View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ASU Digital Repository

Representing Hybrid Transition Systems in an Action Language Modulo ODEs

by

Nikhil Loney

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2017 by the
Graduate Supervisory Committee:

Joohyung Lee, Chair
Georgios Fainekos
Yu Zhang

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Several physical systems exist in the real world that involve continuous as well as discrete changes. These range from natural dynamic systems like the system of a bouncing ball to robotic dynamic systems such as planning the motion of a robot across obstacles. The key aspects of effectively describing such dynamic systems is to be able to plan and verify the evolution of the continuous components of the system while simultaneously maintaining critical constraints. Developing a framework that can effectively represent and find solutions to such physical systems prove to be highly advantageous. Both hybrid automata and action languages are formal models for describing the evolution of dynamic systems. The action language $\mathcal{C}+$ is a rich and expressive language framework to formalize physical systems, but can be used only with physical systems in the discrete domain and is limited in its support of continuous domain components of such systems. Hybrid Automata is a well established formalism used to represent such complex physical systems at a theoretical level, however it is not expressive enough to capture the complex relations between the components of the system the way $\mathcal{C}+$ does.

This thesis will focus on establishing a formal relationship between these two formalisms by showing how to succinctly represent Hybrid Automata in an action language which in turn is defined as a high-level notation for answer set programming modulo theories (ASPMT) — an extension of answer set programs in the first-order level. Furthermore, this encoding framework is shown to be more effective and expressive than Hybrid Automata by highlighting its ability in allowing states of a hybrid transition system to be defined by complex relations among components that would otherwise be abstracted away in Hybrid Automata. The framework is further realized in the implementation of the system CPLUS2ASPMT, which takes advantage of state of the art ODE(Ordinary Differential Equations) based SMT solver DREAL

to provide support for ODE based evolution of continuous components of a dynamic system.

ACKNOWLEDGMENTS

First and foremost, I want to thank my adviser, Dr. Joohyung Lee. I decided to pursue my Master's in Computer Science at Arizona State University to help gain expertise and increase my knowledge in the field of Artificial Intelligence and I can say without a doubt this is exactly what Dr. Lee helped me achieve. I was in his class for AI in Fall'15 and immediately grew to appreciate his teaching and advising techniques. Dr. Lee worked closely with me throughout my time at ASU, patiently guiding me through my research as well as providing suggestions for relevant coursework. I am fortunate and grateful to have such an experienced, patient and dedicated adviser. Thank you Dr. Lee.

I would also like to thank all my teammates, the members of the Automated Reasoning Group: Samidh Talsania, Zhun Yang, Yi Wang, Manjula Malaiarasan and Brandon Gardell. They provided crucial and valuable inputs for my work through several discussions and group seminars. I would also like to thank my friends Rushali Malde, Kishore Narendran, Sanjeet Phatak, Onkar Ghag, Narayan Kanhere, Kevin Vira and many more for providing me with the support and guidance to carry out my research as well as providing welcome distractions. Thank you all.

Finally I would like to thank my family. They have always been very loving, encouraging and supportive throughout my life. My parents have never denied me from pursuing what I was truly interested in and provided me with the support, resources, encouragement and guidance to do so. I would like to specially thank my mother. She has been my strongest light of guidance and pushed me in the right direction whenever I went astray. Thank you Mom, Dad and Rahul.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 7

2.1 Review: Hybrid Automata 7

2.2 Review: ASPMT 10

2.3 Review: $\mathcal{C}+$ 11

2.3.1 Syntax of $\mathcal{C}+$ 11

2.3.2 Semantics of $\mathcal{C}+$ 12

2.4 Related Works 13

2.4.1 PDDL+ 13

2.4.2 Action Language $\mathcal{H}$ 14

3 REPRESENTING LINEAR HYBRID AUTOMATA WITH CONVEX INVARIANTS BY $\mathcal{C}+$ MODULO THEORIES 16

3.1 Representation 16

3.2 Example 21

3.3 Beyond Linear Hybrid Automata 25

3.3.1 Representation 25

3.3.2 Example 27

3.4 Proofs 31

3.4.1 Proof of Lemma 1 31

3.4.2 Proof of Lemma 2 37

3.4.3 Proof of Lemmas 3 and 4 39

CHAPTER Page

4 $\mathcal{C}+$ MODULO ODE ... 41

4.1 New Causal Laws for Expressing Continuous Evolutions of ODEs .. 41

4.2 Corresponding Representation of New Causal Laws in ASPMT 43

5 REPRESENTING HYBRID TRANSITION SYSTEMS IN $\mathcal{C}+$ MODULO ODE ... 46

5.1 Representation ... 46

5.2 Example ... 49

5.3 Turning in the Input Language of dReal ... 54

5.4 Proofs ... 56

5.4.1 Proof of Lemma 11 ... 56

5.4.2 Proof of Lemma 12 ... 61

6 IMPLEMENTATION ... 65

6.1 ASPMT2SMT ... 65

6.1.1 Architecture ... 66

6.1.2 Syntax Restrictions ... 67

6.2 CPLUS2ASPMT ... 68

6.3 Syntax of New Constructs ... 69

6.3.1 In CPLUS2ASPMT ... 69

6.3.2 In Extended ASPMT2SMT ... 70

6.4 Limitations ... 71

6.5 Examples ... 72

6.5.1 Water Tank Example - Example 1 ... 73

6.5.2 Turning Car Example - Example 3 ... 74

6.6 Results ... 77

7 GOING BEYOND HYBRID AUTOMATA 80
7.1 Additive Fluents 80
7.2 Complex Definition of States of Hybrid Systems 81
8 CONCLUSION 83
REFERENCES 85
APPENDIX
A ABBREVIATIONS IN $\mathcal{C}+$ 87
B WATER TANK EXAMPLE 91
C 2 BALL EXAMPLE 96
D CAR EXAMPLE 107

LIST OF TABLES

Table Page

6.1 Runtime Comparison of Hybrid Automata Solvers - Runtime(s) 77

6.2 Encoding Size (Number of Lines) Comparison 78

LIST OF FIGURES

Figure Page

1.1 Snippet of Traffic World in the Input Language of Cplus2ASP 2

1.2 (a) Feasible Plan (b) Infeasible Plan 4

3.1 Hybrid Automata for Water Tank System. 21

3.2 Hybrid Automata of Two Ball Example 28

5.1 Hybrid Automata of Car example 50

6.1 Architecture of System ASPMT2SMT as Shown In (Bartholomew and Lee (2014)) 66

6.2 Architecture of System cplus2ASPMT 69

6.3 Output of Example 1 74

6.4 Output of Example 3 76

C.1 Output of Example 2 106

Chapter 1

INTRODUCTION

Both hybrid automata and action languages are formal models for describing the evolution of dynamic systems. Action language $\mathcal{C}+$ (Giunchiglia *et al.* (2004)) is an extensively used language currently available for formalizing discrete domain problems. It is written using causal laws and syntactic aspects of the action language like **if** and **after** that implicitly takes care of time step expansion, which make $\mathcal{C}+$ easy to use and understand for describing complex transition systems. The ability of $\mathcal{C}+$ to represent properties of actions and fluents as well capturing complex relations among fluents make it a rich and expressive language. Setting up causal laws for a physical system in this way helps capture the complex dependencies between components of the system. The Causal Calculator (CCalc) (Giunchiglia *et al.* (2004)) is an implementation of the action language $\mathcal{C}+$. The semantics of the language of CCalc is related to default logic and logic programming and uses ideas of satisfiability planning for its computations. The system CPLUS2ASP (Babb and Lee (2013)), a successor of CCalc, is a system designed to perform a modular translation of action descriptions written for CCalc into Answer Set Programs(ASP) (Baral (2003)) and uses an ASP solver to find the satisfiable models for a given problem. However, the obvious downfall of using this framework is that in most works on action languages, the transitions are limited to discrete changes only, so they are not appropriate to model systems that involve continuous components governed by ODEs like in hybrid automata or even simple real valued fluents.

Hybrid Automata is a well known mathematical model used extensively to describe complex physical systems. In (Henzinger (1996)) we see that any Hybrid Automata

% If a car is at the end of a segment and will not leave then it will stay where it is

caused $speed(C) = 0$ **if** $-(nextSegment(C) = none)$ & $-willLeave(C)$.

% A car can't have a next segment unless it has travelled

%to the end of its current segment

caused $nextSegment(C) = none$ **if** $node(C) = none$.

caused $nextSegment(C) = none$ **if** $segment(C) = Sg$ &

$node(C) = startNode(Sg)$ & $positiveOrientation(C)$.

caused $nextSegment(C) = none$ **if** $segment(C) = Sg$ &

$node(C) = startNode(Sg)$ & $-positiveOrientation(C)$.

% Only cars which have selected a new segment can leave

constraint $willLeave(C) \rightarrow -(nextSegment(C) = none)$.

Figure 1.1: Snippet of Traffic World in the Input Language of CPLUS2ASP

has two main components:

- Continuous components are the real valued variables of the Hybrid Automata whose rate of change with time is governed by some ordinary differential equation.

- Discrete components are the control modes/switches of the Hybrid Automata represented as vertices in a graph.

The focus of hybrid automata is to model continuous transitions as well as discrete changes, but the discrete components are simply abstracted away not to be able to represent various properties of actions nor complex relations among fluents like action languages do. In (Akman *et al.* (2001)), the Traffic World problem is represented

using action language solver Cplus2ASP. A small snippet of the code can be seen in Figure1.1. The first rule defines the speed of each car when the car does not leave its segment. This is captured using the relation among the fluent constants *nextSegment* and *willLeave*. These fluent constants are further defined by additional constants that can be seen in the remaining rules in the code snippet. Due to the nature in which states are defined in Hybrid Automata, such complex dependencies cannot be effectively captured. Action languages can be used to perform defeasible causal reasoning, reasoning about additive fluents, recursive definition of constants, aggregate computation and more in transition systems which is not possible in Hybrid Automata.

In (Lee and Meng (2013)), an extension of action language $\mathcal{C}+$ (Giunchiglia *et al.* (2004)) is shown to model hybrid transition systems. The main idea of the extension is to reformulate the propositional $\mathcal{C}+$ in terms of Answer Set Programming Modulo Theories (ASPMT) — a tight integration of answer set programs and satisfiability modulo theories (SMT) to allow SMT-like effective first-order reasoning in ASP. Based on the reduction of the tight fragment of ASPMT to SMT, the extended $\mathcal{C}+$ can be turned into SMT formulas so that SMT solvers can be used for computing the language. [1] However, this method does not ensure that an invariant holds during continuous changes. For example, consider the problem of a robot navigating through the pillars as in Figure1.2, where the circles represent pillars that the robot has to avoid collision with.

In order to ensure that the robot does not collide into the pillars during continuous transitions, checking the invariants at each discrete time points is not sufficient because it could generate an infeasible plan, such as (b). This is related to the chal-

[1] The translation was done manually in that paper. One of the contributions of this paper is an implementation that automates the process.

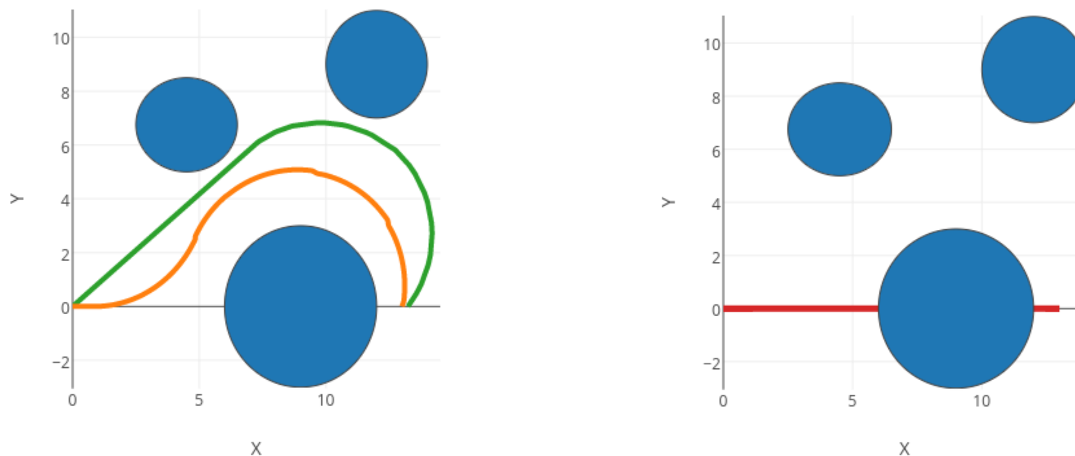

Figure 1.2: (a) Feasible Plan (b) Infeasible Plan

lenge in integrating high-level task planning and low-level motion planning, where the plans generated by task planners may fail often in motion planners.

This thesis will describe how to succinctly represent hybrid automata in an action language which in turn is defined as a high level notation for ASPMT. Starting with a simple case, the description will go over how to represent linear hybrid automata with convex invariants by an action language modulo theories. A further translation into SMT allows for computing them using state-of-the-art SMT solvers that support arithmetic over reals.

