Моделирование и анализ информационных систем. Т. 25, № 5 (2018), с. 465–480
Modeling and Analysis of Information Systems. Vol. 25, No 5 (2018), pp. 465–480

Верификация программ
Program Verification

DOI: 10.18255/1818-1015-2018-5-465-480

UDC 004.942

A Control Flow Graph Based Approach to Make the Verification of Cyber-Physical Systems Using KeYmaera Easier

Baar T., Staroletov S.

Received September 10, 2018

Abstract. KeYmaera is an interactive theorem prover and is used to verify safety properties of cyber-physical systems (CPSs). It implements a Dynamic Logic for Hybrid Programs (HPs), while a HP models a CPS very precisely. Verifying properties of a given system in KeYmaera can become a challenge for a user since the proof is authored in a classical sequent calculus framework and a successful proof requires from the user intimate knowledge of the available calculus rules. Another barrier for widespread application of KeYmaera is the purely textual representation of current proof goals, what requires from the user very good training, experience, and patience.

In this paper, we present an alternative verification approach based on KeYmaera, which drastically improves usability and minimizes user interaction. The main idea is to let the user annotate invariants and contracts to states of the hybrid automaton.

Thus, the user can employ the graphical representation of the modelled system and is not bound to the purely textual form of hybrid programs as in KeYmaera. Based on the user-provided contracts, one can generate proof obligations, which are much simpler than the original proof goal in KeYmaera.

The article is published in the authors' wording.

Keywords: CPS, KeYmaera, proof contracts, verification, hybrid systems, usability, interactive provers

For citation: Baar T., Staroletov S., "A Control Flow Graph Based Approach to Make the Verification of Cyber-Physical Systems Using KeYmaera Easier", *Modeling and Analysis of Information Systems*, **25**:5 (2018), 465–480.

On the authors:
Thomas Baar, orcid.org/0000-0002-8443-1558, PhD,
Hochschule für Technik und Wirtschaft Berlin University of Applied Sciences, Germany
75 A Wilhelminenhofstrasse, D-12459, Berlin, Germany, e-mail: thomas.baar@htw-berlin.de

Sergey Staroletov, orcid.org/0000-0001-5183-9736, PhD,
Polzunov Altai State Technical University,
46 Lenina avenue, Barnaul, Altai region, 656038 Russian Federation, e-mail: serg_soft@mail.ru

1. Motivation

A cyber-physical system (CPS) is a system that tightly combines software with physical components. The state of a CPS consists of the discrete state of its software and the analogous state of its physical parts. Safety analysis of CPSs must take into account

physical laws, which apply to physical parts as well as the code structure of the software part [8].

The notion of hybrid automaton (HA) [3, 9] has proven to be useful for the precise description of the behaviour of CPSs. Like a classical UML state machine [5], a hybrid automaton consists of states, transitions between states, and state variables. Transitions can carry annotations for both an execution condition and an action. An action changes the value of a state variable upon executing the transition. New in hybrid automata is, that the value of (some) state variables (called *continuous state variables*) can change according to given differential equations when the system has entered a long-running state. This extension of hybrid automata to classical UML state machines reflects the physical parts of the modeled system. For examples, the current position (z) of the system changes according to the current velocity by $z' = v$, where z' denotes the derivation of z over time.

Logic-based analysis of a given hybrid automaton has been thoroughly investigated by Platzer in [12] and became practically feasible by the tool KeYmaera [14]. This tool is an interactive theorem prover and allows the user to formally prove safety properties taken both discrete and continuous state variables into account. However, KeYmaera does not work directly on the hybrid automaton but needs as input a so-called *hybrid program* (HP). A hybrid automaton can be seen as the control flow graph of a hybrid program.

In this paper, we propose an approach to overcome some of the obstacles the user faces when authoring a proof using KeYmaera. One enormous problem is the complexity of proofs due to the length and complexity of the system implementation represented by a hybrid program. KeYmaera expects a proof goal of form $preCond \rightarrow [\alpha]postCond$, where α is the hybrid program representing the *whole* system implementation.

Instead, our approach applies the idea already formulated in 1967 by Robert W. Floyd [7] for flowchart verification on the verification of a CPS: The user is allowed to annotate the control flow graph of hybrid program α with fine grained knowledge about intermediate states. This additional knowledge can be given in form of invariants (similar to loop invariants) and contracts for long-running states. Based on the provided invariants and contracts, one can generate proof obligations, which are much simpler than the original proof goal in KeYmaera and can often be automatically discarded.

2. Verification of CPSs using KeYmaera

In KeYmaera, a CPS is modelled in form of a Hybrid Program (HP), for which properties expressed in Dynamic Logic can be proven. A HP is built on variables (always of type $float$), derivations of (continuous) variables, arithmetic expressions, first-order formulas for conditions on the current state, and a simple execution language with operators for assignment ($:=$), sequential execution ($;$), non-deterministic repetition ($*$), and others. For a detailed introduction to HP and the logic of KeYmaera, the reader is referred to [13].

