# Collaborative Verification-Driven Engineering of Hybrid Systems

Stefan Mitsch, Grant Olney Passmore and André Platzer

**Abstract.** Hybrid systems with both discrete and continuous dynamics are an important model for real-world physical systems. The key challenge is how to ensure their correct functioning w.r.t. safety requirements. Promising techniques to ensure safety seem to be model-driven engineering to develop hybrid systems in a well-defined and traceable manner, and formal verification to prove their correctness. Their combination forms the vision of verification-driven engineering. Despite the remarkable progress in automating formal verification of hybrid systems, the construction of proofs of complex systems often requires significant human guidance, since hybrid systems verification tools solve undecidable problems. It is thus not uncommon for verification teams to consist of many players with diverse expertise. This paper introduces a verification-driven engineering toolset that extends our previous work on hybrid and arithmetic verification with tools for (i) modeling hybrid systems, (ii) exchanging and comparing models and proofs, and (iii) managing verification tasks. This toolset makes it easier to tackle large-scale verification tasks.

**Keywords.** formal verification, hybrid system, model-driven engineering.

## 1. Introduction

Computers that control physical processes form so-called *cyber-physical systems* (CPS), which are today pervasively embedded into our lives. For example, cars equipped with adaptive cruise control form a typical CPS that is responsible for controlling acceleration on the basis of distance sensors. Further prominent examples can be found in many safety-critical areas, such as in factory automation, medical equipment, automotive, aviation, and railway industries. From an engineering viewpoint, CPSs can be described as a *hybrid system* in terms of discrete control decisions (the cyber-part, e. g., setting the acceleration of a car) and differential equations modeling the entailed physical continuous dynamics (the physical part, e. g., motion) [40]. More advanced models include aspects of distributed hybrid systems or stochasticity [44], but are not addressed in this paper.

The key challenge in engineering hybrid systems is the question of how to ensure their correct functioning in order to avoid incorrect control decisions w.r.t. safety requirements (e. g., a car with adaptive cruise control will never collide with a car driving ahead). Especially promising techniques

This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, NSF EXPEDITION CNS-0926181, and under Grant Nos. CNS-1035800 and CNS-0931985, by DARPA under agreement number FA8750-12-2-0291, and by the US Department of Transportation's University Transportation Center's TSET grant, award# DTRT12GUTC11. Passmore was also supported by the UK's EPSRC [grants numbers EP/I011005/1 and EP/I010335/1]. Mitsch was also supported by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under grant FFG FIT-IT 829598, FFG BRIDGE 838526, and FFG Basisprogramm 838181, and by the ERC under grant PIOF-GA-2012-328378.

© Springer Basel 2014
Mitsch, S., Passmore, G.O. & Platzer, A. Math.Comput.Sci. (2014) 8: 71. doi:10.1007/s11786-014-0176-y

to ensure safety seem to be model-driven engineering (MDE) to incrementally develop systems in a well-defined and traceable manner and formal verification to mathematically prove their correctness. Together, these techniques form the vision of *verification-driven engineering* (VDE) [24]. Despite the remarkable progress in automating formal verification of hybrid systems, still many interesting and complex verification problems remain that are hard to solve in practice with a single tool by a single person.

Because hybrid systems are undecidable, hybrid systems verification tools work over an undecidable theory, and so verifying complicated systems within them often requires significant human guidance. This need for human guidance is true even for *decidable* theories utilized within hybrid systems verification [7], such as the first-order theory of non-linear real arithmetic (also called the theory of *real closed fields* or RCF), a crucial component of real-world verification efforts. Though decidable, RCF is fundamentally infeasible (it is worst-case doubly exponential in the number of variables [8]), which poses a problem for the automated verification of hybrid systems. Much expertise is often needed to discharge arithmetical verification conditions in a reasonable amount of time and space, expertise requiring the use of deep results in real algebraic geometry. It is thus not uncommon for serious hybrid systems verification teams to consist of many players, some with expertise in control theory and dynamical systems, some in software engineering, some in mathematical logic, some in real algebraic geometry, and so on. Hence, modeling languages that convey a model to a broad and possibly heterogeneous audience together with well-established project management techniques to coordinate team members are crucial to achieve effective collaborative large-scale verification of hybrid systems. Successful examples of team-based large-scale verification of non-hybrid systems include the operating system kernel seL4 [22] in Isabelle/HOL and the Flyspeck project [17], and show, that indeed collaboration is key for proving large systems.

This paper introduces a VDE toolset of modeling and verification tools for hybrid systems (including a backend deployment for project management and collaboration support). The toolset applies proof decomposition in-the-large across multiple verification tools, basing on the completeness of differential dynamic logic ($\mathsf{d}\mathcal{L}$ [40, 43]), which is a real-valued first-order dynamic logic for hybrid programs, a program notation for hybrid systems. The VDE toolset Sphinx extends our previous work on the deductive verification tool KeYmaera [47] and on the nonlinear real arithmetic verification tools RAHD [38] and MetiTarski [39] with tools for (i) graphical and textual modeling of hybrid systems in $\mathsf{d}\mathcal{L}$, (ii) exchanging and comparing models and proofs via a central source repository, and (iii) exchanging knowledge and tasks through a project management backend.

**Structure of the paper** In the next section, we give an overview of related work. In Section 3 we introduce our architecture of a verification-driven engineering toolset, and describe implementation and features of its components. Section 4 introduces an autonomous robotic ground vehicle as application example. Finally, in Section 5 we conclude the paper with an outlook on real-world application of the toolset and possible directions for future work.

## 2. Related Work

Model-driven engineering in a collaborative manner has been successfully applied in the embedded systems community. Efforts, for instance, include transforming between different UML models and SysML [18], modeling in SysML and transforming these models to the simulation tool Orchestra [2], integration of modeling and simulation in Cosmic/Cadena [15], or modeling of reactive systems and integration of various verification tools in Syspect [12].

Recent surveys on verification methods for hybrid systems [1], modeling and analysis of hybrid systems [10], and modeling of cyber-physical systems [11], reveal that indeed many tools are available for modeling and analyzing hybrid systems, but in a rather isolated manner. Supporting collaboration on formal verification by distributing tasks among members of a verification team

in a model-driven engineering approach has not yet been the focus. Although current verification tools for hybrid systems (e. g., PHAVer [13], SpaceEx [14]), as well as those for arithmetic (e. g., Z3 [9]) are accompanied by modeling editors of varying sophistication, they are not yet particularly well-prepared for collaboration either. Developments in collaborative verification of source code by multiple complementary static code checkers [6], modular model-checking (e. g., [26]), and extreme verification [19], however, indicate that this is indeed an interesting field. Most notably, usage of online collaboration tools in the Polymath project has led to an elementary proof of a special case of the density Hales-Jewett theorem [16].

The Unified Modeling Language (UML [20]) is a standardized language for object-oriented modeling. But without extension it is not particularly well suited for modeling hybrid systems [5]. Therefore, the profiling mechanism of UML was used to extend the standardized UML languages SysML [18] for modeling hardware and software components of complex systems and MARTE [31] for modeling real-time and embedded systems. These extensions [5, 27, 50] to add better support for hybrid modeling to UML. However, they propose profiles for the UML Statechart formalism, since those languages base on hybrid automata as underlying principle. We, instead, use hybrid programs and therefore extensions of UML Activity Diagrams are a more natural way of modeling.

- Unlike [18, 2, 15], who focus on exchanging models, we also facilitate collaboration on formal verification.
- Unlike [13, 14, 9], who focus on one aspect of verification, we provide modeling and collaboration tools that should make it easier for domain experts to work in verification teams and exchange models and verification results between different tools.
- Unlike [13, 9], who focus on verification tools, we also work on modeling support and collaboration.
- Unlike [5, 27, 50], who define a hybrid automaton semantics for UML Statecharts, we define a hybrid program semantics for UML Activity Diagrams.

## 3. The VDE Toolset Sphinx

Verification teams often comprise experts with diverse heterogeneous background and accustomed to different modeling and verification tools. In order to integrate different modeling and verification tools, the verification-driven engineering toolset Sphinx[1] proposed in this paper follows a model-driven architecture: metamodels for different modeling and proof languages form the basis for manipulating, persisting, and transforming models. The notion of a model here denotes an instance of a metamodel, i. e., it comprises models, proofs, and strategies. Following the definition of the OMG[2], a metamodel defines a language to formulate models: one example for a metamodel is the grammar of $\mathsf{d}\mathcal{L}$, which, among others, defines language elements for non-deterministic choice, sequential composition, assignment, repetition, and differential equations. An example for a model is given in Section 3.1: it describes a simple water tank as a set of formulas, differential equations, and other $\mathsf{d}\mathcal{L}$ language elements. The model conforms to the grammar of $\mathsf{d}\mathcal{L}$, and thus is an instance of the $\mathsf{d}\mathcal{L}$ metamodel. Fig. 1 gives an overview of the toolset architecture: the $\mathsf{d}\mathcal{L}$ metamodel, $\mathsf{d}\mathcal{L}$ proof metamodel, arithmetic metamodel, and arithmetic proof metamodel each represent an interface between tools and to the backend.

**$\mathsf{d}\mathcal{L}$ metamodel.** The hybrid modeling components (textual and graphical editors for $\mathsf{d}\mathcal{L}$, as well as model comparison) manipulate models that conform to the $\mathsf{d}\mathcal{L}$ metamodel. A transformation runtime transforms between models in $\mathsf{d}\mathcal{L}$ and their textual form read by KeYmaera.

**Hybrid Program UML.** The hybrid program UML profile extends UML with hybrid system concepts that can be translated to $\mathsf{d}\mathcal{L}$ models.

[1] http://http://www.cs.cmu.edu/~smitsch/sphinx.html [2] http://www.omg.org

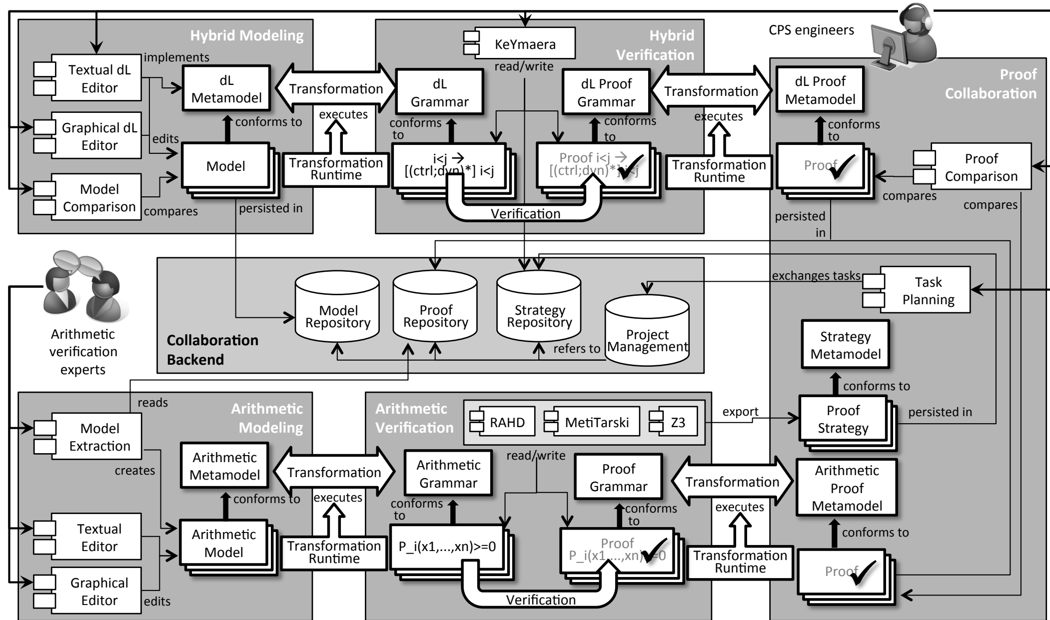



FIGURE 1. Overview of components in the verification-driven engineering toolset

**d$\mathcal{L}$ proof metamodel.** The proof comparison component reads proofs that conform to the d$\mathcal{L}$ proof metamodel. These proofs may either be closed ones (completed proofs, nothing else to be done) or partial proofs (to be continued). A transformation runtime transforms between proofs in Sphinx and their textual form as generated by KeYmaera.

