Dependency Analysis for Hybrid Programs

Yong Kiam Tan
yongkiat@cs.cmu.edu

May 9, 2017

1 Introduction

Formally verifying the correctness of programs in a program logic provides strong static guarantees about the program's runtime behaviour. Unlike static analysis techniques, proving such correctness properties typically requires some form of user interaction, e.g. the user might need to manually state appropriate loop invariants. In the context of $\mathsf{d}\mathcal{L}$, this problem is somewhat exacerbated because one has to reason about both discrete loop invariants, and continuous differential invariants for hybrid programs.

In this paper, we first explore a version of dependency analysis that we have designed and implemented for hybrid programs in KeYmaera X. We claim that such an analysis is non-trivial in the non-deterministic setting of $\mathsf{d}\mathcal{L}$, and we point out the subtleties in our definition. We then make use of dependency analysis in two application areas: finding differential invariants, and generating a variable order for quantifier elimination.

2 Dependency Analysis

Given a hybrid program α, its semantics, $[\![\alpha]\!]$, is defined as a transition relation between states. For example, the semantics of an assignment to x is defined as:

$$(u, w) \in [\![x := e]\!] \iff w = u_x^{u[\![e]\!]}$$

i.e. u transitions to a state where x takes on the value of e in u. We assume that such a semantics has been defined, e.g. following [Platzer, 2017].

A natural question to ask is if we can determine a relationship between $w(x)$ and $u(y)$ for some set of variables $y \in Y$. In particular, we shall focus on the problem of finding a set of variables Y, whose initial values can affect the final value of x. We call Y the set of variables that x *depends* on. Such a set could be used e.g. as a heuristic when searching for loop invariants.

Note that we can also look for more fine grained forms of relationships. For example, one could look for sign or linear/quadratic relationships between variables. These problems can be handled e.g. using abstract interpretation techniques [Cousot and Cousot, 1977], but they quickly become difficult once ordinary differential equations (ODEs) are involved. Thus, we leave them out of scope for this paper.

2.1 Definition

Let $FV(e)$ be the set of variables mentioned in the term e, and $FV(Q)$ for the set of free variables in first-order formulas Q[1]. For a set of variables X, $D_\alpha(X)$ is the set of variables such that the final value of X after running α depends only on the initial values of those variables.

[1] We assume weak-test $\mathsf{d}\mathcal{L}$ for the purposes of this paper.

$$D_{y:=e}(X) = \bigcup_{x \in X} \begin{cases} FV(e) & \text{if } x = y \\ \{x\} & \text{otherwise} \end{cases}$$

$$D_{\alpha;\beta}(X) = D_\alpha(D_\beta(X))$$

$$D_{\alpha \cup \beta}(X) = D_\alpha(X) \cup D_\beta(X)$$

The subset of rules given above follow mostly immediately from the semantics. For example, $D_{\alpha;\beta}$ first finds the variables that X depends on across a run of β before transitively finding the set of variables that those depend on across a run of α.

$$D_{?Q}(X) = X \cup FV(Q)$$

The first slightly surprising rule is the one for tests. By definition, the values of X are unchanged by the test program. However, we might informally think of this as X being "undefined" in the final state if the test failed. For example, for the test $?y > 0$, there are no final states if $y \leq 0$ initially, so y might cause the values of X to be "undefined". This intuition will become clearer in the example for the loop rule.

For a loop, our dependency needs to be correct regardless of how many times the loop is run. As a starting point, X must depend on itself since we can skip the loop. Following the reasoning for $D_{\alpha;\beta}$, if X depends on a set Y across one run of α, it must also transitively depend on any set of variables that Y depends on. Therefore, we define:

$$D_{\alpha^*}(X) = \bigcup_{i \in \mathbb{N}} D^i_\alpha(X)$$

$$D^0_\alpha(S) = S$$

$$D^{i+1}_\alpha(S) = D_\alpha(D^i_\alpha(S))$$

To see how this works, consider the following program that cycles the values of x, y, z:

$$(t := x; x := y; y := z; z := t)^*$$

Following the semantics, this loop can be repeated for any number of times, which means that all the variables are interdependent. Our analysis progressively computes this. For example,

$$D^0_\alpha(\{x\}) = \{x\}$$

$$D^1_\alpha(\{x\}) = \{y\}$$

$$D^2_\alpha(\{x\}) = \{z\}$$

$$D_{\alpha^*}(\{x\}) = \{x, y, z\}$$

Here, we can also see the importance of our definition of $D_{?Q}(X)$. Consider the program

$$(x := x + 1; y := y + 1; ?y \leq 10)^*$$

This is an implicit for-loop indexed on y. Without the test, we have possible final values $x \in \{x_0 + k | k \in \mathbb{N}\}$, i.e. x is independent of the initial value of y. On the other hand, y restricts the number of times that the loop can be executed so we only have $x \in \{x_0 + k | y_0 + k \leq 10, k \in \mathbb{N}\}$.