2.1. Running Example: Simple Velocity Controller

As an illustrative example, we introduce a simple velocity controller. The velocity v of the controlled system (e.g. a car or a train) is set by the controller either to a fixed velocity v_0 or to 0 (zero). Note that the controlled system is moving if $v > 0$, i.e. the system's position (encoded by z) changes for a time-period Δ with $z = z + v * \Delta$. The change of the system's position based on the current velocity v is a *physical law*, which holds independently from the considered controller and has to be taken into account for all long-running states the systems can be in. An alternative (and widely-established) notation for this law is $z' = v$, what is more general than the above $z = z + v * \Delta$, since velocity v might now even change over time.

Our simple velocity controller periodically updates the chosen velocity based on the information how far away from an obstacle (whose position is encoded with m) the system currently is. If the distance to the obstacle $m - z$ is greater than what the system can move within a period ϵ (encoded in our program as variable SB), the system will keep velocity $v = v_0$. Otherwise controller sets $v = 0$, what means that the system stops (very abruptly). The safety property we want to prove is, that the controller never stops the system too late, i.e. under all circumstances we will have $z < m$.

Our example is actually a simplified version of the tutorial example given in [13] and formulated as a Hybrid Program α as follows:

$$
\begin{array}{ll}
 & \{ \\
 & \quad SB := m - \epsilon * v_0; \\
 & \quad if \quad z < SB \\
\alpha = & \quad then \quad \{v := v_0; t := 0; z' = v, t' = 1 \,\&\, t \leq \epsilon\} \\
 & \quad else \quad \{v := 0; z' = v\} \\
 & \quad endif \\
 & \}*
\end{array}
$$

The Hybrid Program α has the form of the nondeterministic repetition $*$ of a block $\{\ldots\}$ while within the block we have a sequence of statements (separated by ;). Nondeterministic repetition means, that the annotated block can be executed arbitrarily often, including 0 times.

Inside the block, the first statement is the assignment $SB := m - \epsilon * v_0$. The second (and last) statement in the block is an $if - then - else$ statement.

The then-branch is a block consisting of assignments $v := v_0$ and $t := 0$ followed by the last statement in the block: a reference to a long-running state with properties $z' = v, t' = 1 \,\&\, t \leq \epsilon$. Note that after entering a long-running state, the system can remain in this state as long as the state's *domain constraint* (here $t \leq \epsilon$) permits. However, the system can leave the long-running state at any time (nondeterministically) and the program α proceeds with executing the next statement. The differential equation $z' = v$ is the physical law already discussed above while $t' = 1$ is a helper equation for a new variable t. Each long-running state can be annotated with an already mentioned domain constraint, which indicates conditions that must hold as long as the systems stays in this long-running state. In other words: The system must leave the long-running state at latest when the domain constraints flips from $true$ to $false$. In our case, the domain constraint is $t \leq \epsilon$. Together with the equation $t' = 1$ the domain constraint

ensures, that the system stays a maximum period of ϵ in this long-running state.

The else-branch $v := 0; z' = v$ is similar to the then-branch and consists of an assignment and a long-running state.

In the remaining paper, we would like to prove for our hybrid program α the safety property that the system will never reach the obstacle at position m, when it is started in a position smaller than m. Formally, this safety property reads as:

$$z < m \wedge \epsilon > 0 \wedge v_0 > 0 \rightarrow [\alpha] z < m$$

3. Our Approach: Graphical Representation and Contracts

Starting with the textual hybrid program α shown above, we extract the control flow graph (CFG) of α as shown in Fig. 1. The only difference to the original definition of α, that the two long-running states in the program are now named with *driving* and *stopped*.

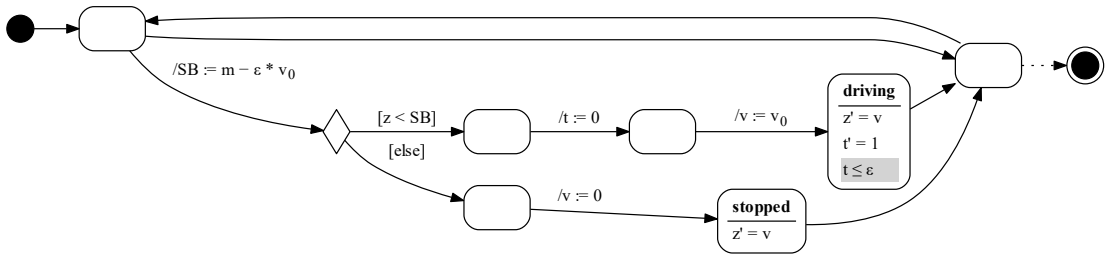

Fig. 1. Control Flow Graph for α (Hybrid Automaton)

The program α is executed by entering the graph via the start node (left side) and following the transitions between the nodes. Transitions can be annotated with an assignment (e.g. $SB := m - \epsilon * v_0$) or with a condition (e.g. $z < SB$). The diamond in the graph represents an if-then-else statement. The nodes for the long-running state contain the annotated differential equations and the domain constraint (gray background). Note that our CFG is very close to the notation of Hybrid Automata [9].