However, many practical problems in hybrid systems contain non-linear polynomials, trigonometric functions and differential equations that cannot be represented by linear hybrid automata. While solving the formulas with these functions is undecidable in general, there is a recent breakthrough in the work of SMT solving. In (Gao *et al.* (2013a)), the authors presented "δ-complete decision procedures" for such SMT formulas, leading to the concept of "satisfiability modulo ODE." [2] SMT solver

[2]A δ-complete decision procedure for an SMT formula F returns false if F is unsatisfiable, and returns true if its syntactic "numerical perturbation" of F by bound δ is satisfiable, where $\delta > 0$ is number provided by the user to bound on numerical errors. The method is practically useful since it is not possible to sample exact values of physical parameters in reality.

dReal is an implementation of SAT modulo ODEs and was shown to be useful for encoding the general class of hybrid automata.

Based on these recent works, this thesis further extends the $\mathcal{C}+$ representation of hybrid automata to non-linear hybrid automata by resorting to *ASP modulo ODE*.

We introduce two new abbreviations of causal laws, one for representing the evolution of continuous variables as specified by ODEs and another for describing invariants during continuous transitions, which stands for formulas whose universal quantification is over bounded time intervals. The simplicity of this method is thanks to the generality of the framework of ASPMT and its relation to SMT that enables algorithmic improvements in SMT to be carried over to the ASPMT setting.

The System cplus2aspmt has been implemented based on these translations, which allows for succinct representation of hybrid transition systems that can be computed by turning an action description in $\mathcal{C}+$ into the input language dReal. By making use of dReal, the system provides support for continuous processes that are governed by ordinary differential equations. The System cplus2ASPMT can be used for reasoning about hybrid transition systems, where other action language implementations, such as the Causal Calcualtor, Cplus2ASP (Babb and Lee (2013)), and coala (Gebser *et al.* (2010)) cannot be used.

The key contribution of this thesis is a framework that can be used to represent and reason about hybrid transition systems described using Hybrid Automata as well as those that involve more complex internal relations among components of the system that cannot be expressed in Hybrid Automata. By doing so, the framework enjoys the advantage of the structure of the Hybrid Automata formalism as well as the freedom to capture essential dependencies with the help of action language specific constructs and perform defeasible causal reasoning in an effective way. This framework is not just syntactic sugar and is realized in the prototype implementation

of system cplus2ASPMT. In this manner, the research culminates in an expressive and effective tool to represent and reason about hybrid transition systems.

This thesis will be organized as follows. In Chapter2, we give a review of hybrid automata, action language $\mathcal{C}+$ and ASPMT to set up the terminologies used for the translations. Chapter3 presents how to represent linear hybrid automata with convex invariants in $\mathcal{C}+$ modulo theory of reals. Chapter4 introduces new abbreviations of causal laws that can be used for modeling invariants and flow using the concept of SAT modulo ODE, and Chapter5 uses these new constructs to represent the general class of non-linear hybrid automata and how to reduce it to the input language of dReal. Chapter6 will cover the implementation of system cplus2aspmt, a variant of the system cplus2asp, which uses dReal as the back-end SMT solver.

Chapter 2

BACKGROUND

2.1 Review: Hybrid Automata

We review the definition of Hybrid Automata (Henzinger (1996); Alur *et al.* (2000)), formulated in terms of a logical language by representing arithmetic expressions by first-order formulas under the background theories, such as QF_NRA (Quantifier-Free Non-linear Real Arithmetic) or QF_NRA_ODE (Quantifier-Free Non-linear Real Arithmetic with Ordinary Differential Equations). We assume the set $\mathcal{R}$ of all real numbers. By $\mathcal{R}_{\geq 0}$ we denote the set of all non-negative reals and $\mathcal{R}^n = \{(r_1, \ldots, r_n) \mid r_i \in \mathcal{R}, 1 \leq i \leq n\}$. Let X be a set of real variables. An arithmetic expression over X is an atomic formula constructed using functions and predicates from the background signature and elements from $\mathcal{R} \cup X$. Let $A(X)$ be an arithmetic expression over X and let x be a tuple of real numbers whose length is the same as the length of X. By $A(x)$, we mean the expression obtained from A by replacing variables in X with the corresponding values in x. For an arithmetic expression with no variables, we say that A *is true* if the expression is evaluated to true in the background theory.

A *Hybrid Automaton* $\mathcal{H}$ consists of the following components:

- **Variables.** A finite list of real-valued variables $X = (X_1, \ldots, X_n)$. The number n is called the *dimension* of $\mathcal{H}$. We write $\dot{X}$ for the list $(\dot{X}_1, \ldots, \dot{X}_n)$ of dotted variables, representing first derivatives during continuous change, and X' for the set $(X'_1, \ldots, X'_n)$ of primed variables, representing the values at the conclusion

of discrete change. $X_0 \subseteq X$ is the set of initial states. We use lower case letters to denote the values of these variables.

- **Control Graph.** A finite directed graph $\langle V, E \rangle$. The vertices are called *control modes*, and the edges are called *control switches*.

- **Initial, invariant, and flow conditions.** Three vertex labelling functions, Init, Inv, and Flow, that assign to each control mode $v \in V$ three first-order formulas:

  - $\mathsf{Init}_v(X)$ is a first-order formula whose free variables are from X. The formula constrains the initial condition.

  - $\mathsf{Inv}_v(X)$ is a first-order formula whose free variables are from X. The formula constrains the value of the continuous part of the state while the mode is v.

  - $\mathsf{Flow}_v(X, \dot{X})$ is a set of first-order formulas whose free variables are from $X \cup \dot{X}$. The formula constrains the continuous variables and their first-order derivatives.

- **Events.** A finite set Σ of symbols called *h-event*s and a function, $\mathsf{hevent} : E \to \Sigma$, that assigns to each edge a unique h-event.

- **Guard.** For each control switch $e \in E$, $\mathsf{Guard}_e(X)$ is a first-order formula whose free variables are from X.

- **Reset.** For each control switch $e \in E$, $\mathsf{Reset}_e(X, X')$ is a first-order formula whose free variables are from $X \cup X'$.

Notation: Upper case X_i is a variable, and lower case x_i is a value of X_i.

A *labelled transition system* consists of the following components:

- **State Space:** A set Q of states and a subset $Q_0 \subseteq Q$ of initial states.

- **Transition Relations:** A set A of labels. For each label $a \in A$, a binary relation $\rightarrow^a$ on the state space Q. Each triple $q \rightarrow^a q'$ is called a *transition*.

The *Hybrid Transition System* T_H of a Hybrid Automaton H is the directed graph obtained from H as follows.

- The set Q of states is the set of all (v, x) such that $v \in V$, $x \in \mathcal{R}^n$, and $\mathsf{Inv}_v(x)$ is true.

- $(v, x) \in Q_0$ iff both $\mathsf{Init}_v(x)$ and $\mathsf{Inv}_v(x)$ are true.

- The transitions are labelled by members from $A = \Sigma \cup \mathcal{R}_{\geq 0}$.

- $(v, x) \rightarrow^\sigma (v', x')$, where $(v, x), (v', x') \in Q$ and σ is an h-event in Σ, is a transition if there is an edge $e = (v, v') \in E$ such that:

  (1) $\mathsf{hevent}(e) = \sigma$,

  (2) the sentence $\mathsf{Guard}_e(x)$ is true, and

  (3) the sentence $\mathsf{Reset}_e(x, x')$ is true.Inv

- $(v, x) \rightarrow^\delta (v, x')$, where $(v, x), (v', x') \in Q$ and δ is a nonnegative real, is a transition if there is a differentiable function $f : [0, \delta] \rightarrow \mathcal{R}^n$, with the first derivative $\dot{f} : [0, \delta] \rightarrow \mathcal{R}^n$ such that:

  (1) $f(0) = x$ and $f(\delta) = x'$,

  (2) for all real numbers $\epsilon \in [0, \delta]$, $\mathsf{Inv}_v(f(\epsilon))$ is true and, for all real numbers $\epsilon \in (0, \delta)$, $\mathsf{Flow}_v(\dot{f}(\epsilon), f(\epsilon))$ is true. The function f is called the *witness* function for the transition $(v, x) \rightarrow^\delta (v, x')$.

2.2 Review: ASPMT

Formally, an SMT instance is a formula in many-sorted first-order logic, where some designated function and predicate constants are constrained by some fixed background interpretation. SMT is the problem of determining whether such a formula has a model that expands the background interpretation (Barrett *et al.* (2009)). Some of the background theories relevant to this paper are as follows:

- *QF_LRA* (Only Quantifier free linear real arithmetic): It includes the following binary function constants that represent arithmetic functions: $+$, $-$, $\times$ and $/$ and the following binary predicates that represent comparison operators: $\geq$, $\leq$, $<$ and $>$. In essence, Boolean combinations of inequations between linear polynomials over real variables.

- *QF_NRA* (Only Quantifier free non-linear real arithmetic): It includes the following binary function constants that represent arithmetic functions: $+$, $-$, $*$ and $/$ and the following binary predicates that represent comparison operators: $\geq$, $\leq$, $<$ and $>$. It is similar to *QF_LRA* with the difference that it is additionally capable of handling non-linear polynomials, trigonometric functions(*sin*, *cos*, *tan*) and other non-linear theories.

- *QF_NRA_ODE* (Only Quantifier free non-linear real arithmetic for ODEs): This is a special background theory for dealing with ODEs. In addition to *QF_NRA*, *QF_NRA_ODE* adds support for functions like integral and $\forall^t$.

The syntax of ASPMT is the same as that of SMT. Let σ^{bg} be the (many-sorted) signature of the background theorybg. An interpretation of σ^{bg} is called a *background interpretation* if it satisfies the background theory. For instance, in the theory of reals, we assume that σ^{bg} contains the set $\mathcal{R}$ of symbols for all real numbers, the

set of arithmetic functions over real numbers, and the set $\{<, >, \leq, \geq\}$ of binary predicates over real numbers. Background interpretations interpret these symbols in the standard way.

Let σ be a signature that is disjoint from σ^{bg}. We say that an interpretation I of σ satisfies F w.r.t. the background theory bg, denoted by $I \models_{bg} F$, if there is a background interpretation J of σ^{bg} that has the same universe as I, and $I \cup J$ satisfies F. For any ASPMT sentence F with background theory σ^{bg}, interpretation I is a stable model of F relative to$\mathbf{c}$ (w.r.t. background theory σ^{bg}) if $I \models_{bg} \mathrm{SM}[F; \mathbf{c}]$.

2.3 Review:$\mathcal{C}+$

2.3.1 Syntax of $\mathcal{C}+$

$\mathcal{C}+$ is defined based on propositional stable model semantics. In this section we formulate in terms of ASPMT. We refer the reader to (Bartholomew and Lee (2013)) for the definition of ASPMT. [1]

We consider a many-sorted first-order signature σ that is partitioned into three sub-signatures: the set σ^{fl} of object constants called *fluent constants*, the set σ^{act} of object constants called *action constants*, and the background signatureσ^{bg}. The signature σ^{fl} is further partitioned into the set σ^{sim} of *simple* fluent constants and the set σ^{sd} of *statically determined* fluent constants.

A *fluent formula* is a formula of signature$\sigma^{fl} \cup \sigma^{bg}$. An *action formula* is a formula of$\sigma^{act} \cup \sigma^{bg}$ that contains at least one action constant and no fluent constants.

A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{2.1}$$

where F and G are fluent formulas.

[1] This is not novel; $\mathcal{C}+$ was formulated in terms of ASPMT in (Lee *et al.* (2013)).

An *action dynamic law* is an expression of the form(2.1) in which F is an action formula and G is a formula.

A *fluent dynamic law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{2.2}$$

where$F$ and$G$ are fluent formulas and H is a formula, provided that$F$ does not contain statically determined constants.

A *causal law* is a static law, an action dynamic law, or a fluent dynamic law. An *action description* is a finite set of causal laws.

2.3.2 Semantics of $\mathcal{C}$+

For a signature σ and a nonnegative integer i, expression $i : \sigma$ is the signature consisting of the pairs $i : c$ such that $c \in \sigma$, and the value sort of $i : c$ is the same as the value sort of c. Similarly, if s is an interpretation of σ, expression $i : s$ is an interpretation of $i : \sigma$ such that $c^s = (i : c)^{i:s}$.

For any action description D of signature $\sigma^{fl} \cup \sigma^{act} \cup \sigma^{bg}$ and any nonnegative integer m, the ASPMT program D_m is defined as follows. The signature of D_m is $0{:}\sigma^{fl} \cup \dots \cup m{:}\sigma^{fl} \cup 0{:}\sigma^{act} \cup \dots \cup (m{-}1){:}\sigma^{act} \cup \sigma^{bg}$. By $i : F$ we denote the result of inserting $i :$ in front of every occurrence of every fluent and action constant in a formula$F$.

ASPMT program D_m is the conjunction of

$$i{:}G \rightarrow i{:}F$$

for every static law (2.1) in D and every $i \in \{0, \dots, m\}$, and for every action dynamic law (2.1) in D and every $i \in \{0, \dots, m{-}1\}$;

$$(i{+}1){:}G \wedge i{:}H \rightarrow (i{+}1){:}F$$

for every fluent dynamic law (2.2) in D and every $i \in \{0, \dots, m-1\}$.