**Arithmetic metamodel.** Arithmetic editors (not yet implemented) manipulate arithmetic models. Again a transformation runtime transforms between models expressed in terms of the arithmetic metamodel and the corresponding textual input (e. g., SMT-LIB syntax [3]) as needed by arithmetic tools, such as RAHD, MetiTarski, or Z3.

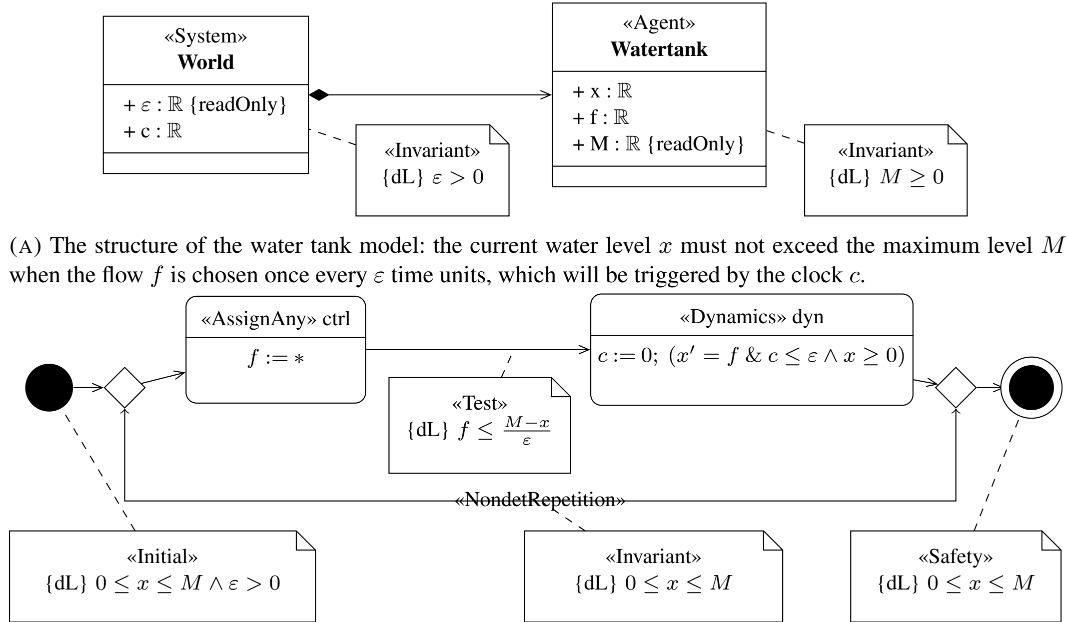
**Arithmetic proof metamodel.** Finally, the proof comparison component reads arithmetic proofs expressed in terms of the arithmetic proof metamodel, and transformed to and from the arithmetic tool's (textual) format by a transformation runtime.

### 3.1. A Hybrid Water Tank Example

We illustrate the notion of hybrid systems and our hybrid programs and hybrid program UML profile by means of the classical water tank example: a water tank should not overflow when the flow in or out of the water tank is chosen once every time interval. The hybrid program UML model is shown in Fig. 2. We will use the hybrid program UML syntax informally here and later introduce it in detail in Section 3.5.3.

The system introduces a global clock $c$ and a bound on the loop execution time $\varepsilon$, which must be strictly positive as indicated by the invariant $\varepsilon > 0$ attached to the class *World*. The system further consists of one agent, the *WaterTank*, which is characterized by the current water level $x$, the current flow $f$ and the maximum level $M$. The maximum level is constant (*readOnly*) and positive, as defined in the attached invariant $M \geq 0$.

The behavior of the system is a single loop with two actions: the *ctrl* action chooses a new nondeterministic flow that will not exceed the water tank's maximum capacity (cf. the test $\frac{M-x}{\varepsilon}$). The subsequent continuous evolution *dyn* resets the clock $c$ and evolves the water level in the tank according to the chosen flow along the differential-algebraic equation $x' = v \;\&\; c \leq \varepsilon \wedge x \geq 0$ (the




(A) The structure of the water tank model: the current water level $x$ must not exceed the maximum level $M$ when the flow $f$ is chosen once every $\varepsilon$ time units, which will be triggered by the clock $c$.

(B) The behavior of the water tank model: the controller *ctrl* chooses a new nondeterministic value for the flow $f$, such that it satisfies the subsequent test before it is passed to the continuous dynamics *dyn*. Controller and continuous dynamics are repeated nondeterministically many times.

FIGURE 2. Example of a hybrid system: a water tank

constraints ensure that the clock will not exceed a certain limit $c \leq \varepsilon$ and the water level will always be positive $x \geq 0$).

The specification about the water tank is annotated as constraints on the initial and the final node: when we start the water tank model in a state where the current level is within the limits of the water tank, then all runs of the model should keep the water level within the limits.

### 3.2. Development and Verification Process by Example

Let us exemplify the toolset with a virtual walk-through a collaborative verification scenario. We begin with modeling a hybrid system as in the water tank example above using the graphical and textual $\mathsf{d}\mathcal{L}$ editors. The resulting model, which conforms to the $\mathsf{d}\mathcal{L}$ metamodel, is transformed on-the-fly during editing by the transformation runtime to a textual input file, and loaded into KeYmaera. In KeYmaera, we apply various strategies for proving safety of our hybrid system model, but may get stuck at some difficult arithmetic problem. We mark the corresponding node in the partial proof and save it in KeYmaera's textual output format. The proof collaboration tool transforms the partial proof text file into a model of the partial proof. We persist the hybrid model and the model of the partial proof in the model and proof repository, respectively. Then we create a request for arithmetic verification (ticket) in the project management repository using the task planning component. The assignee of the ticket accesses the linked partial proof, and extracts an arithmetic verification model from the marked proof node. Then a transformation runtime creates the textual input for one of the arithmetic verification tools. In this tool, a proof for the ticket can be created, along with a proof strategy that documents the proof. Such a proof strategy is vital for replaying the proof later, and for detecting whether or not the arithmetic proof still applies if the initial model changed. Both proof and proof strategy, are imported into the proof collaboration tool and persisted to the corresponding repository. The ticket is closed, together with the node on the original proof (if the arithmetic proof

is complete; otherwise, the progress made is reported back). We fetch the new proof model version from the repository and inspect it using the proof comparison component. Then we transform the proof model into its textual form, load KeYmaera and continue proving our hybrid system from where we left off, but now with one goal closed. In case the corresponding arithmetic prover is connected to KeYmaera, we could even load the proof strategy from the strategy repository and repeat it locally.

### 3.3. KeYmaera: Hybrid System Verification

KeYmaera[3] [47] is a verification tool for hybrid systems that combines deductive, real algebraic, and computer algebraic prover technologies. It is an automated and interactive theorem prover for a natural specification and verification logic for hybrid systems. KeYmaera supports differential dynamic logic ($\mathsf{d}\mathcal{L}$) [40, 41, 42, 43], which is a real-valued first-order dynamic logic for hybrid programs, a program notation for hybrid systems. KeYmaera supports hybrid systems with nonlinear discrete jumps, nonlinear differential equations, differential-algebraic equations, differential inequalities, and systems with nondeterministic discrete or continuous input.

For automation, KeYmaera implements a number of automatic proof strategies that decompose hybrid systems symbolically in differential dynamic logic and prove the full system by proving properties of its parts [42]. This compositional verification principle helps scaling up verification, because KeYmaera verifies a big system by verifying properties of subsystems. Strong theoretical properties, including relative completeness results, have been shown about differential dynamic logic [40, 43] indicating how this composition principle can be successful.

KeYmaera implements fixedpoint procedures [45] that try to compute invariants of hybrid systems and differential invariants of their continuous dynamics, but may fail in practice. By completeness [43], this is the only part where KeYmaera's automation can fail in theory. In practice, however, also the decidable parts of dealing with arithmetic may become infeasible at some point, so that interaction with other tools or collaborative verification via Sphinx is crucial.

At the same time, it is an interesting challenge to scale to solve larger systems, which is possible according to completeness but highly nontrivial. For systems that are still out of reach for current automation techniques, the fact that completeness proofs are compositional can be exploited by interactively splitting parts of the hybrid systems proof off and investigating them separately within Sphinx. If, for instance, a proof node in arithmetic turns out to be infeasible within KeYmaera, this node could be verified using a different tool connected to Sphinx.

KeYmaera has been used successfully for verifying case studies from train control [48], car control [28, 29, 33], air traffic management [30, 46], robotic obstacle avoidance [32], and robotic surgery [25]. These verification results illustrate how some systems can be verified automatically while others need more substantial user guidance. The KeYmaera approach is described in detail in a book [42].

In order to guide domain experts in modeling discrete and continuous dynamics of hybrid systems, the case studies, further examples, and their proofs are included in the KeYmaera distribution. When applying proof strategies manually by selection from the context menu in the interactive theorem prover, KeYmaera shows only the applicable ones sorted by expected utility. Preliminary collaboration features include marking and renaming of proof nodes, as well as extraction of proof branches as new subproblems. These collaboration features are used for interaction with the arithmetic verification tools and the collaboration backend described below.

### 3.4. Arithmetic Verification

Proofs about hybrid systems often require significant reasoning about multivariate polynomial inequalities, i.e., reasoning within the *theory of real closed fields* (**RCF**). Though **RCF** is decidable, it is fundamentally infeasible (hyper-exponential in the number of variables). It is not uncommon

[3] http://symbolaris.com/info/KeYmaera.html

for hybrid system models to have tens or even hundreds of real variables, and **RCF** reasoning is commonly the bottleneck for nontrivial verifications. Automatic **RCF** methods simply do not scale, and manual human expertise is often needed to discharge a proof's arithmetical subproblems.

**RCF** infeasibility is not just a problem for hybrid systems verification. Real polynomial constraints are pervasive throughout the sciences, and this has motivated a tremendous amount of work on the development of feasible proof techniques for various special classes of polynomial systems. In the context of hybrid systems verification, we wish to take advantage of these new techniques as soon as possible.

Given this fundamental infeasibility, how might one go about deciding large **RCF** conjectures? One approach is to develop a battery of efficient proof techniques for different practically useful fragments of the theory. For example, if an $\exists$ **RCF** formula can be equisatisfiably transformed into an $\wedge\vee$-combination of strict inequalities, then one can eliminate the need to consider any irrational real algebraic solutions when deciding the formula. Tools such as RAHD, Z3 and MetiTarski exemplify this heterogeneous approach to **RCF**, and moreover allow users to define *proof strategies* consisting of heuristic combinations of various specialised proof methods. When faced with a difficult new problem, one works to develop a proof strategy which can solve not only the problem at hand but also other problems sharing similar structure. Such strategies, though usually constructed by domain experts, can then be shared and utilised as automated techniques by the community at large.

