A Metamodel-Based Approach for Adding Modularization to KeYmaera's Input Syntax

Thomas Baar(✉)

Department of Engineering I, Hochschule für Technik und Wirtschaft (HTW) Berlin, Wilhelminenhofstraße 75A, 12459 Berlin, Germany
thomas.baar@htw-berlin.de

Abstract. The theorem prover KeYmaera allows (1) to describe Cyber-Physical Systems (CPSs) in terms of a Hybrid Program (HP), (2) to specify properties for the defined system, and (3) to formally verify these properties using a tailored logic called Differential Dynamic Logic (DDL).

The syntax of Hybrid Programs is rather poor and covers only the most basic program statements, such as assignment, test, sequential execution, and iteration. The decision to keep the syntax of HPs very simple has different consequences: An advantage is that also the verification calculus can be kept relatively simple. On the downside we have that even small programs are hard to understand and that the programmer is forced to program using a copy-and-paste style, which obviously hampers maintenance. The most significant drawback, however, is the absence of modularization and a library concept; making the development and verification of bigger systems a huge burden.

In this paper, we identify several problems of KeYmaera's input syntax and illustrate them with examples. To overcome these problems, we first describe the original syntax in form of a metamodel. Then, we propose to extend this metamodel with established programming concepts such as subprogram and abrupt termination. We illustrate our extensions by using a new graphical concrete syntax. Examples from a recent KeYmaera tutorial serve for our paper as illustration examples.

Keywords: Cyber-Physical System (CPS) · Safety property verification · Theorem proving · Language design · Domain-Specific Language (DSL) · Metamodel

1 Motivation

A Cyber-Physical System (CPS) is a system existing in the real world, which usually consists of both cyber and physical components. The behaviour of a cyber component is determined by the (computer) program, which is executed

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - project number 415309034.

N. Bjørner et al. (Eds.): PSI 2019, LNCS 11964, pp. 125–139, 2019.
https://doi.org/10.1007/978-3-030-37487-7_11

on this component while the behaviour of a physical component follows laws from physics, e.g. for torque, acceleration, velocity, etc. An important subset of CPSs are control systems consisting of sensors, processors, and actuators, whose correct functioning is of upmost importance and should be assured by formal verification techniques.

A hybrid system is a formal model of a CPS. To capture the behaviour of cyber components, the hybrid system needs the notion of programs. The behaviour of physical components are modelled by law in physics, which are formulated in terms of ordinary differential equations (ODEs). The theorem prover KeYmaera is able to formally verify properties of hybrid systems formulated in differential dynamic logic (DDL) [13,18]. In this paper, we analyse DDL as used by KeYmaera as input format. We point out some obstacles of the chosen input syntax and make proposals to overcome them.

One of the main problems of the used DDL is, that this single formalism is used for three different purposes, namely, to (i) describe the system to be analysed (*system description*), to (ii) formulate the properties to be hold for the system (*system specification*), and to (iii) formulate proofs (*system verification*). Note that a proof is a tree of DDL-formulas where each connection between nodes of the proof tree must be justified by one rule of the used proof calculus.

Thus, the very same DDL formalism serves quite different purposes and there are some cases, in which it is hard to say, which purpose a given DDL artefact actually serves. For example, the user of KeYmaera is sometimes forced to reformulate a system description in a non-intuitive way, just to make a property of this system verifiable. In other words, the property *about* the system one would like to prove has a strong influence on the way one describes the system itself! Note that - ideally - one should be able to formulate the system description fully independent from the properties one would like to prove - usually later - about the system. As we illustrate with a model of the very simple bouncing ball example, this independence is sometimes not possible. This makes the usage of KeYmaera rather an art than an engineering discipline.

The input syntax for KeYmaera is very rudimentary and forces the user to describe a system is a Big Blob, since modularization, e.g. by subsystems or subprograms, is simply syntactically not possible. In our analysis, we identify also other weaknesses, for example that the correct function of evolutional states rely on executing the right statement before entering the state or that evolutional states usually share a high portion of ODEs. Unfortunately, the current syntax makes it impossible to let an evolutional state 'inherit' from an already defined evolutional state to prevent a copy-paste style in the system description.