For ODEs, we first consider them in vectorial form $\vec{x}' = \vec{e}$ without evolutionary domain constraints. If X does not overlap with $\vec{x}'$, then X remains constant across the ODE. Otherwise, we simply take all the variables that are mentioned in the system.

$$D_{\vec{x}'=\vec{e}}(X) = \begin{cases} X & \text{if } X \cap \vec{x} = \emptyset \\ X \cup \vec{x} \cup \bigcup_{e \in \vec{e}} FV(e) & \text{otherwise} \end{cases}$$

This is certainly an over-approximation, e.g. in the system $x' = 1, y' = 1$, solving the ODE gives $x = x_0 + t$, which only depends on its initial value of x_0 and how long we run the ODE. However, suppose we changed the last ODE to $y' = y^2$. In that case, the final values that x can take becomes dependent on the initial value of y. If $y_0 = 0$, then $x = x_0 + t, y = 0$ is the solution for all t. However, if $y_0 > 0$, then the system has an asymptote at $t = \frac{1}{y_0}$ which it cannot cross, and so the range of final values for x is $[x_0, x_0 + \frac{1}{y_0})$.

To handle evolution domain constraints, we follow the reasoning given for ordinary tests above: Q defines a boundary for the evolution of $\vec{x}$, and so different values for the variables mentioned in Q may lead to different final values for $\vec{x}$.

$$D_{\vec{x}'=\vec{e}\&Q}(X) = D_{\vec{x}'=\vec{e}}(X) \cup FV(Q)$$

Our important use-case for this definition would be for individual variables, i.e. $X = \{x\}$. We write $D_\alpha(x)$ as a shorthand for this case.

2.2 Correctness

We say that two states u, v agree on X if $\forall x \in X. u(x) = v(x)$. Furthermore, we assume that $FV(e)$ and $FV(Q)$ have been appropriately defined so that the following coincidence properties for terms and formulas [Platzer, 2016] hold:

Proposition 1 (Coincidence for terms and formulas)**.** *If u, v agree on $FV(e)$, then $u[\![e]\!] = v[\![e]\!]$, and if u, v agree on $FV(Q)$, then $u \in [\![Q]\!] \iff v \in [\![Q]\!]$.*

With these assumptions, we can show that D_α satisfies the following:

Theorem 1 (Restricted coincidence)**.** *If u, v agree on $D_\alpha(X)$ and $(u, u') \in [\![\alpha]\!]$, there exists $(v, v') \in [\![\alpha]\!]$ such that u', v' agree on X.*

Proof. We show this by induction on the form of α

Case $y := e$. By definition, $u' = u_y^{u[\![e]\!]}$ and $v' = v_y^{v[\![e]\!]}$. For any $x \in X$, if $x = y$, then $FV(e) \in D_{y:=e}(X)$, therefore, u, v agree on $FV(e)$ and $v[\![e]\!] = u[\![e]\!]$ by coincidence for terms. Hence, $u'(x) = u[\![e]\!] = v[\![e]\!] = v'(x)$. Otherwise, $\{x\} \in D_{y:=e}(X)$, so u, v agree on x, which is unchanged by the assignment. Therefore $u'(x) = u(x) = v(x) = v'(x)$.

Case $\alpha; \beta$. By definition there exists u'', $(u, u'') \in [\![\alpha]\!], (u'', u') \in [\![\beta]\!]$. Since u, v agree on $D_\alpha(D_\beta(X))$, by IH on α, there exists a state v'' such that $(v, v'') \in [\![\alpha]\!]$, (v'', u'') agree on $D_\beta(X)$. By IH on β, there exists v' such that u', v' agree on X. Therefore, we have that $(v, v') \in [\![\alpha; \beta]\!]$, and (u', v') agree on X.

Case $\alpha \cup \beta$. By definition either $(u, u') \in [\![\alpha]\!]$ or $(u, u') \in [\![\beta]\!]$. In the first case, u, v agree on $D_\alpha(X) \cup D_\beta(X)$ and therefore also on $D_\alpha(X)$. The conclusion follows by IH on α. The other direction is symmetric.

Case $?Q$. By definition, $u \in [\![Q]\!]$, $u = u'$ and u, v agree on X. Moreover, since u, v also agree on $FV(Q)$, coincidence for formulas implies $v \in [\![Q]\!]$. Therefore, $v' = v$ is a suitable witness.

Case α^*. By definition, $(u, u') \in [\![\alpha]\!]^n$ for some $n \in \mathbb{N}$, we proceed by induction on n. If $n = 0$, $u = u'$, but $X = D_\alpha^0(X) \subseteq D_{\alpha^*}(X)$. Therefore, (u, v) agree on X and $v' = v$ is a suitable witness. In the inductive case, we have $(u, u'') \in [\![\alpha]\!], (u'', u') \in [\![\alpha]\!]^n$. We have $D_\alpha^{n+1}(X) \subseteq D_{\alpha^*}(X)$, so by the

outer IH on α, there exists $(v, v'') \in [\![\alpha]\!]$, (v'', u'') agree on $D^n_\alpha(X)$. By the inner IH, there exists v' such that $(v'', v') \in [\![\alpha]\!]^n$ and (u', v') agree on X. Therefore, v' is a suitable witness.

