains. For example, differential dynamic logic is developed for the safety of cyber-physical systems [37, 35], and linear dynamic logic is often used for AI-based planning [20]. Various tools and techniques have been developed for the verification of formulas expressed with these dynamic logics. However, relational verification is often not well-explored across many of these application domains. It would be beneficial if we can incorporate relational reasoning into these dynamic logics and leverage existing tools for relational verification.

A lightweight yet efficient approach for expressing relational properties is extending the host logic with a *biprogram* construct [38, 6], which explicitly specifies a pair of programs and thus allows reasoning about executions of the pair. We argue that extending dynamic logics with a biprogram provides intuitive and effective support for expressing relational properties. Elements from both programs can be directly used in expressing relational properties. Modalities can be designed to quantify over executions of the program pair. As such, a dynamic logic can be cleanly extended with first-class relational reasoning.

We contribute a general extension, called the REL extension, that builds upon the biprogram construct to support first-class relational reasoning for different dynamic logics. A REL extension has three major components[3]:

– REL programs that specify pairs of programs. A *biprogram* (α, β) can specify a *left* program (i.e., α) and a *right* program (i.e., β), of the host logic.
– REL modalities that express executions of program pairs. They play the same role as modalities in dynamic logics: express different quantifications over executions of program pairs. For example, REL modality of necessity $[\![(\alpha, \beta)]\!] \phi_{\mathfrak{r}}$ expresses that *for all* executions of α and β, $\phi_{\mathfrak{r}}$ holds at their last states.
– REL formulas that directly capture relational properties. For example, formula
$$(\lfloor x \rfloor_{\text{L}} < \lfloor x \rfloor_{\text{R}}) \rightarrow [\![(\alpha, \beta)]\!] (\lfloor x \rfloor_{\text{L}} < \lfloor x \rfloor_{\text{R}})$$
specifies that any pair of executions of α and β preserves the natural order on the values of x. (the projections $\lfloor \cdot \rfloor_{\text{L}}$ and $\lfloor \cdot \rfloor_{\text{R}}$ of variable x respectively refer to its value in the left and right executions).

We design the REL extension in a general and abstract manner, by focusing on the core constructs, i.e., programs and formulas, of all dynamic logics. It can be instantiated for different host dynamic logics to support domain-specific program constructs, i.e., constructs specific to the host logic. We conduct a case study by instantiating the REL extension for differential dynamic logic ($\mathsf{d}\mathcal{L}$) [34, 36], a well-known logic for verifying safety of cyber-physical systems. The instantiation is able to express key relational properties in a succinct way. Another case study on linear dynamic logic on finite traces [20] (LDL$_f$), a logic used by the AI community for reasoning about temporal constraints, can be found in the extended version of this paper.

Verifying REL formulas specified by REL instantiations can benefit from existing verification tools. We introduce an encoding of REL formulas that permit reusing existing tools to verify these formulas, inspired by the technique of self-composition [8], which reduces relational verification of a program to standard

[3] *Color scheme:* we use blue color for syntactic constructs of host logics, and red color for those of the REL extension.

verification of a composition of two copies of the program. We develop a sound and complete encoding for the instantiation for $\mathsf{d}\mathcal{L}$.

The REL extension naturally combines the general program constructs and domain-specific constructs, thereby promoting the verification of relational properties by integrating techniques for both general relational reasoning and domain-specific relational reasoning, i.e., relational reasoning on host logic specific constructs, e.g., $\mathsf{d}\mathcal{L}$ programs modeling system dynamics. To investigate the usefulness of REL extension in promoting verification, we conduct an experiment on the instantiation for $\mathsf{d}\mathcal{L}$. We implement several variants of its encoding and demonstrate that these variants can integrate techniques for general relational reasoning as well as domain-specific relational reasoning. We test these variants on a set of benchmarks and successfully verify all benchmarks using a theorem prover developed for $\mathsf{d}\mathcal{L}$, with little to no manual effort. Some of the benchmarks cannot be automatically verified by prior tools to our knowledge.

Contribution. The key contribution of this paper is the design and validation of a general extension for dynamic logics to support lightweight yet effective first-class relational reasoning. In particular, we make the following contributions:

– A lightweight extension for dynamic logics to support first-class relational reasoning. Using the modalities, the REL extension can naturally and concisely express relational properties involving mixed quantifications. We present a set of practical example properties (Section 3).
– A case study that demonstrates the applicability of the REL extension. The instantiation can express key domain-specific relational properties that are difficult to express with the host logic (Section 4).
– An incrementally improvable encoding of REL formulas that can integrate general relational reasoning and domain-specific reasoning. We experiment with the encoding for $\mathsf{d}\mathcal{L}$ on a set of benchmarks. The results are promising: we can use an existing tool for $\mathsf{d}\mathcal{L}$ to verify non-trivial relational properties, with little to no manual effort. (Section 5).

Section 2 introduces the syntax and semantics of PDL. Section 6 discusses related work and Section 7 concludes.

2 Propositional Dynamic Logic

Propositional dynamic logic (PDL) is a subsystem of most, if not all, dynamic logics. It extends propositional logic with modalities to reason about program executions. The language of regular PDL has expressions of two sorts: (1) propositions or formulas and (2) programs. Its syntax is defined upon Π_0, a set of atomic programs, and Φ_0, a set of atomic propositions. Programs and propositions are mutually inductively defined from the atomic ones as shown in Figure 1. Programs include the operations of Kleene algebra with tests [30]: sequential composition, nondeterministic choice, nondeterministic repetition, and test of a formula. Formulas include the standard propositional connectives and program necessity $[\alpha]\phi$. Program existence $\langle\alpha\rangle\phi$ can be encoded with program necessity and vice versa, e.g., $[\alpha]\phi = \neg\langle\alpha\rangle\neg\phi$. Common abbreviations for logical connectives apply.

The semantics of PDL formulas and programs is interpreted over a Kripke structure $(\mathcal{W}, \mathcal{R}, \mathcal{V})$, where $\mathcal{W}$ is a nonempty set of states, $\mathcal{R}$ is a mapping from the set $\mathit{\Pi}_0$ of atomic programs into binary relations on $\mathcal{W}$, and $\mathcal{V}$ is a mapping from the set $\mathit{\Phi}_0$ of atomic propositions into subsets of $\mathcal{W}$. That is, $\mathcal{R}(P) \subseteq \mathcal{W}\times\mathcal{W}$ for $P \in \mathit{\Pi}_0$, and $\mathcal{V}(A) \subseteq \mathcal{W}$ for $A \in \mathit{\Phi}_0$. $\mathcal{R}$ and $\mathcal{V}$ are extended inductively to give meanings to all programs and formulas of PDL as shown in Figure 1. We write $\omega \models \phi$ if formula ϕ is true at state ω, i.e., $\omega \in \mathcal{V}(\phi)$. We write $[\![\alpha]\!]$ to denote the semantics of α, i.e., if $(\omega, \nu) \in [\![\alpha]\!]$, then there is an execution of α that starts in state ω and ends in state ν.

Program: $\alpha, \beta \ ::= \ P \in \mathit{\Pi}_0 \mid ?\phi \mid \alpha\,;\beta \mid \alpha \cup \beta \mid \alpha^*$
Formula: $\phi, \psi \ ::= \ \top \mid A \in \mathit{\Phi}_0 \mid \neg\phi \mid \phi \wedge \psi \mid [\alpha]\phi$

Semantics of formulas ϕ, ψ

$$\begin{array}{rl}
\omega \models \top & \text{iff } \omega \in \mathcal{W} \\
\omega \models A & \text{iff } \omega \in \mathcal{V}(A) \\
\omega \models \neg\phi & \text{iff } \omega \not\models \phi \\
\omega \models \phi \wedge \psi & \text{iff } \omega \models \phi \text{ and } \omega \models \psi \\
\omega \models [\alpha]\phi & \text{iff } \nu \models \phi \text{ for all states } \nu \text{ with } (\omega,\nu) \in [\![\alpha]\!]
\end{array}$$

Semantics of programs α, β

$$\begin{array}{rl}
[\![P]\!] = & \mathcal{R}(P) \text{ for } P \in \mathit{\Pi}_0 \\
[\![?\phi]\!] = & \{(\omega,\omega) \mid \omega \models \phi\} \\
[\![\alpha\,;\beta]\!] = & \{(\omega,\nu) \mid \exists\,\mu, (\omega,\mu) \in [\![\alpha]\!] \text{ and } (\mu,\nu) \in [\![\beta]\!]\} \\
[\![\alpha \cup \beta]\!] = & [\![\alpha]\!] \cup [\![\beta]\!] \\
[\![\alpha^*]\!] = & [\![\alpha]\!]^* \text{ the transitive, reflexive closure of } [\![\alpha]\!]
\end{array}$$

Fig. 1: Syntax and Semantics of PDL

Relations and states in PDL are *abstract*, i.e., states in PDL are abstract points and atomic programs in PDL are abstract binary relations. Such a level of abstraction lets us focus on the fundamental design of the REL extension.

3 The REL Extension

This section introduces the design of the REL extension in the setting of PDL. We present its syntax and semantics and then showcase example REL formulas.