The transition system represented by an action description$D$ consists of states (vertices) and transitions (edges). A *state* is an interpretation s of σ^{fl} such that $0\!:\!s \models_{bg} \text{SM}[D_0;\ 0\!:\!\sigma^{sd}]$. A *transition* is a triple $\langle s, e, s' \rangle$, where s and s' are interpretations of σ^{fl} and e is an interpretation of σ^{act}, such that

$$(0\!:\!s) \cup (0\!:\!e) \cup (1\!:\!s') \models_{bg} \text{SM}[D_1;\ (0\!:\!\sigma^{sd}) \cup (0\!:\!\sigma^{act}) \cup (1\!:\!\sigma^{fl})]\ .$$

The definition of the transition system above implicitly relies on the following property of transitions:

Theorem 1 *For every transition $\langle s, e, s' \rangle$, s and s' are states.*

The following theorem states the correspondence between the stable models of D_m and the paths in the transition system represented by D:

Theorem 2

$$(0\!:\!s_0) \cup (0\!:\!e_0) \cup (1\!:\!s_1) \cup (1\!:\!e_1) \cup \cdots \cup (m\!:\!s_m)$$
$$\models_{bg} \text{SM}[D_m;\ (0\!:\!\sigma^{sd}) \cup (0\!:\!\sigma^{act}) \cup (1\!:\!\sigma^{fl}) \cup (1\!:\!\sigma^{act}) \cup \cdots \cup (m{-}1\!:\!\sigma^{act}) \cup (m\!:\!\sigma^{fl})]$$

iff each triple $\langle s_i, e_i, s_{i+1} \rangle$ $(0 \le i < m)$ is a transition.

It is not difficult to check that ASPMT program D_m that is obtained from action description D is always tight. Functional completion (Bartholomew and Lee (2013)) on ASPMT can be applied to turn D_m into an SMT instance.

2.4 Related Works

2.4.1 PDDL+

PDDL 2.1 (Fox and Long (2003)) introduced numeric fluents and durative actions to represent and reason about continuous time and resource. The framework is further

extended to allow autonomous processes and event in PDDL+ (Fox and Long (2006)). While the former is similar to our simple encoding approach, the start-process-stop model in PDDL+ can be represented in our framework by representing a process as an inertial fluent. (Shin and Davis (2005)) extended SAT-based planning framework to cover an extension of PDDL+ language using SAT-based arithmetic constraint solvers. In (Shin and Davis (2005)), durative actions are always understood as the start action, continuous action and end action. Hybrid planning problems expressed in PDDL+ (Fox and Long (2006)) model mixed discrete and continuous change. Prior work on PDDL+ plan synthesis (Bogomolov *et al.* (2014); Coles *et al.* (2012); Coles and Coles (2014); Shin and Davis (2005)) assumes that continuous change is linear or handle nonlinear change by discretization (Penna *et al.* (2009)). Our framework is not restricted to the reasoning of hybrid transition systems whose continuous change is linear.

In (Bryce *et al.* (2015)) an SMT encoding of a PDDL+ action description is proposed that is able to perform reasoning about hybrid transition systems whose continuous change is non-linear. This is closely related to what we present in our framework. (Bryce *et al.* (2015)) proposes a syntactic translation of a PDDL+ instance to an equivalent SMT encoding but does not provide an implementation in automating this translation. Our framework is not simply syntactic sugar and provides an implementation for automated SMT translation as seen in system CPLUS2ASPMT.

2.4.2 Action Language $\mathcal{H}$

Our approach is similar to the Action Language $\mathcal{H}$ (Chintabathina (2008)) which is a recent extension of action language to reason about continuous process. One notable difference is there, each state represents an interval of time, rather than a particular timepoint. This yields a different notion of Hybrid Transition systems than

what is described in Hybrid Automata. Instead of using SMT solvers, an implementation of $\mathcal{H}$ is by translation into the language $\mathcal{AC}$ (Mellarkod *et al.* (2008)), which extends ASP with constraints. Action Language $\mathcal{H}$ does not have action dynamic laws, and consequently does not allow additive fluents. While each discrete state represents an interval of time, action language $\mathcal{H}$ does not provide support for continuous evolution via ODEs or constraint checking for each intermediate time point of continuous evolution.

Chapter 3

REPRESENTING LINEAR HYBRID AUTOMATA WITH CONVEX INVARIANTS BY $\mathcal{C}+$ MODULO THEORIES

In this chapter, we will present a framework for representing a specific case of Linear Hybrid Automata in $\mathcal{C}+$ Modulo Theories and go on to prove the correspondence of the transition systems. This chapter also presents certain classes of Hybrid Automata that can be represented in $\mathcal{C}+$ in a similar fashion.

3.1 Representation

Linear hybrid automata (Henzinger (1996)) are a special case of hybrid automata where (i) the initial, invariant, flow, guard, reset conditions are boolean combinations of linear inequalities, and (ii) the free variables of flow conditions are from $\dot{X}$ only. Further we assume that for each $\mathsf{Inv}_v(X)$ from each control mode v, the set of values of X that makes $\mathsf{Inv}_v(X)$ true forms a convex region. [1] For instance, this is the case if $\mathsf{Inv}_v(X)$ is a *conjunction* of linear inequalities.

We show how a linear hybrid automata H can be turned into an action description D_H in $\mathcal{C}+$, and extend this representation to non-linear hybrid automata in the next section. Let H be a hybrid automaton. We first define the signature of action description D_H as follows.

- For each real-valued variable X_i in H, a fluent constant X_i of sort $\mathcal{R}$.
- For each edge $e \in E$ and the corresponding $\mathsf{hevent(e)} \in \Sigma$, a boolean valued action constant $\mathsf{hevent(e)}$.

[1] A set X is *convex* if for any $x_1, x_2 \in X$ and any θ with $0 \leq \theta \leq 1$, we have $\theta x_1 + (1-\theta)x_2 \in X$.

- An action constant *Dur* of sort nonnegative reals.
- A Boolean action constant *Wait*.
- A fluent constant *Mode* of sort V.

The $\mathcal{C}+$ action description D_H consists of the following causal laws. We use lower case letter x_i for denoting a real-valued variable. [2] For lists of object constants and variables $Y = (Y_1, \ldots, Y_n)$ and $Z = (Z_1, \ldots, Z_n)$, by $Y = Z$, we denote the conjunction $(Y_1 = Z_1) \wedge \cdots \wedge (Y_n = Z_n)$.

- **Exogenous constants**:

$$\textbf{exogenous } X_i \quad (X_i \in X)$$

$$\textbf{exogenous } \mathsf{hevent}(e)$$

$$\textbf{exogenous } Dur$$

  Intuitively, these causal laws assert that the values of the fluents can be arbitrary. The action constant *Dur* is to record the Dur that each transition takes. Discrete transitions are assumed to have duration 0.

- **Discrete transitions:** For each control switch $e = (v_1, v_2) \in E$:
  - **Guard**:

$$\textbf{nonexecutable } \mathsf{hevent}(e) \textbf{ if } \neg \mathrm{Guard}_e(X)$$

    The causal law asserts that an h-event cannot be executed if its guard condition is not satisfied.

  - **Reset**:

$$\textbf{constraint } \mathsf{Reset}_e(x, x') \textbf{ if } X = x' \textbf{ after } X = x \wedge \mathsf{hevent}(e) \wedge Mode = v_1$$

[2] This is not to be confused with the notational convention for hybrid automata, where x_i is a real value.

The causal law asserts that if an h-event executed in the state with the corresponding mode, the discrete transition set the new value of fluent X in accordance with the reset condition.

– **Mode Update** :

$$\textbf{nonexecutable } \mathsf{hevent}(e) \textbf{ if } \mathit{Mode} \neq v_1$$
$$\mathsf{hevent}(e) \textbf{ causes } \mathit{Mode} = v_2$$
$$\textbf{inertial } \mathit{Mode} = v \qquad\qquad (v \in V)$$

The first causal law asserts an additional constraint for an h-event to be executable (when the state is in the corresponding mode). The second causal law resets the new control mode. The third causal law asserts the commonsense law of inertia on the control mode: the mode does not change when no action affects it.

– **Duration**:

$$\mathsf{hevent}(e) \textbf{ causes } \mathit{Dur}=0$$

During discrete transitions, duration is set to 0.

- **Continuous Transitions:** For each control mode $v \in V$:

– **Wait**:

$$\textbf{default } \mathit{Wait}=\text{TRUE}$$
$$\mathsf{hevent}(e) \textbf{ causes } \mathit{Wait}=\text{FALSE}$$

Wait is an auxiliary constant that is true when no h-events are executed, in which case continuous transitions should occur.

– **Flow**: For each $X_i \in X$,

$$\textbf{constraint } \mathsf{Flow}_v((x' - x)/t) \textbf{ if } X = x' \textbf{ after } X = x \wedge Mode = v \wedge Dur = t \wedge Wait = \text{TRUE} \quad (\delta > 0)$$

$$\textbf{constraint } x' = x \textbf{ if } X = x' \textbf{ after } X = x \wedge Mode = v \wedge Dur = 0 \wedge Wait = \text{TRUE} \tag{3.1}$$

These causal laws assert that when no h-event is executed (i.e., *Wait* is true), the next values of the continuous variables are determined by the flow condition.

– **Invariant**:

$$\textbf{constraint } \mathsf{Inv}_v(X) \textbf{ if } Mode = v$$

The causal law asserts that in each state, the invariant condition for the control mode should be true.

It is easy to see from the assumption on the flow condition of linear hybrid automata that the witness function exists and is unique; furthermore it is linear over time.

Also note that it is sufficient to check the invariant condition in each state only, not during the transitions, because of the assumption that the invariant condition is convex and the flow condition is linear, from which it follows that

$$\forall t \in [0, \delta](\mathsf{Inv}_v(f(0)) \;\wedge\; \mathsf{Inv}_v(f(\delta)) \rightarrow \mathsf{Inv}_v(f(t))) \tag{3.2}$$

is true, where f is the witness function.

The following theorem asserts the correctness of the translation.

Theorem 3 *There is a 1:1 correspondence between the paths of the transition system of a Linear Hybrid automata H with convex invariants and the paths of the transition system of the action description D_H.*

The proof is immediate from the following two lemmas. By a path in a transition system, we mean the sequence of edges. [3] First, we show that every path in the labelled transition system of T_H is a path in the transition system described by D_H.

Notation: We say that an interpretation I of σ satisfies F w.r.t. the background theory bg, denoted by $I \models_{bg} F$, if $I \cup J^{bg}$ satisfies F.

Lemma 1 *For any path*

$$p = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_{m-1}} (v_m, x_m)$$

in the labelled transition system of H, let

$$p' = \langle s_0, a_0, s_1, a_1, \dots, s_m \rangle,$$

where each s_i is an interpretation of fluent constants and each a_i is an interpretation of action constants such that, for $i = 0, \dots m-1$,

- $s_0 \models_{bg} (mode, X) = (v_0, x_0)$;
- $s_{i+1} \models_{bg} (mode, X) = (v_{i+1}, x_{i+1})$ $(i = 0, \dots, m-1)$, *where*
  - *if* $\sigma_i = \mathsf{hevent}(v_i, v_{i+1})$, *then* $a_i \models_{bg} Dur = 0$ *and* $a_i \models_{bg} wait = \text{FALSE}$;
  - *if* $\sigma_i \in \mathcal{R}_{\geq 0}$, *then* $a_i \models_{bg} Dur = \sigma_i$ *and* $a_i \models_{bg} wait = \text{TRUE}$;

Then, p' is a path in the transition system D_H.

Next, we show that every path in the transition system of D_H is a path in the labelled transition system of H.

[3] The start node of a path may not necessarily satisfy the initial conditions, but it is easy to check. Dropping this requirement simplifies the statement.

Lemma 2 *For any path*

$$q = (s_0, a_0, s_1, a_1, \ldots, s_m)$$

in the transition system of D_H, let

$$q' = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{m-1}} (v_m, x_m),$$

where each $v_i \in V$ and each $x_i \in R^n$ for $i = 0, \ldots, m$ such that

- $s_i \models_{bg} (Mode, X) = (v_i, x_i)$*;*
- σ_i *is*
  - $\mathsf{hevent}(v_i, v_{i+1})$ *if* $a_i \models_{bg} \mathsf{hevent}(v_i, v_{i+1})$*;*
  - $a_i(Dur)$ *otherwise.*

Then, q' is a path in the transition system of T_H.

3.2 Example

This section highlights the use of the earlier mentioned representation in representing the example of a simple water tank.

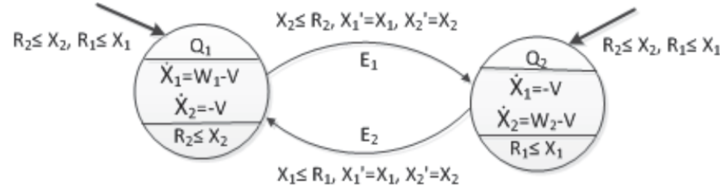

Figure 3.1: Hybrid Automata for Water Tank System.

Example 1 *The Hybrid Automata in Figure 3.1 describes a water tank example with 2 tanks X_1 and X_2. Here R_1 and R_2 are constants that describe the lower bounds of the level of water in the respective tanks. W_1 and W_2 are constants that define the*

rate at which water is being added to the respective tanks and V is the constant rate at which water is draining from the tanks. We apply the restriction that you can add water to only 1 tank at a time.

Assuming $W_1 = W_2 = 7.5$, $V = 5$, $R_1 = R_2 = 0$ and initially the level of water in the respective tanks are $X_1 = 0, X_2 = 8$, then the goal is to find the best way to add water to each of the tanks with the passage of time.