Case $\vec{x}' = \vec{e}\&Q$[2]. We have a solution to the ODE of duration r, $\phi(0) = u, \phi(r) = u'$ satisfying $\phi \vDash \vec{x}' = \vec{e}\&Q$. If $X \cap \vec{x} = \emptyset$, then $\phi \vDash \vec{x}' = \vec{e}\&Q$ implies that ϕ holds X constant, so u, u' agree on X. Since u, v also agree on X, we pick $v = v'$, i.e. the ODE is evolved for 0 time from v. It remains only to show that Q holds initially for v, but this must be true because by definition of $D_{\vec{x}'=\vec{e}\&Q}(X)$, u, v also agree on $FV(Q)$. Therefore, since $\phi(0) = u \in [\![Q]\!]$, we have $v \in [\![Q]\!]$.

In the other case, (u, v) agree on $X \cup \vec{x} \cup \bigcup_{e \in \vec{e}} FV(e)$. Here, we define $\psi(\tau) = \phi(\tau)^{v(\vec{x}^C)}_{\vec{x}^C}$, i.e. $\psi(\tau)$ is identical to $\phi(\tau)$ for all the variables mentioned in the system, but it takes on the values of the initial state v for other variables. We claim that $\psi \vDash \vec{x}' = \vec{e}\&Q$, and that $\psi(r)$ agrees with $\phi(r) = u'$ on X.

Firstly, ψ keeps all the variables not in the system $x \in \vec{x}^C$ constant, and $\psi(0) = \phi(0)^{v(\vec{x}^C)}_{\vec{x}^C} = u^{v(\vec{x}^C)}_{\vec{x}^C} = v$, where the last equality follows since u, v agree on $\vec{x}$.

We now need to show that ψ respects the system. For the evolution domain constraint, we have $\phi(\tau) \in [\![Q]\!]$, but $\psi(\tau)$ agrees with $\phi(\tau)$ except on constant variables $\vec{x}^C$. In particular we need to show that they agree on $FV(Q) \cap \vec{x}^C$, and thus, also on $FV(Q)$. Note that $\vec{x}^C$ is held constant in both ϕ and ψ, i.e. $\phi(\tau), \phi(0) = u$ agree on $\vec{x}^C$ and similarly for $\psi(0), \psi(\tau) = v$. By definition, u, v agree on $FV(Q)$ and therefore, $\phi(\tau), \psi(\tau)$ agree on $FV(Q) \cap \vec{x}^C$.

To see that ψ obeys the ODE, first note that it is suitably continuous for each $\vec{x}_i$ as long as ϕ is. Moreover, $\frac{d\psi(t)(\vec{x}_i)}{dt}(\tau) = \frac{d\phi(t)(\vec{x}_i)}{dt}(\tau) = \phi(\tau)[\![\vec{e}_i]\!]$. Thus, we only need to show that $\phi(\tau), \psi(\tau)$ agree on $FV(\vec{e}_i)$. This is done in a similar way to evolution domain constraints: $\phi(\tau), \psi(\tau)$ agree except on $\vec{x}^C$, but these are held constant across the ODE. By definition, u, v agree on $FV(\vec{e}_i)$, and so $\phi(\tau), \psi(\tau)$ agree on $FV(\vec{e}_i) \cap \vec{x}^C$ and we are done.

We have established a suitable witness $v' = \psi(r), (v, v') \in [\![\vec{x}' = \vec{e}\&Q]\!]$. It remains to show that u', v' agree on X. The proof is similar to the above: $\phi(r), \psi(r)$ already agree on $\vec{x}$ so we only need to show that they agree on $\vec{x}^C \cap X$. These are held constant across the ODE, so it is sufficient that u, v agree on X, which is true by assumption.

□

The theorem also justifies a heuristic when searching for preconditions P of a formula $P \to [\alpha]Q$. We only need to search for P whose free variables are in $D_\alpha(FV(Q))$.

Corollary 1. *If $P \to [\alpha]Q$, then there is a precondition R such that $P \to R$, $R \to [\alpha]Q$, where $FV(R) \subseteq D_\alpha(FV(Q))$.*

Proof. Suppose that P is a suitable precondition making the formula valid. If $FV(P) \subseteq D_\alpha(FV(Q))$, then we are done. Otherwise, it mentions some free variables $\vec{x} \cap D_\alpha(FV(Q)) = \emptyset$, we claim that $R \equiv \exists \vec{x} P$ is a weaker precondition where $\vec{x}$ is not free. Note that $P \to R$, since $\vec{x}$ is free in P.