3.1 Syntax and Semantics

The REL extension extends PDL with programs, modalities, and formulas, as follows. It builds on $\mathit{\Pi}_\mathfrak{r}$, a set of REL atomic programs, and $\mathit{\Phi}_\mathfrak{r}$, a set of REL atomic propositions. REL programs are analogous to PDL, with the addition of a biprogram construct (α, β), where α and β specify two PDL programs to run, respectively, by the *left* and *right* execution. The syntax of REL formulas is also analogous to PDL, with the addition of projection formulas $\lfloor\phi\rfloor_{\mathtt{B}}$, which refers to a PDL formula ϕ in one of the two executions specified by $\mathtt{B} \in \{\mathtt{L}, \mathtt{R}\}$. The REL modality of necessity $[\![\alpha_\mathfrak{r}]\!]\phi_\mathfrak{r}$ can be used to encode the REL modality of existence $\langle\!\langle\alpha_\mathfrak{r}\rangle\!\rangle\phi_\mathfrak{r}$ and vice versa, e.g., $[\![\alpha_\mathfrak{r}]\!]\phi_\mathfrak{r} = \neg\langle\!\langle\alpha_\mathfrak{r}\rangle\!\rangle\neg\phi_\mathfrak{r}$.

The semantics of REL programs and formulas is interpreted over the semantics of REL atomic programs $\mathcal{R}_\mathfrak{r}(P_\mathfrak{r})$ (for $P_\mathfrak{r} \in \mathit{\Pi}_\mathfrak{r}$) and REL atomic propositions $\mathcal{V}_\mathfrak{r}(A_\mathfrak{r})$ (for $A_\mathfrak{r} \in \mathit{\Phi}_\mathfrak{r}$), as well as the semantics of PDL programs and formulas.

$\mathcal{R}_\mathfrak{r}(P_\mathfrak{r})$ is a transition relation between two pairs of states: if $((\omega_{\text{L}}, \omega_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \in \mathcal{R}_\mathfrak{r}(P_\mathfrak{r})$, then an execution of the left program specified by $P_\mathfrak{r}$ runs from ω_{L} to ν_{L}, and an execution of the right program of $P_\mathfrak{r}$ runs from ω_{R} to ν_{R}. $\mathcal{V}_\mathfrak{r}(A_\mathfrak{r})$ is a relation on states, i.e., $\mathcal{V}_\mathfrak{r}(A_\mathfrak{r}) \subseteq \mathcal{W} \times \mathcal{W}$ for $A_\mathfrak{r} \in \Phi_\mathfrak{r}$. Semantics of all REL programs and formulas, as shown in Figure 2, is defined inductively from the semantics of REL atomic programs and formulas, and the projection formulas. A projection formula $\lfloor \phi \rfloor_{\text{B}}$ tests if PDL formula ϕ holds in the state specified by B (left or right). REL modality of necessity $[\![\alpha_\mathfrak{r}]\!] \phi_\mathfrak{r}$ holds in bi-state $(\omega_{\text{L}}, \omega_{\text{R}})$ if and only if formula $\phi_\mathfrak{r}$ holds in any pairs of states that are reachable from $(\omega_{\text{L}}, \omega_{\text{R}})$ by running $\alpha_\mathfrak{r}$. We write $\llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}}$ to denote the semantics of $\alpha_\mathfrak{r}$, and $(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r}$ if the formula $\phi_\mathfrak{r}$ holds in bi-state $(\omega_{\text{L}}, \omega_{\text{R}})$.

REL **program**: $\alpha_\mathfrak{r}, \beta_\mathfrak{r} ::= P_\mathfrak{r} \in \Pi_\mathfrak{r} \mid ?\phi_\mathfrak{r} \mid \alpha_\mathfrak{r} ; \beta_\mathfrak{r} \mid \alpha_\mathfrak{r} \cup \beta_\mathfrak{r} \mid \alpha_\mathfrak{r}^* \mid (\alpha, \beta)$

REL **formula**: $\phi_\mathfrak{r}, \psi_\mathfrak{r} ::= A_\mathfrak{r} \mid \neg\phi_\mathfrak{r} \mid \phi_\mathfrak{r} \wedge \psi_\mathfrak{r} \mid [\![\alpha_\mathfrak{r}]\!] \phi_\mathfrak{r} \mid \lfloor \phi \rfloor_{\text{B}}$ $(\text{B} \in \{\text{L}, \text{R}\})$

Semantics of REL **program** $\alpha_\mathfrak{r}$ $\beta_\mathfrak{r}$

$$\begin{aligned}
\llbracket P_\mathfrak{r} \rrbracket_{\text{RL}} &= \mathcal{R}_\mathfrak{r}(P_\mathfrak{r}) \text{ for } P_\mathfrak{r} \in \Pi_\mathfrak{r} \\
\llbracket ?\phi_\mathfrak{r} \rrbracket_{\text{RL}} &= \{((\omega_{\text{L}}, \omega_{\text{R}}), (\omega_{\text{L}}, \omega_{\text{R}})) \mid (\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r}\} \\
\llbracket \alpha_\mathfrak{r} ; \beta_\mathfrak{r} \rrbracket_{\text{RL}} &= \{((\omega_{\text{L}}, \omega_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \mid \text{exists } \mu_{\text{L}}, \mu_{\text{R}} \text{ that} \\
&\qquad ((\omega_{\text{L}}, \omega_{\text{R}}), (\mu_{\text{L}}, \mu_{\text{R}})) \in \llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}} \text{ and } ((\mu_{\text{L}}, \mu_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \in \llbracket \beta_\mathfrak{r} \rrbracket_{\text{RL}}\} \\
\llbracket \alpha_\mathfrak{r} \cup \beta_\mathfrak{r} \rrbracket_{\text{RL}} &= \{((\omega_{\text{L}}, \omega_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \mid \llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}} \cup \llbracket \beta_\mathfrak{r} \rrbracket_{\text{RL}}\} \\
\llbracket \alpha_\mathfrak{r}^* \rrbracket_{\text{RL}} &= \llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}}^* \text{ the transitive, reflexive closure of } \llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}}, \text{ where } \alpha_\mathfrak{r}^0 \text{ is } (?\top, ?\top) \\
\llbracket (\alpha, \beta) \rrbracket_{\text{RL}} &= \{((\omega_{\text{L}}, \omega_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \mid (\omega_{\text{L}}, \nu_{\text{L}}) \in \llbracket \alpha \rrbracket \text{ and } (\omega_{\text{R}}, \nu_{\text{R}}) \in \llbracket \beta \rrbracket \}
\end{aligned}$$

Semantics of REL **formula** $\phi_\mathfrak{r}$ $\psi_\mathfrak{r}$

Truth of formula $\phi_\mathfrak{r}$ in bi-state $\omega_{\text{L}}, \omega_{\text{R}}$, denoted $(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r}$ is defined as follows:

$$\begin{aligned}
(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} A_\mathfrak{r} &\text{ iff } (\omega_{\text{L}}, \omega_{\text{R}}) \in \mathcal{V}_\mathfrak{r}(A_\mathfrak{r}) \\
(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \neg\phi_\mathfrak{r} &\text{ iff } (\omega_{\text{L}}, \omega_{\text{R}}) \not\models_{\text{RL}} \phi_\mathfrak{r} \\
(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r} \wedge \psi_\mathfrak{r} &\text{ iff } (\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r} \text{ and } (\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \psi_\mathfrak{r} \\
(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} [\![\alpha_\mathfrak{r}]\!] \phi_\mathfrak{r} &\text{ iff } \forall\, \nu_{\text{L}}, \nu_{\text{R}} \text{ that } ((\omega_{\text{L}}, \omega_{\text{R}}), (\nu_{\text{L}}, \nu_{\text{R}})) \in \llbracket \alpha_\mathfrak{r} \rrbracket_{\text{RL}}, (\nu_{\text{L}}, \nu_{\text{R}}) \models_{\text{RL}} \phi_\mathfrak{r} \text{ holds} \\
(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \lfloor \phi \rfloor_{\text{B}} &\text{ iff } \omega_{\text{L}} \models \phi \text{ when } \text{B} = \text{L} \text{ or } \omega_{\text{R}} \models \phi \text{ when } \text{B} = \text{R}
\end{aligned}$$

Fig. 2: Syntax and semantics of REL programs and formulas

3.2 Quantification over Executions

Expressing relational properties often involves quantifications over executions, e.g., for all executions of a program α, there exists an execution of β that gets the same results. We can use REL formulas to easily and *succinctly* express all quantifications over two executions by using (and mixing) the necessity and existence modalities, as shown below. We write σ_α (or σ_β) to denote an execution of program α (or β) and σ_α[-1] to denote the last state of σ_α. The first two entries show how to encode quantifications without alternations. The other four show how to encode quantifications with alternations, by combining the biprogram construct and a special program $?\top$ (i.e., *skip* in common programming languages). For example, *forall, exists*, i.e., $\forall \sigma_\alpha\ \exists \sigma_\beta$ is encoded as $[\![(\alpha, ?\top)]\!] \langle\!\langle (?\top, \beta) \rangle\!\rangle$, where the modality $[\![(\alpha, ?\top)]\!]$ quantifies over executions of program α, and the modality $\langle\!\langle (?\top, \beta) \rangle\!\rangle$ quantifies over executions of program β.