### 3.5. Modeling and Proof Collaboration

In order to interconnect the variety of specialized verification procedures introduced above, Sphinx follows a model-driven engineering approach: it introduces metamodels for the included modeling and proof languages. These metamodels provide a clean basis for model creation, model comparison, and model transformation between the formats of different tools. This approach is feasible, since in principle many of those procedures operate over the theory **RCF**, or at least share a large portion of symbols and their semantics. One could even imagine that very same approach for exchanging proofs between different proof procedures, since proofs in RCF, in theory, can all be expressed in the same formal system. Currently, proofs in Sphinx are exchanged merely for the sake of being repeated in the original tool (although KeYmaera already utilizes many such tools and hence is able to repeat a wide variety of proofs).

In the case of textual languages, Sphinx uses the Eclipse Xtext[4] framework to obtain metamodels directly from the language grammars (cf. Fig. 3, obtained from the $\mathsf{d}\mathcal{L}$ grammar [40]), together with other software artifacts, such as a parser, a model serializer, and a textual editor with syntax highlighting, code completion, and cross referencing. These metamodels are the basis for creating models in $\mathsf{d}\mathcal{L}$, as well as for defining transformations between $\mathsf{d}\mathcal{L}$ and other modeling languages. The models in $\mathsf{d}\mathcal{L}$ make use of mathematical terms, and are embedded in KeY files since KeYmaera uses the KeY [35] format for loading models and saving proofs. In the following sections, we introduce $\mathsf{d}\mathcal{L}$ in more detail and describe the support for creating $\mathsf{d}\mathcal{L}$ models and working on proofs in Sphinx.

**3.5.1. Differential Dynamic Logic.** For specifying and verifying correctness statements about hybrid systems, we use *differential dynamic logic* $\mathsf{d}\mathcal{L}$ [40, 42, 43], which supports *hybrid programs* as a program notation for hybrid systems. The syntax of hybrid programs is summarized together with an informal semantics in Table 1; the metamodel introduced in Fig. 3 reflects this syntax. The sequential composition «$\alpha;\ \beta$» expresses that $\beta$ starts after $\alpha$ finishes (e. g., first let a car choose its acceleration, then drive with that acceleration). The non-deterministic choice «$\alpha \cup \beta$» follows either $\alpha$ or $\beta$ (e. g., let a car decide nondeterministically between accelerating and braking). The non-deterministic repetition operator «$\alpha^*$» repeats $\alpha$ zero or more times (e. g., let a car choose a new acceleration arbitrarily often). Discrete assignment «$x := \theta$» instantaneously assigns the value of the term $\theta$ to the variable $x$ (e. g., let a car choose a particular acceleration), while «$x := *$» assigns an

[4] www.eclipse.org/Xtext

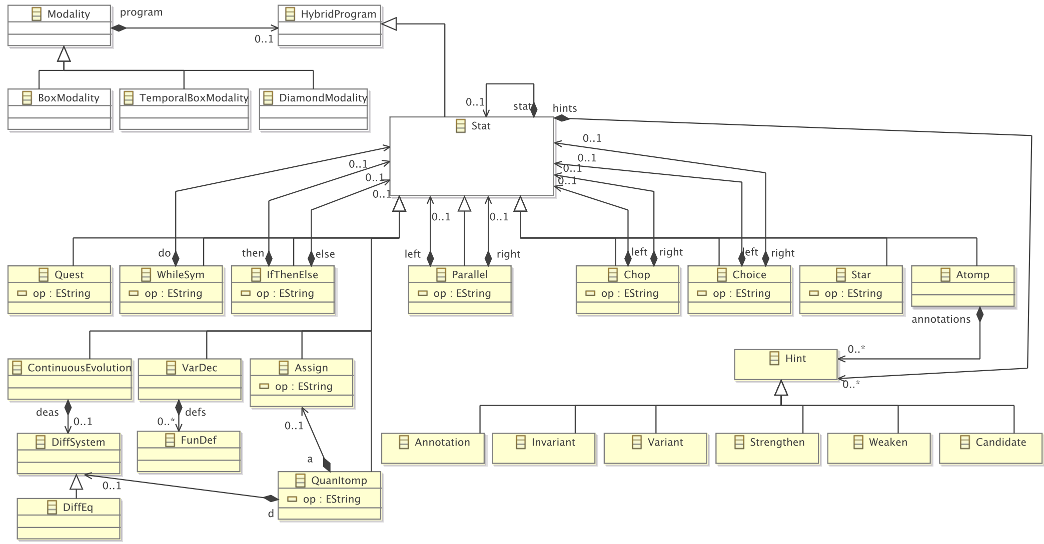



FIGURE 3. The d$\mathcal{L}$ metamodel extracted from the input grammar of KeYmaera

TABLE 1. Statements of hybrid programs

| Statement | Metamodel element | Effect |
|---|---|---|
| $\alpha;\ \beta$ | Chop | sequential composition, performs HP $\alpha$ and then HP $\beta$ afterwards |
| $\alpha\ \cup\ \beta$ | Choice | nondeterministic choice, follows either HP $\alpha$ or HP $\beta$ |
| $\alpha^*$ | Star | nondeterministic repetition, repeats HP $\alpha$ $n \geq 0$ times |
| $x := \theta$ | Assign (term) | discrete assignment of the value of term $\theta$ to variable $x$ (jump) |
| $x := *$ | Assign (wild card term) | nondeterministic assignment of an arbitrary real number to $x$ |
| $\big(x_1' = \theta_1, \dots,$ | ContinuousEvolution | continuous evolution of $x_i$ along differential equation system |
| $x_n' = \theta_n\ \&\ F\big)$ | DiffSystem | $x_i' = \theta_i$, restricted to maximum domain or invariant region $F$ |
| $?F$ | Quest | check if formula $F$ holds at current state, abort otherwise |
| if$(F)$ then $\alpha$ else $\beta$ | IfThenElse | perform HP $\alpha$ if $F$ holds, perform HP $\beta$ otherwise |
| while$(F)$ do $\alpha$ end | WhileSym | perform HP $\alpha$ as long as $F$ holds |
| $[\alpha]\phi$ | BoxModality | d$\mathcal{L}$ formula $\phi$ must hold after all executions of HP $\alpha$ |
| $\langle\alpha\rangle\phi$ | DiamondModality | d$\mathcal{L}$ formula $\phi$ must hold after at least one execution of HP $\alpha$ |

arbitrary value to $x$ (e. g., let a car choose any acceleration). «$x' = \theta\ \&\ F$» describes a continuous evolution of $x$ within the evolution domain $F$ (e. g., let the velocity of a car change according to its acceleration, but always be greater than zero). The test «$?F$» checks that a particular condition expressed by $F$ holds, and aborts if it does not (e. g., test whether or not the distance to a car ahead is large enough). A typical pattern that involves assignment and tests, and which will be used subsequently, is to limit the assignment of arbitrary values to known bounds (e. g., limit an arbitrarily chosen acceleration to the physical limits of a car, as in $x := *; ?x \geq 0$). The deterministic choice «if$(F)$ then $\alpha$ else $\beta$» executes $\alpha$ if $F$ holds, and $\beta$ otherwise (e. g., let a car accelerate only when it is safe; brake otherwise). Finally, «while$(F)$ do $\alpha$ elihw» is a deterministic repetition that executes $\alpha$ as long as $F$ holds.

To specify the desired correctness properties of hybrid programs, differential dynamic logic (d$\mathcal{L}$) provides modal operators $[\alpha]$ and $\langle\alpha\rangle$, one for each hybrid program $\alpha$. When $\phi$ is a d$\mathcal{L}$ formula (e.g., a simple arithmetic constraint) describing a safe state and $\alpha$ is a hybrid program, then the d$\mathcal{L}$ formula $[\alpha]\phi$ states that all states reachable by $\alpha$ satisfy $\phi$. Dually, d$\mathcal{L}$ formula $\langle\alpha\rangle\phi$ expresses that

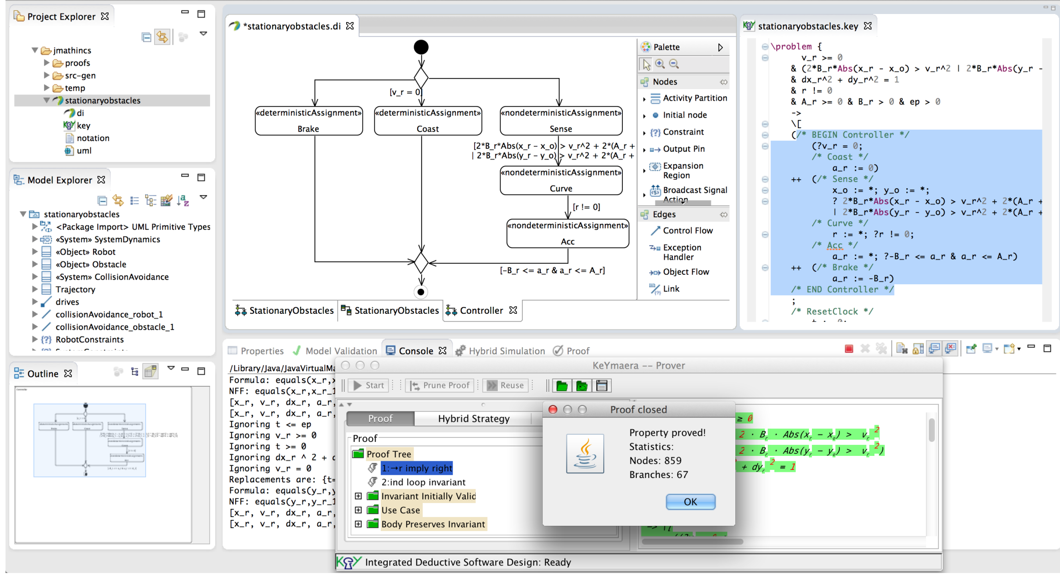



FIGURE 4. Textual and graphical syntax, proof in KeYmaera. The text editor selection highlights the controller part of the graphical editor to the left of the textual editor.

there is a state reachable by the hybrid program $\alpha$ that satisfies $d\mathcal{L}$ formula $\phi$. The set of $d\mathcal{L}$ formulas is generated by the following EBNF grammar (where $\sim \in \{<, \leq, =, \geq, >\}$ and $\theta_1, \theta_2$ are arithmetic expressions in $+, -, \cdot, /$ over the reals):

$$\phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \forall x \phi \mid \exists x \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

Thus, besides comparisons ($<, \leq, =, \geq, >$), $d\mathcal{L}$ allows one to express negations ($\neg\phi$), conjunctions ($\phi \wedge \psi$), universal ($\forall x \phi$) and existential quantification ($\exists x \phi$), as well as the already mentioned state reachability expressions ($[\alpha]\phi$, $\langle\alpha\rangle\phi$).

**3.5.2. Creating Models.** Sphinx currently includes $d\mathcal{L}$ as generic modeling language to create models of hybrid and cyber-physical systems. The concrete textual syntax and $d\mathcal{L}$ editor are created from the $d\mathcal{L}$ metamodel and shown in Fig. 4, together with a concrete graphical syntax based on UML and the KeYmaera prover attached through the console.

In order to facilitate creation of textual models in $d\mathcal{L}$, Sphinx includes templates of common model artifacts (e. g., ODEs of linear and circular motion). These templates, when instantiated, allow in-place editing and automated renaming of the template constituents. As usual in the Eclipse platform, such templates can be easily extended and shared between team members.

Since generic modeling languages, such as $d\mathcal{L}$ for hybrid systems, tend to incur a steep learning curve, the Sphinx platform can be extended with dedicated domain-specific languages (DSL). Such DSLs should be designed to meet the vocabulary of a particular group of domain experts. They can be included into Sphinx in a similar fashion to the generic modeling language $d\mathcal{L}$, i. e., in the form of Eclipse plugins that provide the DSL metamodel and the modeling editor.