In addition to identifying problems of KeYmaera's input syntax, we also make proposals to overcome these problems. In order to describe our solutions at the right level of abstraction, our solution proposal will address the abstract syntax - which we define in form of a metamodel - instead of the textual concrete syntax. In order to stress the independence of our solution proposals from the concrete syntax, we will employ also a graphical syntax, which is close to the Abstract Syntax Tree (AST).

2 Background

We first review the logical basis of the prover KeYmaera.

2.1 Dynamic Logic (DL)

The term Dynamic Logic (DL) was coined for the first time by Harel et al. in [7], which is based on the work of Pratt [16] and Hoare/Floyd [4,8]. A recent review on the history of Dynamic Logic is given by Pratt in [17].

Dynamic Logic has a long tradition in analysing *programs running on a machine*. (First-Order) Dynamic Logic allows for a program α to formulate properties for the pre- and post-state of the program's execution. Syntactically, DL formulas are built on top of arithmetic terms and arithmetic atomic formulas, such as $x < 5 + 3$. The set of DL formulas is closed under the logical junctors $\wedge, \vee, \rightarrow, \leftrightarrow$, under the quantifiers $\forall$ $\exists$, and under the parametrized modalities $[\alpha]$ (*box*), $<\alpha>$ (*diamond*), where α is a program. A program is syntactically defined as a tree of statements. We have *assignment* (:=), *test* (?), *skip* (*skip*[1]) as atomic statements and *nondeterministic choice* ($\cup$), *sequential composition* (;), and *iteration* ($*$) as composed statements. Furthermore, some derived statements (known as *syntactic sugar*) are allowed. For example, the program *if c then s1 else s2 endif* is defined as an abbreviation for $(?c; s1) \cup (?\neg c; s2)$. In the version of DL supported by KeYmaera, all terms (e.g. 3 + 8) including variables are of type Real, so there is no support for a sophisticated type system. For a thorough introduction to Dynamic Logic in syntax and semantics, the reader is referred to [6].

Semantically, a formula of form $\phi \rightarrow [\alpha]\psi$ claims that program α, when started in a state in which ϕ holds, might not terminate or, in case it actually terminates, will result always in a state, in which ψ holds. The second modality <> (diamond), which can occur in DL-formulas as well, has a different semantics: $<\alpha>\ \psi$ claims that program α terminates and that for at least one post-state the formula ψ holds (note, that α can behave non-deterministically).

As a concrete example, let us consider the formula

$$x > 0 \rightarrow [\mathit{if}\ x > 0\ \mathit{then}\ x := x - 1\ \mathit{else}\ x := -25\ \mathit{endif}; x := x + 1]\ x > 0 \quad (1)$$

The program α within the box modality is the sequential composition (operator ;) of an if-statement and an assignment (operator :=). The claim, formulated by (1) about program α reads as follows: Whenever α is started in a state, in which $x > 0$ holds, then $x > 0$ must also hold once α has terminated (note, that termination of α is not part of the claim). Formula (1) is actually valid, i.e. under all circumstances the formula is evaluated to true (see [6] for a formal definition of validity).

It is rather easy to argue informally on the validity of (1): This implication evaluates only to false, when its premise evaluates to true and its conclusion to

[1] Since *skip* can be simulated by ? *true* it is not supported by all versions of KeYmaera.

false. The premise is $x > 0$. Under this assumption, when executing program α, the then-branch of the first statement (if-statement) is always taken and decreases variable x by one. In the second statement, the value of x is again increased by one, so the value of x in the post-state – let us denote it by x_{post} – is $x_{post} = x_{pre} - 1 + 1$, while x_{pre} denotes the value of variable x in the pre-state. The conclusion of (1) can thus be reduced to the proof obligation $x_{pre} - 1 + 1 > 0$, which can never evaluate to false if we assume $x_{pre} > 0$. Fortunately, we do not have to rely on informal argumentation for showing the validity of (1) but can also use the theorem prover KeYmaera, which proves (1) fully automatically.

Please note that the formulas of DL do not make any claim about the execution time of program α, but only formulate properties on the relationship of α's pre- and post-states. You might just think all statements within program α being executed instantaneously, i.e. their execution does not take any time. This is an important difference to the extension of DL, called Differential Dynamic Logic (DDL), we consider next.