Consider $u \in [\![\exists \vec{x} P]\!]$, by definition, there exists $v = u^{\vec{c}}_{\vec{x}} \in [\![P]\!]$ for some instantiation $\vec{c}$ of the quantifier. Since $P \to [\alpha]Q$ is valid, $v \in [\![[\alpha]Q]\!]$. Now, u differs from v only on $\vec{x}$, and so they agree on on $D_\alpha(FV(Q))$. Consider any $(u, u') \in [\![\alpha]\!]$, by the theorem, there exists $(v, v') \in [\![\alpha]\!]$ where u', v' agree on $FV(Q)$. Moreover, $v' \in [\![Q]\!]$, therefore, by coincidence for formulas, $u' \in [\![Q]\!]$ and thus, $u \in [\![[\alpha]Q]\!]$. Note further that quantifier elimination (QE) yields an equivalent precondition $QE(\exists \vec{x} P)$, which means the extra existential quantifier is not required. □

2.3 Tightening the Analysis

As a syntactic analysis, our definition of $D_\alpha(x)$ is necessarily an over-approximation. It is possible, however, to tighten the analysis, especially in the case of ODEs. Consider the system

$$x' = v, v' = a \& v \geq 0$$

[2]For brevity, we omit explicitly mentioning the coincidence lemma.

Our naive analysis tells us that x depends on $\{x, v, a\}$, but also that v depends on $\{x, v, a\}$. Solving the system gives $x = x_0 + vt + \frac{at^2}{2}, v = v_0 + at$, which exists for all $t \geq 0$, but v is clearly not dependent on the initial value of x.

Let us assume instead that the ODE $\vec{x}' = \vec{e}$ has a global solution for all (non-negative) time and initial values. Then, we can instead use:

$$D_{\vec{x}'=\vec{e}}(X) = T$$

where T is the smallest set satisfying $X \subseteq T$ and $\vec{x}_i \in T \to FV(\vec{e}_i) \in T$, i.e. the transitive closure of all the variables mentioned in X across the system.

The idea behind this is that anything not in T can only affect the values of variables in T if they restrict the time evolution of the ODE. Since we assumed that a global solution exists for the ODE, this restriction cannot occur. More formally, consider the unique symbolic global solution Φ taking initial states to solutions. There exist solutions $\Phi(u), \Phi(v)$ satisfying the ODE starting in u, v respectively. If we inspect these solutions under the reduced ODE system involving only $\vec{x}' \cap T$, then they start in states that agree on T. By uniqueness, they must remain the same on $\vec{x}' \cap T$, and hence, also on T since the other variables are held constant. Therefore, the final states also agree on $X \subseteq T$.

A special case of ODEs satisfying this property are linear systems of the form $\vec{x}' = A\vec{x}$, where A does not mention $\vec{x}$. These have a global solution given by $\Phi(\vec{x_0}, t) = e^{tA}\vec{x_0}$, and are particularly easy to check syntactically.

We can take the linearity idea even further. Consider the following system:

$$x' = 1, y' = xy$$

The system is non-linear by construction so the linearity check fails and we are forced to conclude that x, y are interdependent. However, solving the first equation gives $x = x_0 + t$. Plugging this into the latter gives $y = y_0 e^{x_0 t + \frac{t^2}{2}}$, which exists for all time. We see that y depends on x_0, y_0 but x is only dependent on x_0.

Instead of requiring that the entire ODE be linear, we can instead require only that the rest of the system not involving T, i.e. $\vec{x} \cap T^C$, is itself linear. This would allow us to correctly conclude that x only depends on x_0 above. Note that this works even if $\vec{x} \cap T$ itself is non-linear. For example, the following modified system again only has finite time solutions, but x is still not dependent on y_0.

$$x' = x^2, y' = xy$$

The rough sketch here is to similar to the argument for a vectorial version of the inverse DG axiom [Platzer, 2012, 2016]. It crucially relies on the fact that the ODE system $\vec{x} \cap T^C$ is linear in $\vec{x} \cap T^C$. For any solution ϕ of $\vec{x} \cap T$, we are able to construct a corresponding unique solution for $\vec{x} \cap T^C$ that exists as long as ϕ does[3]. Augmenting ϕ with this unique solution extends ϕ to a full solution of $\vec{x}$. Thus, the presence of $\vec{x} \cap T^C$ in the original system does not affect the evolution of $\vec{x} \cap T$, and so it does not affect $X \subseteq T$.

Note that in order to extend the definition to ODEs with evolution domain constraints, we transitively close over $X \cup FV(S)$ instead of X in the definition of T above. This is slightly different from our original definition, as can be seen from the following system: $t' = 1, v' = a \& v \geq 0$. If we did not close over S, then the analysis incorrectly reports that t only depends on t, v initially but it clearly depends on a as well.