In a second step, our diagram is extended with the safety property to be proven as shown in Fig. 2. This diagram contains notes for a pre- and a post-condition. The tool KeYmaera is supposed to prove now $pre \rightarrow [\alpha] post$, but due to the complexity of α, it often becomes a challenge to manually create a proof using KeYmaera for this claim.

In a third step, we want to make the work of the KeYmaera user much easier. The idea is to provide a system description that already contains key facts for proving the correctness as shown in Fig. 3.

We allow the user to add so-called *proof contracts*. A proof contract can be a pre-/post-condition attached to a long-running state or an invariant attached to a normal state. For example, the desired property $z < m$ basically holds in every state of the

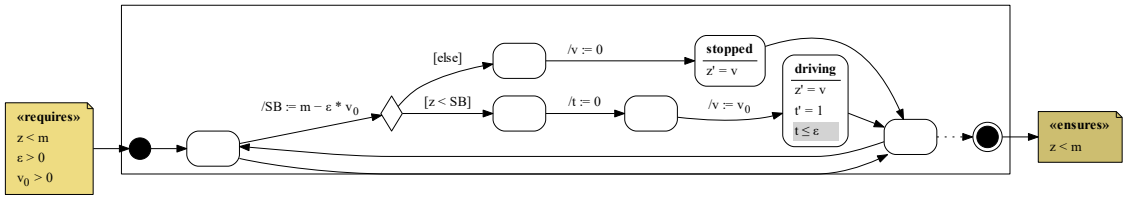

Fig. 2. System specification for α including pre-/post-condition

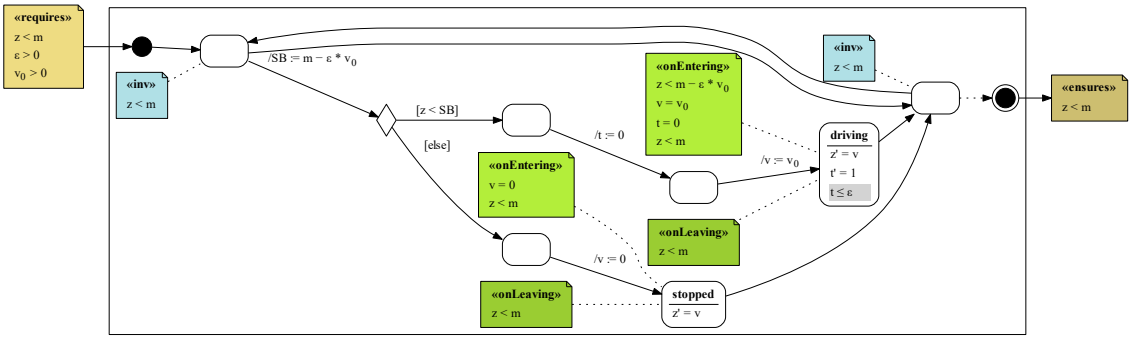

Fig. 3. Proof contracts have been added

control flow graph and especially in the nodes after the start state and before the end state (indicated in Fig. 3 by notes with stereotype ≪inv≫). The long-running state *driving* has now attached a pre-condition $z < m - \epsilon * v_0 \wedge v = v_0 \wedge z < m$ (indicated by note with stereotype ≪onEntering≫; the lines within the note are implicitly connected with logical conjunction $\wedge$) and the post-condition $z < m$ (indicated by a note with stereotype ≪onLeaving≫).

3.1. Generation of Proof Obligations for the Control Flow Graph

Once we have annotated the control flow graph with additional proof contracts, we want to know whether the annotated graph is still correct. Correctness basically means, that between any annotated states s_{pre} and s_{post}, which are connected by a *transition path* $t_1, \ldots, t_n$, the property specified for s_{post} holds, whenever the system evolves from state s_{pre} with its specified properties and executes transitions $t_1, \ldots, t_n$.

To illustrate the approach, all transition paths resulting into proof obligations are marked in Fig. 4. The proof obligations can be grouped according to their characteristics. We discuss here only the most important aspects of the proof obligations. The full version of all proof obligations can be found in appendix A. They are written in the input syntax of KeYmaera.