2.2 Differential Dynamic Logic (DDL)

DDL [12] is an extension of DL, which means that every DL formula is also a DDL formula. The same way as DL formulas, a DDL formula usually makes a claim about a program α. However, since DDL formulas are mainly used to describe the behaviour of Cyber-Physical Systems, we rather say that program α *encodes the behaviour of the CPS* instead of α *is executed on a machine*, as we do for programs α of pure DL formulas.

The only difference between DL and DDL is a new kind of statement called *continuous evolution statement* (or simply *evolution statement*), which is allowed to occur in programs α. When during the execution of α a continuous evolution statement is reached, then the execution of this statement *takes time* and the system will stay in the corresponding *evolution state* for a while. Note that this is a new semantic concept of DDL and marks an important difference to pure DL!

Executing the evolution statement means for the modelled CPS to stay in the evolution state as long as it wishes (the time to stay is - in general - chosen non-deterministically). However, the modeller has two possibilities to restrict the time period the system stays in the evolution state: The first possibility is to add a so-called *domain constraint* to the evolution statement, which is a first-order formula and which is separated from the rest of the statement by & (ampersand). The domain constraint semantically means that the system cannot stay longer in the evolution state than the time at which the constraint is evaluated to *true*. In other words: at latest when the evaluation of the domain constraint switches from *true* to *false*, the system has to leave the evolution state.

The second possibility to restrict the time period is to have a sequential composition of an evolution statement followed by a test statement. Theoretically, the machine can leave the evolution state at any time, but if the following test evaluates to *false*, then this branch of execution is dismissed for the logical analysis of the system behaviour. Thus, an evolution statement immediately followed

by a test statement is a general technique to force the system to remain in the evolution state as long as the test condition is evaluated to *false*.

Bouncing Ball as a Simple CPS. We illustrate both the usage of an evolution statement as well as the two mentioned techniques to control the time the system will stay in the evolution state by the following bouncing ball example:

$$\alpha_{BB} \equiv (\{x' = v, v' = -g \;\&\; x \geq 0\}; ?x = 0; v := -cv)* \tag{2}$$

The behaviour of the bouncing ball is described with the help of a new kind of variables, called *continuous variables*. For example, variable x is always a non-negative number and encodes the ball's position and variable v encodes velocity, which can be both positive (going up) or negative (going down). The constant g is the gravitation acceleration and greater 0. The constant c is the damping coefficient, a number between 0 and 1.

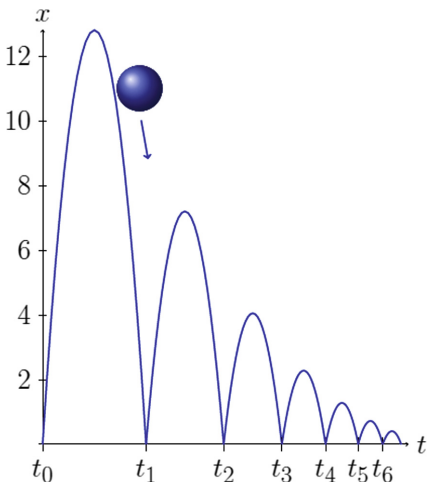

Fig. 1. Sample trajectory of a bouncing ball (Source: [13, p. 98])

The structure of α_{BB} is that of an iteration (operator *) over a sequence (operator ;) of an evolution statement (enclosed by the curly braces), followed by a test (operator ?), followed by an assignment (operator :=). The program α_{BB} is read as follows: The systems starts in a state with given values for variables x and v. These values are not specified yet, but later, we will force the start position x_0 to be a positive number while the start velocity v_0 is allowed to be positive, zero, or negative. As long as the system stays in the first evolution state, the values of x, v will change continuously over time according to physical laws. Thus, the continuous variables x, v represent rather functions $x(t), v(t)$ over time t. The relevant physical laws for x, v are expressed by the two differential equations: $x' = v, v' = -g$.

The latter means that the velocity decreases constantly over time due to the gravitational force of the earth. Fortunately, this ODE has a simple polynomial solution, which facilitates the analysis of the whole system considerably: $v(t) = v_0 + -g * t$. Analogously, depending on the changing velocity v, the position x of the bouncing ball changes with $x(t) = x_0 + v_0 * t + \frac{-g}{2}t^2$.