The Hybrid Automata description as well as the corresponding representation in $\mathcal{C}+$ is as follows.

Hybrid Automata Components

- Variables:
  - $X_1, X_1', \dot{X}_1$
  - $X_2, X_2', \dot{X}_2$
- States:
  - Q_1(mode=1)
  - Q_2(mode=2)
- Directed Graph: The graph is given above
- Invariants:
  - $Inv_{Q_1}(X) : X_2 \geq \mathrm{R}_2$
  - $Inv_{Q_2}(X) : X_1 \geq \mathrm{R}_1$
- Flow:

- $flow_{Q_1}(X) : \dot{X}_1 = \mathrm{W}_1 - \mathrm{V} \ \wedge \ \dot{X}_2 = -\mathrm{V}.$
- $flow_{Q2}(X) : \dot{X}_1 = -\mathrm{V} \ \wedge \ \dot{X}_2 = \mathrm{W}_2 - \mathrm{V}.$

- Jump:
  - $Guard_{q1,q2}(X) : X_2 \leq \mathrm{R}_2.$
  - $Guard_{q2,q1}(X) : X_1 \leq \mathrm{R}_1.$
  - $Reset_{q1,q2}(X, X') : X'_1 = X_1 \ \wedge \ X'_2 = X_2.$
  - $Reset_{q2,q1}(X, X') : X'_1 = X_1 \ \wedge \ X'_2 = X_2.$

C+ Action Description

$q \in \{Q_1, Q_2\}$; t, x_1, x_2 are variables of sort $\mathcal{R}_{\geq 0}$. $\mathrm{W}_1, \mathrm{W}_2, \mathrm{V}$ are fixed real numbers

Simple fluent constants:	Sort:
X_1, X_2	$\mathcal{R}_{\geq 0}$
Mode	$\{Q_1, Q_2\}$
Action constants:	Sort:
E_1, E_2, *Wait*	Boolean
Dur	$\mathcal{R}_{\geq 0}$

% Exogenous constants:

exogenous X_1, X_2, E_1, E_2, Dur

% Guard:

nonexecutable E_1 **if** $\neg(X_2 \leq R_2)$

nonexecutable E_2 **if** $\neg(X_1 \leq R_1)$

% Reset:

constraint $(X_1, X_2) = (x_1, x_2)$ **after** $(X_1, X_2) = (x_1, x_2) \wedge E_1 \wedge Mode = Q_1$

constraint $(X_1, X_2) = (x_1, x_2)$ **after** $(X_1, X_2) = (x_1, x_2) \wedge E_2 \wedge Mode = Q_2$

% Mode:

nonexecutable E_1 **if** $\neg(Mode = Q_1)$ **nonexecutable** E_2 **if** $\neg(Mode = Q_2)$

E_1 **causes** $Mode = Q_2$ E_2 **causes** $Mode = Q_1$

inertial $Mode = q \quad (q \in \{Q_1, Q_2\}$

% Duration:

E_1 **causes** $Dur = 0$ E_2 **causes** $Dur = 0$

% Wait:

default $Wait = \textsc{true}$

E_1 **causes** $Wait = \textsc{false}$ E_1 **causes** $Wait = \textsc{false}$

% Flow:

constraint $((x'_1 – x_1)/t, (x'_2 – x_2)/t) = (\mathrm{W}_1 - \mathrm{V}, -\mathrm{V})$ **if** $(X_1, X_2) = (x'_1, x'_2)$

after $(X_1, X_2) = (x_1, x_2) \wedge Mode = Q_1 \wedge Dur = 0 \wedge Wait = \textsc{true}$

constraint $((x'_1 – x_1)/t, (x'_2 – x_2)/t) = (-\mathrm{V}, \mathrm{W}_2 - V)$ **if** $(X_1, X_2) = (x'_1, x'_2)$

after $(X_1, X_2) = (x_1, x_2) \wedge Mode = Q_2 \wedge Dur = 0 \wedge Wait = \textsc{true}$

constraint $(x'_1, x'_2) = (x_1, x_2)$ **after** $(X_1, X_2) = (x'_1, x'_2)$

$\wedge Mode = q \wedge Dur = 0 \wedge Wait = \textsc{true} \quad (q \in \{Q_1, Q_2\})$

% Invariant

constraint $X_2 \geq \mathrm{R}_2$ **if** $Mode = Q_1$ **constraint** $X_1 \geq \mathrm{R}_1$ **if** $Mode = Q_2$

3.3 Beyond Linear Hybrid Automata

Note that formula (3.2) is not necessarily true in general even when $\mathsf{Inv}_v(X)$ is a Boolean combination of linear (in)equalities (disjunction over them may yield a non-convex invariant). The robot example introduction is another such instance.

Let's now assume $\mathsf{Flow}_v(X, \dot{X})$ is the conjunction of formulas of the form $\dot{X}_i = g(X)$ for each X_i, where g is a Lipschitz continuous function whose variables are from X only. [4] In this case, it is known that the witness function f exists and is unique. This is a common assumption imposed on hybrid automata.

3.3.1 Representation

The encoding in the previous section still works even when the flow condition is non-linear as long as for the unique witness function for each transition, formula (3.2) is true. In such a case, we modify the **Flow** representation as

- **Flow**: For each $v \in V$ and $X_i \in X$,

  $$\textbf{constraint } X_i = f_i(\delta) \textbf{ after } X = x \land \mathit{Mode} = v \land \mathit{Dur} = \delta \land \mathit{Wait} = \text{TRUE}$$

  where $f_i : [0, \delta] \to \mathcal{R}^n$ is the witness function such that

  (1) $f_i(0) = x_i$ and

  (2) for all reals $\epsilon \in [0, t]$, $\mathsf{Flow}_v(\dot{f_i}(\epsilon), f(\epsilon))$ is true.

Theorem 3 and Lemmas 1, 2 still remain true for this class of hybrid automata assumed in the beginning of this section if we consider this version of D_H instead

[4] A function $f : \mathcal{R}^n \to \mathcal{R}^n$ is called *Lipschitz continuous* if there exists $\lambda > 0$ such that for all $x, x' \in \mathcal{R}^n$,

$$|f(x) - f(x')| < \lambda |x - x'|.$$

of the previous one. The following modified theorem shows the correctness of the translation.

Theorem 4 *There is a 1:1 correspondence between the paths of the transition system of a Hybrid automata H with convex invariants and the paths of the transition system of the action description D_H as obtained using the translation mentioned in this section.*

The proof is immediate from the following two lemmas. By a path in a transition system, we mean the sequence of edges. [5] First, we show that every path in the labelled transition system of T_H is a path in the transition system described by D_H.

Notation: We say that an interpretation I of σ satisfies F w.r.t. the background theory bg, denoted by $I \models_{bg} F$, if $I \cup J^{bg}$ satisfies F.

Lemma 3 *For any path*

$$p = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_{m-1}} (v_m, x_m)$$

in the labelled transition system of H, let

$$p' = \langle s_0, a_0, s_1, a_1, \dots, s_m \rangle,$$

where each s_i is an interpretation of fluent constants and each a_i is an interpretation of action constants such that, for $i = 0, \dots m{-}1$,

- $s_0 \models_{bg} (mode, X) = (v_0, x_0)$*;*
- $s_{i+1} \models_{bg} (mode, X) = (v_{i+1}, x_{i+1})$ *$(i = 0, \dots, m{-}1)$, where*
  - *if* $\sigma_i = \mathsf{hevent}(v_i, v_{i+1})$*, then* $a_i \models_{bg} Dur{=}0$ *and* $a_i \models_{bg} wait{=}\text{FALSE}$*;*

[5] The start node of a path may not necessarily satisfy the initial conditions, but it is easy to check. Dropping this requirement simplifies the statement.

– *if* $\sigma_i \in \mathcal{R}_{\geq 0}$*, then* $a_i \models_{bg} Dur = \sigma_i$ *and* $a_i \models_{bg} wait = \text{TRUE}$*;*

Then, p' *is a path in the transition system* D_H*.*

Next, we show that every path in the transition system of D_H is a path in the labelled transition system of H.

Lemma 4 *For any path*

$$q = (s_0, a_0, s_1, a_1, \ldots, s_m)$$

in the transition system of D_H*, let*

$$q' = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{m-1}} (v_m, x_m),$$

where each $v_i \in V$ *and each* $x_i \in R^n$ *for* $i = 0, \ldots, m$ *such that*

- $s_i \models_{bg} (Mode, X) = (v_i, x_i)$*;*
- σ_i *is*
  - $\mathsf{hevent}(v_i, v_{i+1})$ *if* $a_i \models_{bg} \mathsf{hevent}(v_i, v_{i+1})$*;*
  - $a_i(Dur)$ *otherwise.*

Then, q' *is a path in the transition system of* T_H*.*

3.3.2 Example

Example 2 *Consider the two balls with different elasticity falling to the ground. This example involves 2 balls with the same elasticity falling to the ground. Ball 1 starts at 2 units from the ground while ball 2 starts at 3 units from the ground. Our aim is to formalize this problem and observe the transition system of these 2 balls with time.*

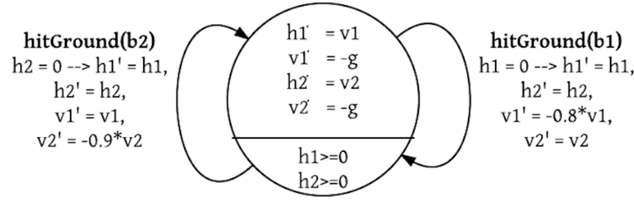

Figure 3.2: Hybrid Automata of Two Ball Example

The **Flow** *condition for Ball* $b1$ *is represented as:*

$$\textbf{constraint } V_1 = v + (-g) \cdot t \textbf{ after } V_1 = v \wedge Dur = t \wedge Wait = \text{TRUE}$$

$$\textbf{constraint } H_1 = h + v \cdot \delta - (0.5) \cdot g \cdot t \cdot t \textbf{ after } H_1 = h \wedge Dur = t \wedge Wait = \text{TRUE}.$$

For this example, it is clear that (3.2) *is true.*

The Hybrid Automata description as well as the corresponding representation is as follows.

Hybrid Automata Components

The control graph can be seen in Figure 3.2

- Variables:
  - $h1, h1', \dot{h1}$
  - $h2, h2', \dot{h2}$
  - $v1, v1', \dot{v1}$
  - $v2, v2', \dot{v2}$
- States:
  - s_0(Corresponds to $mode = 1$)
- Hevents:
  - $hitGround_b1$

- $hitGround_b2$

- Directed Graph: The graph is given above

- Invariants:

  - $inv_{s_0}(h1, h2) : h1 \geq 0 \wedge h2 \geq 0$

- Flow:

  - $flow_{s_0}(h1, h2, v1, v2) : \dot{h1} = v1 \;\wedge\; \dot{h2} = v2 \;\wedge\; \dot{v1} = -g \;\wedge\; \dot{v2} = -g.$

- Jump:

  - $Guard_{(s_0,s_0)}(h1, h2, v1, v2) : h1 = 0 \;\wedge\; h2 = 0$
  - $Reset_{(s_0,s_0)}(h1, h2, v1, v2) : h1' = 0 \wedge h2' = h2 \wedge v1' = -0.8{*}v1 \wedge v2' = v2$
  - $Reset_{(s_0,s_0)}(h1, h2, v1, v2) : h1' = h1 \wedge h2' = 0 \wedge v1' = v1 \wedge v2' = -0.9{*}v2$

C+ Action Description

$q \in \{S_0\}$; t,h_1,h_2 are variables of sort $\mathcal{R}_{\geq 0}$. v_1,v_2 are variables of sort $\mathcal{R}$.

b ranges over $\{B_1, B_2\}$. G is a fixed real number

Simple fluent constants:	Sort:
$Height(b)$	$\mathcal{R}_{\geq 0}$
$Velocity(b)$	$\mathcal{R}$
$Mode$	$\{Q_1, Q_2\}$

Action constants:	Sort:
$HitGround(b),\ Wait$	Boolean
Dur	$\mathcal{R}_{\geq 0}$

% Exogenous constants:

exogenous $Height(b), Velocity(b), HitGround(b), Dur$

% Guard:

nonexecutable $HitGround(b)$ **if** $\neg(Height(b) = 0)$

% Reset:

constraint $(Height(B_1), Velocity(B_1), Height(B_2), Velocity(B_2))$

$= (h_1, -0.8 * v_1, h_2, v_2)$ **after** $(Height(B_1), Velocity(B_1), Height(B_2),$

$Velocity(B_2)) = (h_1, v_1, h_2, v_2) \ \wedge HitGround(B_1) \wedge Mode = Q_1$

constraint $(Height(B_1), Velocity(B_1), Height(B_2), Velocity(B_2))$

$= (h_1, -0.9 * v_1, h_2, v_2)$ **after** $(Height(B_1), Velocity(B_1), Height(B_2),$

$Velocity(B_2)) = (h_1, v_1, h_2, v_2) \ \wedge HitGround(B_2) \wedge Mode = Q_1$

% Mode:

nonexecutable $HitGround(b)$ **if** $\neg(Mode = S_0)$

$HitGround(b)$ **causes** $Mode = S_0$

inertial $Mode = q \quad (q \in \{S_0\})$

% Duration:

$HitGround(b)$ **causes** $Dur = 0$

% Wait:

default $Wait = \text{TRUE}$

$HitGround(b)$ **causes** $Wait = \text{FALSE}$

% Flow:

constraint $Velocity(b) = v_1 + (-G) \cdot t$ **after** $Velocity(b) = v_1$

$\wedge \ Mode = S_0 \ \wedge Dur = t \wedge Wait = \text{TRUE}$

constraint $Height(b) = h_1 + v_1 \cdot \delta - (0.5) \cdot G \cdot t \cdot t$ **after** $Height(B) = h_1$

$\wedge\ Mode = S_0\ \wedge Dur = t \wedge Wait = \text{TRUE}.$

% Invariant

constraint $Height(b) \geq 0$ **if** $Mode = S_0$

3.4 Proofs

3.4.1 Proof of Lemma 1

Lemma 1

p' is a path in the transition system D_H.