In the next section we describe the Hybrid Program UML profile, an extension to the UML for graphical modeling of hybrid programs that should make it easier to convey the main features of a hybrid system to a broader audience.

**3.5.3. The Hybrid Program UML Profile.** The Hybrid Program UML profile follows a fundamental principle of UML in that it separates modeling the structure of a hybrid system from modeling its

behavior. Currently, Sphinx supports class diagrams for modeling the structure of a hybrid system, since hybrid programs do not yet support modules. Future work includes the addition of composite structure diagrams as used in [5, 50], and the introduction of proof rules that exploit the additional structural information during the verification process. For modeling behavior, we use activity diagrams instead of the UML statecharts used in existing hybrid system UML profiles [5, 27, 50], since activity diagrams model control flow more akin to hybrid programs. UML statecharts are a language to model system behavior as graphs where the vertices are the states of the system and the edges represent transitions between states. Thus, with some extension UML statecharts are suitable to represent hybrid automata (cf. [5, 27, 50]). UML activity diagrams, in contrast, are a language to model system behavior as graphs where the vertices are actions or decisions and the edges represent control flow. The notion of control flow between actions (statements) makes activity diagrams more suitable to model (computational) processes [20], such as our hybrid programs.

We use model transformation to define the semantics of the Hybrid Program UML profile relative to $\mathsf{d}\mathcal{L}$. Besides defining the semantics of the Hybrid UML profile, model transformations can be implemented as model transformation specifications (e. g., using the Atlas transformation language ATL [21]) and executed to transform models back and forth between the Hybrid Program UML profile and their hybrid program counterparts. Since hybrid automata can be encoded in hybrid programs [42, Appendix C],[5] we define both, a hybrid program semantics and a hybrid automatic semantics, for the Hybrid Program UML profile. Note, however, that hybrid automata, when encoded in $\mathsf{d}\mathcal{L}$, are often less natural to express and also less efficient to verify than well-structured hybrid programs, because they lack program structure that could be exploited during the proof and require additional variables to identify the states of the automaton.

We use *UML profiles* as extension mechanism to provide hybrid system modeling concepts that are not yet present in standard UML. Profiles are the standard way to extend UML with domain-specific modeling concepts [20]. A UML profile is defined by specifying *stereotypes* and *constraints*. A stereotype is applicable to a particular element of the UML (e. g., a classifier) and adds additional modeling capabilities to the original UML element. For example, standard UML actions have a body that we can use to capture an atomic hybrid program, such as deterministic assignment or differential equations. When we want to describe additional information, such as differential invariant constraints or evolution domain constraints of a differential-algebraic equation, we can introduce a stereotype *Dynamics* for UML actions that adds the necessary modeling abilities to actions. Constraints can be added to profiles in order to restrict properties to only admissible values, derive property values from other properties, or otherwise check the consistency of a UML model. UML provides the Object Constraint Language (OCL, [37]) for defining such constraints. In the following paragraphs we describe our profiles for modeling the structure and the behavior of a hybrid system, which was already informally used in Section 3.1.

**System Structure.** Hybrid programs in $\mathsf{d}\mathcal{L}$ use variables and functions over the reals as modeling primitives. Further structuring mechanisms, such as classes, are not yet supported in $\mathsf{d}\mathcal{L}$. In order to still capture and communicate the intended structure of a hybrid program, we provide stereotypes for UML class diagrams[6]. Fig. 5 shows the stereotypes currently available in the Hybrid UML profile for modeling the structure of a hybrid system.

A hybrid system usually consists of multiple *entities* [36], which are either *objects* that may evolve through manipulation but not by themselves, or *agents* whose evolution is driven by the decisions of a controller (e. g., the robot agent that has to avoid obstacles).

[5] The transformation from hybrid automata into hybrid programs follows the same principle as implementing a finite automaton in a programming language. The converse transformation from hybrid programs into hybrid automata is based on the transition structure induced by the semantics of hybrid programs [42, 44].

[6] In principle, a single class would already be a valid structure for a hybrid program. It is useful to split the system into multiple separate classes corresponding to different entities in the system.

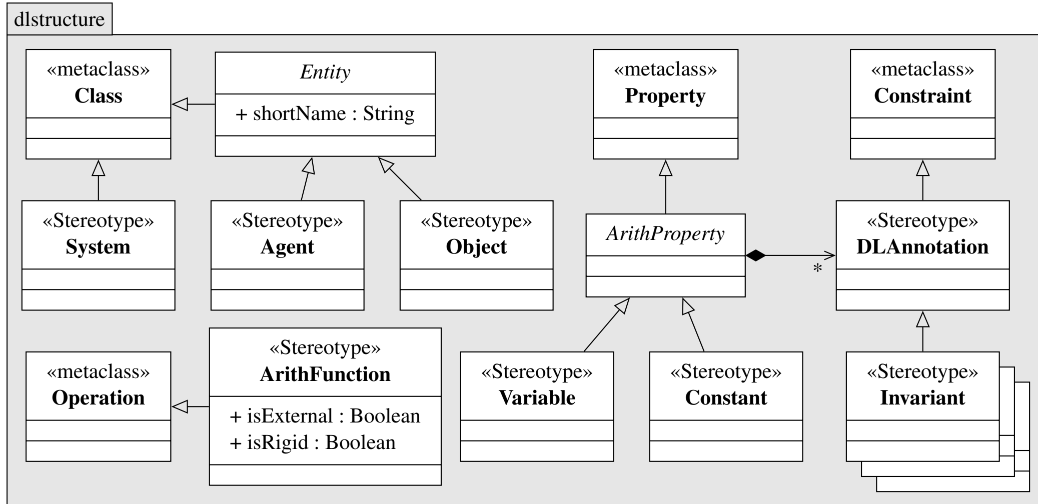



FIGURE 5. The Hybrid Program UML profile: structure of hybrid programs

Entities are usually characterized by some *properties* (e. g., the robot's position) [23]. These properties can be discerned into *constant* properties (cf. *Constant*: their value can be read but not written, e. g., the position of a stationary obstacle), whereas others can change and are therefore called *fluent* [49] or *variable* (cf. *Variable*: their value can both be read and written, e. g., the position of the robot). These stereotypes can be equivalently modeled using standard UML notation *readOnly* for properties. Additional constraints may apply to properties (e. g., a minimum positive braking force $B > 0$, or bounds on the acceleration $-B \leq a \leq A$). We can model these constraints using the stereotype *Invariant* already in the structure of the system, if the constraints have to be satisfied throughout model execution; otherwise, they are rather part of system behavior. We use $\mathsf{d}\mathcal{L}$ to formalize those constraints, since OCL does not support arbitrary arithmetic expressions. Some properties in a hybrid program are shared among all entities (e. g., time). A class marked with the stereotype *system* can capture such shared knowledge.

We allow further decomposition of agents into multiple classes. These classes are linked to their respective agent via the *association* concept of UML. If we want to emphasize that an instance of some class is owned by at most one agent at a time, we use *composition* (e. g., a robot's internal control variables could be factored into a dedicated control state, which no other robot has access to). If we want to share instances of a class between multiple agents, we use a standard *association* instead. No further annotation with stereotypes is necessary.

**System Behavior.** Hybrid programs know essentially two kinds of actions that can change the state of a system: instantaneous jumps (i. e., assignment) are part of the discrete control structure of a hybrid system, and differential equations are part of the continuous dynamics of a hybrid system. Fig. 6 shows the stereotypes for modeling the discrete and continuous dynamics of a hybrid system.

Activity diagrams, as introduced briefly above, provide modeling concepts to represent actions (opaque actions are essentially atomic blackbox actions), decisions, guarded control flow, and constraints. In hybrid program UML we distinguish the following actions (actions are represented as rounded rectangles with a name compartment and a body compartment in UML): Nondeterministic assignment (*AssignAny*) chooses any value for the variable. Deterministic assignment (*AssignTerm*) chooses the value defined by an arithmetic term for the variable. Continuous evolution (*Dynamics*) evolves the values of variables along a differential equation system that stays within a maximum evolution domain. The differential invariant of the differential equation system, if known, can be

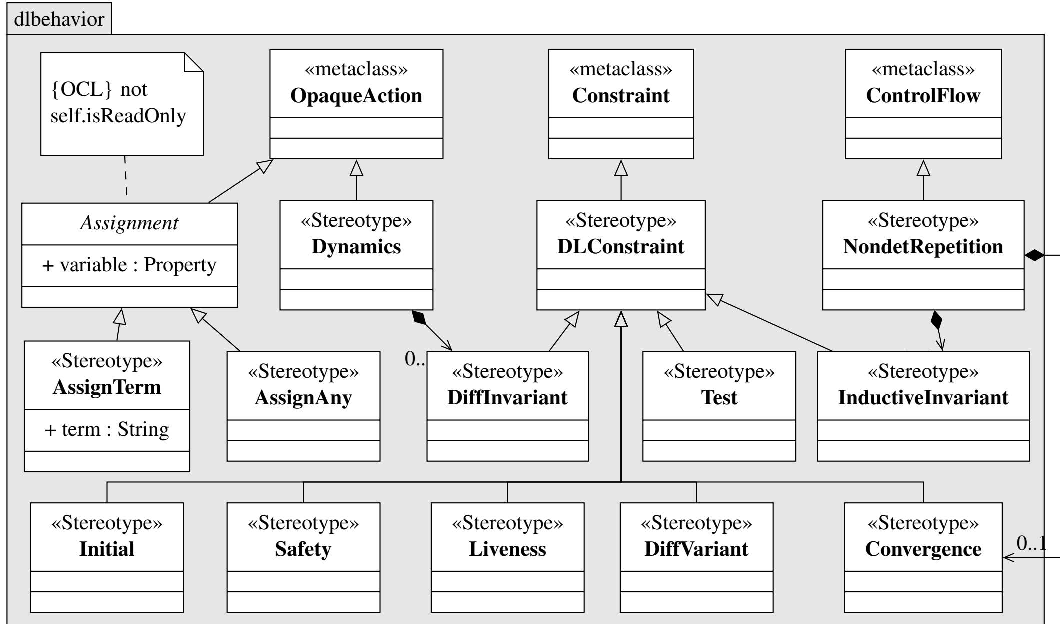



FIGURE 6. The Hybrid Program UML profile: behavior of hybrid programs

annotated as a constraint to the dynamics action. The common features of deterministic and nondeterministic assignment are factored into the abstract base class *Assignment*.

The control flow of a hybrid program can be defined with three composition operations for hybrid program statements: nondeterministic choice, sequential composition, and nondeterministic repetition. Nondeterministic choice can be modeled with standard UML notation for decisions (splitting and merging nodes), while sequential composition is control flow. We introduce a stereotype *NondetRepetition* for control flow (i. e., nondeterministic repetitions will be backwards edges), which can be annotated with a constraint to specify an inductive loop invariant.

Tests in hybrid programs ensure that a particular condition is satisfied in subsequent program statements. They are modeled as algebraic constraints on control flows. Further useful constraints are either part of the specification language (*Initial*, *Safety*, *Liveness*) to express correctness criteria, or guide the verification process but do not influence system behavior (*InductiveInvariant*, *DiffInvariant*, *Convergence*, *DiffVariant*).

We define the semantics of Hybrid Program UML relative to $\mathsf{d}\mathcal{L}$ by transformation specifications from the Hybrid Program UML profile to the hybrid program metamodel. Currently, we support two kinds of transformations: *well-structured* activity diagrams[7] can be transformed into well-structured hybrid programs with loops, whereas arbitrary activity diagrams can be transformed into $\mathsf{d}\mathcal{L}$ automata, which are hybrid automata embedded into hybrid programs with additional variables and tests to represent the states.