2.4 Coincidence and Restricted Coincidence

We named Theorem 1 *restricted* coincidence because it is a more refined form of the coincidence lemma for programs [Platzer, 2016]. The coincidence lemma shows that for any set V such that $FV(\alpha) \subseteq V$,

[3]Ignoring the case where singularities (from dividing by variables in $\vec{x} \cap T$) occur in the RHS of $\vec{x} \cap T^C$. A simple syntactic check can help to rule these out as well.

two states agreeing on V continue to agree on V after α. We can apply this lemma directly to get a coarse form of dependency analysis. Suppose that we are trying to find the dependencies for a set X. By the coincidence lemma, if two states agree on $FV(\alpha) \subseteq X \cup FV(\alpha)$, then they agree on $X \subseteq X \cup FV(\alpha)$ after α, which is exactly the required property given in Theorem 1.

The coarse analysis does not account for the fact that we are only requiring states to agree on X after α. Our analysis, D_α, is essentially coincidence, but *restricted* to only require X to coincide, rather than $FV(\alpha) \cup X$. For example, consider $\alpha \equiv x' = v, v' = a \& v \geq 0$ and $X = \{v\}$. Using the coarse analysis gives us $FV(\alpha) \cup X = \{x, v, a\}$, while our analysis gives a smaller set, $D_\alpha(X) = \{v, a\}$.

2.5 Separating Control and Data

In our definition of $D_\alpha(x)$ above, we frequently run into situations where a dependency arises only because of control flow. In particular, we would have liked to have a notion that only involves data, i.e. ignoring all the control logic:

$$\tilde{D}_{?Q}(X) = X$$

$$\tilde{D}_{\vec{x}'=\vec{e}\&Q}(X) = D_{\vec{x}'=\vec{e}}(X)$$

Here, $\tilde{D}_\alpha$ is the same as D_α except with the two control rules replaced. Consider our ODE example, $x' = v, v' = a \& v \geq 0$, the original analysis would give us $D_\alpha(a) = a, v$, whereas ignoring control flow yields $\tilde{D}_\alpha(a) = a$. One could conversely define a set of control dependencies that only track the variables which affect X using control only. Unfortunately, we have not yet found a good way to characterise either of these types of dependencies since they seem to be inherently intertwined.

An alternative, but closely related, way to understand dependencies is to examine how data flows in a forward manner. A standard forward data-flow analysis question is to determine, at each program point, which assignments to variables could have flowed to that point, i.e. the reaching definitions. We assume that every operator in the HP is labelled with a unique index, and that there is a special index 0 statement for the start of the program. We define the reaching definitions analysis for HPs in terms of a equations between In(-coming) and Out(-going) definitions generated by a program α at its associated index n. $I(n)$ and $O(n)$ are maps from variable names to sets of indices, $I(n)$ gives the set of definitions that might reach program point n, and $O(n)$ give the set of definitions that exit program point n.

$$(x := e)_n \to O(n) = I(n)[x \mapsto \{n\}]$$

$$(\alpha_i; \beta_j)_n \to O(i) = I(j), O(n) = O(j), I(i) = I(n)$$

$$(\alpha_i \cup \beta_j)_n \to O(n) = O(i) \cup O(j), I(i) = I(n), I(j) = I(n)$$

$$(?Q)_n \to O(n) = I(n)$$

$$(\alpha_i^*)_n \to O(n) = O(i), I(i) = I(n) \cup O(i)$$

$$(\vec{x}' = \vec{e}\&Q)_n \to O(n) = I(n)[\vec{x} \mapsto \{n\}] \cup I(n)$$

where $S[\vec{x} \mapsto n](x_i) = n$ and union on maps indicates point-wise unions.

The set of equations generated by α_n can be solved using a fixed-point iteration, e.g. using Kildall's method Kildall [1973], and explicitly setting the initial conditions $I(n) = \{x \mapsto \{0\}\}$. Such an analysis is typically used in compilers to soundly perform code optimisations and transformations. In our context, data-flow analysis could be used to help users identify finer dependencies in their models, but we have not explored this direction further.

2.6 Implementation

We have implemented the dependency analysis as described above in KeYmaera X. Here, we discuss some implementation notes not covered by the preceding discussion.

1. We need to detect linear ODEs in KeYmaera X's free term grammar. For this, we use a simple approximation: we check that each $\vec{e_i}$ only mentions monomials over $\vec{x_i}$ with degree 0 or 1. This is sound but incomplete, e.g. $x^2 - x^2$ has degree 0 in x but our analysis would return degree 2.

2. In both linear ODEs and loops, we need to find a transitive closure over the variables that could be mentioned in the program or system. In both cases, the number of variables that can be mentioned syntactically is finite, so it is sufficient for us to apply a fixed-point iteration until we reach convergence.

3. The definitions shown above do not account for function, propositional or program symbols[4].In fact, the only way to have a sound analysis for program constants is to set $D_c(X) = \mathbb{V}$, i.e the set of all variables. This is not a useful case, so we omitted it from our implementation.

   However, KeYmaera X represents constants using nullary functions, and so it is useful to at least be able to track variable dependencies on function symbols. Our implementation returns both the variables and function symbols that may affect X.

4. We use a flag to determine whether to include the control dependencies, i.e whether to compute D_α or $\tilde{D}_\alpha$. This turned out to be a useful relaxation in some cases.