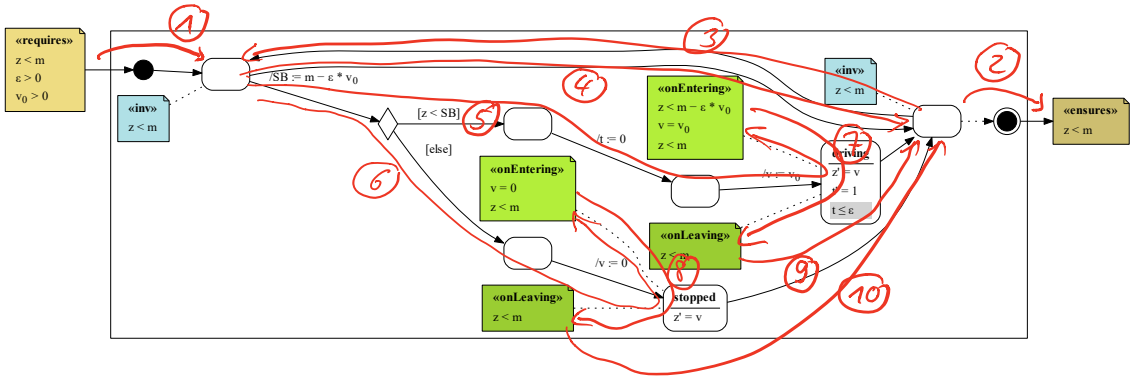

Fig. 4. Transition paths resulting into proof obligations

Transition paths between ≪requires≫/≪ensures≫ properties and first/last state of the system (①, ②)

The control flow graph starts always with a start node and finishes with a final node. The start/final node has a successor/predecessor node respectively, which are annotated with an invariant (in our case, the invariant is in both cases $z < m$, cmp. Fig. 4). Two proof obligations must now ensure, that

- the ≪requires≫ properties entails the invariant of the first node (i.e., the successor node of the start state) – see transition path ①
- the ≪ensures≫ properties is entailed by the invariant of the last node (i.e., the predecessor node of the final state) – see transition path ②

In mathematical notation we have:
$REQUIRES \rightarrow INV_{first}$
$INV_{last} \rightarrow ENSURES$

Transitions paths between invariant nodes (③, ④)

There are two nodes in the system that are annotated with an invariant (called the first/last node, see above). Both nodes are connected by a direct transition in each direction. In our case, these two transitions connect the two states directly and they do not have any annotation (no condition, no assignment). However, annotations would be allowed as well as a sequence of transitions to establish the connection between two invariant nodes.

If two invariant nodes n_1, n_2 are connected with a transition path $t_1, \ldots, t_n$, then the proof obligation has to ensure that the invariant of n_2 is entailed by invariant of n_1 followed by the execution of $\{t_1; \ldots; t_n\}$

In mathematical notation we have:
$INV_{n_1} \rightarrow [\{t_1; \ldots; t_n\}] INV_{n_2}$

Transitions paths between invariant node and ≪onEntering≫ property (⑤, ⑥)

There might be also a transition path $t_1, \ldots, t_n$ connecting an invariant node n_1 with a long-running state n_2. The long-running state must be annotated with an ≪onEntering≫ property. The proof obligation has to ensure that the ≪onEntering≫ property is entailed by invariant of n_1 followed by the execution of $\{t_1; \ldots; t_n\}$

In mathematical notation we have:

$INV_{n_1} \rightarrow [\{t_1; \ldots; t_n\}] ONENTERING_{n_2}$

Note that this is the first proof obligation in our example, in which $\{t_1; \ldots; t_n\}$ is not an empty sequence since the transitions are really annotated with condition/assignment (cmp. appendix A).

Relating ≪onEntering≫ and ≪onLeaving≫ properties for each long-running state (⑦, ⑧)

For each long-running state n, there is a proof obligation that the ≪onLeaving≫ property is entailed by the given ≪onEntering≫ property and the differential equations including the domain constraint attached to the long-running state. Proving this entailment might be non-trivial and usually needs some additional help from the user.

Let's consider node *stopped* for a rather simple example. We have to prove that from the ≪onEntering≫ property $v = 0 \wedge z < m$ and the differential equation $z' = v$ the ≪onLeaving≫ property $z < m$ follows. However, the value of variable z in the ≪onLeaving≫ property might be different from the value for z in the ≪onEntering≫ property. Furthermore, we have to deal with the differential equation $z' = v$, for which our formalism for proof obligations (first-order logic) is not made for.

To cope with this problem, we introduce separate versions of the variables in the ≪onLeaving≫ property and substitute the original variables with the new version. For example, the ≪onLeaving≫ property $z < m$ become $z_{out} < m$ when we introduce z_{out} for z. In addition, we let the user formulate additional formulas to resolve the differential equations attached to the long-running state. In case of state *stopped*, the user might resolve $z' = v$ to $z_{out} = z + v * \Delta$, where Δ encodes the time the systems stays in the long-running state *stopped*. Sometimes, it is important to know that $\Delta >= 0$ holds.