$$\begin{aligned}
\forall \sigma_\alpha \forall \sigma_\beta.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \llbracket (\alpha, \beta) \rrbracket \phi_{\mathfrak{r}} \\
\exists \sigma_\alpha \exists \sigma_\beta.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \langle\!\langle (\alpha, \beta) \rangle\!\rangle \phi_{\mathfrak{r}} \\
\forall \sigma_\alpha \exists \sigma_\beta.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \llbracket (\alpha, ?\top) \rrbracket \langle\!\langle (?\top, \beta) \rangle\!\rangle \phi_{\mathfrak{r}} \\
\forall \sigma_\beta \exists \sigma_\alpha.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \llbracket (?\top, \beta) \rrbracket \langle\!\langle (\alpha, ?\top) \rangle\!\rangle \phi_{\mathfrak{r}} \\
\exists \sigma_\alpha \forall \sigma_\beta.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \langle\!\langle (\alpha, ?\top) \rangle\!\rangle \llbracket (?\top, \beta) \rrbracket \phi_{\mathfrak{r}} \\
\exists \sigma_\beta \forall \sigma_\alpha.((\sigma_\alpha[\text{-}1], \sigma_\beta[\text{-}1]) \models_{\text{RL}} \phi_{\mathfrak{r}}) &\equiv \langle\!\langle (?\top, \beta) \rangle\!\rangle \llbracket (\alpha, ?\top) \rrbracket \phi_{\mathfrak{r}}
\end{aligned}$$

Fig. 3: Encoding of quantifications

Notation: For brevity, we write $\llbracket (\alpha, \beta) \rangle\!\rangle \phi_{\mathfrak{r}}$ and $\langle\!\langle (\alpha, \beta) \rrbracket \phi_{\mathfrak{r}}$ for formulas $\llbracket (\alpha, ?\top) \rrbracket \langle\!\langle (?\top, \beta) \rangle\!\rangle \phi_{\mathfrak{r}}$, and $\langle\!\langle (\alpha, ?\top) \rangle\!\rangle \llbracket (?\top, \beta) \rrbracket \phi_{\mathfrak{r}}$ respectively from now on. We can design other syntactic sugar to further improve the usability of REL.

3.3 Expressing Relational Properties

The REL extensions express relational properties intuitively. We show some interesting examples here. Some examples involve syntax more than PDL, but their meanings can be easily inferred. Note that example properties shown here are general: they can be similarly expressed in most dynamic logics.

Refinement Relation. A program α refines another program β if the behaviors of β subsume those of α. That is, all states reachable from a state ω by following a transition of α could also be reached from ω by following *some* transitions of β. Such a property can be encoded using the $\forall \sigma_\alpha, \exists \sigma_\beta$ quantification:

$$(\lfloor A \rfloor_{\text{L}} \leftrightarrow \lfloor A \rfloor_{\text{R}}) \rightarrow \llbracket (\alpha, \beta) \rangle\!\rangle (\lfloor A \rfloor_{\text{L}} \leftrightarrow \lfloor A \rfloor_{\text{R}})$$

Where programs α and β refer to the same set of atomic propositions and programs. The formula $(\lfloor A \rfloor_{\text{L}} \leftrightarrow \lfloor A \rfloor_{\text{R}})$ is a shorthand for $\bigwedge_{i \in 1 \ldots n} (\lfloor A_i \rfloor_{\text{L}} \leftrightarrow \lfloor A_i \rfloor_{\text{R}})$, which encodes that the left and right states are identical.

Noninterference. Noninterference [39, 25] is a well-known strong information security property that guarantees that public outputs of a system do not reveal any confidential information (i.e., confidentiality), or dually that untrusted inputs of a system do not modify trusted contents (i.e., integrity). Noninterference has various variants, especially in a language that involves nondeterminism. Here, we express a few variants.

First, a common notion of noninterference for confidentiality of a *deterministic* program ensures that attackers who have access to the program's low-security input and output won't be able to infer the program's high-security input. Intuitively, the property states that for any executions of a deterministic program α receiving the same low-security inputs, they should produce the same low-security outputs. Such a noninterference notion is a $\forall\forall$ property:

$$(\lfloor A_{in} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{in} \rfloor_{\text{R}}) \rightarrow \llbracket (\alpha, \alpha) \rrbracket (\lfloor A_{out} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{out} \rfloor_{\text{R}})$$

Where atomic propositions A_{in} and A_{out} represent, respectively, low-security inputs and outputs of α. The modality $\llbracket (\alpha, \alpha) \rrbracket$ ensures that α produces the same low-security output if the same low-security input is received.

Various forms of noninterference can be defined when α has nondeterminism. One of them is *possibilistic noninterference* [39] or *nondeducibility* [2], which states that an attacker cannot infer confidential information *for certain* in a program with non-observable nondeterminism. Intuitively, the attacker cannot observe the nondeterministic choices made by program α, so the attacker is not certain if the information received is confidential. Such a noninterference notion can be expressed with the *forall, exists* quantifications over executions of α, i.e.,

$$(\lfloor A_{in} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{in} \rfloor_{\text{R}}) \rightarrow [\![(\alpha, \alpha) \rangle\!\rangle (\lfloor A_{out} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{out} \rfloor_{\text{R}})$$

To properly reason about two executions in the presence of nondeterminism, some variants of noninterference require (some of) the nondeterminism in the two executions to resolve in the same way [48]. For example, a nondeterministic choice in a program may represent a user's decision, which is assumed to be public input, and so the resolution of the nondeterministic choice should be the same in both executions. Under such circumstances, we can use the REL nondeterministic choice, i.e., $\alpha_{\mathfrak{r}} \cup \beta_{\mathfrak{r}}$, to express that two executions take the same choice. The following formula shows an example of such a noninterference, where $\alpha_{\mathfrak{r}}$ is $((?A_{in}\,;\,?A_{out},\ ?A_{in}\,;\,?A_{out}))$ and $\beta_{\mathfrak{r}}$ is $((?\neg A_{out},\ ?\neg A_{out}))$:

$$(\lfloor A_{in} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{in} \rfloor_{\text{R}}) \rightarrow [\![\alpha_{\mathfrak{r}} \cup \beta_{\mathfrak{r}}]\!] (\lfloor A_{out} \rfloor_{\text{L}} \leftrightarrow \lfloor A_{out} \rfloor_{\text{R}})$$

This formula holds: two executions with the same nondeterministic choice, i.e., either the first branch or the second, would lead to the same value of A_{out}.

Lock-step Properties. Note that the semantics of $\alpha_{\mathfrak{r}}^*$ means the left and right programs specified in $\alpha_{\mathfrak{r}}$ would always loop for the same number of iterations. Combining it with other constructs, we can express interesting relational properties on loops that involve lock-step. For example, the following formula expresses a *relational invariant* of a loop with a biprogram. In particular, an invariant ϕ_{inv} holds at the end of every loop iteration of $(\alpha, \beta)^*$: $\phi_{inv} \rightarrow [\![(\alpha, \beta)]\!] \phi_{inv}$.

Loop Alignments. As shown in prior work, verifying relational properties on loops may require careful *alignment* in executing the left and right loops. The REL extension can readily and clearly express these alignments. Consider the example (DoubleSquare) adapted from [41, 3] on integers x, y, and z:

$$\lfloor x \rfloor_{\text{L}} = \lfloor x \rfloor_{\text{R}} \rightarrow [\![y := 0\,;\ (z := 2x, z := x);\ (body^*, body^*)]\!]$$
$$[\![(?z \leq 0, ?z \leq 0);\ (?\top, y := 2y)]\!] \lfloor y \rfloor_{\text{L}} = \lfloor y \rfloor_{\text{R}}$$

Where $body \equiv ?z \geq 0;\ z := z - 1;\ y := y + x$. An effective alignment for verifying this property can be expressed as a formula that replaces $(body^*, body^*)$ with $(body\,;\,body, body)^*$ which aligns two loop iterations of the left execution with one iteration of the right.

For general $\forall\forall$ formulas of the form $[\![(body^*, body^*)]\!] \phi_{\mathfrak{r}}$, we can express the alignments required for verifying such a formula in a general manner, as follows:

$$[\![(?Gb;\ (body, body) \cup ?Gl;\ (body, \top) \cup ?Gr;\ (\top, body))^*]\!] \phi_{\mathfrak{r}}$$

Where formulas Gb, Gl, and Gr specify the conditions for executing, respectively, both the left and right loops, the left loop only, and the right loop only. For example, we can get an effective alignment for verifying the example above by letting $Gb \equiv \lfloor x \rfloor_{\text{L}} = \lfloor x \rfloor_{\text{R}}$, $Gl \equiv \lfloor x \rfloor_{\text{L}} \geq \lfloor x \rfloor_{\text{R}}$, $Gr \equiv \lfloor false \rfloor_{\text{L}} \wedge \lfloor false \rfloor_{\text{R}}$.