5. The definition of D_α can be used to induce a variable dependency graph. For each variable x, we add an edge from x to every variable in $D_\alpha(x)$. From the graph structure, we can extract (1) a (partial) topological ordering for variables, and (2) the strongly connected components on variables.

As an example, consider the following model for Lab 3 Question 6:

```
{
  {{
    acc:=*; ?(0 <= acc & acc <= A &
    ((x-ox)^2 + (y-oy)^2)^(1/2) > (v * T + acc * T^2/2 + (v+acc*T)^2 / (2*B)))
  );} ++ acc:=-B;}
  t:=0;
  {y' = x*v/rad, x' = -y*v/rad, v'= acc, t' = 1 & v >= 0 & t < T }
}*
```

The strict D_α analysis gives us the following dependencies:

Variable	Variable Dependencies	Constant Dependencies
t	{t,v,x,y}	{ox, T, oy, rad, A, B}
v	{v,x,y}	{ox, T, oy, rad, A, B}
acc	{acc, x, y, v}	{ox, T, oy, rad, A, B}
x	{x,v,y}	{ox, T, oy, rad, A, B}
y	{y,v,x}	{ox, T, oy, rad, A, B}

This is overly strict for some purposes, especially with respect to the constant dependencies. Turning off the control flow checks, i.e. computing $\tilde{D}_\alpha$ instead, gives us the following (smaller) but incorrect set of dependencies:

[4]Except in $FV(\cdot)$, where it looks for variables mentioned in their arguments.

Variable	Variable Dependencies	Constant Dependencies
t	{t,v,x,y}	{B, rad}
v	{v}	{B}
acc	{acc}	{B}
x	{x,v,y}	{B, rad}
y	{y,v,x}	{B, rad}

3 Differential Invariant Generation

In this section, we explain our variant of differential invariant generation [Platzer and Clarke, 2009] using the definition of D_α for ODEs. We focus only on goals of the form $P \to [\vec{x}' = \vec{e}\&Q]R$; the extension to global loop invariants is described in [Platzer and Clarke, 2009]. We start by recalling the process described in [Platzer and Clarke, 2009], known as differential saturation, which relies on a form of dependency analysis. The latter part of this section explains our variation on the algorithm.

3.1 Differential Saturation

The main idea behind differential saturation is to repeatedly enrich the evolution domain with suitable differential invariants. For example, if we managed to prove $P \to \phi$ and $\phi \to [\vec{x}' = \vec{e}\&Q]\phi$, then we can reduce our original goal using a differential cut to $P \to [\vec{x}' = \vec{e}\&Q \wedge \phi]R$. Observe that the enriched evolution domain $Q \wedge \phi$ might become strong enough to imply the postcondition, in which case we can prove the goal using differential weakening. Otherwise, the enriched domain could also be used to prove new differential invariants.

The process relies on an input list of likely differential induction candidates, which it tries in order. Importantly, these induction candidates are provided parametrically, and the parameters are instantiated by the algorithm as it attempts to prove the corresponding differential invariants. To illustrate this, consider the following ODE, where a point is following a circular trajectory of radius 1.

$$x^2 + y^2 = 1 \to [x' = -y, y' = x]R$$

Given a set of variables $V = \{x_1, ..., x_n\}$ and a degree d, we define a parametric polynomial, p, as follows:

$$p := \sum_{i_1+...+i_n \leq d} a_{i_1,...,i_n} x_1^{i_1} ... x_n^{i_n}$$

where the coefficients $a_{i_1,...,i_n}$ are fresh symbolic parameters.

If $V = \{x, y\}$, then a parametric polynomial of degree 2 is given by

$$ax^2 + bxy + cy^2 + dx + ey + f$$

where a, b, c, d, e, f are the fresh parameters.

Assuming the coefficients are constant, we can compute their symbolic Lie derivative along the ODE's vector field. In the example above, this would be

$$\begin{aligned} p' &= 2axx' + b(x'y + y'x) + 2cyy' + dx' + ey' \\ &= -2axy + b(x^2 - y^2) + 2cxy - dy + ex \end{aligned}$$

Now, suppose we wanted to use p as an equational differential invariant[5], then we need to show the following:

$$P \vdash p = 0$$

$$Q \vdash p' = 0$$

[5]Similarly for inequalities like $p \geq 0$.

In our case, comparing coefficients for $p' = 0$ gives us $a = c, b = 0, d = 0, e = 0$. Substituting this back into the first equation, leaves us with $x^2 + y^2 = 1 \vdash ax^2 + cy^2 + f = 0$. One of the solutions[6] is $a = 1, c = 1, f = -1$. The algorithm can thus enrich the evolution domain with the invariant $x^2 + y^2 - 1 = 0$, i.e. the point stays on a circle of radius 1.