For the long-running state *stopped*, we come up with the following proof obligations: $(v = 0 \wedge z < m) \wedge z_{out} = z + v * \Delta \wedge 0 <= \Delta \rightarrow z_{out} < m$

In mathematical notation we have:

$(ONENTERING_n \wedge AXIOMS\_FOR\_DIFFEQS \wedge$

$DOMAIN\_CONSTRAINT_n \wedge DOMAIN\_CONSTRAINT_n[v \leftarrow v_{out}])$

$\rightarrow ONLEAVING_n[v \leftarrow v_{out}]$

for all variables v that might change their value in state n. Note that $f[v \leftarrow v_{out}]$ denotes the substitution of v by v_{out} in f.

Transitions paths between «onLeaving» property and invariant node (⑨, ⑩)

If a long-running node n_1 is succeeded by a transition path $t_1, \ldots, t_n$ going to an invariant node n_2, we need a proof obligation showing that after leaving n_1 and executing the transition path $t_1, \ldots, t_n$, the invariant of n_2 is entailed.

In mathematical notation we have:
$ONLEAVING_{n_1} \rightarrow [\{t_1; \ldots; t_n\}]INV_{n_2}$

3.2. Discarding the generated proof obligations using KeYmaera

We used KeYmaera version 3.6.17 to show the validity of generated proof obligations. Our KeYmaera installation was 'pure' in the sense that no background prover such as Reduce, Z3, or Mathematica was configured.