Motion Planning. The REL extension can intuitively express many relational properties on motion planning or path finding, e.g., for driving or robots. For example, an important property in motion planning is finding the optimal path (of all possible paths). Given a control system modeled as α, the following $\exists\forall$

formula specifies that an optimal path exists (the left execution arrives no-later than all possible executions):

$$(\lfloor\phi\rfloor_{\text{L}} \wedge \lfloor\phi\rfloor_{\text{R}}) \rightarrow \langle\!\langle(\alpha,\alpha)]\!](\lfloor\psi\rfloor_{\text{R}} \rightarrow \lfloor\psi\rfloor_{\text{L}})$$

Where $\lfloor\phi\rfloor_{\text{L}} \wedge \lfloor\phi\rfloor_{\text{R}}$ specifies that both executions start at the same location, and $\lfloor\psi\rfloor_{\text{R}} \rightarrow \lfloor\psi\rfloor_{\text{L}}$ specifies that the left execution arrives no-later than other executions of α. We can further express useful variants of this property. For example, an optimal path may exist only if certain actions are triggered:

$$(\lfloor\phi\rfloor_{\text{L}} \wedge \lfloor\phi\rfloor_{\text{R}}) \rightarrow \langle\!\langle((\alpha_1; \beta; \alpha_2), \alpha)]\!](\lfloor\psi\rfloor_{\text{L}} \rightarrow \lfloor\psi\rfloor_{\text{R}})$$

Where β specifies these actions. And α_1 and α_2 respectively model possible actions the system can take before and after β.

4 Case Study: Differential Dynamic Logic

This section presents a case study of the REL extension on differential dynamic logic [34, 37, 36] (d$\mathcal{L}$), a well-known first-order dynamic logic that enables verifying high-level cyber-physical systems models featuring real arithmetic, nondeterminism, and differential equations. Programs in d$\mathcal{L}$ are referred to as *hybrid programs* [37]. They are a formalism for modeling systems that have both continuous and discrete behaviors. Hybrid programs can express continuous evolution (as differential equations) as well as discrete transitions.

Figure 4 shows the syntax of constructs that d$\mathcal{L}$ add over PDL and their semantics. Variables are real-valued and can be deterministically assigned ($x := \theta$, where θ is an arithmetic term) or nondeterministically assigned ($x := *$). The hybrid program $x' = \theta \& \phi$ expresses the continuous evolution of variables x: given the current value of variable x, the system follows the differential equation $x' = \theta$ for some (nondeterministically chosen) amount of time so long as the formula ϕ, the *evolution domain constraint*, holds for all of that time. Note that x can be a vector of variables and then θ is a vector of terms of the same dimension. Atomic formulas of d$\mathcal{L}$ are comparisons of terms, i.e., $(\theta \sim \delta)$.

Similar to PDL, the semantics of d$\mathcal{L}$ [34, 36] is a Kripke semantics where the Kripke model's worlds are the states of the system. Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{V}$ denote the set of variables. A state is a map $\omega : \mathbb{V} \mapsto \mathbb{R}$ assigning a real value $\omega(x)$ to each variable $x \in \mathbb{V}$. The set of all states is denoted by STA.

With d$\mathcal{L}$, we are often interested in formulas of the form $\phi_{pre} \rightarrow [\alpha]\phi_{post}$: if ϕ_{pre} is true then ϕ_{post} holds after any possible execution of α. The hybrid program α often has the form $(ctrl\,;\,plant)^*$, where $ctrl$ models atomic actions of the control system and does not contain continuous parts (i.e., differential equations); and $plant$ models the evolution of the physical environment and has the form of $x' = \theta \& \phi$. That is, the system is modeled as unbounded repetitions of a controller action followed by an update to the physical environment.

Consider, as an example, an autonomous vehicle that needs to stop before hitting an obstacle [37]. Figure 5 shows a d$\mathcal{L}$ model of such an autonomous vehicle. Let d be the vehicle's distance from the obstacle. The desired *safety condition* (ϕ_{post}) is that d is positive, i.e., not hitting the obstacle. Let v and

Semantics of d$\mathcal{L}$ terms $\theta, \delta ::= x \mid c \mid \theta \oplus \delta$

$$\begin{aligned} \omega[\![x]\!] &= \omega(x) \text{ and } \omega[\![c]\!] = c \\ \omega[\![\theta \oplus \delta]\!] &= \omega[\![\theta]\!] \oplus \omega[\![\delta]\!] \ (\oplus \text{ denotes arithmetic for } \oplus \in \{+, \times\}\ ) \end{aligned}$$

Semantics of d$\mathcal{L}$ programs $\alpha, \beta ::= x := \theta \mid x := * \mid x' = \theta \& \phi \mid \cdots$

$$\begin{aligned} [\![x := \theta]\!] &= \{(\omega, \nu) \mid \nu(x) = \omega[\![\theta]\!] \text{ and for all other variables } y \neq x,\ \nu(y) = \omega(y)\} \\ [\![x := *]\!] &= \{(\omega, \nu) \mid \nu(y) = \omega(y) \text{ for all variables } y \neq x\} \\ [\![x' = \theta \& \phi]\!] &= \{(\omega, \nu) \mid \text{exists a solution } \varphi : [0, r] \to \textsc{Sta} \text{ of } x' = \theta \text{ with } \varphi(0) = \omega, \\ &\qquad \varphi(r) = \nu, \text{ and } \varphi(t) \models \phi \text{ for all } t \in [0, r]\} \end{aligned}$$

Semantics of d$\mathcal{L}$ formulas $\phi, \psi ::= \theta \sim \delta \mid \forall x.\ \phi \mid \cdots$

$\omega \models \theta \sim \delta$ iff $\omega[\![\theta]\!] \sim \omega[\![\delta]\!]$ ($\sim$ denotes comparison for $\sim \in \{=, \leq, <, \geq, >\}$)

$\omega \models \forall x.\ \phi$ iff $\nu \models \phi$ for all states ν that agree with ω except for the value of x

Fig. 4: Syntax and semantics of d$\mathcal{L}$ programs and formulas in addition to PDL

a be the vehicle's velocity and acceleration. Let t be the time elapsed since the controller was last invoked. The program *plant* describes how the physical environment evolves over time interval ϵ: distance changes according to $-v$ (i.e., $d' = -v$), velocity changes according to the acceleration (i.e., $v' = a$), and time passes at a constant rate (i.e., $t' = 1$). The differential equations evolve within the time interval $t \leq \epsilon$ and if v is non-negative (i.e., $v \geq 0$).

$$\begin{aligned} \phi_{pre} &\equiv A \geq 0 \land B \geq 0 \land 2Bd > v^2 \\ \phi_{post} &\equiv d > 0 \\ \psi &\equiv 2Bd > v^2 + (A + B)(A\epsilon^2 + 2v\epsilon) \\ accel &\equiv ?\psi\,;\ a := A \\ brake &\equiv a := -B \\ ctrl &\equiv (accel \cup brake)\,;\ t := 0 \\ plant &\equiv d' = -v, v' = a, t' = 1 \& (v \geq 0 \land t \leq \epsilon) \\ \phi_{safety} &\equiv \phi_{pre} \to [(ctrl\,;\ plant)^*]\phi_{post} \end{aligned}$$

Fig. 5: d$\mathcal{L}$ model of an autonomous vehicle

The hybrid program *ctrl* models the vehicle's controller. The vehicle can either accelerate at A or brake at $-B$. The controller chooses nondeterministically between these options. Programs *accel* and *brake* express the controller accelerating or braking. The controller can accelerate only if ψ is true, which captures that the vehicle can accelerate for the next ϵ seconds only if doing so would still allow it to brake in time. The formula to be verified is ϕ_{safety}. Given a precondition ϕ_{pre}, the axioms and proof rules of d$\mathcal{L}$ can be used to prove the safety condition ϕ_{post}. The theorem prover KeYmaera X [23] provides tool support.

4.1 d$\mathcal{L}_{\textsc{rel}}$: A REL Instantiation for d$\mathcal{L}$

We build d$\mathcal{L}_{\textsc{rel}}$, a REL instantiation for d$\mathcal{L}$. d$\mathcal{L}_{\textsc{rel}}$ instantiates the atomic programs and formulas of the REL extension with constructs designed for d$\mathcal{L}$. The syntax and semantics of these constructs are shown in Figure 6. (We omit the other constructs shown in Figure 2.) The relational atomic programs, i.e., $P_{\mathfrak{r}}$, in d$\mathcal{L}_{\textsc{rel}}$ are a relational version of deterministic assignment ($x := \theta$), nondeterministic assignment ($x := *$), and continuous evolution ($x' = \theta \& \phi$). The relational atomic formulas, i.e., $A_{\mathfrak{r}}$, in d$\mathcal{L}_{\textsc{rel}}$ are comparisons of d$\mathcal{L}_{\textsc{rel}}$ terms, i.e., $\theta_{\mathfrak{r}} \sim \delta_{\mathfrak{r}}$. A d$\mathcal{L}_{\textsc{rel}}$ term can be either a *projection* term: $\lfloor \theta \rfloor_{\mathtt{B}}$, which refers to a d$\mathcal{L}$ term θ in one of the two executions specified by $\mathtt{B} \in \{\mathtt{L}, \mathtt{R}\}$, or an arithmetic operation of two d$\mathcal{L}_{\textsc{rel}}$ terms, i.e., $\theta_{\mathfrak{r}} \oplus \delta_{\mathfrak{r}}$.