The domain constraint $x \geq 0$ mentioned in the evolution statement allows the system to remain in the evolution state only as long as x is non-negative. Theoretically, the system can leave at any time the evolution state, but the next statement is the test $?x = 0$. Thus, if the system leaves the evolution state with $x > 0$, then this computational branch will be discarded. Thus, when verifying properties of the system we can rely on the system leaving the evolution state only when $x = 0$, meaning when the ball touches the ground. The following assignment $v := -cv$ encodes that the ball goes up again: The negative value v due to the ball falling down will change instantaneously to a positive value (multiplication with $-c$) but the absolute value of v decreases since the ball loses energy when touching the ground and changing the move direction. Figure 1 shows how the position x of a bouncing ball might change over time (sample trajectory).

3 Problems in Using KeYmaera's Input Syntax

Differential Dynamic Logic as introduced above is supported by KeYmaera and allows to verify formally important properties of technical system as demonstrated in numerous case studies from different domains, e.g. aircrafts [9,14], trains [15], and robots [11].

However, the used input syntax to formulate properties in form of DDL formulas suffers from numerous problems that are described in the following. The solutions we propose to overcome these problems are discussed in Sect. 4.

(1) Invariant specification is not directly supported in DDL. Besides describing the behaviour of hybrid systems as done with program α_{BB} for the bouncing ball, the main purpose of DDL is to specify also properties of such systems. Typical and in practice very important properties are so-called *safety properties*, saying that the system never runs into a 'bad situation'. Let's encode a 'bad situation' with $\neg\psi$. We can show the absence of $\neg\psi$ by proving that in all reachable system states formula ψ holds, i.e. ψ is an invariant. If we assume all statements except the evolution state are executed instantaneously, then showing invariant ψ actually means to show that ψ holds while the system stays in any of its evolution states. However, the modality operators provided by DDL allow only to describe the state *after* the program has terminated. For example, for the bouncing ball system α_{BB} defined in (2) we can prove very easily

$$x = 0 \rightarrow [\alpha_{BB}]x = 0 \tag{3}$$

Note, however, that $x = 0$ has not been proven to be an invariant! If we want to express the interesting invariant, that position x remains all the time within the interval $[0, H]$, while H encodes the system's initial position and if velocity v is initially 0, we have to admit that the formula

$$H > 0 \wedge v = 0 \wedge x = H \wedge 0 < c \wedge c < 1 \rightarrow [\alpha_{BB}]x \leq H \tag{4}$$

is provable, but does NOT encode $x \leq H$ being an invariant because this formula does not say anything about x and H *while* the system stays in the evolution state $\{x' = v, v' = -g \ \& \ x \geq 0\}$, which is part of α_{BB}. In order to prove $x \leq H$ being an invariant the user is forced to reformulate α_{BB} to

$$\alpha'_{BB} \equiv (\{x' = v, v' = -g \ \& \ x \geq 0\}; (skip \cup (?x = 0; v := -cv))* \tag{5}$$

This, however, would be an example for choosing the system description depending on the property we would like to prove! We consider this as bad style.

(2) Evolution state definition cannot be reused. Evolution statements have to contain all ODEs that should hold in the corresponding states. If a program contains multiple evolution statements, then all ODEs usually have to be copied for all these statements, since an ODE normally encodes a physical law that holds in each of the evolution states. Currently, the syntax of KeYmaera does not allow to define all ODEs once and then to reuse this definition for all occurring evolution statements. This lack of reuse results in a copy-and-paste style for describing a system. As an example, we refer to Example 3a from the KeYmaera-tutorial [18], page 10, Eq. (20): $\{p' = v, v' = -a \ \& \ v \geq 0 \wedge p + \frac{v^2}{2B} \leq S\} \cup \{p' = v, v' = -a \ \& \ v \geq 0 \wedge p + \frac{v^2}{2B} \geq S\}$ Here, the definition of the two evolution states (in curly braces) are very similar and defined by copy-and-paste.

(3) Evolution state definition is not encapsulated. In the KeYmaera-tutorials [12,18], there is a frequently applied pattern to ensure that the system stays in an evolution state $ev \equiv \{\ldots \& \ldots\}$ for at most time ϵ. This is achieved by extending the definition of ev to $ev' \equiv \{\ldots, t' = 1 \& \ldots \wedge t \leq \epsilon\}$ while t is a fresh continuous variable. Together with the ODE $t' = 1$, the additional domain constraint $t \leq \epsilon$ forces the system to leave ev' at latest after time ϵ has elapsed. However, this refined definition of ev works only, if the value of t has been set beforehand to 0. In order to achieve this, the statement ev is usually substituted by $t := 0; ev'$. While this pattern works basically in practice, the definition of ev' is not encapsulated and prevents compositionality of programs.