The remaining questions are: (1) which polynomial candidates p, or equivalently, which sets V we consider, and (2) how we order these polynomial candidates. Trivially, we could consider V to be all the variables mentioned in the ODE. Unfortunately, this quickly makes solving for the constraints infeasible. Instead, the algorithm solves both (1) and (2) together by considering the variables in dependency order. To handle (1), recall our property in Corollary 1: it is sufficient to consider invariants $\phi \to [\alpha]\phi$ such that ϕ is closed under the dependency order, i.e. $FV(\phi) = D_\alpha(FV(\phi))$. Moreover, if a variable x depends on a closed set X, then it makes sense for us to first attempt to find invariants in X before attempting to find invariants in $x \cup X$. This solves the ordering problem (2).

For example, consider the following (sub)-system of equations described in Platzer and Clarke [2009]:

$$\alpha \equiv x_1' = d_1, x_2' = d_2, d_1' = -\omega d_2, d_2' = \omega d_1, t' = 1$$

Note that this is a linear system, so by analysing the dependencies for each variable we get:

$$D_\alpha(d_1) = \{\omega, d_1, d_2\}$$
$$D_\alpha(d_2) = \{\omega, d_1, d_2\}$$
$$D_\alpha(x_1) = \{\omega, d_2, d_1, x_1\}$$
$$D_\alpha(x_2) = \{\omega, d_2, d_1, x_2\}$$
$$D_\alpha(t) = \{t\}, D_\alpha(\omega) = \{\omega\}$$

We omit ω because it is constant across the ODE. The strongly connected components of the induced dependency graph are the sets: $\{t\}, \{d_1, d_2\}, \{x_1\}, \{x_2\}$. Moreover, x_1, x_2 depend on d_1, d_2, so we generate invariants from the sets (in order): $\{t\}, \{d_1, d_2\}, \{d_1, d_2, x_1, x_2\}$.

3.2 Invariant Generation

Our current implementation of invariant generation is a simplified form of differential saturation. Instead of aiming to prove the whole goal directly via saturation, we instead seek only to advise the user on potential differential invariants that could be true about their ODE system. Thus, we implemented most of the preceding description of differential saturation except we only apply the derivative constraint when solving for the parameters of differential invariants:

$$Q \vdash p' = 0$$

Naturally, our resulting differential invariants are not as constrained as the ones generated by differential saturation. Instead, we leave remaining parameters unconstrained and report the result to the user. For example, in the rotational ODE system, we report the parametric rotational invariant $ax^2 + ay^2 + f = 0$ where a, f are fresh names instead of constraining them according to the precondition. We leave it up to the user to appropriately instantiate these parameters once they have decided on an appropriate precondition.

As a simple way to generate the appropriate parameter constraints, our implementation uses partial quantifier elimination on $\forall \vec{x}.(Q \to p' = 0)$. Here, $\vec{x}$ quantifies over all the variables occurring in the LHS of the ODE system, so that the generated invariant only has free variables that are constant, i.e. involving the fresh variables we introduced, and actual constants for the ODE. We then inspect the resulting formula heuristically for potential instantiations of parameters that can generate useful invariants. In the rotational example, the result of quantifier elimination is the formula $c = a \wedge b =$

[6] Notice that solutions are non-unique, for example, $a = 2, c = 2, f = -2$ is also a solution.

$0 \wedge d = 0 \wedge e = 0$. We simply set all the free parameters as required by the conjuncts to obtain the parametric invariant shown above. In some other cases, we get output formulas that contain disjuncts and inequalities. For these, we heuristically generate instantiations, e.g.$h \geq 0$ leads to an instantiation $h = 0$, and then double check that they do in fact produce appropriate invariants.

We experimented with our invariant generator on a few example systems, and the results are summarised in Table 3.2, fresh variables are represented by f_i. Unfortunately, saturation appears to degrade the performance of partial quantifier elimination, and so we have disabled it for these results. The invariants we generated make some intuitive sense with respect to the ODEs. For example, the elliptical invariant shows that if we start on any ellipse centred about the origin, then we stay on it.

Similarly, in the straight line motion example, setting $f_1 = 1, f_0 = -v_0$ gives us the standard motion equation $v = v_0 + at$. In fact, we also found some larger invariants that we could not fit into the table. In the straight line motion case, we also have the following parametric invariant:

$$f_0 + a^2 f_1 t^2 + \frac{1}{2}at(-2f_2 + f_3 t - 4f_1 v) + v(f_2 - f_3 t + f_1 v) + f_3 x = 0$$

If we set $f_2 = f_1 = 0, f_3 = 1, f_0 = -x_0$, then we get $-x_0 + \frac{1}{2}at^2 - vt + x = 0$. If we further substitute the previous equation for v here, we get $x = x_0 + v_0 t + \frac{1}{2}at^2$, which is exactly the solution for x. Interestingly, we did not need any differential cuts to derive this invariant, even though it is needed in general.