Semantics of d$\mathcal{L}_{\text{REL}}$ term: $\theta_{\mathfrak{r}},\ \delta_{\mathfrak{r}} ::= \lfloor\theta\rfloor_{\mathsf{B}} \mid \theta_{\mathfrak{r}} \oplus \delta_{\mathfrak{r}}$

$$(\omega_{\text{L}},\omega_{\text{R}})[\![\lfloor\theta\rfloor_{\mathsf{B}}]\!]_{\text{RL}} = \omega_{\text{L}}[\![\theta]\!] \text{ if } \mathsf{B}=\mathsf{L} \text{ or } \omega_{\text{R}}[\![\theta]\!] \text{ if } \mathsf{B}=\mathsf{R}$$

$$(\omega_{\text{L}},\omega_{\text{R}})[\![\theta_{\mathfrak{r}} \oplus \delta_{\mathfrak{r}}]\!]_{\text{RL}} = (\omega_{\text{L}},\omega_{\text{R}})[\![\theta_{\mathfrak{r}}]\!]_{\text{RL}} \oplus (\omega_{\text{L}},\omega_{\text{R}})[\![\delta_{\mathfrak{r}}]\!]_{\text{RL}} \ \ (\oplus \text{ denotes arithmetic for } \oplus \in \{+,\times\}$$

Semantics of d$\mathcal{L}_{\text{REL}}$ program: $\alpha_{\mathfrak{r}},\ \beta_{\mathfrak{r}} ::= x := \theta \mid x := * \mid x' = \theta \,\&\phi \mid \cdots$

$$[\![x := \theta]\!]_{\text{RL}} = [\![(x := \theta, x := \theta)]\!]_{\text{RL}}$$

$$[\![x := *]\!]_{\text{RL}} = [\![(x := *, x := *)\,;\ ?(\lfloor x\rfloor_{\text{L}} = \lfloor x\rfloor_{\text{R}})]\!]_{\text{RL}}$$

$$[\![x' = \theta \,\&\phi]\!]_{\text{RL}} = [\![(t := 0, t := 0)\,;\, (x' = \theta, t' = 1\&\phi, x' = \theta, t' = 1\&\phi)\,;\, ?(\lfloor t\rfloor_{\text{L}} = \lfloor t\rfloor_{\text{R}})]\!]_{\text{RL}}$$

Semantics of d$\mathcal{L}_{\text{REL}}$ formula: $\phi_{\mathfrak{r}},\ \psi_{\mathfrak{r}} ::= \theta_{\mathfrak{r}} \sim \delta_{\mathfrak{r}} \mid \cdots$

$$(\omega_{\text{L}},\omega_{\text{R}}) \models_{\text{RL}} \theta_{\mathfrak{r}} \sim \delta_{\mathfrak{r}} \text{ iff } (\omega_{\text{L}},\omega_{\text{R}})[\![\theta_{\mathfrak{r}}]\!]_{\text{RL}} \sim (\omega_{\text{L}},\omega_{\text{R}})[\![\delta_{\mathfrak{r}}]\!]_{\text{RL}} \ (\sim \text{ is arithmetic comparison of } \sim)$$

Fig. 6: Syntax and semantics of d$\mathcal{L}_{\text{REL}}$ programs and formulas in addition to REL

Semantically, program $[\![x := \theta]\!]_{\text{RL}}$ indicates that the same deterministic assignment is run by the left and right executions. The program $[\![x := *]\!]_{\text{RL}}$ ensures both executions having the same value for variable x. $[\![x' = \theta \,\&\phi]\!]_{\text{RL}}$ enforces a constraint that the *durations* used by the physical evolution in both executions are the same (i.e., $?(\lfloor t\rfloor_{\text{L}} = \lfloor t\rfloor_{\text{R}})$ where t is a fresh continuous variable added into both executions). Such a design is useful since we often want to compare executions of two systems only if they execute for the same period of duration.

$$\begin{aligned}
\psi &\equiv 2Bd > v^2 + (A+B)(A\epsilon^2 + 2v\epsilon)\\
accel &\equiv (?\psi, ?\psi)\,;\ a := A\\
brake &\equiv a := -B\\
ctrl_t &\equiv (\quad (?temp > T\,;\, thermo := -1)\\
&\qquad \cup (?temp < T\,;\, thermo := 1)\\
&\qquad \cup (?temp = T) \quad)\\
ctrl'_t &\equiv (\quad (?temp > T\,;\, thermo := -2)\\
&\qquad \cup (?temp < T\,;\, thermo := 2)\\
&\qquad \cup (?temp = T) \quad)\\
ctrl_v &\equiv (accel \cup brake)\,;\ t := 0\\
plant &\equiv d' = -v, v' = a, t' = 1, temp' = thermo\\
&\qquad \&\ (v \geq 0 \wedge t \leq \epsilon)\\
\phi_{\mathfrak{r}} &\equiv (\lfloor d\rfloor_{\text{L}} = \lfloor d\rfloor_{\text{R}} \wedge \lfloor v\rfloor_{\text{L}} = \lfloor v\rfloor_{\text{R}})\\
\phi_{robust} &\equiv \phi_{\mathfrak{r}} \rightarrow [\![((ctrl_t, ctrl'_t);\ ctrl_v;\ plant)^*]\!] \phi_{\mathfrak{r}}
\end{aligned}$$

Fig. 7: A d$\mathcal{L}_{\text{REL}}$ example on robustness

Figure 7 shows examples of d$\mathcal{L}_{\text{REL}}$ programs and formulas adapted from existing work [48]. It presents a design of an autonomous vehicle with velocity control and interior temperature control. Its velocity control is the same as the example presented in Figure 5. For temperature control, the vehicle detects the interior temperature ($temp$), and then chooses one of the two control modes, specified respectively in program $ctrl_t$ and $ctrl'_t$. Here, both modes compare the current temperature with a target temperature T, and then set the thermostat accordingly. The modes differ only in the values of $thermo$. The temperature changes according to $thermo$.

A system designer may want a *robust* velocity control, i.e., the choice of modes for temperature control won't interfere with the vehicle's control of velocity. This relational property is expressed as a d$\mathcal{L}_{\text{REL}}$ formula at the last line (i.e., ϕ_{robust}). Intuitively, the formula says for two runs of the vehicle, that have different modes of temperature control, if the vehicle starts with the same position and velocity (the premise $\phi_{\mathfrak{r}}$ of the implication), makes the same control decisions for acceleration and brake ($ctrl_v$), and runs for the same duration ($plant$), it would end with the same position and velocity (the conclusion $\phi_{\mathfrak{r}}$ of the implication). The validity of this formula suggests that the vehicle has robust velocity control.

Expressing such a relational property with $\mathsf{d}\mathcal{L}_{\text{REL}}$ is straightforward and much more succinct than the original version [48].

We investigate the expressive power of $\mathsf{d}\mathcal{L}_{\text{REL}}$ and prove that it is equally expressive as $\mathsf{d}\mathcal{L}$, stated with the following theorem:

Theorem 1. $\mathsf{d}\mathcal{L}_{\text{REL}}$ *and* $\mathsf{d}\mathcal{L}$ *are equally expressive.*

The theorem holds as every $\mathsf{d}\mathcal{L}$ formula can be encoded as an equivalent $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula and vice versa. We prove this equivalence by constructing a sound and complete encoding of $\mathsf{d}\mathcal{L}_{\text{REL}}$ in $\mathsf{d}\mathcal{L}$ in Section 5. Such a $\mathsf{d}\mathcal{L}_{\text{REL}}$ design promotes verification, as all $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas can be verified using the tools for $\mathsf{d}\mathcal{L}$ and leverage their future advances.

Note that it's possible to make a different design so $\mathsf{d}\mathcal{L}_{\text{REL}}$ is more expressive. For example, we could introduce new atomic relational program constructs that cannot be expressed with $\mathsf{d}\mathcal{L}$. For example, recent work [14, 49] introduce a notion of simulation distance in the setting of $\mathsf{d}\mathcal{L}$, a relational property that quantitatively captures the behavior distance between two $\mathsf{d}\mathcal{L}$ programs. The notion requires calculating the *infimum* of distances between a reachable state of one program to the set of reachable states of another program. We can add a new atomic relational formula to help express such a relational property, and such a atomic construct cannot be expressed with $\mathsf{d}\mathcal{L}$ due to the infimum calculation.

5 Verification Techniques

Verifying REL formulas benefits from existing tools. To explore this benefit, we introduce an encoding that transforms REL formulas into formulas of the host logic, so existing tools and techniques developed for the host logic can be reused. The encoding can integrate verification techniques for general relational reasoning as well as domain-specific reasoning. Thanks to the integration, it promotes automated verification of relational properties, especially domain-specific ones. To demonstrate its usefulness, we conduct an experiment with the encoding for $\mathsf{d}\mathcal{L}_{\text{REL}}$ on a set of benchmarks. The results show that we can verify the benchmarks with an existing tool developed for $\mathsf{d}\mathcal{L}$, with little to no manual effort.

5.1 Host Logic Encoding

The encoding is inspired by self-composition [22, 8, 45, 33], a proof technique often used for proving noninterference for deterministic programs. We explain self-composition first, and then introduce the encoding in the setting of $\mathsf{d}\mathcal{L}_{\text{REL}}$.

Self-Composition. To develop an intuition for how the self-composition technique is used to prove noninterference, consider the problem of checking whether low-security outputs of a deterministic program reveal high-security inputs. Construct two copies of the program, renaming the program variables so that the variables in the two copies are disjoint. Set the low-security inputs in both copies to identical values but allow the high-security inputs to take different values.