(4) Missing notion of subprogram (or function call in general). Once the examples in the KeYmaera-tutorials [12,18] become a little bit more complicate, they are given in a composed form, e.g. Example 3a from [18, p. 10]: $init \rightarrow [(ctrl; plant)*]req$ where $init \equiv \ldots$, $ctrl \equiv \ldots$, $plant \equiv \ldots$, $req \equiv \ldots$ Presenting a DDL problem in such a composed form highly improves readability. However, the usage of such a composed notation is impossible for the input file of KeYmaera. While one could imagine to

introduce new relational symbols *init*, *req* and to constrain their interpretation by subformulas *init* ↔ ..., *req* ↔ ..., it is currently impossible to define subprograms *ctrl* and *plant* and to compose the resulting program from these subprograms.

4 A Metamodel-Based Approach to Solve Identified Problems

The problems identified above can be overcome by incorporating language concepts from object-oriented programming languages and statecharts into the input syntax of KeYmaera. In order to discuss the incorporated new language concepts at the right level of abstraction, we formulate our proposal in form of a changed metamodel for KeYmaera's input syntax. As a starting point, we present the metamodel of the current syntax.

4.1 Metamodel of Current KeYmaera Syntax

Metamodeling [5] is a widely adopted technique to specify the abstract syntax of modelling and programming languages. One well-known language definition is that of the Unified Modeling Language (UML) [19].

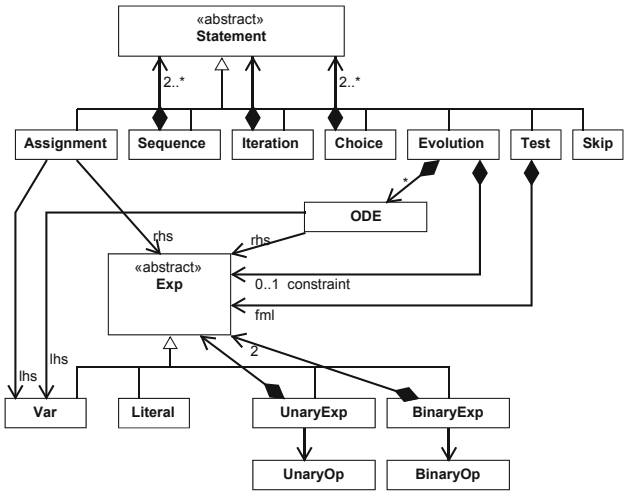

Fig. 2. Metamodel of KeYmaera's input syntax

Figure 2 shows a sketch of the metamodel of KeYmaera's current input syntax with focus on statements within a program. All meta-associations with multiplicity greater than 1 are assumed to be ordered. If the multiplicity on

a meta-association is missing, then 1 is the default value. The metaclass `Exp` represents expressions of both type *Real* (e.g. 5 + x) and of type *Boolean* (e.g. x < 10).

A concrete program α for KeYmaera can be represented by an instance of the metamodel. This instance is equivalent to the result obtained by parsing this program, i.e. the abstract syntax tree (AST).

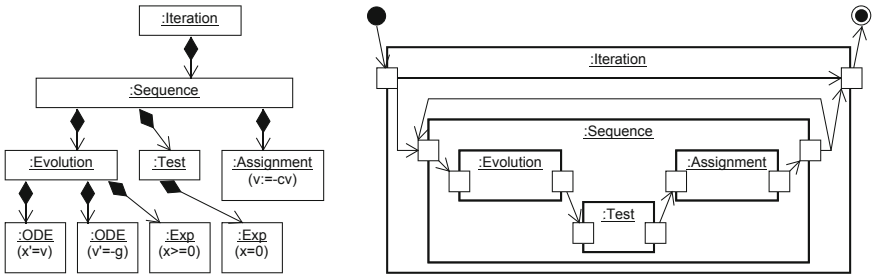

Fig. 3. Instance of the metamodel (left) and control-flow inspired graphical syntax (right) for bouncing ball program (α_{BB})

The left part of Fig. 3 shows the metamodel instance for the bouncing ball program $\alpha_{BB} \equiv (\{x' = v, v' = -g \;\&\; x \geq 0\}; ?x = 0; v := -cv)*$ as defined in (2). In the right part we see an AST-aligned graphical representation of the same program: Each kind of statement is represented by a block with input and output pins. The control flow is visualized by directed edges connecting two pins. The pre-/post-states of the program execution are represented by the symbol for start/final state known from UML's statemachine [19].