**The Hybrid Program Semantics of Hybrid Program UML.** Currently, all constants and variables are handled globally as a flat structure (i. e., the structure present in a Hybrid Program UML model is not yet translated to hybrid programs). The hybrid programs $\alpha$ and $\beta$ in Table 2 represent either one of the atomic actions in Hybrid Program UML (assignment, nondeterministic assignment, or continuous dynamics) or a well-structured part of an activity diagram. An edge between two actions

[7] Well-structured activity diagrams consist of properly nested loops and branches and define a unique initial and final node.

$\boxed{\alpha}$ and $\boxed{\beta}$ corresponds to a sequential composition of the corresponding transformed hybrid programs $\alpha$ and $\beta$, cf. (1). A guard on an edge is transformed into a sequential composition with an intermediate test, cf. (2). A decision node and a matching merge node with a forward edge and a backward edge is translated into a nondeterministic repetition, cf. (3). If the forward edge is missing this means at least one repetition, cf. (4). Finally, a decision node with a matching merge node (but without a back edge) is transformed into a nondeterministic choice, cf. (5).

TABLE 2. The hybrid program semantics of Hybrid Program UML

| | HP UML | Hybrid Program | Description |
|---|---|---|---|
| (1) | $\boxed{\alpha} \rightarrow \boxed{\beta}$ | $\alpha;\ \beta$ | Control flow is a sequential composition |
| (2) | $\boxed{\alpha} \overset{[F]}{\rightarrow} \boxed{\beta}$ | $\alpha;\ ?F;\ \beta$ | Guarded control flow is a sequential composition with intermediate test |
| (3) | $\rightarrow \Diamond \rightarrow \boxed{\alpha} \rightarrow \Diamond \rightarrow$ (with backedge and forward edge) | $\alpha^*$ | Decision node and merge node linked with backedge and forward edge is a nondeterministic repetition |
| (4) | $\rightarrow \Diamond \rightarrow \boxed{\alpha} \rightarrow \Diamond \rightarrow$ (with backedge) | $\alpha;\ \alpha^*$ | Decision node and merge node linked with backedge is a at least one repetition |
| (5) | $\rightarrow \Diamond$ $\rightarrow\boxed{\alpha}\rightarrow$ / $\rightarrow\boxed{\beta}\rightarrow$ $\Diamond \rightarrow$ | $\alpha \cup \beta$ | Decision node and matching merge node are a nondeterministic choice (analogous for more than two branches) |

**The Hybrid Automaton Embedding Semantics of Hybrid Program UML.** The hybrid automaton embedding in hybrid programs matches smaller patterns in an activity diagram. It is thus applicable to a wider range of activity diagrams, which not even have to be well-structured (i. e., arbitrary state jumps are allowed). As a downside, the transformation preserves no explicit program structure (e. g., sequence of statements) that could be exploited during verification. This means that sufficient information about the program structure has to be conveyed in the system invariant. This practice significantly increases verification effort.

The hybrid automaton embedding constructs an automaton-structure from hybrid program notation instead of explicit program structure [42]. It uses an additional variable $s$ to keep track of the current state of the automaton. A unique identifier per vertex and edge of the activity diagram identifies the automaton states. The hybrid automaton embedding is then a nondeterministic choice over all the states, and the control flow is translated into updates of the current state with the respective follow-up state, as summarized in Table 3.

**3.5.4. Hybrid System Simulation.** An interesting opportunity for inspecting the behavior of a hybrid system during the modeling phase (prior to verification) is provided by Mathematica 9, which is able to simulate and plot hybrid system behavior using a combination of *NDSolve* and *WhenEvent* conditions[8]. We envision transforming corresponding excerpts of $\mathsf{d}\mathcal{L}$ to Mathematica for visualizing plots of the dynamic behavior of a hybrid program over time in Sphinx. Simulations can be useful for debugging system models and quickly conveying intuitions about their behavior to the respective members of the collaborative verification-driven engineering team.

**3.5.5. During the Proof.** Collaboration support in Sphinx includes model and proof comparison tools, both locally and with the model and proof repositories maintained in a central source code repository. For this, not only textual comparison is implemented, but also structural comparison of

[8] www.wolfram.com/mathematica

TABLE 3. The hybrid automaton embedding semantics of Hybrid Program UML

| | HP UML | Hybrid Automaton Embedding in Hybrid Program | Description |
|---|---|---|---|
| (1) | $\boxed{\alpha} \rightarrow \boxed{\beta}$ | $(?s = id(\alpha);\ \alpha;\ s := id(\beta)) \cup (?s = id(\beta);\ \beta)$ | Control flow between two actions is nondeterministic choice of two states: each is a sequential composition of a state test and the actual atomic action, with the transition modeled as assignment of a new state ID |
| (2) | $\boxed{\alpha} \rightarrow \Diamond$ | $?s = id(\alpha);\ \alpha;\ s := id(\Diamond)$ | Control flow between an action and a decision/merge node is a sequential composition of a state test and assignment of a new state ID |
| (3) | $\boxed{\alpha} \overset{[F]}{\rightarrow} \boxed{\beta}$ | $(?s = id(\alpha);\ \alpha;\ ?F;\ s := id(\beta)) \cup (?s = id(\beta);\ \beta)$ | Guarded control flow is a sequential composition with an intermediate test |
| (4) | $\rightarrow \Diamond \begin{smallmatrix} \rightarrow \boxed{\alpha} \\ \rightarrow \boxed{\beta} \end{smallmatrix}$ | $?s = id(\Diamond);\ \big(s := id(\alpha) \cup s := id(\beta)\big)$ | Control flow between a decision node and actions is a nondeterministic choice (analogous for more than two branches) |

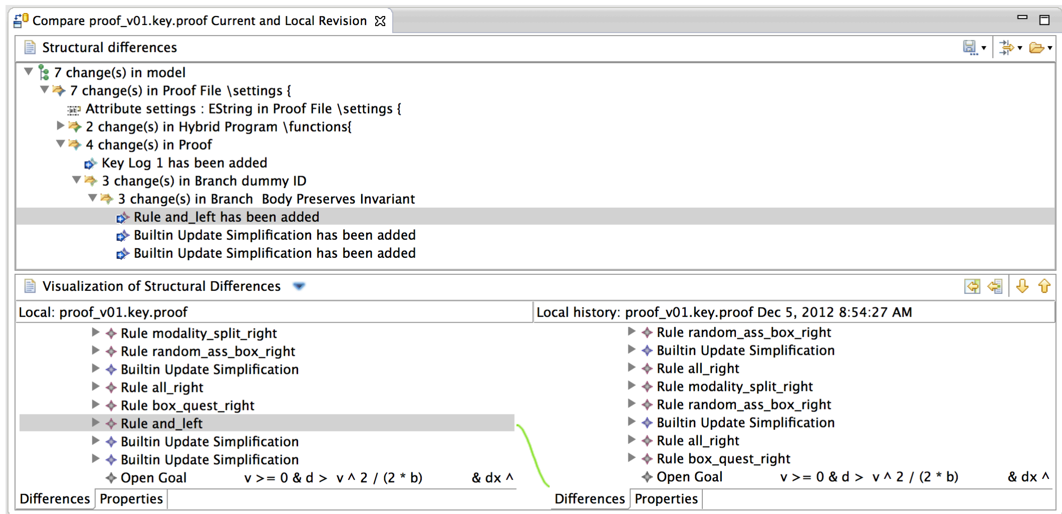



FIGURE 7. Comparison of the structure of two proof versions

models expressed in terms of the d$\mathcal{L}$ metamodel and proofs expressed in terms of the d$\mathcal{L}$ proof metamodel is supported (cf. Fig. 7). Exchanging proofs and inspecting updates on partial proofs is vital especially for collaboration. For example, highlighted changes between different versions of a partial proof lets one easily spot and adopt proof progress made by other team members, go back and forth between versions, and detect conflicts.

Specific unsolved subproblems of a proof (e. g., complex arithmetic problems) can be flagged in KeYmaera and extracted to other tools to further facilitate knowledge and expertise exchange. That makes it easier to participate the verification effort and collaborate in jointly coming up with a solution. An open question, however, concerns the merging of the partial verification results into

a single coherent proof without recourse to external verification steps. In a first step, in Sphinx we only allow exchanging proof strategies that can be executed by KeYmaera. Sphinx injects these proof strategies directly into a $\mathsf{d}\mathcal{L}$ proof instead of an open goal, and KeYmaera checks the injected proof steps for correctness. That way, external (arithmetic) solvers can replace manual verification effort without compromising proof trustworthiness.

Later, actual proof certificates and further proof strategies will be exchanged to further increase trust, and more sophisticated comparisons of proof goals are envisioned to more robustly support replaying proofs.

### 3.6. The Collaboration Backend

The Sphinx modeling tool uses existing Eclipse plugins to connect to a variety of backend source code repositories and online project management tools. As source code repository we utilize Subversion[9] and the Eclipse plugin Subclipse[10]. Currently, Mylyn[11] and its connectors are used for accessing online project management tools (e. g., Bugzilla[12], Redmine[13], or any web-based tool via Mylyn's Generic Web templates connector) and exchanging tickets (i. e., requests for verification). These tickets are the organizational means for collaborating on verification problems and tasks within a working group. Exchange of models and proofs may then be conducted either by attaching files to tickets, or by linking tickets directly to models and proofs in the source code repository. In the latter case, one benefits from the model and proof comparison capabilities of Sphinx. Verification tools (currently KeYmaera), are linked to the modeling tool by implementing extensions to the Eclipse launch configuration. These extensions hook into the context menu of Eclipse (models in $\mathsf{d}\mathcal{L}$ and $\mathsf{d}\mathcal{L}$ proof files in our case) and, on selection, launch an external program.

## 4. Application Example

In this section we illustrate a verification example of an autonomous robot [32] that we collaboratively developed and solved using KeYmaera and geometric relevance filtering. We compare the effort of using KeYmaera interactively, KeYmaera fully automated, and KeYmaera together with geometric relevance filtering connected via Sphinx.

With the increased introduction of autonomous robotic ground vehicles as consumer products—such as autonomous hovers and lawn mowers, or even accepting driverless cars on regular roads in California—we face an increased need for ensuring product safety not only for the good of our consumers, but also for the sake of managing manufacturer liability. One important aspect in building such systems is to make them scrutable, in order to mitigate unrealistic expectations and increase trust [51]. In the design stage of such systems, formal verification techniques ensure correct functioning w.r.t. some safety condition, and thus, increase trust. In the course of this, formal verification techniques can help to make assumptions explicit and thus clearly define what can be expected from the system under which circumstances (before the system is built and executed).

We are going to illustrate a design and verification process that encourages collaboration from high-level graphical models which convey intuition about the system to a broad and possibly heterogeneous audience to detailed formal models, which are suitable for formal verification. For this we will discuss our formal model of an autonomous robotic ground vehicle and its proof. More details on the model and case studies, as well as extensions for moving obstacles, sensor uncertainty, sensor failure, and actuator disturbance can be found in [32].

We will begin with a hierarchically structured graphical model that defines the high-level system behavior, the expected operating environment and the initial conditions under which the robot can be activated safely together with the invariants and safety conditions that the robot will then guarantee. We will complement the high-level model with a more detailed robot controller model.

[9] subversion.apache.org [10] subclipse.tigris.org [11] www.eclipse.org/mylyn
[12] www.bugzilla.org [13] www.redmine.org

Together, these models are translated into d$\mathcal{L}$ and formally verified. Finally, we will discuss a simple one-dimensional model of the robot to exemplify how modeling decisions in d$\mathcal{L}$ can make verification easier.