Now, sequentially compose these two programs together. If the composed program can terminate in a state where the corresponding low-security outputs differ, then the original program does not satisfy noninterference; conversely, if in all executions of the composed program, the low-security outputs are the same, then the original program satisfies noninterference. Intuitively, the composition of the two copies allows a single program to represent two executions of the original program, reducing checking a relational property of the original problem to checking a safety property of the composed program.

Using the insights, our encoding for $\mathsf{d}\mathcal{L}_{\text{REL}}$ builds on two steps: (1) renaming a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula into an equivalent one whose left and right programs use disjoint variables, and (2) making a composition of the two programs.

Renaming $\mathsf{d}\mathcal{L}_{\text{REL}}$ Formulas. To help with renaming, we define *renaming functions* that map all variables accessible by the *right* execution/program to fresh variables. We write $\xi(\phi_{\mathfrak{r}})$ for the formula identical to $\phi_{\mathfrak{r}}$ but whose variables accessible by the right execution have been renamed according to ξ. Renaming functions similarly apply to $\mathsf{d}\mathcal{L}_{\text{REL}}$ programs. We also write $\xi(\omega)$ for the state identical to ω but whose domain variables have been renamed according to ξ.

A translation from $\mathsf{d}\mathcal{L}_{\text{REL}}$ to $\mathsf{d}\mathcal{L}$. We develop a function π that will be used to transform renamed $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas, i.e., $\xi(\phi_{\mathfrak{r}})$, into an equivalent $\mathsf{d}\mathcal{L}$ formula. A key step of π is to convert the biprogram construct (α, β) into a composition of α and β. Also, it directly extracts the contents from all projection constructs, e.g., $\lfloor \phi \rfloor_{\text{B}}$ and $\lfloor \theta \rfloor_{\text{B}}$. The π is defined as follows:

Definition 1 (π for $\mathsf{d}\mathcal{L}_{\text{REL}}$). For a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula $\phi_{\mathfrak{r}}$ whose variables for the left and right executions are disjoint (i.e., $\text{VAR}_{\text{L}}(\phi_{\mathfrak{r}}) \cap \text{VAR}_{\text{R}}(\phi_{\mathfrak{r}}) = \varnothing$), a function π that can transform a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula $\phi_{\mathfrak{r}}$ (or a $\mathsf{d}\mathcal{L}_{\text{REL}}$ program $\alpha_{\mathfrak{r}}$) to an equivalent $\mathsf{d}\mathcal{L}$ formula (or program), is defined inductively as follows.

$$\begin{aligned} \pi(\theta_{\mathfrak{r}} \sim \delta_{\mathfrak{r}}) &= \pi(\theta_{\mathfrak{r}}) \sim \pi(\delta_{\mathfrak{r}}) & \pi(\alpha_{\mathfrak{r}}\, ;\ \beta_{\mathfrak{r}}) &= \pi(\alpha_{\mathfrak{r}})\, ; \pi(\beta_{\mathfrak{r}}) \\ \pi(\neg \phi_{\mathfrak{r}}) &= \neg \pi(\phi_{\mathfrak{r}}) & \pi(\alpha_{\mathfrak{r}} \cup \beta_{\mathfrak{r}}) &= \pi(\alpha_{\mathfrak{r}}) \cup \pi(\beta_{\mathfrak{r}}) \\ \pi(\phi_{\mathfrak{r}} \wedge \psi_{\mathfrak{r}}) &= \pi(\phi_{\mathfrak{r}}) \wedge \pi(\psi_{\mathfrak{r}}) & \pi(?\phi_{\mathfrak{r}}) &= ?\pi(\phi_{\mathfrak{r}}) \\ \pi([\![\alpha_{\mathfrak{r}}]\!]\phi_{\mathfrak{r}}) &= [\pi(\alpha_{\mathfrak{r}})]\pi(\phi_{\mathfrak{r}}) & \pi(\alpha^*_{\mathfrak{r}}) &= (\pi(\alpha_{\mathfrak{r}}))^* \\ \pi(\lfloor \phi \rfloor_{\text{B}}) &= \phi & \pi((\alpha, \beta)) &= \alpha\, ; \beta \end{aligned}$$

And $\pi(\theta_{\mathfrak{r}})$ on terms is inductively defined:

$$\pi(\lfloor \theta \rfloor_{\text{B}}) = \theta \qquad\qquad \pi(\theta_{\mathfrak{r}} \oplus \delta_{\mathfrak{r}}) = \pi(\theta_{\mathfrak{r}}) \oplus \pi(\delta_{\mathfrak{r}})$$

With a renaming function ξ, we can encode a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula $\phi_{\mathfrak{r}}$ as $\pi(\xi(\phi_{\mathfrak{r}}))$.

Theorem 2 (Soundness and completeness of the encoding for $\mathsf{d}\mathcal{L}_{\text{REL}}$). *For states ω_{L} ω_{R}, a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula $\phi_{\mathfrak{r}}$, and a renaming function ξ for $\phi_{\mathfrak{r}}$,*

$$(\omega_{\text{L}}, \omega_{\text{R}}) \models_{\text{RL}} \phi_{\mathfrak{r}} \text{ iff } ((\omega_{\text{L}} \Downarrow \text{VAR}_{\text{L}}(\phi_{\mathfrak{r}})) \otimes (\xi(\omega_{\text{R}} \Downarrow \text{VAR}_{\text{R}}(\phi_{\mathfrak{r}})))) \models \pi(\xi(\phi_{\mathfrak{r}}))$$

Where $\omega \Downarrow V$ denotes the projection of state ω on a set of variables V: the map $\{x \mapsto \omega(x)\}$ for all $x \in V$, and $\otimes$ means the join of two non-overlapping states.

The proof can be done by simultaneous induction on $\phi_{\mathfrak{r}}$ and $\alpha_{\mathfrak{r}}$.

Verifying $\mathsf{d}\mathcal{L}_{\text{REL}}$ Formulas with existing tools. With Theorem 2, we can verify a $\mathsf{d}\mathcal{L}_{\text{REL}}$ formula $\phi_{\mathfrak{r}}$ by verifying a $\mathsf{d}\mathcal{L}$ formula $\pi(\xi(\phi_{\mathfrak{r}}))$, which can be done with the theorem prover KeYmaera X [23]. We encode the example shown in Figure 7 (ϕ_{robust} in particular) as a $\mathsf{d}\mathcal{L}$ formula $\pi(\xi(\phi_{robust}))$, which KeYmaera X is able to prove fully automatically.

Note that π *sequentially* composes the left and right programs, i.e., $\pi((\alpha, \beta)) = \alpha ; \beta$. Though the encoding based on π is sound and complete, it may be challenging to directly verify certain $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas with this encoding. We discuss next how to improve π by integrating techniques for both general relational reasoning and domain-specific relational reasoning.

5.2 Integrating Techniques in the Encoding

A REL instantiation like $\mathsf{d}\mathcal{L}_{\text{REL}}$ naturally blends general relation constructs and domain-specific constructs, e.g., $[\![(x'=\theta\&\phi, y'=\delta\&\psi)\rangle\!\rangle(\lfloor x\rfloor_{\text{L}} \leq \lfloor y\rfloor_{\text{R}})$ expresses a *forall, exists* property on two differential equations. The integration opens great opportunities to utilize advances in both general relational reasoning and domain-specific relational reasoning. Using $\mathsf{d}\mathcal{L}_{\text{REL}}$ as an example, we show how to improve the verification power by retrofitting the encoding π with various techniques.

In particular, we develop four variants of π, each with increasing verification power, achieved by incorporating additional *heuristics* in π. We either develop these heuristics or adapt them from prior work. We test these variants on a set of benchmarks, by (1) applying these variants on the benchmarks to generate $\mathsf{d}\mathcal{L}$ formulas, and then (2) use the theorem prover KeYmaera X to verify the generated formulas. Figure 8 shows the benchmarks and the results.

Benchmarks. The benchmarks include a set of example properties from this paper and some existing work. In particular, we collect many motivating examples from [3], as it focuses on Kleene Algebra with Tests, which is directly relevant to dynamics logics. We are able to verify all examples from [3], except for two examples that require native support of integers (Ex.2.2) and arrays (Ex.2.3), which are not natively supported by KeYmaera X. In addition, we verify a few other representative and non-trivial case studies from existing work.

We elaborate on the four variants of π and their corresponding results. Note that, as shown in Figure 8, the default π can automatically verify three of these benchmarks.

Benchmarks \ Variants	π	π_{align}	$\pi_{\forall\exists}$	π_{ode}	π_{syn}
Fig.7 (Fig.8 in [48])	A	A	A	A	A
Ex.2.1 in [3]		S	S	S	S
Ex.2.4 in [3]	A	A	A	A	A
Ex.7.4 in [3]	A	A	A	A	A
Ex.7.5 in [3]			A	A	A
Ex.7.6 in [3]			A	A	A
Ex.1				A	A
Fig.8 in [49]				S	S
Cs.1 in [29]					UA
Cs.2 in [29]					UA

Fig. 8: Using variants of π to verify non-trivial relational properties. For notation, *Ex.* and *Cs.* respectively denote the *example properties* and *case studies* in the cited work. Table cells indicate how much automation KeYmaera X can verify the generated $\mathsf{d}\mathcal{L}$ formulas. Notations A, S, and UA mean, respectively, fully automated, automated with less than 3 manual proof steps, and automated after user inputs.