Lemma 5 *Given a linear function $f(t)$ over $t \in [0, \sigma]$ for some $\sigma \in \mathcal{R}_{\geq 0}$ such that $f : [0, \sigma] \to \mathcal{R}$ and a conjunction of linear inequalities $P(X)$ over $X \in \mathcal{R}_{\geq 0}$. Then,*

$$\forall \epsilon \in (0, \sigma)(P(f(0)) \wedge P(f(\sigma)) \to P(\epsilon)).$$

Proof. Since $P(X)$ is a conjunction of linear inequalities, we know from (Sec 2.2.4, Boyd and Vandenberghe (2004)) values of X that satisfies $P(X)$ must form a convex region [6] in $\mathcal{R}^n$. Since a straight line is basically a linear function and $f(t)$ is a linear function, it follows that for any $\epsilon \in (0, \sigma)$, $P(f(\epsilon))$ is true, if $P(f(0))$ and $P(f(\sigma))$ are true. ∎

Lemma 6 *Let H be a linear hybrid automaton and*

$$(v, x) \xrightarrow{\sigma} (v, x')$$

[6] A convex region is a set of points such that, given any two points A, B in that set, the line AB joining them lies entirely within that set. If P is a convex set and $x_1 ... x_k$ are any points in it, then $x = \sum_{i=1}^{k} \lambda_i x_i$ is also in P, where $\lambda_i > 0$ and $\sum i = 1^k \lambda_i = 1$.

be a transition in T_H such that $\sigma \in \mathcal{R}_{\geq 0}$. $f(t) = x + t * (x' - x)/\sigma$ is a linear differentiable function from $[0, \sigma]$ to $\mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma] \rightarrow \mathcal{R}^n$ such that: (1) $f(0) = x$ and $f(\sigma) = x'$ and (2) for all reals $\epsilon \in (0, \sigma)$, both $\mathsf{Inv}_v(f(\epsilon))$ and $\mathsf{Flow}_v(\dot{f}(\epsilon))$ are true.

Proof. We check that f satisfies the above conditions:

- $f(t)$ is differentiable over $t \in [0, \sigma]$.
- It is clear that $f(0) = x$ and $f(\sigma) = x'$.
- Since (v, x) and (v', x') are states of T_H, it follows that $\mathsf{Inv}_v(f(0))$ and $\mathsf{Inv}_v(f(\sigma))$ are true. By Lemma 5 it is clear that for $\epsilon \in (0, \sigma)$, $\mathsf{Inv}_v(f(\epsilon))$ is true.
- Since $(v, x) \xrightarrow{\sigma} (v, x')$ is a transition in T_H, it follows that there is a function f' such that (1) f' is differentiable in $[0, \sigma]$, (2) for any $\epsilon \in (0, \sigma)$, $\mathsf{Flow}_v(\dot{f}'(\epsilon))$ is true, (3) $f'(0) = x$ and $f'(\sigma) = x'$. Since f' is continuous on $[0, \sigma]$ (differentiability implies continuity) and differentiable on $(0, \sigma)$, by mean value theorem [7] , there is a point $c \in (0, \sigma)$ such that $\dot{f}'(c) = (x' - x)/\sigma$. Consequently, $\mathsf{Flow}_v((x' - x)/\sigma)$ is true. As a result, we get $\mathsf{Flow}_v(\dot{f}(\epsilon))$ is true for all $\epsilon \in (0, \sigma)$.

∎

Lemma 7 *For each $i \geq 0$, s_i is a state in the transition system of D_H.*

Proof. By definition we are to show that

$$0 : s_i \models_{bg} SM[(D_H)_0; \phi],$$

while $SM[(D_H)_0; \phi]$ is equivalent to the conjunctions of,

[7] http://en.wikipedia.org/wiki/Mean_value_theorem

$$\leftarrow \neg(\mathsf{Inv}_v(0:X)) \ \wedge\ 0:Mode = v. \tag{3.3}$$

for each $v \in V$. Since p is a path,for each $i \geq 0$,(v_i, x_i) is a state in T_H. By the definition of hybrid transition systems, $\mathsf{Inv}_{v_i}(X_i)$ is true. Hence $0 : s_i \models_{bg}$ (3.3) ∎

Lemma 8 *For each $i \geq 0$, $\langle s_i, a_i, s_{i+1}\rangle$ is a transition.*

Proof. By definition, we are to show that

$$0\!:\!s_i \cup 0\!:\!a_i \cup 1\!:\!s_{i+1} \models_{bg} \text{SM}[(D_H)_1; 0\!:\!\sigma^{act} \cup 1\!:\!\sigma^{fl}]. \tag{3.4}$$

From (Ferraris *et al.* (2011)) it is clear that for any formula F SM$[F]$ is equivalent to $Comp[F]$. Additionally, it is also clear that SM$[F] \ \wedge\ Choice(\mathbf{c})$ is equivalent to SM$[F]$ where $Choice(\mathbf{c})$ represents the choice rules for all non-intensional constants $\mathbf{c}$.

$Comp[(D_H)_1; 0 : \sigma^{act} \cup 1 : \sigma^{fl}]$ is equivalent to the conjunction of following formulas:

- Formula $FLOW$, which are the rules

$$\leftarrow\ \mathsf{Flow}_v(1:X - 0:X/t) \ \wedge\ 0:Mode = v \ \wedge\ 0:Dur = t \ \wedge\ 0:Wait \ \wedge\ t > 0 \tag{3.5}$$

  AND

$$\leftarrow\ 1:X = 0:X \ \wedge\ 0:Mode = v \ \wedge\ 0:Dur = 0 \ \wedge\ 0:Wait \tag{3.6}$$

- Formula INV, which is the rule

$$\leftarrow \neg(\mathsf{Inv}_v(k:X)) \ \wedge\ k:Mode = v. \tag{3.7}$$

  for each $k \in \{0, 1\}$ and each $v \in V$.

- Formula $GUARD$, which is the conjunction of

$$\leftarrow 0 : \mathsf{hevent}(e) \;\wedge\; 0 : \neg\mathsf{Guard}_e(X) \tag{3.8}$$

  for each edge $e \in E$;

- Formula $RESET$, which is the conjunction of

$$\begin{aligned} \leftarrow \neg\mathsf{Reset}_e(x_0, x_1) \;\wedge\; 1 : X = x_1 \\ \wedge\; 0 : \mathsf{hevent}(e) \;\wedge\; 0 : X = x_0 \;\wedge\; 0 : Mode = v. \end{aligned}$$

  for each edge $e \in E$;

- Formula $MODE$, which is the conjunction of

$$\begin{aligned} &\leftarrow 0 : \mathsf{hevent}(e) \;\wedge\; \neg(0 : Mode = v_1). \\ &1 : Mode = v_2 \;\leftrightarrow\; (0 : Mode = v_1 \wedge 0 : \mathsf{hevent}(e)) \vee \\ &\qquad\qquad (1 : Mode = v_2 \;\wedge\; 0 : Mode = v_2). \end{aligned}$$

  for each edge $e = (v_1, v_2) \in E$;

- Formula $DURATION$, which is the rule

$$0 : Dur = 0 \;\leftarrow\; \bigvee_{e \in E} 0 : \mathsf{hevent}(e).$$

  for each edge $e = (v_1, v_2) \in E$;

- Formula $WAIT$, which is the rule

$$\begin{aligned} &0 : Wait = false \;\leftrightarrow\; \bigvee_{e \in E} 0 : \mathsf{hevent}(e). \\ &0 : Wait = true \;\leftrightarrow\; \neg\neg 0 : Wait = true. \end{aligned}$$

We will show that $0 : s_i \;\cup\; 0 : a_i \;\cup\; 1 : s_{i+1}$ satisfies $FLOW$, $GUARD$, $RESET$, $MODE$, $DURATION$, $WAIT$. From the definition of T_H there are 2 cases for the value of σ:

Case 1: $\sigma = \mathsf{hevent}(e)$ where $e = (v_i, v_{i+1})$. It follows from the construction of p' that $(Dur)^{a_i} = 0$, $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$ and $(Wait)^{a_i} = \text{FALSE}$. Since $FLOW$ is trivially satisfied, it is sufficient to consider only $GUARD$, $RESET$, $MODE$, $DURATION$, $WAIT$.

From the fact that:

$$(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$$

is a transition in T_H and that $\sigma_i = \mathsf{hevent}(e)$, it follows from the definition of hybrid transition systems that $\mathsf{Guard}_e(0 : X)$ and $\mathsf{Reset}_e(0 : X, 1 : X)$ is true.

- $GUARD$: It is immediate that $0 : s_i \ \cup\ 1 : s_{i+1} \models_{bg} \mathsf{Guard}_e(0 : X)$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup\ 0 : a_i \ \cup\ 1 : s_{i+1} \models_{bg} GUARD$

- $RESET$: Note that $s_i \models_{bg} (Mode, X) = (v_i, x_i)$ and $s_{i+1} \models_{bg} (Mode, X) = (v_{i+1}, x_{i+1})$. It is immediate that $0 : s_i \ \cup\ 1 : s_{i+1} \models_{bg} \mathsf{Reset}_e(0 : X, 1 : X), 0 : s_i \models_{bg} 0 : Mode = v_i$, $0 : s_i \models_{bg} 0 : X = x_i$ and $0 : s_i \models_{bg} 1 : X = x_{i+1}$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup\ 0 : a_i \ \cup\ 1 : s_{i+1} \models_{bg} RESET$

- $MODE$: Note that $s_i \models_{bg} (Mode, X) = (v_i, x_i)$ and $s_i \models_{bg} (Mode, X) = (v_{i+1}, x_{i+1})$. It is immediate that $0 : s_i \models_{bg} 0 : Mode = v_i$ and $1 : s_{i+1} \models_{bg} 1 : Mode = v_{i+1}$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup\ 0 : a_i \ \cup\ 1 : s_{i+1} \models_{bg} MODE$.

- $DURATION$: Since $(Dur)^{a_i} = 0$ and $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup\ 0 : a_i \ \cup\ 1 : s_{i+1} \models_{bg} DURATION$

- $WAIT$: Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, and $(Wait)^{a_i} = \text{FALSE}$ it follows that $0 : s_i \ \cup\ 0 : a_i \ \cup\ 1 : s_{i+1} \models_{bg} WAIT$

Case 2: $\sigma_i \in \mathcal{R}_{\geq 0}$. By the construction of p', $Dur^{a_i} = \sigma_i$, $Wait^{a_i} = \text{TRUE}$ and $(\mathsf{hevent}(e))^{a_i} = \text{FALSE}$ for every $e = (v, v') \in E$. Since $GUARD$, $RESET$, $MODE$,

$DURATION$, $WAIT$ is trivially satisfied, it is sufficient to consider only $FLOW$ and INV.

From the fact that

$$(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$$

is a transition of T_H and that $\sigma_i \in \mathcal{R}_{\geq 0}$, it follows from the definition of hybrid transition systems that

(a) $v_i = v_{i+1}$, and

(b) there is a differentiable function $f : [0, \sigma_i] \rightarrow \mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma_i] \rightarrow \mathcal{R}^n$ such that: (1) $f(0) = x_i$ and $f(\sigma_i) = x_{i+1}$ and (2) for all reals $\epsilon \in (0, \sigma_i)$, both $\mathsf{Inv}_{v_i}(f(\epsilon))$ and $\mathsf{Flow}_{v_i}(\dot{f}(\epsilon))$ are true.

We check the following:

- INV: From the fact that (v_i, x_i) and (v_{i+1}, x_{i+1}) are states in T_H, by the definition of hybrid transition systems, $\mathsf{Inv}_{v_i}(x_i)$ and $\mathsf{Inv}_{v_{i+1}}(x_{i+1})$ are true. Note that $s_i \models_{bg} (Mode, x) = (v_i, x_i)$ and $s_{i+1} \models_{bg} (Mode, x) = (v_{i+1}, x_{i+1})$. As a result,

$$0 : s_i \models_{bg} \leftarrow \neg(\mathsf{Inv}_{v_i}(0 : X)) \ \wedge \ 0 : Mode = v.$$

$$1 : s_{i+1} \models_{bg} \leftarrow \neg(\mathsf{Inv}_{v_i}(1 : X)) \ \wedge \ 1 : Mode = v.$$

  Hence $0{:}s_i \cup 0{:}a_i \cup 1{:}s_{i+1} \models_{bg} INV$

- $FLOW$:
  - If $\sigma_i = 0$, then $Dur^{e^i} = 0$. From (b), $x_i = x_{i+1} = f(0)$. As a result $X^{s_i} = X^{s_{i+1}}$ and it follows that $0{:}s_i \cup 0{:}a_i \cup 1{:}s_{i+1} \models_{bg}$ (3.6).
  - If $\sigma_i > 0$, then $Dur^{e^i} > 0$. By Lemma 6, $f(t) = x_i + t * (x_{i+1} - x_i)/\sigma_i$ is a differentiable function that satisfies all the conditions in (b). As a result, $\mathsf{Flow}_{v_i}((x_{i+1} - x_i)/\sigma_i)$ is true and thus $0{:}s_i \cup 0{:}e_i \cup 1{:}s_{i+1} \models_{bg} \mathsf{Flow}_{v_i}((1{:}x - 0{:}x)/Dur)$. It follows that $0{:}s_i \cup 0{:}e_i \cup 1{:}s_{i+1} \models_{bg}$ (3.5).