### 4.1. Hierarchical Graphical Modeling

First, we construct a high-level model of the structure and the behavior of the robotic ground vehicle and of the assumptions about the environment it is operating in. Fig. 8 uses the Hybrid Program UML profile to model the structure of the robotic obstacle avoidance algorithm.

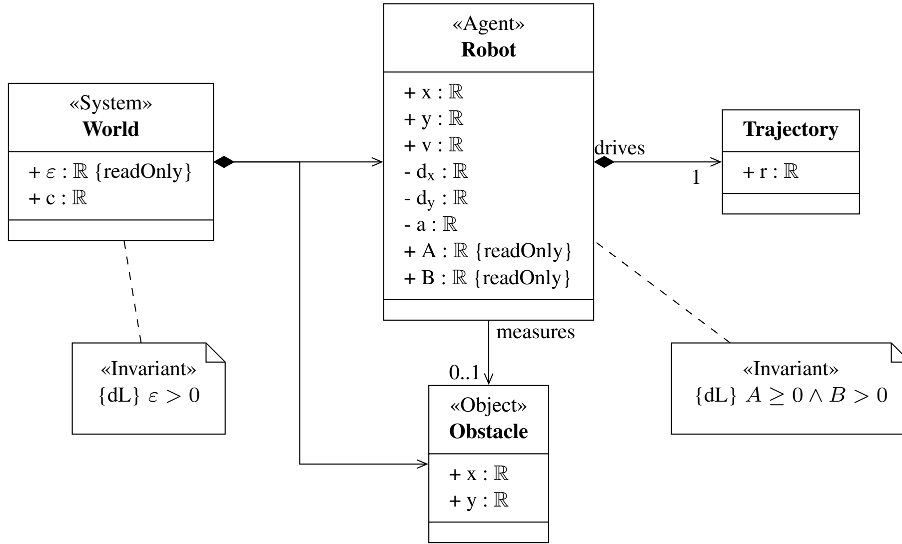



FIGURE 8. The structure of the robotic obstacle avoidance model

The system class *World* provides a global clock $c$ and ensures a cycle time of at most $\varepsilon$ time units (i. e., any controller in the system will run at least once every $\varepsilon$ time units). The state of a robot is characterized by its position $(x, y)$ and orientation $(d_x, d_y)$ in two dimensions and its linear velocity $(v)$. The robot can control its linear acceleration within certain bounds $(a \in [A, B])$ and choose a new trajectory. It measures the position of the nearest obstacle to make decisions about its trajectory.

The high-level behavior of the robotic obstacle avoidance algorithm is modeled in a hierarchical activity diagram using our Hybrid Program UML profile. Fig. 9 shows the high-level behavior with the controller and the dynamics. In this example, the dynamics is a non-linear differential-algebraic equation that describes the robot's motion on a circular segment: $x' = vd_x, y' = vd_y, d_x' = -\frac{vd_y}{r}, d_y' = \frac{vd_x}{r}, v' = a \;\&\; v \geq 0 \wedge c \leq \varepsilon$. The high-level behavior further details the initial condition under which the obstacle avoidance algorithm is safe to start and the safety condition that we want to be true for all executions (in these conditions we use $p_r = (x, y)$ to denote the position of the robot and $p_o$ to denote the position of the obstacle in two dimensions).

A model of such high-level behavior is useful to communicate major design decisions, such as the expected operating environment and the most important constraints that the system must obey. It also consolidates more detailed models that may have been produced by different members of a verification team. As future work we will integrate composite structure diagrams, as in [**?**], to make the interfaces between those detailed models explicit. A more detailed model of the controller complements the high-level *ctrl* block with detailed implementation-specific decisions, as shown in Fig. 10.

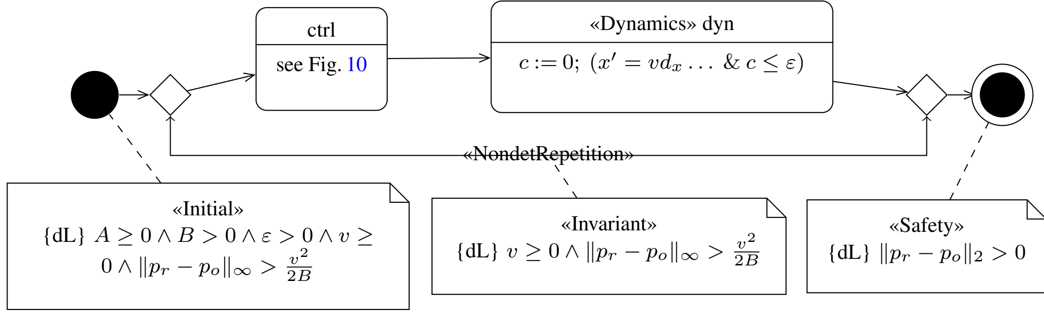



FIGURE 9. Overview of the behavior of the robotic obstacle avoidance model

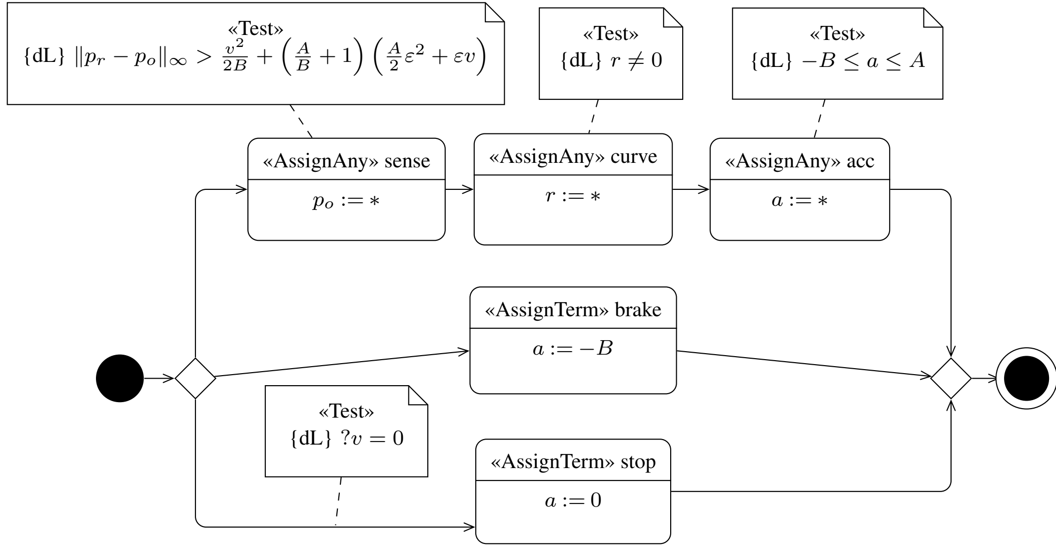



FIGURE 10. The controller of the robotic obstacle avoidance model

The robot has three control options (top to bottom in Fig. 10): If the robot's current state is safe with respect to the sensed position of the nearest obstacle, then the robot may choose a new curve and accelerate with any rate within its physical bounds. For this, we utilize the modeling pattern introduced above: we assign an arbitrary value to the robot's acceleration state ($a := *$), which is then restricted to any value from the interval $[-B, A]$ using a test ($? - B \leq a \leq A$). The robot can brake ($a := -B$), which we want to be an emergency action that should be executed with minimal time delay (i. e., we want braking to be safe even when the robot relies on previously sensed obstacle positions). Finally, if the robot is stopped ($?v = 0$), it may choose to remain in its current spot ($a := 0$).

To stay always safe the robot must account for (i) its own braking distance ($\frac{v^2}{2B}$), (ii) the distance it may travel with its current velocity ($\varepsilon v$) until it is able to initiate braking, and (iii) the distance needed to compensate the acceleration $A$ that may have been chosen in the worst case. For a complete model of the robotic obstacle avoidance algorithm and further variants as a hybrid program we refer to [32].

In the following paragraphs we discuss how the structure of that model and the resulting proof structure can be exploited to facilitate collaboration during the proof.

The d$\mathcal{L}$ proof calculus provides proof rules to syntactically decompose a hybrid program into smaller, easier provable pieces. Such a proof unfolds into many subgoals that often can be handled separately. Proof 1 sketches the proof structure of the robot obstacle avoidance safety proof together with the proof rules used in the proof sketch[14]. The names in the proof are the abbreviations that we introduced in the graphical model as placeholders for more complicated formulas, which get expanded when necessary. The three control choices of *ctrl* are transformed by the proof rule $[\cup]r$ into a conjunction, which is further split by the proof rule $\wedge$r into separate branches in the proof.

**Proof 1** Proof sketch of the robot obstacle avoidance algorithm

$$([\cup]r)\ \frac{\Gamma \vdash [\alpha]\phi \wedge [\beta]\phi, \Delta}{\Gamma \vdash [\alpha \cup \beta]\phi, \Delta} \qquad (\wedge\mathrm{r})\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \qquad ([;])\ \frac{[\alpha][\beta]\phi}{[\alpha;\beta]\phi} \qquad (\mathrm{Wr})\ \frac{\vdash}{\vdash \phi} \qquad (\mathrm{Wl})\ \frac{\vdash}{\phi \vdash}$$

$$
\mathrm{[;]}\ \dfrac{
  [\cup]r,\wedge r\ \dfrac{
    \mathrm{expert\ A}\ \dfrac{
      \mathrm{Wl,Wr}\ \dfrac{
        \mathrm{QE}\ \dfrac{*}{\tilde{\phi} \vdash \tilde{\psi}}
      }{\phi \ldots \vdash \psi \ldots}
    }{\phi \vdash [\mathit{sense};\ \mathit{curve};\ \mathit{acc}][\mathit{dyn}]\psi}
    \qquad
    [\cup]r,\wedge r\ \dfrac{
      \mathrm{B}\ \dfrac{\ldots}{\phi \vdash [\mathit{brake}][\mathit{dyn}]\psi}
      \quad
      \mathrm{C}\ \dfrac{\ldots}{\phi \vdash [?v=0;\ \mathit{stop}][\mathit{dyn}]\psi}
    }{\phi \vdash [(\mathit{brake}) \cup (?v=0;\ \mathit{stop})][\mathit{dyn}]\psi}
  }{\phi \vdash [(\mathit{sense};\ \mathit{curve};\ \mathit{acc}) \cup (\mathit{brake}) \cup (?v=0;\ \mathit{stop})][\mathit{dyn}]\psi}
  \ \Big/\ \phi \vdash [\mathit{ctrl}][\mathit{dyn}]\psi
}{\phi \vdash [\mathit{ctrl};\ \mathit{dyn}]\psi}
$$

These branches can be handled separately by different verification team members, who apply further proof rules of the d$\mathcal{L}$ proof calculus to continue the proof (cf. branches expert A, B and C). Towards the leaves of a branch the proof rules of d$\mathcal{L}$ increasingly eliminate hybrid program elements by turning them into first-order real arithmetic formulas. These formulas are often hard to prove, because the d$\mathcal{L}$ proof rules are not designed to automatically identify and eliminate unnecessary context information (e. g., in the brake branch $\phi$ still contains information about acceleration). Quantifier elimination, which is the final step to proof correctness of a first-order real arithmetic formula, is doubly exponential in the formula size [8]. This means that we want to reduce the number of variables at the leaves of the proof as much as possible. At this stage collaboration across different verification tools becomes possible: we can ship off the formulas to an arithmetic tool or expert to discover what information is unnecessary and can then weaken the formulas in the sequent (Wl,Wr) before we invoke the quantifier elimination procedure (QE).