π_{align}: π + **heuristics on alignment**. As shown by existing work [3, 41], verifying relational properties often requires effective *alignment* of programs and computations so that relational invariants can be easier specified. Consider the following property from existing work [3] written in $\mathsf{d}\mathcal{L}_{\text{REL}}$:

$$\lfloor n\rfloor_{\text{L}}{=}\lfloor n\rfloor_{\text{R}}{\to}[\![\, i := n\,;\ r := 1\,]\!][\![\,(body^*, body^*);\ (?i = 0, ?i = 0)\,]\!]\lfloor r\rfloor_{\text{L}} = \lfloor r\rfloor_{\text{R}}$$

Where $body \equiv\ ?i \neq 0;\ r := r \times i;\ i := i - 1$. An effective alignment for verifying this property is to rewrite $(body^*, body^*)$ into $(body, body)^*$ which run the left and right executions in lockstep. We develop π_{align} that adds into π a heuristic that encodes a proof rule from existing work (Rule (3) from [3]) that soundly performs such a rewrite. Apply π_{align} to the property above produces a $\mathsf{d}\mathcal{L}$ formula that can be verified with KeYmaera X with only one manual step.

$\pi_{\forall\exists}$: π_{align} + $\forall\exists$ **heuristics**. Certain $\forall\exists$ properties on loops can be reduced to $\forall\forall$ properties for easier verification. In particular, the following reasoning is sound, as the lockstep of $(\alpha, \beta)^*$ presents a witness for the $\forall\exists$ property:

$$\textit{if }\ \phi_{\mathfrak{r}}{\to}[\![\,(\alpha,\beta)^*\,]\!]\psi_{\mathfrak{r}}\ \textit{ and }\ \phi_{\mathfrak{r}}{\to}\langle\!\langle\,(?\top,\beta)^*\,\rangle\!\rangle\top\ \textit{ then }\ \phi_{\mathfrak{r}}{\to}[\!\langle\,(\alpha^*,\beta^*)\,\rangle\!\rangle\psi_{\mathfrak{r}}$$

Here, the condition $\phi_{\mathfrak{r}}{\to}\langle\!\langle\,(?\top,\beta)^*\,\rangle\!\rangle\top$ ensures β has valid executions from $\phi_{\mathfrak{r}}$.

This rule can be generalized to support reasoning with more program constructs, such as $[\!\langle\,(\alpha_1;\ \alpha^*;\ \alpha_2, \beta_1;\ \beta^*;\ \beta_2)\,\rangle\!\rangle\psi_{\mathfrak{r}}$.

We develop $\pi_{\forall\exists}$ by adding these $\forall\exists$ rules above into π_{align}, and test it on benchmarks from existing work (Ex.7.5 and Ex.7.6 from [3]). KeYmaera X is able to verify the generated $\mathsf{d}\mathcal{L}$ formulas automatically.

π_{ode}: $\pi_{\forall\exists}$ + **heuristics on dynamics**. In addition to integrating general relational reasoning (showcased in variants π_{align} and $\pi_{\forall\exists}$), we can integrate domain-specific relational reasoning. For example, consider the following example on a relationship between two dynamics of exponential decay:

$$\lfloor x\rfloor_{\text{L}}{\geq}\lfloor x\rfloor_{\text{R}} \wedge \lfloor x\rfloor_{\text{L}} \geq 0 \wedge A > 0 \wedge A \geq B{\to}[\!\langle\,(x' = A \times x, x' = B \times x)\,\rangle\!\rangle\lfloor x\rfloor_{\text{L}}{\geq}\lfloor x\rfloor_{\text{R}} \quad (1)$$

This property cannot be verified with KeYmaera X using the variants above. However, this $\forall\exists$ property should hold since the x value of the right dynamic would always be lower if *the two dynamics evolve for the same duration of time*. We can derive heuristics based on this observation. Consider a formula of the form $\phi_{\mathfrak{r}}{\to}[\!\langle\,(x'{=}\,\theta\&\phi_1, y'{=}\,\delta\&\phi_2)\,\rangle\!\rangle\psi_{\mathfrak{r}}$ (this formula has been properly renamed, so the left and right programs refer to disjoint sets of variables). After applying the π function, we get a $\mathsf{d}\mathcal{L}$ formula of the form: $\phi{\to}[x'{=}\,\theta\&\phi_1]\langle y'{=}\,\delta\&\phi_2\rangle\psi$, which can be reasoned with the following heuristic:

$$\textit{if }\ \phi{\to}[x'{=}\,\theta, y'{=}\,\delta\&(\phi_1 \wedge \phi_2)]\psi\ \textit{ and }\ \phi{\to}[x'{=}\,\theta, y'{=}\,\delta](\phi_1 \to \phi_2)$$
$$\textit{then }\ \phi{\to}[x'{=}\,\theta\&\phi_1]\langle y'{=}\,\delta\&\phi_2\rangle\psi$$

The first condition of this heuristic merges two dynamics into one and forces the two dynamics to evolve for the same duration of time. A formula of the form $\phi{\to}[x'{=}\,\theta, y'{=}\,\delta\&(\phi_1 \wedge \phi_2)]\psi$ is often easier to prove. The second condition on evolution constraints indicates there exists executions of the right dynamics if the left has valid executions. The second condition is needed to ensure *there exists an execution of* β for a $[\!\langle\,(\alpha,\beta)\,\rangle\!\rangle\phi_{\mathfrak{r}}$ modality.

We develop π_{ode} by adding this heuristic into $\pi_{\forall\exists}$, and we manage to automatically verify EQ.1 in KeYmaera X. In addition to Eq.1, we apply π_{ode} to a non-trivial case study on a water tank from existing work [49]. Its main property is a $\forall\exists$ one on a system-level model of the water tank. The function π_{ode} first applies the heuristic added in $\pi_{\forall\exists}$, and then applies the heuristic added in π_{ode}. We manage to verify the $\mathsf{d}\mathcal{L}$ formula generated by π_{ode} on this property with only one manual proof step.

π_{syn}: π_{ode} + User Inputs. We further investigate the utility of the encoding by adding a heuristic derived from recent work on relational verification of $\forall\forall$ properties on dynamics (Thm.24[29]). The heuristic aims to reason with $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas of the form: $\phi_{\mathfrak{r}} \rightarrow [\![(x' = \theta \& \phi_1, y' = \delta \& \phi_2)]\!] \psi_{\mathfrak{r}}$. However, instead of fixing the duration passed by the left and right dynamics as in π_{ode}, this heuristic asks for user inputs that would allow the two dynamics to *synchronize* at different time points, e.g., compare the values of x and y after the left and right dynamics evolve for different durations.

We develop a variant π_{syn} that asks users to provide inputs when it sees appropriate $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas, and then produces corresponding $\mathsf{d}\mathcal{L}$ formulas. We test π_{syn} with two case studies from [29]: after obtaining the correct inputs, π_{syn} produces $\mathsf{d}\mathcal{L}$ formulas that can be automatically verified by KeYmaera X.

Discussion. Our experiments demonstrate the feasibility of integrating techniques into the encoding, thereby promoting automated verification of relational properties. Note that we manually develop all these variants of π. How to systematically and automatically integrate different techniques is an interesting and useful future work. The main difficulty is, given an input REL formula, to identify the heuristics to use in its transformation. We plan to develop algorithms to automate the selection of heuristics.

6 Related Work

BiKAT.. The closest related work is the BiKAT [3], an extension of KAT to help reason with the alignments of relational properties through equational reasoning. A main focus of BiKAT is to ensure *adequacy*: the aligned program should capture all computations of the originial program. Our biprogram construct is analogous to the design in BiKAT. However, a major difference between our work and BiKAT is applicability: the main focus of BiKAT is alignments of two programs, while we also concern domain-specific reasoning. The reasoning rules for BiKAT can be encoded in the translations (as shown in Section 5.2), but BiKAT cannot be used to reason with many $\mathsf{d}\mathcal{L}_{\text{REL}}$ formulas. In addition, the design of modalities promotes expressing complex relational properties concisely, which is not a motivation behind BiKAT.

Relational Reasoning for Dynamic Logic.. Beckert et al. have done a series of work on extending dynamic logic with trace modalities [10, 9, 26]. They further apply trace modalities to check secure information flow in the setting of concurrent programs [26]. Their work focus on first-order dynamic logic with deterministic programs. The limitation to deterministic programs is significant because nondeterminism is essential to the utility of dynamic logics, especially in AI-enabled systems [24]. Gutsfeld et al. introduce an expressive extension to propositional dynamic logic to check hyperproperties by introducing path quantifiers [27]. Algorithms for model checking these hyperproperties are introduced. Tool support is left for future work. In comparison, the REL extension goes beyond propositional dynamic logic, and it can reuse existing tools to verify relational properties.

Relational Reasoning for Differential Dynamic Logic.. Various approaches have been proposed to analyze specific relational properties in $\mathsf{d}\mathcal{L}$. A primitive is introduced to express a refinement relation between two hybrid programs [31]. An expressive modal logic based on $\mathsf{d}\mathcal{L}$ has been introduced to reason with nondeducibility [13]. Neither work provides tool support. Kolčák et al. introduce a relational extension of $\mathsf{d}\mathcal{L}$ that focuses on reasoning about $\forall\forall$ on two dynamics [29]. Our translation for $\mathsf{d}\mathcal{L}_{\text{REL}}$ is able to integrate this work. Xiang et al. introduce a formal framework in the setting of $\mathsf{d}\mathcal{L}$ for modeling and analyzing the robustness of cyber-physical systems under sensor attacks [48]. In our case study for $\mathsf{d}\mathcal{L}$, we express and verify this robustness property with $\mathsf{d}\mathcal{L}_{\text{REL}}$, which is more succinct than the original example.