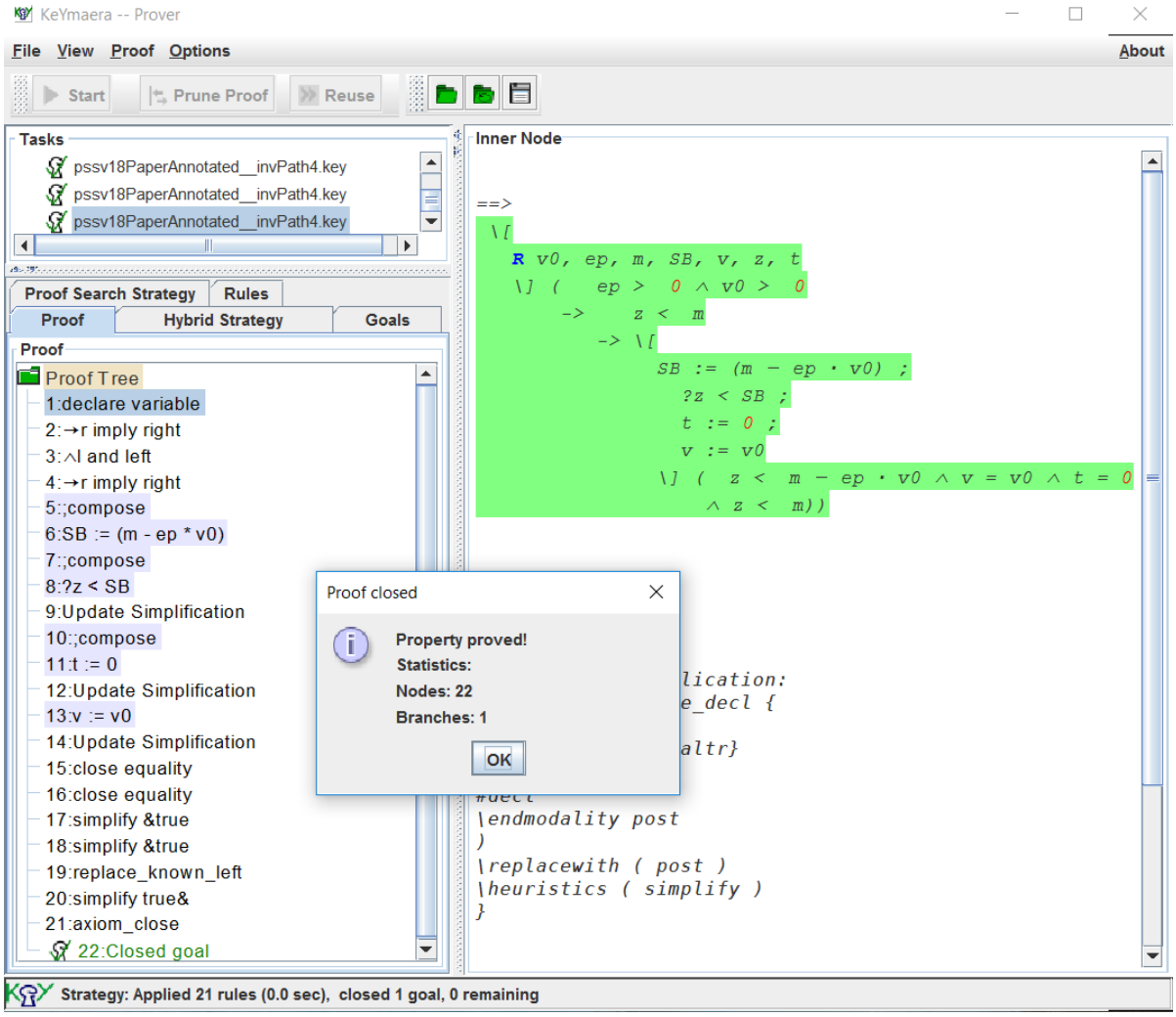

Fig. 5. Automatic proof of obligation using KeYmaera

Nevertheless, all generated obligations except of one could be proven automatically without any additional user interaction. Fig. 5 shows as a typical example the automatic proof of the non-trivial obligation ⑤. The only proof obligation that could not be discarded automatically was obligation ⑦, because the proof exploits transitivity of the $<$ relationship. Nonetheless, the prover PRINCESS [15] could prove also this obligation fully automatically.

4. Related Work

According to the nature of CPSs, we can identify three main layers for their specification: (1) transition automata with discrete jumps, (2) continuous dynamic calculations in each state (what makes the system to be a hybrid system) and (3) safety properties which are interesting for a user to check.

In this section we browse among some known techniques to verify such specifications and describe issues one can face. We use a simple demo system with the velocity controller and the simple goal $z < m$.

Firstly, the user's goals about constant or unexpected behaviour can be easily transformed into LTL(Linear Time Logic) formulas with temporal operators "always" and "eventually". For example, if the property $z < m$ is supposed to hold in all reachable states during execution of the CPS, then we can express this as $[]z < m$.

But after expressing the goals we will get a major issue when we try to describe the behaviour model of a system: It is near to impossible to implement the continuous (or at least close to continuous) dynamic behaviour in each state, even if we hard code the mathematical expressions and solve it without loose of accuracy. The main problem here is an explosion of the number of internal states and memory being used in a verifier to express such a system.

According to the design of a well-known tool in Model Based Checking world, Spin's Promela language doesn't include floating point arithmetic to the models, because the purpose of the language is to encourage abstraction from the computational aspects and focusing on the verification of process interaction [2]. So we need a more than integer-based arithmetic and it cannot be done in most of the cases.

Next, we can move to tools that use rather complex automaton models, especially timed automata. One great representative is Uppaal [4]. It offers construction of such extended automata, check invariants and can verify properties expressed with modalities (i.e. always predicate). For example, if we would like to test $z < m$ during a particular system run, we can check it dynamically by putting $z < m$ as an invariant in desired states or statically verify that goal by using a query with $E[](z < m)$. To describe the system in the Uppaal, we should implicitly create the behaviour automaton. We can introduce control variables and during transitions we can update them by calling our functions that are being written in a code which almost looks like C. The creators of Uppaal made a big step from ordinary discrete automaton – they introduce a SMC(Statistical Model Checking) extension [6] that offers making controlled non-determined transitions, adds double datatype to user's code, adds floating-point clocks type (user can specify the delta step for it in the settings) and they even target to model and verify hybrid systems by introducing time based derivatives in the invariants.

The main disadvantage of writing code for hybrid systems in Uppaal (as in some other tools) is that we should program it almost implicitly using the offered language and it is hard to write complicated ODEs or other types of mathematical models. Uppaal supports time-based derivatives, we can use for checking invariants when staying in a state (as an additional way to check the correctness of a mathematical model implementation, see Figure 6).

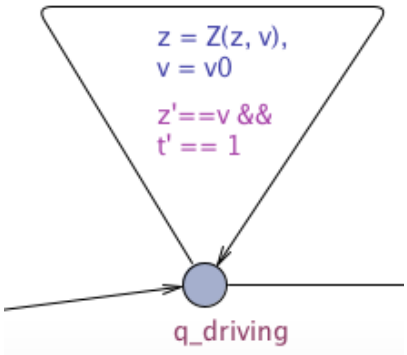

Fig. 6. Derivatives of an invariant on a Uppaal model

Lastly, we refer to the rich world of verification tools for C. Note, that a huge amount of mathematical libraries has been written in C. The modular platform for static analysis Frama-C [10] can prove a lot of types of C programs, it uses the deductive approach and extends the Hoare logic to work with pointers, memory and various type conversions. The floating point arithmetic is supported. They use a Weakest Precondition (WP) method and the verification of the program in this case will consist of calculating the weakest precondition from the end to the beginning of the function code and setting up the problem of proving the reverse derivation to the theorem prover (an internal and some externals interactive provers can be used). So, it is a very strict method and to prove the function, all the precondition, post-conditions, changes the variables, loops, internal function calls must be annotated in a special form (see the invariant with $z < m$ on Figure 7). The ISO-standardized language ACSL [1] is used.