4.2 Solutions for Identified Problems

Based of the graphical notation introduced above we discuss now solutions for the problems listed in Sect. 3.

(1) Invariant specification is not directly supported in DDL. As described in Sect. 3, the modal operator $[\alpha]$ refers always to the post-state represented by the final state node in Fig. 3, right part. However, for checking an invariant we need a reference to the state after each evolution statement has been finished. This moment in the execution is represented by the output-pin of the `Evolution` state. What is needed in the program semantics is a direct edge from each output-pin of each `Evolution` state to the final state, as shown in Fig. 4 by the green edge. This concept is known as *abrupt termination*.

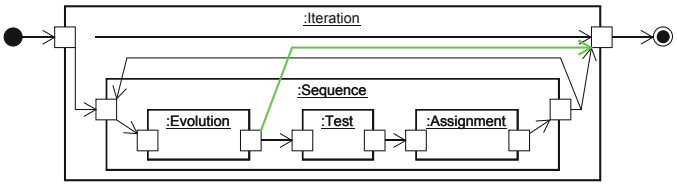

Fig. 4. Solution for invariant specification problem

Note that abrupt termination could be realized without any change of the input syntax of KeYmaera since it requires merely a changed control-flow for the existing statements.

(2) Evolution state definition cannot be reused. Often, the very same ODEs and constraints occur again and again in multiple evolution statements, which hampers readability. To prevent this, our proposal is to introduce the *declaration of* named evolution statements which can be referenced by other evolution statement to - for example - inherit from them ODEs and constraints. The relevant change of the metamodel is shown in Fig. 5.

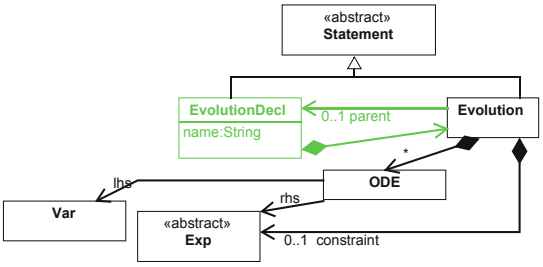

Fig. 5. Solution for evolution state reuse problem

One problem still to be discussed is, whether the declaration of an evolution state can occur at an arbitrary location in the program or should be rather done prior to the program as a global declaration. This question refers to the important issue of which scope the identifier introduced by the declaration (see metaattribute `name`) should actually have. Since resolving the scope of an identifier is rather a problem when parsing a program, this issue is out of scope for this paper.

(3) Evolution state definition is not encapsulated. As demonstrated in the problem definition, an evolution statement sometimes works only as intended when a variable has been set beforehand to the right value. Practically this means, the evolution state EV is always prepended by an assignment $ASGN$, so $(ASGN\,;\,EV)$ has to occur always for correctness. In order to get rid of dependencies of evolution state to assignments from the context

(which prevents a simple reuse of EV within a different context), we propose to extend the evolution state with optional additional statements that are always executed when entering or leaving the state. This state extension is well-known as entry-/exit-actions from UML statemachines. The relevant change of the metamodel is shown in Fig. 6.

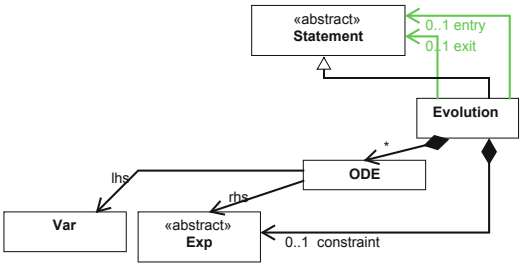

Fig. 6. Solution for evolution state encapsulation problem

(4) Missing notion of subprogram. One of the most basic concepts in programming is the possibility to encapsulate (a block of) statements with a given name and to reuse these statements at various locations of the program. This concept is usually called *subprogram*, *procedure*, or *method*; depending whether parameters are used or not. In general, this is a very old, proven and well understood concept, so that we introduce only the most simple variant in our solution proposal here (cmp. Fig. 7).