∎

Using the lemmas defined and proved above, we prove Lemma 2 as follows:

Proof. By Lemma 7, each s_i is a state of D_H. By Lemma 8, each $\langle s_i, a_i, s_{i+1}\rangle$ is a transition of D_H. So p' is a path in the transition system of D_H. ∎

3.4.2 Proof of Lemma 2

Lemma 2

q' is a path in the transition system of T_H.

Lemma 9

(a) For each $i \geq 0$, (v_i, x_i) is a state in T_H, and

(b) (v_0, x_0) is an initial state in T_H.

Proof.

(a) By definition, we are to show that $\mathsf{Inv}_{v_i}(x_i)$ is true. Since each s_i is a state in the transition system of D_H, by definition,

$$0\!:\!s_i \models_{bg} \mathrm{SM}[(D_H)_0; \emptyset]. \tag{3.9}$$

Note that $\mathrm{SM}[(D_H)_0; \emptyset]$ is equivalent to the conjunction of the formula:

$$0\!:\!\mathsf{Inv}_v(x) \leftarrow 0\!:\!Mode = v \tag{3.10}$$

for each location $v \in V$. Since $(Mode)^{s_i} = v_i$, it follows that $s_i \models_{bg} \mathsf{Inv}_{v_i}(x)$. Since $x^{s_i} = x_i$, it follows that $\mathsf{Inv}_{v_i}(x_i)$ is true.

(b) We are to show that $\mathsf{Init}_{v_0}(x_0)$ is true. Since we express $\mathsf{Init}_{v_0}(x_0)$ as is in D_H. It is trivially entailed. ∎

Lemma 10 *For each $i \geq 0$, $(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$ is a transition in T_H.*

Proof. From the fact that (s_i, a_i, s_{i+1}) is a transition of D_H, by definition we know that

$$0\!:\!s_i \cup 0\!:\!a_i \cup 1\!:\!s_{i+1} \models_{bg} \mathrm{SM}[(D_H)_1; 0\!:\!\sigma^{act} \cup 1\!:\!\sigma^{fl}]. \tag{3.11}$$

$Comp[(D_H)_1; 0 : \sigma^{act} \cup 1 : \sigma^{fl}]$ is equivalent to the conjunction of $FLOW$, $RESET$, $GUARD$, $DURATION$, $MODE$, $WAIT$.

Consider two cases:

Case 1: There exists an edge $e = (v, v')$ such that $(\mathsf{hevent}(e))^{a_i} = t$. Since $Mode^{s_i} = v_i$ and $Mode^{s_{i+1}} = v_{i+1}$, it follows that (v, v') must be (v_i, v_{i+1}).As a result, $(\mathsf{hevent}(e))^{a_i} = t$. It follows from the definition that $\sigma_i = \mathsf{hevent}(e)$.

- Since $0 : s_i \cup 0 : a_i \cup 1 : s_{i+1} \models_{bg} GUARD$, $X^{s_{i+1}} = x_{i+1}$ and $X^{s_i} = x_i$, it is immediate that $\mathsf{Guard}_e(x_i)$ is true.
- Since $0 : s_i \cup 0 : a_i \cup 1 : s_{i+1} \models_{bg} RESET$, $X^{s_{i+1}} = x_{i+1}$ and $X^{s_i} = x_i$, it is immediate that $\mathsf{Reset}_e(x_i, x_{i+1})$ is true.

(b) By Lemma 9, (v_i, x_i) and (v_{i+1}, x_{i+1}) are states.

Case 2: There exists an edge $e = (v, v')$ such that $(\mathsf{hevent}(e))^{a_i} = \text{FALSE}$ for $(v, v') \in E$. By construction, $(Dur)^{a_i} = \sigma_i$ for some $\sigma_i \in \mathcal{R}_{\geq 0}$. By Lemma 9, (v_i, x_i) and (v_{i+1}, x_{i+1}) are states. From $MODE$, it follows that $Mode^{s_i} = Mode^{s_{i+1}}$. As a result, $v_i = v_{i+1}$. We are to show that there is a differentiable function $f : [0, \sigma_i] \to \mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma_i] \to \mathcal{R}^n$ such that: (1) $f(0) = x_i$ and $f(\sigma_i) = x_{i+1}$ and (2) for all reals $\epsilon \in (0, \sigma_i)$, both $\mathsf{Inv}_{v_i}(f(\epsilon))$ and $\mathsf{Flow}_{v_i}(\dot{f}(\epsilon))$ are true. Define $f(t) = x_i + t * (x_{i+1} - x_i)/\sigma_i$. We check that f satisfies the above conditions:

- $f(t)$ is differentiable over $[0, \sigma_i]$.

- It is clear that $f(0) = x_i$ and $f(\sigma_i) = x_{i+1}$.

- We check that for any $\epsilon \in (0, \sigma)$, $\mathsf{Inv}_v(f(\epsilon))$ is true. From $i : s_i \cup (i+1) : s_{i+1} \models_{bg}$ (3.7), it follows that $\mathsf{Inv}_{v_i}(f(0))$ and $\mathsf{Inv}_{v_i}(f(\sigma_i))$ are true. By Lemma 5 it is clear that for $\epsilon \in (0, \sigma)$, $\mathsf{Inv}_v(f(\epsilon))$ is true.

- We check that for any $\epsilon \in (0, \sigma)$, $\mathsf{Flow}_{v_i}(\dot{f}(\epsilon))$ is true. We only consider the case where $\sigma_i > 0$ because otherwise is trivial (there is no $\epsilon \in (0, 0)$). From (3.5), it follows that $\mathsf{Flow}_{v_i}((f(\sigma_i) - f(0))/\sigma_i)$ is true. Since $f(t)$ is a linear function, it follows that for any $\epsilon \in (0, \sigma_i)$, $\dot{f}(\epsilon) = (f(\sigma_i) - f(0))/\sigma_i$. As a result, $\mathsf{Flow}_{v_i}(\dot{f}(\epsilon))$ is true

From above, we conclude that (10) is a transition. ∎

Using the lemmas defined and proved above, we prove Lemma 2 as follows:

Proof. By Lemma 9 (a), each (v_i, x_i) is a state in T_H. By Lemma 10, each $(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$ is a transition in T_H. So q' is a path in T_H. If $s_0 \models_{bg} \mathsf{Init}_{v_0}$ then by Lemma 9 (b), (v_0, x_0) is a initial state in T_H. ∎

3.4.3 Proof of Lemmas 3 and 4

As we have mentioned earlier the Theorem 3 and Lemmas 1, 2 still remain true for this class of hybrid automata assumed in the beginning of this section if we consider this version of D_H instead of the previous one. Only a few parts of the proof need to be modified to take care of this updated consideration.

Equation 3.5 is modified and represented as follows:

$$\leftarrow \ 1 : X_i = f_i(\delta) \ \wedge \ 0 : X = x \wedge 0 : \mathit{Mode} = v \wedge 0 : \mathit{Dur} = \delta \wedge 0 : \mathit{Wait} = \text{TRUE}$$

where $f_i : [0, \delta] \to \mathcal{R}^n$ is the witness function such that

(1) $f_i(0) = x_i$ and

(2) for all reals $\epsilon \in [0, t]$, $\mathsf{Flow}_v(\dot{f_i}(\epsilon), f(\epsilon))$ is true.

The proof for Lemma 3 follows from Lemma 1. The only modification to be made is to Lemma 8. The following will be the modified proof for *FLOW* in *Case 2*:

- If $\sigma_i = 0$, then $Dur^{e^i} = 0$. From (b), $x_i = x_{i+1} = f(0)$. As a result $X^{s_i} = X^{s_{i+1}}$ and it follows that $0\!:\!s_i \cup 0\!:\!a_i \cup 1\!:\!s_{i+1} \models_{bg}$ (3.6).

- If $\sigma_i > 0$, then $Dur^{e^i} > 0$. From (3.6) we know that f_i is a differentiable function that satisfies all the conditions in (b). Since we consider $\textsc{Flow}(X, \dot{X})$ to be the conjunction of formulas of the form $\dot{X}_i = g(X)$ for each X_i, where g is a Lipschitz continuous function whose variables are from X, hence f_i is the only solution to $\dot{X}_i$. It follows that $0\!:\!s_i \cup 0\!:\!e_i \cup 1\!:\!s_{i+1} \models_{bg}$ (3.5).

The proof of Lemma 4 follows from Lemma 3. The only modification to be made is to Lemma 10. The following will be the modified proof for *FLOW* in *Case 2*: From (3.6) we know that f_i is a differentiable function that satisfies all the conditions for differentiable function f defined for a transition in Hybrid Automata. As a result, $\mathsf{Flow}_{v_i}(\dot{f}(\epsilon))$ is true.

Chapter 4

$\mathcal{C}+$ MODULO ODE

In this section we introduce two new abbreviations of causal laws to express the continuous evolutions governed by ODEs. As usual, we assume ODEs are Lipschitz continuous in order to ensure that the solutions to the ODEs are unique.

4.1 New Causal Laws for Expressing Continuous Evolutions of ODEs

We assume the set σ^{fl} of fluent constants contains a set σ^{diff} of real valued fluent constants $X = (X_1, \ldots, X_n)$ called *differentiable* fluent constants, and an inertial fluent constant *Mode*, which ranges over a finite set of control modes. Intuitively, the values of differentiable fluent constants are governed by some ODEs associated with each value of *Mode*. We also assume that *Dur* is an exogenous action constant of sort $\mathcal{R}_{\geq 0}$.

Below are the two new abbreviations.

- A *rate declaration* is an expression of the form:

$$\textbf{derivative of } X_i \textbf{ is } F_i(X) \textbf{ if } Mode = v \tag{4.1}$$

  for each differentiable fluent constant $X_i \in \sigma^{diff}$ and for each value v of *Mode*, where $F_i(X)$ is a fluent formula over $\sigma^{bg} \cup \sigma^{diff}$. We assume that an action description has a unique rate declaration for each X_i and v. This declaration can be shown using the two ball example where we know that the rate of change of height of the ball is dependent on its velocity in $Mode = 1$.

$$\textbf{derivative of } Height(B_1) \textbf{ is } velocity(B_1) \textbf{ if } Mode = 1$$

[1] For (4.1), by $d/dt[X_i](v)$ we denote the formula $F_i(X)$. Let θ_v be the list $d/dt[X_1](v), \ldots, d/dt[X_n](v)$ for all differentiable fluent constants $X_1, \ldots, X_n$ in σ^{diff}. The set of rate declarations expands into the following causal laws:

$$\textbf{constraint } (X_1, \ldots, X_n) = (x_1 + y_1, \ldots, x_n + y_n) \textbf{ after } (X_1, \ldots, X_n) = (x_1, \ldots, x_n) \ \wedge \ (y_1, \ldots, y_n) = \int_0^\delta (d/dt[X_1](v), \ldots, d/dt[X_n](v))dt \ \wedge \ Mode = v \ \wedge \ Dur = \delta \ \wedge \ Wait = \text{TRUE} \tag{4.2}$$

where $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ are real variables. This can be demonstrated once again with the two ball example where H_1, V_1, H_2, V_2 represent $Height$ and $Velocity$ of $ball_1$ and $ball_2$.

$$\textbf{constraint } (H_1, H_1, H_2, V_2) = (x_1 + y_1, \ldots, x_4 + y_4) \textbf{ after } (H_1, H_1, H_2, V_2) = (x_1, \ldots, x_4) \ \wedge \ (y_1, \ldots, y_4) = \int_0^\delta (d/dt[H_1](1), \ldots, d/dt[V_2](1))dt \ \wedge \ Mode = 1 \ \wedge \ Dur = \delta \ \wedge \ Wait = \text{TRUE} \tag{4.3}$$

- An *invariant law* is an expression of the form

$$\textbf{always_t } F(X) \textbf{ if } Mode = v \tag{4.4}$$

where $F(X)$ is a fluent formula of signature $\sigma^{diff} \cup \sigma^{bg}$. This can be seen using the two ball example

$$\textbf{always_t } Height(b) \geq 0 \textbf{ if } Mode = 1$$

[1]Listing complete ODEs could be viewed as a strong condition. However, even in the state-of-the-art system dReal, the integral construct only accepts complete ODE systems. It does not yet support a parallel composition, where each mode of a single automaton corresponds to a partial ODE system. Also note that flow conditions cannot be decomposed in general, since variables in ODEs evolve simultaneously over continuous time. For this reason, existing SMT techniques use the standard non-compositional encoding for networked hybrid automata.