To verify the hybrid system with Frama-C, the mathematical model for it should be solved (by direct or numerical methods) and annotated in C (and annotations can occupy huge places in a code). There is no explicit way to write it in the terms of mathematics.

A similar approach is pursuited by Ariadne [11], a framework implemented in C++. The user can encode a CPS in form of a hybrid automaton including its requirements as instances of C++ classes. More precisely, there are special classes for declaring states, transitions and invariants of the modeled system. After defining the system, the user can execute code to do reachability analysis and prove properties of the described system, especially for safety verification. Ariadne allows parametric verification, which exhaustively checks all possible values of the parameters and determines for which values the component obeys the guarantees. For setting up physical formulas in a model, the user has to use overloaded operations, which are not fully supported by the framework yet. Also, a graphical system representation is not supported yet and enforces the user to work with plain C++-code all time.

```
/*@ requires z0_0 ≥ 0;
    requires z0_0 < m;
    requires eps > 0;
    requires v0_1 > 0;
    requires dt > 0;
    ensures \result < \old(m);
 */
double model(double z0_0, double v0_1, double m, double eps, double dt)
{
  double SB = (double)0;
  double z_0 = z0_0;
  double v = v0_1;
  states q = q_start;
  double t_0 = (double)0;
  int weRun = 1;
  /*@ loop invariant z_0 ≥ 0;
      loop invariant z_0 < m;
      loop invariant t_0 ≥ 0;
      loop invariant weRun ≡ 1 ∨ weRun ≡ 0;
      loop assigns z_0, v, t_0, SB, weRun;
  */
  while (weRun) {
    SB = m - eps * v0_1;
    if (z_0 < SB) {
      t_0 = (double)0;
      /*@ loop invariant z_0 ≥ 0;
          loop invariant z_0 < m;
          loop invariant t_0 ≥ 0;
          loop invariant t_0 ≤ eps;
          loop assigns z_0, v, t_0;
      */
      while (t_0 <= eps) {
        v = v0_1;
      }
      z_0 = Z(t_0,(int)v,z_0);
      t_0 += dt;
    }
```

Fig. 7. Annotations in the simple hybrid model in C

5. Conclusion and Future Work

In this paper, we discussed one of the biggest barrier of verification tools such as KeYmaera to get widely acceptance in industry: They assume the user to be highly trained in mathematical logic and to know in detail the system's proof rules. In addition, a particular problem of KeYmaera is the representation of a proof. The actual proof of a system property can be saved by KeYmaera, but inspection of it by the user is very hard, since the key ideas of a proof are cluttered by many other proof rule applications, which are necessary to get a formal proof right.

Based on a simple but typical example, we illustrated our new approach to let the user annotate key proof facts to the system description itself. As a result, there are much more proof obligations to be proven by KeYmaera, but they are much simpler now and require much less user interaction while the formality of the proof is preserved.

So far, we treated the illustrated example as a pen-and-pencil case study. The next step will be to build a prototypical front-end tool, that allows the user to specify the system graphical as shown in Fig. 3 and which will generate the proof obligations for KeYmaera automatically.

References

[1] Baudin P. et al., *ACSL: ANSI/ISO C Specification Language. Version 1.12*, https://frama-c.com/download/acsl_1.12.pdf.

[2] *Spin: Promela reference. float - floating point numbers*, http://spinroot.com/spin/Man/float.html.

[3] Alur R. et al., "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems", *Hybrid Systems*, Lecture Notes in Computer Science, **736**, 1993, 209–229, https://doi.org/10.1007/3-540-57318-6_30.

[4] Behrmann G. et al., "A tutorial on uppaal", *Formal methods for the design of real-time systems*, Springer, 2004, 200–236.

[5] Booch G. et al., *The unified modeling language user guide – covers UML 2.0*, Second Edition, Addison Wesley object technology series, Addison-Wesley, 2005.

[6] David A. et al., "Uppaal SMC tutorial", *International Journal on Software Tools for Technology Transfer*, **17**:4 (2015), 397–415.

[7] Floyd R.W., "Assigning Meanings to Programs", *Proceedings of Symposium on Applied Mathematics*, **19** (1967), 19–32.

[8] *Hybrid Systems*, Lecture Notes in Computer Science, **736**, ed. Robert L. Grossman et al., 1993.

[9] Henzinger Thomas A., "The Theory of Hybrid Automata", *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, 1996, 278–292.

[10] Kirchner F. et al., "Frama-C: A software analysis perspective", *Formal Aspects of Computing*, **27**:3 (2015), 573–609.

[11] Nuzzo P. et al., "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems", *Proceedings of the IEEE*, **103**:11 (2015), 2104–2132.

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

[13] Platzer A., "Logic and Compositional Verification of Hybrid Systems (Invited Tutorial)", *Computer Aided Verification, 23rd International Conference*, Lecture Notes in Computer Science, **6806**, 2011, 28–43.