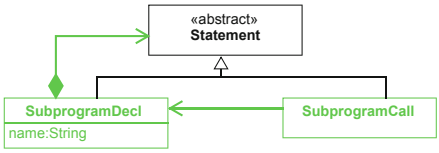

Fig. 7. Solution for missing subprogram problem

5 Towards the Realization of Solution Proposals

In this section we review possible realization options for the proposed solution. Finally, we give a recommendation for one realization option.

5.1 Realization by Extending the Prover KeYmaera

The prover KeYmaera mainly consists of a parser for the input syntax and a calculus in form of proof rules, which even can be changed by the user. In addition, there are some technical components such as (i) a prover engine for

applying proof rules to create a formal proof, (ii) adapters to incorporate external proof systems such as *Z3* or *Mathematica*, and (iii) a GUI to control the proof editing process. However, all these technical components are out of scope for this paper.

For our proposals it is worth to distinguish pure syntactic changes from those, that have an impact on the calculus used by KeYmaera. To the latter belong the support of abrupt termination (problem (1)) and the possibility to invoke subprograms (problem (4)). These changes would require to considerably extend KeYmaera's calculus. While such an extension requires intimate knowledge of the underlying proof engine, it is nevertheless possible, as the prover KeY [1][2] demonstrates. KeY is an interactive verification tool for programs implemented in the language Java and its calculus covers all the subtleties of a real world programming language, including *function calls*, *call stack*, *variable scope*, *abrupt termination* by throwing an exception, *heap analysis*, etc.

Pure syntactic changes among our proposals, i.e. addressing problems (2), (3), could be realized in KeYmaera just by extending the parser. Note that the creation of an alternative concrete input syntax is also topic of the ongoing project called Sphinx [10] carried out by the authors of KeYmaera. Sphinx aims to add a graphical frontend to the prover and will allow the user to specify a program in a pure graphical syntax (similar to our graphical notation proposed in Fig. 3, right part).

The general problem with any deep change of the prover KeYmaera is the technical knowledge it requires. Furthermore, there are good reasons to keep a version of the tool with the original syntax due to its simplicity, what makes it much simpler to use KeYmaera for teaching than, for example, its predecessor KeY. However, a new version of KeYmaera with deep changes is hard to maintain as the original KeYmaera might evolve in future. For these reasons, deep changes can hardly be done by others than the original authors of KeYmaera themselves.

5.2 Realization by Creating a Frontend-DSL

An alternative and flexible approach is the development of a frontend-DSL to incorporate the new language concepts introduced in Sect. 4.2. The main idea is to develop a new Domain-Specific Language according to the given metamodel. Note that the metamodel covers merely the abstract syntax and keeps some flexibility for the concrete syntax. Modern frameworks for defining DSLs such as Xtext and Sirius even allow to have for one DSL *multiple* representations (i.e. *concrete syntaxes*) supported by corresponding editors, e.g. a textual syntax and a graphical syntax. Figure 8 shows the general architecture of such a tool. Note that the new tool will allow the user to interact synchronously with both a textual and a graphical editor to create a model. However, the new models cannot be simply transformed to input files for the original KeYmaera, because the new syntax supports some semantically new concepts such as abrupt termination or subprogram invocation stack. It is the task of the ProofManagement component

[2] Historically, the prover KeY is the predecessor of KeYmaera.

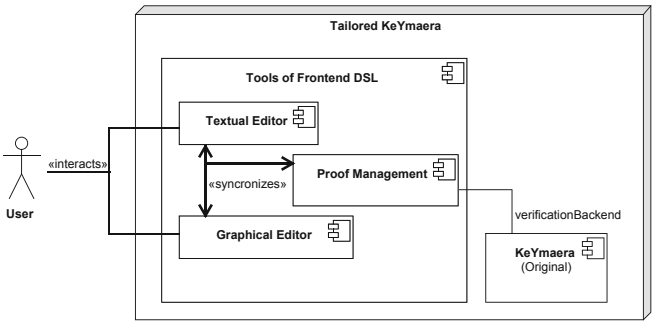

Fig. 8. Architecture of solution using a frontend-DSL

to split tasks - for instance to prove an invariant - into smaller proof obligations, which can be formulated as formulas of differential dynamic logic (DDL) and to pass these obligations to the original KeYmaera tool as verification backend. How an invariant task can be split into smaller proof obligations is demonstrated based on a concrete example in [2].