Each invariant law (4.4) in an action description is expanded into

$$\textbf{constraint } \forall t \forall x \Bigg((0 \leq t \leq \delta) \wedge \\ \big(x = (x_1, \ldots, x_n) + \int_0^t (d/dt[X_1](v), \ldots, d/dt[X_n](v))dt \rightarrow F(x)\big) \Bigg) \\ \textbf{after } (X_1, \ldots, X_n) = (x_1, \ldots, x_n) \wedge Mode = v \wedge Dur = \delta \wedge Wait = \text{TRUE}. \tag{4.5}$$

4.2 Corresponding Representation of New Causal Laws in ASPMT

We slightly extend the ASPMT signature $i : \sigma$ such that it is the signature consisting of the pairs $i : c$ such that $c \in \sigma \ \wedge \ c \notin \sigma^{diff}$, and the value sort of $i : c$ is the same as the value sort of c. If $c \in X$ then $i : c$ consists of $0 : X_0$ for $i = 0$ and $i - 1 : c_t$ for $i \geq 1$, and the value sort of $0 : c_0$ and $i - 1 : c_t$ is the same as the value sort of c. Similarly, if s is an interpretation of σ, expression $i : s$ is an interpretation of $i : \sigma$ such that $c^s = (i : c)^{i:s}$.

We also extend $i : F$ such that $i : F$ describes a formula F by:

- Replacing every occurrence of c with $0 : c_0$ if $c \in X$ and $i = 0$
- Replacing every occurrence of c with $i - 1 : c_t$ if $c \in X$ and $i \geq 1$
- Replacing every occurrence of c with $i : c$ if $c \in \sigma$ and $c \notin X$

for every $i \in \{0, \ldots, m-1\}$. Each rate declaration law (4.1) is represented in ASPMT as

$$d/dt[X_i] = F_i(X) \ \leftarrow \ Mode = v. \tag{4.6}$$

We use the earlier example to demonstrate this:

$$d/dt[Height(B_1)] = Velocity(B_1) \ \leftarrow \ Mode = 1.$$

Each expanded rate declaration law (4.2) is represented in ASPMT as:

$$
\begin{aligned}
&Case\ i=0:\ (0:X_{1t},0:X_{2t},...,0:X_{nt}) = (x_1+y_1,\ldots,x_n+y_n) \leftarrow \\
&\qquad (y_1,\ldots,y_n) = \int_0^d \theta_v dt\ \wedge\ 0:Mode=v\ \wedge\ 0:Dur=d\ \wedge\ 0:Wait\ \wedge \\
&\qquad 0:X_{10}=x_1\ \wedge\ 0:X_{20}=x_2\ \wedge\ ...\ 0:X_{n0}=x_n.,\\
&Case\ i\geq 1:\ (i:X_{1t},i:X_{2t},...,i:X_{nt}) = (x_1+y_1,\ldots,x_n+y_n) \leftarrow \\
&\qquad (y_1,\ldots,y_n) = \int_0^d \theta_v dt\ \wedge\ i:Mode=v\ \wedge\ i:Dur=d\ \wedge\ i:Wait\ \wedge \\
&\qquad i-1:X_{1t}=x_1\ \wedge\ i-1:X_{2t}=x_2\ \wedge\ ...\ i-1:X_{nt}=x_n..
\end{aligned}
\tag{4.7}
$$

for every $i \in \{0,\ldots,m-1\}$. We use the earlier example to demonstrate this:

$$
\begin{aligned}
&Case\ i=0:\ (0:H_{1t},0:V_{1t},0:H_{2t},0:V_{2t}) = (x_1+y_1,\ldots,x_4+y_4) \leftarrow \\
&\qquad (y_1,\ldots,y_4) = \int_0^d \theta_1 dt\ \wedge\ 0:Mode=1\ \wedge\ 0:Dur=d\ \wedge\ 0:Wait\ \wedge \\
&\qquad 0:H_{10}=x_1\ \wedge\ 0:V_{10}=x_2\ \wedge\ 0:H_{20}=x_3\ 0:V_{20}=x_4.,\\
&Case\ i\geq 1:\ (i:H_{1t},i:V_{1t},i:H_{2t},i:V_{2t}) = (x_1+y_1,\ldots,x_4+y_4) \leftarrow \\
&\qquad (y_1,\ldots,y_n) = \int_0^d \theta_v dt\ \wedge\ i:Mode=v\ \wedge\ i:Dur=d\ \wedge\ i:Wait\ \wedge \\
&\qquad i-1:H_{10}=x_1\ \wedge\ i-1:V_{10}=x_2\ \wedge\ i-1:H_{20}=x_3\ i-1:V_{20}=x_4.
\end{aligned}
\tag{4.8}
$$

Each expanded invariant law (4.5) is represented in ASPMT as:

$$\begin{aligned}\leftarrow\ \neg\forall t\forall x\Bigg(&(0\le t\le\delta)\wedge\\ &\big(x=(x_1,\dots,x_n)+\int_0^t(d/dt[X_1](v),\dots,d/dt[X_n](v))dt\rightarrow F(x)\big)\Bigg)\\ &\wedge\ (i-1:X_1,\dots,i-1:X_n)=(x_1,\dots,x_n)\wedge i-1:\mathit{Mode}=v\\ &\wedge i-1:\mathit{Dur}=\delta\wedge i-1:\mathit{Wait}=\textsc{true}. \qquad (4.9)\end{aligned}$$

for every $i\in\{1,\dots,m-1\}$.

We use the earlier example to demonstrate this:

$$\begin{aligned}\leftarrow\ \neg\forall t\forall x\Bigg(&(0\le t\le\delta)\wedge\\ &\big(x=(x_1,\dots,x_4)+\int_0^t(d/dt[H_1](1),\dots,d/dt[V_2](1))dt\rightarrow(x_1,x_3)\ge 0\big)\Bigg)\\ &\wedge\ (i-1:H_1,\dots,i-1:V_2)=(x_1,\dots,x_4)\wedge i-1:\mathit{Mode}=1\\ &\wedge i-1:\mathit{Dur}=\delta\wedge i-1:\mathit{Wait}=\textsc{true}. \qquad (4.10)\end{aligned}$$

Chapter 5

REPRESENTING HYBRID TRANSITION SYSTEMS IN $\mathcal{C}+$ MODULO ODE

In this chapter, we revisit the encoding of hybrid automata in language $\mathcal{C}+$. This time, we represent it in the language of $\mathcal{C}+$ modulo ODE using the new causal laws introduced in the previous chapter.

5.1 Representation

The signature is the same as before. Implicit rules and discrete transition laws are the same as before.

The translation consists of the same rules as the one in Chapter 3 except for the rules that account for continuous transitions. Each variable in hybrid automata is identified with a differential fluent constant. The flow and invariant conditions are modified as follows.

- Continuous Transition:
  - **Flow**: We assume that flow conditions are written as a set of $\dot{X}_i = F_i(X)$ for each X_i in σ^{diff} where F_i is a formula whose free variables are from X only. We assume there is only one such formula for each X_i.

    For each $v \in V$ and $X_i \in X$, D_H includes a rate declaration

    $$\textbf{derivative of } X_i \textbf{ is } F_i(X) \textbf{ if } \mathit{Mode} = v$$

    which describes the flow of each differential fluent constant X_i for all possible values of Mode.

– **Invariant**: For each $v \in V$, D_H includes an invariant law

$$\textbf{constraint } \mathsf{Inv}_v(X) \textbf{ if } Mode = v$$

$$\textbf{always_t } \mathsf{Inv}_v(X) \textbf{ if } Mode = v$$

The new **always_t** law ensures the invariant is true even during the transition.

Note that we do not need to extend language $\mathcal{C}+$ for this purpose except it refers to the new background theory QF_NRA_ODE.

Theorem 3 and Lemmas 1, 2 still remain true when we extend non-linear hybrid automata assumed in the beginning of this section if we consider this version of D_H instead of the previous one. The extended theorem statements are as follows

Theorem 5 *There is a 1:1 correspondence between the paths of the transition system of a Hybrid automata H and the paths of the transition system of the action description D_H as obtained using the translation described in this section.*

The proof is immediate from the following two lemmas. By a path in a transition system, we mean the sequence of edges. [1] First, we show that every path in the labelled transition system of T_H is a path in the transition system described by D_H.

Notation: We say that an interpretation I of σ satisfies F w.r.t. the background theory bg, denoted by $I \models_{bg} F$, if $I \cup J^{bg}$ satisfies F.

Lemma 11 *For any path*

$$p = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{m-1}} (v_m, x_m)$$

[1] The start node of a path may not necessarily satisfy the initial conditions, but it is easy to check. Dropping this requirement simplifies the statement.

in the labelled transition system of H*, let*

$$p' = \langle s_0, a_0, s_1, a_1, \ldots, s_m \rangle,$$

where each s_i *is an interpretation of fluent constants and each* a_i *is an interpretation of action constants such that, for* $i = 0, \ldots m{-}1$*,*

- $s_0 \models_{bg} (mode, X) = (v_0, x_0)$*;*
- $s_{i+1} \models_{bg} (mode, X) = (v_{i+1}, x_{i+1})$ *(*$i = 0, \ldots, m{-}1$*), where*
  - *if* $\sigma_i = \mathsf{hevent}(v_i, v_{i+1})$*, then* $a_i \models_{bg} Dur{=}0$ *and* $a_i \models_{bg} wait{=}\text{FALSE}$*;*
  - *if* $\sigma_i \in \mathcal{R}_{\geq 0}$*, then* $a_i \models_{bg} Dur{=}\sigma_i$ *and* $a_i \models_{bg} wait{=}\text{TRUE}$*;*

Then, p' *is a path in the transition system* D_H*.*

Next, we show that every path in the transition system of D_H is a path in the labelled transition system of H.

Lemma 12 *For any path*

$$q = (s_0, a_0, s_1, a_1, \ldots, s_m)$$

in the transition system of D_H*, let*

$$q' = (v_0, x_0) \xrightarrow{\sigma_0} (v_1, x_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{m-1}} (v_m, x_m),$$

where each $v_i \in V$ *and each* $x_i \in R^n$ *for* $i = 0, \ldots, m$ *such that*

- $s_i \models_{bg} (Mode, X) = (v_i, x_i)$*;*
- σ_i *is*
  - $\mathsf{hevent}(v_i, v_{i+1})$ *if* $a_i \models_{bg} \mathsf{hevent}(v_i, v_{i+1})$*;*
  - $a_i(Dur)$ *otherwise.*

Then, q' *is a path in the transition system of* T_H*.*

5.2 Example

Example 3 *We take the case where there is a car of length 1 unit moving at a constant speed of 1 unit. The car is initially at origin where* $x = 0$ *and* $y = 0$ *and* $\theta = 0$*. Additionally there are walls defined by the equations* $(x-6)^2 + y^2 = 9$,$(x-5)^2 + (y-7)^2 = 4$,$(x-12)^2 + (y-9)^2 = 4$*. The goal is to find a plan such that the car ends up at* $x = 13$ *and* $y = 0$ *without hitting the walls. The dynamics of the car is as follows:*

Moving Straight

$$\frac{d[x]}{dt} = \cos(\theta),\ \frac{d[y]}{dt} = \sin(\theta),\ \frac{d[theta]}{dt} = 0$$

Turning Left

$$\frac{d[x]}{dt} = \cos(\theta),\ \frac{d[y]}{dt} = \sin(\theta),\ \frac{d[theta]}{dt} = \tan(\frac{\pi}{18})$$

Turning Right

$$\frac{d[x]}{dt} = \cos(\theta),\ \frac{d[y]}{dt} = \sin(\theta),\ \frac{d[theta]}{dt} = \tan(-\frac{\pi}{18})$$

The invariant in the other example were just simple logical formulas. For this particular example we wish to show that the car must not hit the obstacle. For that to be true, the car must be outside the round wall. Hence the invariant for any state in this example would be the logic formulas $(x-6)^2 + y^2 > 9$,$(x-5)^2 + (y-7)^2 > 4$,$(x-12)^2 + (y-9)^2 > 4$.

The Hybrid Automata description as well as the corresponding representation is as follows.

Hybrid Automata Components

The control graph for Example 3 can be seen in Figure 5.1.