[14] Quesel J.-D. et al., "How to Model and Prove Hybrid Systemswith KeYmaera:A Tutorial on Safety", *STTT*, **18**:1 (2016), 67-91.

[15] Rümmer Ph., *Princess Homepage*, http://www.philipp.ruemmer.org/princess.shtml.

A Generated Proof Obligations

For the running example of this paper introduced in Sect. 2. and for the additional proof contracts formulated in Sect. 3. the following proof obligations were generated (in KeYmaera syntax). For the numbering of each proof obligation (PO) the reader is referred to Sect. 3.1.

1.1. PO 1

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
z < m & ep > 0 & v0 > 0
->
z < m
)
}
```

1.2. PO 2

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
z < m
->
z < m
)
}
```

1.3. PO 3

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
->
( z < m)
)
)
}
```

1.4. PO 4

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
->
( z < m)
)
)
}
```

1.5. PO 5

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
-> \[
SB := m - ep * v0; ? z < SB; t := 0; v := v0
\]
( z < m - ep * v0 & t = 0 & v = v0 & z < m)
)
)
}
```

1.6. PO 6

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
-> \[
SB := m - ep * v0; ? ! z < SB; v := 0
\]
( v = 0 & z < m)
)
)
}
```

1.7. PO 7

For this proof obligation on node *driving* the user provided the additional formula
$z_{out} = z + v * tdiff \wedge t_{out} = t + tdiff$

```
\problem {
\[ R v0, ep, m, SB, v, z, t, z_out, t_out, tdiff ; \] (
ep > 0 & v0 > 0
-> (
(z < m - ep * v0 & v = v0 & t = 0 & z < m) &
(0 <= tdiff) &
(t <= ep) &
(t_out <= ep) &
(z_out = z + v * tdiff & t_out = t + tdiff)
->
z_out < m
)
)
}
```

1.8. PO 8

For this proof obligation on node *stopped* the user provided the additional formula
$z_{out} = z + v * tdiff$

```
\problem {
\[ R v0, ep, m, SB, v, z, t, z_out, t_out, tdiff ; \] (
ep > 0 & v0 > 0
-> (
(v = 0 & z < m) &
(0 <= tdiff) &
(true) &
(true) &
(z_out = z + v * tdiff)
->
z_out < m
)
)
}
```

1.9. PO 9

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
->
( z < m)
)
)
}
```

1.10. PO 10

```
\problem {
\[ R v0, ep, m, SB, v, z, t ; \] (
ep > 0 & v0 > 0
-> (
z < m
->
( z < m)
)
)
}
```

Баар Т., Старолетов С.М., " Упрощение процесса верификации кибер-физических систем с использованием подхода с графом потока управления в средстве KeYmaera", *Моделирование и анализ информационных систем*, **25**:5 (2018), 465–480.

DOI: 10.18255/1818-1015-2018-5-465-480

Аннотация. KeYmaera является средством интерактивного доказательства теорем и используется для проверки свойств безопасности кибер-физических систем (CPS). Проверка таких свойств в интерактивном режиме может быть осложнена, поскольку доказательство осуществляется с использованием классического секвентного логического исчисления и успешное доказательство требует от пользователя глубоких знаний о доступных правилах, имеющихся в логике исчисления. Еще одним препятствием для широкого применения KeYmaera является представление текущих целей только в виде текста, что предполагает хорошую подготовку пользователя для построения успешных доказательств. В этой статье мы представляем альтернативный метод верификации для KeYmaera, который значительно повышает удобство использования и минимизирует работу пользователей. Основная идея заключается в том, чтобы позволить пользователю добавлять аннотации в виде инвариантов и контрактов к состояниям гибридной программы. В нашем подходе пользователь может использовать графический язык представления моделируемой системы, что позволяет ему не работать с чисто текстовым форматом гибридных программ, являющимся входным для средства KeYmaera. Исходя из предоставленных пользователем контрактов, можно получать доказательства, которые гораздо проще, чем исходная цель доказательств в KeYmaera, и которые могут быть обработаны в большинстве случаев полностью автоматически. Статья публикуется в авторской редакции.

Ключевые слова: кибер-физические системы, KeYmaera, контракты, верификация, гибридные системы, интерактивные доказатели теорем

Об авторах:
Баар Томас, orcid.org/0000-0002-8443-1558, профессор,
Университет прикладных технических и экономических наук г. Берлина, Германия
Wilhelminenhofstrasse 75 A, D-12459, Berlin, Germany, e-mail: thomas.baar@htw-berlin.de

Старолетов Сергей Михайлович, orcid.org/0000-0001-5183-9736, канд. физ.-мат. наук, доцент
Алтайский государственный технический университет им. И.И. Ползунова
проспект Ленина, 46, г. Барнаул, Алтайский край, 656038, Российская Федерация,
e-mail: serg_soft@mail.ru