We compare different proof variants of the robot obstacle avoidance algorithm to highlight the potential reduction in proof effort when developers with different expertise collaborate on a proof. Table 4 compares the number of proof branches, the total number of proof steps, the number of manually executed steps, the number of manually executed weaken operations, the number of exported goals and goals solved by the external tool, the proof execution duration, and the memory used during the arithmetic in the proof. The baseline (line 1 in Table 4) is a proof with manual optimization to reduce branching. We created two further variants of the proof: the partly mechanic variant manually weakened obviously unnecessary contextual information to reduce the number of branches in the proof; the fully mechanic variant branches fully automated by KeYmaera. Both variants were proved fully interactively (cf. lines 2 and 5 in Table 4), fully automated in KeYmaera (cf. lines 3 and 6 in Table 4), and automated with real-arithmetic formulas exported to geometric relevance filtering (cf. lines 4 and 7 in Table 4).

The interesting result is that geometric relevance filtering can solve many of the cases introduced by the fully mechanic branching, while it **Todo: fails** on the same highly complex problems

[14] The d$\mathcal{L}$ proof calculus is explained in detail in [42, 43]

as in the partly mechanic case. This means that the external tool directs the manual effort that is still needed in both variants to the interesting cases, while it takes care of much of the tedious work.

### 4.2. Model Variants and Proof Structure

Since it is hard to come up with a fully verifiable model that includes all the details right from the beginning, the models discussed in the previous section and in our previous case studies [33, 34, 32] are the result of different modeling and verification variants. In the process of creating these models, different assumptions and simplifications were applied until we reached the final versions.

In this section, we discuss how various design decisions influence the structure of a proof and thereby the verification workload.

**4.2.1. Modeling.** We use a simplified model of a robot on a one-dimensional track [34]. In this example, navigation of a robot is considered safe, if the robot is able to stay within its assigned area (e. g., on a track) and does not actively crash with obstacles. Since we cannot guarantee reasonable behavior of obstacles, however, the robot is allowed to passively crash (i. e., while obstacles might run into the robot, the robot will never move into a position where the obstacle could not avoid a collision).

Model 1 shows a textual $\mathsf{d}\mathcal{L}$ model of a hybrid system comprising the control choices of an autonomous robotic ground vehicle, the control choices of a moving obstacle, and the continuous dynamics of the system. The system represents the common controller-plant model: it repeatedly executes control choices followed by dynamics, cf. (1). The control of the robot is executed in parallel to that of the obstacle, cf. (2).

Once again, the robot has three options: it can brake unconditionally, cf. (3). If its current state is safe (defined by (6)), then the robot may accelerate with any rate within its physical bounds, cf. (4). Finally, if the robot is stopped, it may choose to remain in its current spot and may or may not change its orientation while doing so, cf. (5). This is expressed again by arbitrary assignment with subsequent test: this time, the test $?o_r^2 = 1$, however, restricts the orientation value to either forwards or backwards ($o_r \in \{1, -1\}$).

To stay safe the robot must account for the worst case braking, travel, and acceleration distance, cf. (6). This safety margin applies to either the upper or the lower bound of the robot's area, depending on the robot's orientation: when driving forward (i. e., towards the upper bound), we do not need a safety margin towards the lower bound, and vice versa. This is expressed by the factors $\frac{1-o_r}{2}$ and

TABLE 4. Proof complexity in KeYmaera with and without collaboration

| | Variant | Proof Size | | Manual Steps | | Exported | Time | RAM |
|---|---|---|---|---|---|---|---|---|
| | | Branches | Steps | All | Weaken | (Solved) | | |
| (1) | Baseline | 67 | 868 | 139 | 84 | 0 | 34.8 (2.4) | 51.7 |
| (2) | Partly mechanic (interactive) | 67 | 935 | 202 | 148 | 0 | 45.8 (11.6) | 52.9 |
| (3) | Partly mechanic (auto) | | | | | | | |
| (4) | Partly mechanic (GRF) | 67 | 980 | 108 | 54 | 18 (13) | 38.1 (4) | 52.2 |
| (5) | Mechanic (interactive) | 87 | 1193 | 440 | 356 | 0 | 46.9 (3.7) | 52.2 |
| (6) | Mechanic (auto) | | | | | | | |
| (7) | Mechanic (GRF) | 87 | 1230 | 139 | 55 | 32 (28) | 46.4 (4.2) | 52.4 |

**Model 1** Single wheel drive without steering (one-dimensional robot navigation)

$$swd \equiv (ctrl; dyn)^* \tag{1}$$
$$ctrl \equiv (ctrl_r \parallel ctrl_o) \tag{2}$$
$$ctrl_r \equiv (a_r := -B) \tag{3}$$
$$\cup\, (?safe;\ a_r := *;\ ?-B \le a_r \le A) \tag{4}$$
$$\cup\, (?v_r = 0;\ a_r := 0;\ o_r := *;\ ?o_r^2 = 1) \tag{5}$$
$$safe \equiv x_{\underline{b}} + \frac{1-o_r}{2}\left(\frac{v_r^2}{2B} + \left(\frac{A}{B}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v_r\right)\right) < x_r < x_{\overline{b}} - \frac{1+o_r}{2}\left(\frac{v_r^2}{2B} + \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v_r\right)\right) \tag{6}$$
$$\wedge\, \|x_r - x_o\| \ge \frac{v_r^2}{2B} + \left(\frac{A}{B}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v_r\right) + V\left(\varepsilon + \frac{v_r + A\varepsilon}{B}\right) \tag{7}$$
$$ctrl_o \equiv \left(?v_o = 0;\ o_o := *;\ ?o_o^2 = 1\right) \tag{8}$$
$$\cup\, (v_o := *;\ ?0 \le v_o \le V) \tag{9}$$
$$dyn \equiv (t := 0;\ x_r' = o_r v_r,\ v_r' = a_r,\ x_o' = o_o v_o,\ t' = 1\ \&\ v_r \ge 0 \wedge v_o \ge 0 \wedge t \le \varepsilon) \tag{10}$$

$\frac{1+o_r}{2}$ , which mutually evaluate to zero (e. g., $\frac{1-o_r}{2} = 0$ when driving forward with $o_r = 1$). The distance between the robot and the obstacle must be large enough to (i) allow the robot to brake to a stand-still, (ii) compensate its current velocity and worst-case acceleration, and (iii) account for the obstacle moving towards the robot with worst-case velocity $V$ while the robot is still not stopped, cf. (7). Note, that w.r.t. the obstacle we have to be more conservative than towards the bounds, because we want to be able to come to a full stop even when the obstacle approaches the robot from behind.

The obstacle, essentially, has similar control options as the robot (with the crucial difference of not having to care about safety): it may either remain in a spot and possibly change its orientation (8), or choose any velocity up to $V$, cf. (9).

**4.2.2. Verification.** We verify the safety of acceleration and orientation choices as modeled in Model 1 above, using a formal proof calculus for d$\mathcal{L}$ [40, 42]. The robot is safely within its assigned area and at a safe distance to the obstacle, if it is able to brake to a complete stop at all times[15]. The following condition captures this requirement as an invariant that we want to hold at all times during the execution of the model:

$$r \text{ stoppable } (o,b) \equiv\ \|x_r - x_o\|\ \ge\ \frac{v_r^2}{2B} + \frac{vV}{b} \wedge x_{\underline{b}} + \frac{1-o_r}{2}\frac{v_r^2}{2B} < x_r < x_{\overline{b}} - \frac{1+or}{2}\frac{vr^2}{2B}$$
$$\wedge\, v_r \ge 0 \wedge o_r^2 = 1 \wedge o_o^2 = 1 \wedge 0 \le v_o \le V$$

The formula states that the distance between the robot to both the obstacle and the bounds is safe, if there is still enough distance for the robot to brake to a complete stop before it reaches either. Also, the robot must drive with positive velocity, the chosen directions of robot and obstacle must be either forwards ($o_r = 1$) or backwards ($o_r = -1$), and the obstacle must use only positive velocities up to $V$.

**Theorem 1 (Safety of single wheel drive).** *If a robot is inside its assigned area and at a safe distance from the obstacle's position $x_o$ initially, then it will not actively collide with the obstacle and*

[15] The requirement that the robot has to ensure an option for the obstacle to avoid a collision is ensured trivially, since the obstacle in this model can choose its velocity directly. In a more realistic model the obstacle would choose acceleration instead; then the robot had to account for the braking distance of the obstacle, too

*stay within its area while it follows the* `swd` *control model (Model 1), as expressed by the provable* $\mathsf{d}\mathcal{L}$ *formula:*

$$r\ \textit{stoppable}\ (o,b) \rightarrow [\textit{swd}]\big((v_r > 0 \rightarrow \|p_r - p_o\| > 0) \wedge x_{\underline{b}} < x_r < x_{\overline{b}}\big)$$

We proved Theorem 1 using KeYmaera. With respect to making autonomous systems more scrutable, such a proof may help in a twofold manner: on the one hand, it may increase trust in the implemented robot (given the assumption that the actual implementation and execution can be traced back to the abstract model). On the other hand, it makes the behavior of the robot more understandable. In this respect, the most interesting properties of the proven model are the definition of *safe* and the invariant, which allow us to analyze design trade-offs and tell us what is always true about the system regardless of its state. As an example, let us consider the distance between the robot and the obstacle that is considered safe: $\|x_r - x_o\| \geq \frac{v_r^2}{2B} + \left(\frac{A}{B}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v_r\right) + V\left(\varepsilon + \frac{v_r + A\varepsilon}{B}\right)$. This distance can be interpreted as the minimum distance that the robot's obstacle detection sensors are required to cover; it is a function of other robot design parameters (maximum velocity, braking power, worst-case acceleration, sensor/processor/actuator delay) and the parameters expected in the environment (obstacle velocity). $\|x_r - x_o\|$ can be optimized w.r.t. different aspects: for example, to find the most cost-efficient combination of components that still guarantees safety, to specify a safe operation environment given a particular robot configuration, or to determine time bounds for algorithm optimization.

With respect to the manual guidance and collaboration needed in such a proof, we had to apply knowledge in hybrid systems and in-depth understanding of the robot model to find a system invariant, which is the most important manual step in the proof above. We further used arithmetic interactions, such as the hiding of superfluous terms to reduce arithmetic complexity, transforming and replacing terms (e. g., substitute the absolute function with two cases, one for negative and one for positive values).

**4.2.3. The Proof Structure of Model Variants.** We now want to discuss the proof structure of different model variants: For example, one can make explicit restrictions on particular variables, such as first letting the robot start in a known direction (instead of an arbitrary direction). Such assumptions and simplifications, of course, are not without implications on the proof. While in some aspect a proof may become easier, it may become more laborious or more complex in another. In this section, we discuss five variants of the single wheel drive model (without obstacle) to demonstrate implications on the proof structure and on the entailed manual guidance needed to complete a proof in KeYmaera.

The following model variants are identical in terms of the behavior of the robot. However, assumptions on the starting direction were made in the antecedent of a provable $\mathsf{d}\mathcal{L}$ formula, and the starting direction as well as the orientation of the robot were explicitly distinguished by disjunction or non-deterministic choice, or implicitly encoded in the arithmetic, as described below.

**Assumed starting direction, orientation by disjunction.** In the first variant, the robot is assumed to start in a known direction, specified in the antecedent of $o_r = 1 \ldots \rightarrow [\textit{swd}](x_{\underline{b}} < x_r < x_{\overline{b}})$. Also, the orientation of the robot is explicitly distinguished by disjunction in $\textit{safe} \equiv (o_r = -1 \wedge x_{\underline{b}} + \ldots < x_r) \vee (o_r = 1 \wedge x_r < x_{\overline{b}} - \ldots)$, and the robot had an explicit choice on turning during stand-still $(?v_r = 0;\ o_r := -o_r; \ldots) \cup (?v_r = 0; \ldots)$.