6 Related Work

The definition of DSLs can be done with numerous technologies, e.g. Xtext, Spoofax, Metaedit, MPS. For realizing a DSL with both a textual and a graphical concrete syntax, the combination Xtext and Sirius is very attractive.

Enriching the prover KeYmaera with a graphical syntax for DDL programs is done in the project Sphinx [10]. The architecture of this tool is pretty similar to our proposal in Fig. 8, but the focus is - in difference to our approach - not the improvement of readability and modularization by making the input syntax richer, but to enable the user to graphically construct a program for DDL.

While enriching a plain, imperative language with concepts from object-oriented programming has been done many times in computing science's history (take the transition from C to C++ or from Modula to Oberon as examples), it is still considered as a challenge. There is an excellent tutorial by Bettini in [3] on how to incorporate into a plain sequential language based on simple expressions additional concepts from object-oriented programming (e.g. *class*, *attribute*, *method*, *visibility*). The resulting language in this tutorial is called *SmallJava* and illustrates almost all technical difficulties when realizing a Java-like programming language in form of a DSL.

7 Conclusion and Future Work

The syntax of programs of differential dynamic logic as supported by the theorem prover KeYmaera have been kept very simple and low level. An advantage of this decision is that also the calculus for proving such programs being correct could be

kept relatively simple and that proofs can be constructed and understood easily. At the downside we have that - once the examples become a little bit more complicate - programs are hard to read, poorly structured, and are impossible to reuse within a different context.

In this paper, we identified four general problems when applying the current program syntax in practice. Furthermore, we made proposals to overcome the identified problems by incorporating proven language concepts from programming languages and from UML's statemachines into KeYmaera's input syntax. These concepts have the potential to make programs scalable and easier to be understood since they foster readability and modularization.

Our proposals have been formulated in form of a changed metamodel representing the abstract syntax of programs. The chosen form for formulating the proposal has the advantage of being very precise while leaving it open, how the changes should actually be realized in a given concrete syntax. Currently, the implementation of a frontend DSL being the main constituent of a *Tailored KeYmaera* tool set is under construction, but not finished yet.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Baar, T., Staroletov, S.: A control flow graph based approach to make the verification of cyber-physical systems using KeYmaera easier. Model. Anal. Inf. Syst. **25**(5), 465–480 (2018)
3. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publisher, Birmingham (2016)
4. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proceedings of Symposium on Applied Mathematics, pp. 19–32. Mathematical Aspects of Computer Science, American Mathematical Society (1967)
5. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley, Hoboken (2008)
6. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundation of Computing. MIT Press, Cambridge (2000)
7. Harel, D., Meyer, A.R., Pratt, V.R.: Computability and completeness in logics of programs (preliminary report). In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, 4–6 May 1977, Boulder, Colorado, USA, pp. 261–268. ACM (1977)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
9. Jeannin, J.-B., et al.: A formally verified hybrid system for the next-generation airborne collision avoidance system. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 21–36. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_2
10. Mitsch, S.: Modeling and Analyzing Hybrid Systems with Sphinx - A User Manual. Carnegie Mellon University and Johannes Kepler University, Pittsburgh and Linz (2013). http://www.cs.cmu.edu/afs/cs/Web/People/smitsch/pdf/userdoc.pdf

11. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Newman, P., Fox, D., Hsu, D. (eds.) Robotics: Science and Systems IX, 24–28 June 2013. Technische Universität Berlin, Berlin (2013)
12. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14509-4
13. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-63588-0
14. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: a case study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_35
15. Platzer, A., Quesel, J.-D.: European train control system: a case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_13
16. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: 17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25–27 October 1976, pp. 109–121. IEEE Computer Society (1976)
17. Pratt, V.: Dynamic logic: a personal perspective. In: Madeira, A., Benevides, M. (eds.) DALI 2017. LNCS, vol. 10669, pp. 153–170. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73579-5_10
18. Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A.: How to model and prove hybrid systems with KeYmaera: a tutorial on safety. STTT **18**(1), 67–91 (2016)
19. Rumbaugh, J.E., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manuel - Covers UML 2.0. Addison Wesley Object Technology Series, 2nd edn. Addison-Wesley, Boston (2005)