Biprogram.. Our biprogram is an adaptation of the work of Pottier and Simonet on information flow analysis for ML [38]. The work introduces an extension of ML that encodes a pair of ML terms as a single bracket construct. The bracket constructs cannot be nested. This is analogous to our design in that the biprogram contains two programs of the host logics. This bracket-based approach has been often used in proving noninterference, e.g., [51, 4]. The name "biprogram" is also used in other recent work on relational reasoning [6, 3].

Relational Program Logic. Another approach for relational verification is to use relational program logic [11, 50, 32]. Benton introduced Relational Hoare Logic for verifying program transformations [11], which provides a general framework for relational correctness proofs. Recent work proposes a notion to evaluate the design of Relational Hoare Logic [32]. Sousa and Dillig introduce a program logic, named Cartesian Hoare Logic (CHL), for verifying k-safety properties [42]. The work has been further extended for proving the correctness of 3-way merge [43]. They introduce a generalized form of Hoare triples to express relations between different program executions, and use SMT solvers to determine their satisfiability. D'Osualdo et al. [21] describe a logic for hyper-triple composition (LHC) based on weakest preconditions that can decompose a hypersafety proof along the boundary of hyper tuples, offering ways of combining multiple k-safety proofs. These relational program logics often are based on weakest pre-condition and only support $\forall\forall$ pre/post k-safety properties. In contrast, the REL extensions naturally support properties with mixed modalities. Moreover, the translations for REL extensions open opportunities in combining general relational reasoning and and domain-specific reasoning. We believe these relational program logics can be also integrated as a part of the translations, which we leave to future work.

Self-Composition and Product Program.. The encodings introduced in this work are inspired by self-composition, which is a common approach for relational reasoning [8, 19, 7, 16]. Approaches based on self-composition are often syntax-directed, that is, they compose two programs that have similar, if not identical, syntactic constructs. Alternatively, property directed self-composition [41] tackles programs that cannot be easily aligned in a composition. It composes programs (or copies of the same program) by finding good alignment between the copies in order to have expressible assertions. The encodings in this work can be viewed as special kinds of self-compositions that shall be adapted for different host logics. The encoding can benefit from advances in self-composition, as we have shown in Section 5.2.

Other Related Work.. Noninterference is a well-known relational property for secure information flow and has been widely studied, e.g., [38, 44, 47, 5, 15]. REL extensions bring in several major benefits for properties like noninterference. First, noninterference reasons about program executions, and modalities in REL capture executions intuitvely. Different variants of noninterference can be concisely expressed, as shown in the examples of Section 3. Second, the notion of noninterference can be easily extended to different domains (host logics), as the REL extension connects the common program constructs with domain-specific constructs. And third, verifying noninterference can leverage more existing tools, as the translations can integrate different techniques.

Relational verification has been studied in the setting of temporal-style logics, such as HyperLTL and HyperCTL, have been introduced to model relational properties [17, 12, 46, 27]. Model checking is often used in verify these properties. In contrast, REL extensions are more expressive on program behaviors than logics like HyperLTL and HyperCTL. Translations for the extensions may leverage the model checkers for HyperLTL or HyperCTL if the correct heuristics are developed, which we leave to future work.

7 Conclusion

We introduce a general and lightweight relational extension for dynamic logics. We expect that the extension can be instantiated for almost any dynamic logic, in the same way that one would expect that almost any dynamic logic could be extended (syntactically and semantically) with first-order operators. Verifying REL formulas can leverage existing tools developed for the host logics. Moreover, the instantiations show great potential in promoting verification by integrating general relational reasoning and domain-specific relational reasoning.

Future work. One immediate next step is to *systematically* integrate techniques for general relational reasoning and domain-specific relational reasoning into the encoding. Proof automation with the translations is an important and interesting future work. Additionally, we believe the natural expression of relational properties by the REL extension opens new opportunities to explore more powerful verification techniques. We plan to develop a novel proof system for the REL extension that allows us to prove relational properties that cannot be proven by existing tools and techniques.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification–The KeY Book: From Theory to Practice, vol. 10001. Springer (2016)
2. Allen, P.: A comparison of non-interference and non-deducibility using CSP. In: CSF. pp. 43–44 (1991)
3. Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An algebra of alignment for relational verification. PLDI **7**, 573–603 (2023)
4. Arden, O., Myers, A.C.: A calculus for flow-limited authorization. In: CSF. pp. 135–149 (2016)
5. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: PLAS. pp. 1–12 (2011)
6. Banerjee, A., Nagasamudram, R., Naumann, D.A., Nikouei, M.: A relational program logic with data abstraction and dynamic framing. arXiv preprint arXiv:1910.14560 (2019)
7. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. pp. 200–214 (2011)
8. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSF. pp. 100–114 (2004)
9. Beckert, B., Bruns, D.: Dynamic logic with trace semantics. In: CADE. pp. 315–329 (2013)
10. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In: IJCAR. pp. 626–641 (2001)
11. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL. pp. 14–25 (2004)
12. Beutner, R., Finkbeiner, B.: Autohyper: Explicit-state model checking for HyperLTL. In: TACAS. pp. 145–163 (2023)
13. Bohrer, B., Platzer, A.: A hybrid, dynamic logic for hybrid-dynamic information flow. In: LICS. pp. 115–124 (2018)
14. Chong, S., Lanotte, R., Merro, M., Tini, S., Xiang, J.: Quantitative robustness analysis of sensor attacks on cyber-physical systems. In: HSCC. pp. 1–12 (2023)
15. Chudnov, A., Naumann, D.A.: Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In: CSF. pp. 189–203 (2018)
16. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: PLDI. pp. 1027–1040 (2019)
17. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: POST. pp. 265–284 (2014)
18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
19. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: PerCom. pp. 193–209 (2005)
20. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI (2013)
21. D'Osualdo, E., Farzan, A., Dreyer, D.: Proving hypersafety compositionally. In: OOPSLA2. pp. 289–314 (2022)
22. Francez, N.: Product properties and their direct verification. Acta informatica **20**(4), 329–344 (1983)

23. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: CADE. pp. 527–538 (2015)
24. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: TACAS. pp. 413–430 (2019)
25. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S&P. pp. 11–20 (1982)
26. Grahl, D.: Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java. Ph.D. thesis, Karlsruhe Institute of Technology (29 Oct 2015)
27. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Propositional dynamic logic for hyperproperties. In: CONCUR (2020)
28. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
29. Kolčák, J., Dubut, J., Hasuo, I., Katsumata, S.y., Sprunger, D., Yamada, A.: Relational differential dynamic logic. In: TACAS. pp. 191–208 (2020)
30. Kozen, D.: Kleene algebra with tests. TOPLAS **19**(3), 427–443 (1997)
31. Loos, S.M., Platzer, A.: Differential refinement logic. In: LICS. pp. 505–514 (2016)
32. Nagasamudram, R., Naumann, D.A.: Alignment completeness for relational hoare logics. In: LICS. pp. 1–13 (2021)
33. Naumann, D.A.: Thirty-seven years of relational hoare logic: remarks on its principles and history. In: ISoLA. pp. 93–116 (2020)
34. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41**(2), 143–189 (2008)
35. Platzer, A.: The complete proof theory of hybrid systems. In: LICS. pp. 541–550 (2012)
36. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. Journal of Automated Reasoning **59**(2), 219–265 (2017)
37. Platzer, A.: Logical foundations of cyber-physical systems, vol. 662. Springer (2018)
38. Pottier, F., Simonet, V.: Information flow inference for ML. In: POPL. pp. 319–330 (2002)
39. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
40. Satake, Y., Unno, H.: Propositional dynamic logic for higher-order functional programs. In: CAV. pp. 105–123 (2018)
41. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: CAV. pp. 161–179 (2019)
42. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: PLDI. pp. 57–69 (2016)
43. Sousa, M., Dillig, I., Lahiri, S.K.: Verified three-way program merge. In: OOPSLA. pp. 145–164 (2016)
44. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Haskell. pp. 95–106 (2011)
45. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: SAS. pp. 352–367 (2005)
46. Wang, Y., Nalluri, S., Pajic, M.: Hyperproperties for robotics: Planning via HyperLTL. In: ICRA. pp. 8462–8468 (2020)
47. Xiang, J., Chong, S.: Co-Inflow: coarse-grained information flow control for Java-like languages. In: S&P (2021)

48. Xiang, J., Fulton, N., Chong, S.: Relational analysis of sensor attacks on Cyber-Physical Systems. In: CSF (2021)
49. Xiang, J., Lanotte, R., Tini, S., Chong, S., Merro, M.: Measuring robustness in cyber-physical systems under sensor attacks. Nonlinear Analysis: Hybrid Systems **56** (2025)
50. Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1-3), 308–334 (2007)
51. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. International Journal of Information Security **6**(2-3), 67–84 (2007)