**Orientation by arithmetic.** In the second variant, we kept the assumed starting direction of the first variant. However, the orientation by disjunction in the definition of *safe* was replaced by using $o_r$ as discriminator value encoded in the arithmetic, as in $\textit{safe} \equiv x_{\overline{b}} - \frac{1+o_r}{2}(\ldots) < x_r < x_{\underline{b}} + \frac{1-o_r}{2}(\ldots)$.

**Arbitrary starting direction by disjunction.** The third variant relaxes the assumption on the starting direction by introducing a disjunction of possible starting directions in the antecedent of the provable formula $(o_r = 1 \vee o_r = -1) \ldots \rightarrow [\textit{swd}](x_{\underline{b}} < x_r < x_{\overline{b}})$.

TABLE 5. Nodes, branches, and manual proof steps of variants

| | Variant | Nodes | Branches | Manual steps | Avoids |
|---|---|---|---|---|---|
| (i) | Assumed starting direction, orientation by disjunction | 387 | 34 | 24 | |
| (ii) | Orientation by arithmetic | 331 | 28 | 25 | $(\vee l)$ |
| (iii) | Arbitrary starting direction by disjunction | 650 | 56 | 44 | |
| (iv) | Arbitrary starting direction by arithmetic | 185 | 17 | 22 | $(\vee l)$ |
| (v) | Replace non-deterministic choice | 160 | 14 | 29 | $([\cup])(\wedge r)$ |

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta} \quad (\vee l)$$

**Arbitrary starting direction by arithmetic.** The fourth variant replaces the disjunction in the antecedent by stating the two orientation options as $o_r^2 = 1$ in $o_r^2 = 1 \ldots \rightarrow [swd](x_{\underline{b}} < x_r < x_{\overline{b}})$.

**Replace non-deterministic choice with arithmetic.** Finally, we replace the non-deterministic turning choice with $(?v_r = 0;\ o_r := *;\ ?o_r^2 = 1; \ldots)$.

Table 5 summarizes the proof structures of the five variants. Unsurprisingly—when considering the rules of the d$\mathcal{L}$ proof calculus [41] as listed in Table 5—disjunctions in the antecedent $(\vee l)$ or in tests of hybrid programs, as well as non-deterministic choices $([\cup])$ increase the number of proof branches and with it the number of manual proof steps. The number of proof branches can be reduced, if we can replace disjunctions in the antecedent (but also conjunctions in the consequent) or non-deterministic choices in the hybrid program by an equivalent arithmetic encoding. Conversely, this means that some arithmetic problems can be traded for easier ones with additional proof branches.

## 5. Conclusion

In this paper, we gave a vision of a verification-driven engineering toolset including hybrid and arithmetic verification tools, and introduced modeling and collaboration tools with the goal of making formal verification of hybrid systems accessible to a broader audience. The current implementation features textual and graphical modeling editors, integration of KeYmaera as a hybrid systems verification tool, model and proof comparison, and connection to various collaboration backend systems.

As a vision of extending collaboration support, it is planned to integrate Wikis and other online collaboration tools (currently, we use Redmine both as project management repository and for knowledge exchange) for exchanging knowledge on proof tactics. Additionally, collaboration with experts outside the own organization can be fostered by linking to Web resources, such as MathOverflow[16]. In such a platform, requests could be forwarded to those experts whose knowledge matches the verification problem best.

Another interesting research direction are refactoring operations to systematically constructing incremental model variants without the need for re-proving the complete model. Refactoring operations are systematic changes applied to a hybrid system model or the properties that we want to prove about them. In a naive way, after a refactoring was applied we would have to reprove all properties about a model. But often a refactoring operation changes only fragments of a model while it leaves the remainder of the model untouched, or the refactored model and properties are systematically reducible to previous proofs by side deduction. A refactoring operation should systematically reduce verification effort by creating new artifacts that are less effort to prove than the complete model.

[16] mathoverflow.net

In a verification-driven engineering process, a refactoring operation creates (i) a refactored model and properties with links to the original model and a description of the applied refactoring; (ii) a correctness conjecture, together with verification tickets in the project management backend. This way, all changes applied to models and their properties via refactoring operations can be provably traced back.

The VDE toolset is currently being tested in a collaborative verification setting between Carnegie Mellon University, the University of Cambridge, and the University of Edinburgh.

## References


[1] R. Alur. Formal verification of hybrid systems. In S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software (EMSOFT)*, pages 273–278. ACM, 2011.

[2] M. Bajaj, A. Scott, D. Deming, G. Wickstrom, M. D. Spain, D. Zwemer, and R. Peak. Maestro—a model-based systems engineering environment for complex electronic systems. In *Proceedings of the 22nd Annual INCOSE International Symposium*, Rome, Italy, 2012. INCOSE.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf, 2012. (last accessed: 2013-01-09).

[4] C. Belta and F. Ivancic, editors. *Hybrid Systems: Computation and Control (part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013*. ACM, 2013.

[5] K. Berkenkötter, S. Bisanz, U. Hannemann, and J. Peleska. The hybriduml profile for uml 2.0. *STTT*, 8(2):167–176, 2006.

[6] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[7] P. Collins and J. Lygeros. Computability of finite-time reachable sets for hybrid systems. In *44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, pages 4688–4693. IEEE, dec. 2005.

[8] J. H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1-2):29–35, Feb. 1988.

[9] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, pages 337–340, Budapest, Hungary, 2008. Springer-Verlag.

[10] B. De Schutter, W. Heemels, J. Lunze, and C. Prieur. Survey of modeling, analysis, and control of hybrid systems. In J. Lunze and F. Lamnabhi-Lagarrigue, editors, *Handbook of Hybrid Systems Control – Theory, Tools, Applications*, chapter 2, pages 31–55. Cambridge University Press, Cambridge, UK, 2009.

[11] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13 –28, jan. 2012.

[12] J. Faber, S. Linker, E.-R. Olderog, and J.-D. Quesel. Syspect - modelling, specifying, and verifying real-time systems with rich data. *International Journal of Software and Informatics*, 5(1-2):117–137, 2011.

[13] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings*, volume 3414 of *LNCS*, pages 258–273. Springer, 2005.

[14] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In S. Q. Ganesh Gopalakrishnan, editor, *CAV*, LNCS. Springer, 2011.

[15] A. S. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Sci. Comput. Program.*, 73(1):39–58, 2008.

[16] T. Gowers and M. Nielsen. Massively collaborative mathematics. *Nature*, 461:879–881, 2009.

[17] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[18] M. C. Hause and F. Thom. An integrated MDA approach with SysML and UML. In *Proc. of the 13th Intl. Conference on Engineering of Complex Computer Systems*, ICECCS '08, pages 249–254, Washington, DC, USA, 2008. IEEE Computer Society.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 332–358. Springer, 2003.

[20] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML @ Work*. dpunkt, 2005.

[21] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[23] M. M. Kokar, C. J. Matheus, and K. Baclawski. Ontology-based situation awareness. *International Journal of Information Fusion*, 10(1):83–98, 2009.

[24] F. Kordon, J. Hugues, and X. Renault. From model driven engineering to verification driven engineering. In *Proc. of the 6th IFIP int. workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 381–393. Springer, 2008.

[25] Y. Kouskoulas, D. Renshaw, A. Platzer, and P. Kazanzides. Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In Belta and Ivancic [4].

[26] O. Kupferman and M. Y. Vardi. Modular model checking. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, COMPOS'97, pages 381–401, London, UK, 1998. Springer.

[27] J. Liu, Z. Liu, J. He, F. Mallet, and Z. Ding. Hybrid marte statecharts. *Frontiers of Computer Science*, 7(1):95–108, 2013.

[28] S. M. Loos and A. Platzer. Safe intersections: At the crossing of hybrid systems and verification. In K. Yi, editor, *ITSC*, pages 1181–1186, 2011.

[29] S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM*, volume 6664 of *LNCS*, pages 42–56. Springer, 2011.

[30] S. M. Loos, D. Renshaw, and A. Platzer. Formal verification of distributed aircraft controllers. In Belta and Ivancic [4].

[31] F. Mallet and R. de Simone. Marte: a profile for rt/e systems modeling, analysis–*and simulation*? In S. Molnár, J. R. Heath, O. Dalle, and G. A. Wainer, editors, *SimuTools*, page 43. ICST, 2008.

[32] S. Mitsch, K. Ghorbal, and A. Platzer. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Robotics: Science and Systems*, 2013.

[33] S. Mitsch, S. M. Loos, and A. Platzer. Towards formal verification of freeway traffic control. In C. Lu, editor, *Proc. of the 2nd Int. Conference on Cyber-Physical Systems (ICCPS)*, pages 171–180. IEEE, 2012.

[34] S. Mitsch, G. O. Passmore, and A. Platzer. A vision of collaborative verification-driven engineering of hybrid systems. In M. Kerber, C. Lange, and C. Rowat, editors, *Do-Form*, pages 8–17. AISB, 2013.

[35] W. Mostowski. The KeY syntax. In B. Beckert, R. Hähnle, and P. H. Schmitt, editors, *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*, pages 599–626. Springer Berlin Heidelberg, 2007.

[36] I. Niles and A. Pease. Towards a standard upper ontology. In *Proc. of the 2nd Int. Conf. on Formal Ontology in Information Systems (FOIS '01)*, pages 2–9, Ogunquit, MN, USA, 2001. ACM.

[37] Object Management Group. OMG object constraint language (OCL). Technical Report formal/2012-01-01, OMG, 2012.

[38] G. O. Passmore. *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. PhD thesis, University of Edinburgh, 2011.

[39] G. O. Passmore, L. C. Paulson, and L. M. de Moura. Real algebraic strategies for MetiTarski proofs. In J. Jeuring, J. A. Campbell, J. Carette, G. D. Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *AISC/MKM/Calculemus*, volume 7362 of *Lecture Notes in Computer Science*, pages 358–370. Springer, 2012.

[40] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008.

[41] A. Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010.

[42] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[43] A. Platzer. The complete proof theory of hybrid systems. In *LICS*, pages 541–550. IEEE, 2012.

[44] A. Platzer. Logics of dynamical systems. In *LICS*, pages 13–24. IEEE, 2012.

[45] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Formal Methods in System Design*, 35(1):98–120, 2009.

[46] A. Platzer and E. M. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[47] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[48] A. Platzer and J.-D. Quesel. European Train Control System: A case study in formal verification. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.

[49] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, Cambridge, MA, USA, 2001.

[50] W. Schäfer and H. Wehrheim. Model-driven development with mechatronic uml. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 533–554. Springer, 2010.

[51] N. Tintarev, N. Oren, K. V. Deemter, R. Kutlak, M. Green, J. Masthoff, and W. Vasconcelos. SAsSy—scrutable autonomous systems. In *to appear in: Proceedings of the 2013 Workshop on Enabling Domain Experts to use Formalised Reasoning (Do-Form)*, 2013.

Stefan Mitsch
Computer Science Department, Carnegie Mellon University
and CIS, Johannes Kepler University
5000 Forbes Ave,
Pittsburgh, PA 15213, USA
e-mail: `smitsch@cs.cmu.edu`

Grant Olney Passmore
LFCS, Edinburgh and Clare Hall, Cambridge
10 Crichton Street,
Edinburgh, UK
e-mail: `grant.passmore@cl.cam.ac.uk`

André Platzer
Computer Science Department, Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213, USA
e-mail: `aplatzer@cs.cmu.edu`