System Type	**ODE System**	**Invariants Generated**
Elliptical motion about origin	$v' = aw, w' = -v$	$f_0 + f_1(v^2 + aw^2) = 0$
Straight line motion	$x' = v, v' = a, t' = 1$	$f_0 + f_1(-at + v) = 0$
Exponential*	$x' = x, t' = 1$	$f_0 + t \geq 0, f_1 + x^2 \geq 0$
Flight dynamics subsystem*	α (see above)	$f_0 + f_1(d_1^2 + d_2^2) = 0,$ $f_2 + f_3(d_2 - \omega x_1) + f_4(d_1 + \omega x_2) = 0,$ $f_5 + t \geq 0$

Table 1: Summary of generated invariants, we only searched for invariants of the form $p \geq 0$ in the cases marked with *.

4 Heuristics for Quantifier Elimination

We also experimented with applying dependency analysis to the problem of finding heuristic variable orderings for calls to the quantifier elimination tactic in KeYmaera X. Using the induced variable dependency graph, we defined the following partial order on variables: $x < y \iff y \in D_\alpha(x) \wedge x \notin D_\alpha(y)$.

To evaluate the usefulness of our analysis, we first needed a series of arithmetic benchmarks. For this purpose, we implemented a simple logger for quantifier elimination (QE) calls. The logger itself is of independent interest because the arithmetic formulae that it logs can also be used for other purposes.

A straightforward solution would have been to produce a new tactic, e.g. `loggedQE`, and have the user call this tactic whenever a call needs to be logged. There are two issues with this approach. Firstly, it would have required extra work to rewrite existing case studies to call the new tactic. Secondly, this does not allow one to log internal QE calls, such as ones made by the ODE solver.

Instead, we directly inserted logger calls into the base QE tactic. This ensures that every call is logged, regardless of where it came from. The logger can also be configured to log extra information, such as a tag/name for each logged entry, and the underlying conclusion of the associated provable. We can make use of this extra information e.g. to analyse variable dependencies.

There are also times when QE is called to decide trivial goals, e.g. $0 + 1 < 2$. As an initial filtering step, we also implemented a measure on input sequents. The logger can be configured to directly filter out sequents whose measure is less than a given value. We currently pick this value rather arbitrarily.

Name	Filter Size	No. Log Entries
Axiomatic ODE Solver Tests	10	414
STTT Tutorial	10	1000
Lab 2 Q2	10	225
Lab 3 Q6	10	24
Chilled Water	40	222
ETCS (No ModelPlex)	40	431

Table 2: Number of collected arithmetic benchmarks.

Table 4 summarizes the collected logs from existing test suites in KeYmaera X. The number of log entries is of interest, e.g. if we aimed to remove QE oracles from the core of KeYmaera X, we would need a tool that can handle all of these QE calls.

We timed four different versions of the QE tactics for their performance on these logs.

QE1 The current QE, which lexicographically orders the variables.

QE2 The heuristic QE following the first heuristic in Huang et al. [2015]. We slightly improved the existing version.

QE3 Augmenting the heuristic order with the partial order induced by D_α.

QE4 Augmenting the heuristic order with the partial order induced by $\tilde{D}_\alpha$.

Unfortunately, we found that most of the calls to QE were fast enough that our ordering modifications did not matter. There were only 2 cases, both in Lab 3 Q6, where the timings significantly differed. In the first case, QE3 took $\approx 5.5s$ while the others took $\approx 12.5s$. In the second, QE1 and QE4 finished in ≈ 270 seconds, while QE2 and QE3 took ≈ 13000 seconds. These two cases are evidence that variable ordering certainly matters, but they do not provide any conclusive evidence that any of the QE heuristics we tried work better than others.

5 Conclusion

We have defined and implemented a dependency analysis, D_α for hybrid programs. The implementation was used in two application areas: (1) invariant generation, and (2) quantifier elimination variable ordering. The invariants we generated for (1) are promising, and it could be useful to develop them further, e.g. using something other than partial quantifier elimination to solve for constraints on parameters. On the other hand, the application to (2) did not provide any conclusive benefits, but the generated artefacts (e.g. a QE logger) might be useful for other purposes.

References

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL `http://doi.acm.org/10.1145/512950.512973`.

Z. Huang, M. England, D. Wilson, J. H. Davenport, and L. C. Paulson. A comparison of three heuristics to choose the variable ordering for cylindrical algebraic decomposition. *ACM Commun. Comput. Algebra*, 48(3/4):121–123, Feb. 2015. ISSN 1932-2240. doi: 10.1145/2733693.2733706. URL `http://doi.acm.org/10.1145/2733693.2733706`.

G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. doi: 10.1145/512927.512945. URL `http://doi.acm.org/10.1145/512927.512945`.

A. Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4):1–38, 2012. doi: 10.2168/LMCS-8(4:16)2012.

A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *Journal of Automated Reasoning*, pages 1–47, 2016. ISSN 1573-0670. doi: 10.1007/s10817-016-9385-1. URL `http://dx.doi.org/10.1007/s10817-016-9385-1`.

A. Platzer. Differential Equations & Domains. `http://symbolaris.com/course/fcps17/02-diffeq.pdf`, 2017.

A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Form. Methods Syst. Des.*, 35(1):98–120, 2009. doi: 10.1007/s10703-009-0079-8.