- Variables:

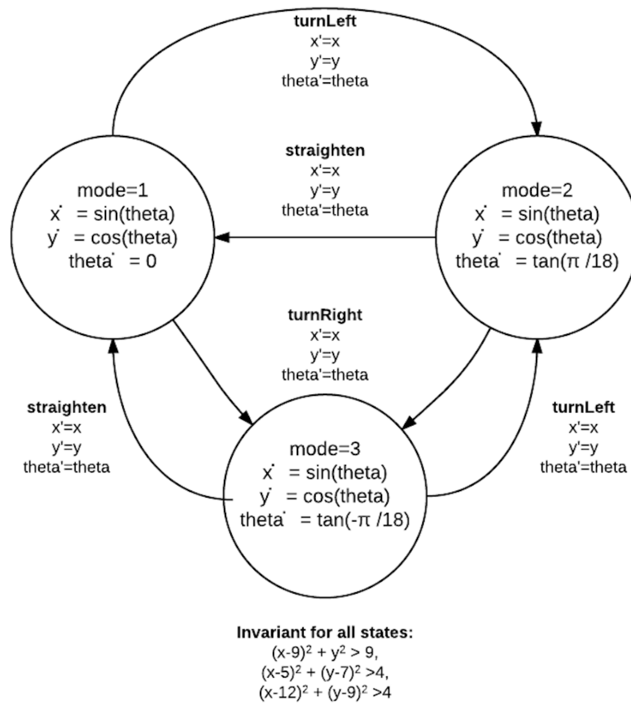

Figure 5.1: Hybrid Automata of Car example

  - $x, x', \dot{x}$
  - $y, y', \dot{y}$
  - $theta, theta', \dot{theta}$

- States:
  - moveStraight($mode = 1$)
  - moveLeft($mode = 2$)
  - moveRight($mode = 3$)

- Directed Graph: The graph is given above

- H-Events:
  - straighten

- turnLeft
- turnRight

- Invariants:
  - $inv(allmodes) : ((x-6)^2 + y^2 > 9) \wedge \ ((x-5)^2 + (y-7)^2 > 4) \wedge \ ((x-12)^2 + (y-9)^2 > 4))$

- Flow:
  - $flow(1)(x, y, theta) : \dot{x} = sin(theta) \ \wedge \ \dot{y} = cos(theta) \wedge \ \dot{theta} = 0.$
  - $flow(2)(x, y, theta) : \dot{x} = sin(theta) \ \wedge \ \dot{y} = cos(theta) \wedge \ \dot{theta} = tan(\pi/18).$
  - $flow(3)(x, y, theta) : \dot{x} = sin(theta) \ \wedge \ \dot{y} = cos(theta) \wedge \ \dot{theta} = tan(-\pi/18).$

- Jump: We club the edges with same hevent together as they have the same effect.
  - $Reset((\{3,1\},\{2,1\})(x, y, theta)) \ : \ x' = x \ \wedge \ y' = y \ \wedge \ theta' = theta.$
  - $Reset((\{1,2\},\{3,2\})(x, y, theta)) \ : \ x' = x \ \wedge \ y' = y \ \wedge \ theta' = theta.$
  - $Reset((\{1,3\},\{2,3\})(x, y, theta)) \ : x' = x \ \wedge \ y' = y \ \wedge \ theta' = theta.$

C+ Action Description

$q \in \{Q_1, Q_2\}$; t is a variable of sort $\mathcal{R}_{\geq 0}$. x, y, $theta$ are variables of sort $\mathcal{R}$.

Simple fluent constants:	Sort:
$X, Y, Theta$	$\mathcal{R}_{\geq 0}$
$Mode$	$\{Q_1, Q_2, Q_3\}$

Action constants:	Sort:
$TurnLeft$, $TurnRight$, $Straighten$ $Wait$	Boolean
Dur	$\mathcal{R}_{\geq 0}$

% Exogenous constants:

exogenous $X, Y, Theta, TurnLeft, TurnRight, Straighten, Dur$

% Reset:

constraint $(X, Y, Theta) = (x, y, theta)$ **after**

$$(X, Y, Theta) = (x, y, theta) \ \wedge straighten$$

constraint $(X, Y, Theta) = (x, y, theta)$ **after**

$$(X, Y, Theta) = (x, y, theta) \ \wedge turnLeft$$

constraint $(X, Y, Theta) = (x, y, theta)$ **after**

$$(X, Y, Theta) = (x, y, theta) \ \wedge turnRight$$

% Mode:

nonexecutable $Straighten$ **if** $(Mode = Q_1)$

nonexecutable $TurnLeft$ **if** $(Mode = Q_2)$

nonexecutable $TurnRight$ **if** $(Mode = Q_3)$

$TurnLeft$ **causes** $Mode = Q_2$

$TurnRight$ **causes** $Mode = Q_3$

$Straighten$ **causes** $Mode = Q_1$

inertial $Mode = q \quad (q \in \{Q_1, Q_2, Q_3\}$

% Duration:

$TurnRight$ **causes** $Dur = 0$

$TurnLeft$ **causes** $Dur = 0$

$Straighten$ **causes** $Dur = 0$

% Wait:

default $Wait = \text{TRUE}$

$TurnLeft$ **causes** $Wait = \text{FALSE}$

$Straighten$ **causes** $Wait = \text{FALSE}$

$TurnRight$ **causes** $Wait = \text{FALSE}$

% Rate Declarations:

derivative of x **is** $\sin(theta)$ **if** $Mode = Q_1$.

derivative of y **is** $\cos(theta)$ **if** $Mode = Q_1$.

derivative of $theta$ **is** 0 **if** $Mode = Q_1$.

derivative of x **is** $\sin(theta)$ **if** $Mode = Q_2$.

derivative of y **is** $\cos(theta)$ **if** $Mode = Q_2$.

derivative of $theta$ **is** $\tan(\pi/18)$ **if** $Mode = Q_2$.

derivative of x **is** $\sin(theta)$ **if** $Mode = Q_3$.

derivative of y **is** $\cos(theta)$ **if** $Mode = Q_3$.

derivative of $theta$ **is** $\tan(-\pi/18)$ **if** $Mode = Q_3$.

% Invariant

constraint $(X-6)*(X-6)+(Y)*(Y) > 9$.

always_t $(X-6)*(X-6)+(Y)*(Y) > 9)$ **if** $Mode = q$. $\quad (q \in \{Q_1, Q_2, Q_3\})$

constraint $(X-5)*(X-5)+(Y-7)*(Y-7) > 4$.

always_t $(X-5)*(X-5)\&(Y-7)*(Y-7)>4$ **if** $Mode = q$.

$$(q \in \{Q_1, Q_2, Q_3\})$$

constraint $(X-12)*(X-12)+(Y-9)*(Y-9)>4$.

always_t $(X-12)*(X-12)+(Y-9)*(Y-9)>4$ **if** $Mode = q$.

$$(q \in \{Q_1, Q_2, Q_3\})$$

5.3 Turning in the Input Language of dReal

System dReal (Gao *et al.* (2013b)) is an SMT solver to check the satisfiability of logic formulas over the real numbers up to a given precision $\delta > 0$ using δ-complete decision procedures, involving various non-linear real functions, such as polynomials, exponentiation, trigonometric functions, and solutions of Lipschitz-continuous ordinary differential equations (ODEs).

The input language of dReal follows Version 2 of SMT-LIB standard, but its ODE extension is not standard. In the language, `t`-variables (variables ending with `_t`) have a special meaning. $c_i_\texttt{t}$ is a t-variable between timepoint i and $i+1$ that progresses in accordance with ODE specified by some flow condition, and is universally quantified to assert that their values during each transition satisfies the invariant condition for that transition (c.f. (4.5)).

For ASPMT formula F that is generated above, By $dr(F)$ we describe a formula F by:

- Replacing every occurrence of $i\!:\!c_t$ in F with $c_i_\texttt{t}$.
- Replacing every occurrence of $i\!:\!c$ in F with c_i.

for every $i \in \{0, \ldots, m-1\}$.

The set θ_v of rate declaration law (4.1) can be expressed in dReal as

$$(\texttt{define-ode flow_}v\ ((\texttt{= d/dt}[X_1]\ F_1), \ldots, (\texttt{= d/dt}[X_n]\ F_n)))$$

An example of this can be showing using earlier example of two balls

$$(\texttt{define-ode flow_1}\ ((\texttt{= d/dt}[H_1]\ V_1), \ldots, (\texttt{= d/dt}[V_2]\ -g)))$$

From (5), we know that the $(d/dt[X_1](v), \ldots, d/dt[X_n](v))$ is obtained from list $\mathsf{Flow_v}$. We also know that every θ_v describes ODE system $flow_v$. The *integral* construct in dReal solves $\texttt{flow_}v$ for each interval $[0, \delta]$ by solving

$$\int_0^{\delta} (d/dt[X_1](v), \ldots, d/dt[X_n](v))\ dt \tag{5.1}$$

and computing the values of $dr(i+1 : X_1), \ldots dr(i+1 : X_n)$ given the initial values $dr(i : X_1), \ldots dr(i : X_n)$.

In the language of dREAL, the integral construct explained above is represented as

$$(\texttt{integral}\ (\texttt{0.}\ \delta\ [X_1, \ldots, X_n]\ \texttt{flow_}v))$$

Using this integral construct, every rule (4.2) can be expressed in dReal as

- if $i = 0$,

$$(\texttt{assert}\ (\texttt{=>}\ (\texttt{and}\ ((\texttt{= mode_0}\ v)\ (\texttt{= wait_0 true})))$$
$$(\texttt{=}\ [X_1\texttt{_0_t}, \ldots, X_n\texttt{_0_t}]$$
$$(\texttt{integral}\ (\texttt{0. duration_0}\ [X_1\texttt{_0}, \ldots, X_n\texttt{_0}]\ \mathsf{Flow_v}))))$$

- if $i > 1$,

$$(\texttt{assert}\ (\texttt{=>}\ (\texttt{and}\ ((\texttt{= mode_}i\ v)\ (\texttt{= wait_}i\ \texttt{true})))$$
$$(\texttt{=}\ [X_1\texttt{_}i\texttt{_t}, \ldots, X_n\texttt{_}i\texttt{_t}]$$
$$(\texttt{integral}\ (\texttt{0. duration_}i\ [X_1\texttt{_}(i-1)\texttt{_t}, \ldots, X_n\texttt{_}(i-1)\texttt{_t}]\ \mathsf{Flow_v}))))$$

This can be demonstrated using the two ball example as follows:

$$\texttt{(assert (=> (and ((= mode_}i\texttt{ 1) (= wait_}i\texttt{ true)))}$$
$$\texttt{(= }[H_1_i_\texttt{t}, \ldots, V_2_i_\texttt{t}]$$
$$\texttt{(integral (0. duration_}i\ [H_1_(i-1)_\texttt{t}, \ldots, V_2_(i-1)_\texttt{t}]\ \mathsf{Flow_1}))))$$

Every invariant law (4.4) describes a formula with the bounded quantifier $\forall^{[0,t]}$ explained earlier for any real value t. This quantifier is succinctly expressed in dReal using the forall_t construct. ASPMT rule (4.5) can be abbreviated in dReal as

$$\texttt{(assert (forall_t } v\ \texttt{[0 duration_}i\texttt{]}\ dr(i : F)\texttt{))}$$

This can be demonstrated using the two ball example as follows:

$$\texttt{(assert (forall_t } 1\ \texttt{[0 duration_}i\texttt{]}\ (\geq Height_B_1\ 0)))$$

5.4 Proofs

5.4.1 Proof of Lemma 11

Lemma 11

p' is a path in the transition system D_H.

Lemma 13 *Let H be a hybrid automaton and*

$$(v, x) \xrightarrow{\sigma} (v, x')$$

be a transition in T_H such that $\sigma \in \mathcal{R}_{\geq 0}$. $f(t) = x + \int_0^{\sigma_i} \dot{f}(t)\, dt$ is a differentiable function from $[0, \sigma]$ to $\mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma] \to \mathcal{R}^n$ such that: (1) $f(0) = x$ and $f(\sigma) = x'$ and (2) for all reals $\epsilon \in (0, \sigma)$, both $\mathsf{nv}_v(f(\epsilon))$ and $\mathsf{Flow}_v(f(\epsilon), \dot{f}(\epsilon))$ are true.

Proof. We know that $\dot{f}$ is a Lipschitz continuous function. From the Picard-Lindelof theorem, we know that there is a unique solution f' with the derivative $\dot{f}$. From the definition of f, it is also clear that $\dot{f}$ is the derivative of f. Consequently $f = f'$.

We know that $(v, x) \xrightarrow{\sigma} (v, x')$ is a transition in T_H, hence there must exist a function with the derivative $\dot{f}$ that satisfies all the conditions. We also know that f is the unique function with the derivative $\dot{f}$. Hence f satisfies all the conditions.

Lemma 14 *For each $i \geq 0$, s_i is a state in the transition system of D_H.*

Proof. By definition we are to show that

$$0 : s_i \models_{bg} SM[(D_H)_0; \phi],$$

while $SM[(D_H)_0; \phi]$ is equivalent to the conjunctions of,

$$\leftarrow \neg(\mathsf{Inv}_v(0 : X)) \ \wedge \ 0 : Mode = v. \tag{5.2}$$

for each $v \in V$. Since p is a path,for each $i \geq 0$,(v_i, x_i) is a state in T_H. By the definition of hybrid transition systems, $\mathsf{Inv}_{v_i}(X_i)$ is true. Hence $0 : s_i \models_{bg}$ (5.2) ∎

Lemma 15 *For each $i \geq 0$, $\langle s_i, a_i, s_{i+1}\rangle$ is a transition.*

Proof. By definition, we are to show that

$$0{:}s_i \cup 0{:}a_i \cup 1{:}s_{i+1} \models_{bg} \mathrm{SM}[(D_H)_1; 0{:}\sigma^{act} \cup 1{:}(\sigma^{fl} - \sigma^{diff})]. \tag{5.3}$$

From (Ferraris *et al.* (2011)) it is clear that for any formula F SM$[F]$ is equivalent to *Comp*$[F]$. Additionally from (Ferraris *et al.* (2011)) it is also clear that SM$[F] \ \wedge \ Choice(\mathbf{c})$ is equivalent to SM$[F]$ where $Choice(\mathbf{c})$ represents the choice

rules for all non-intensional constants **c**.

$Comp[(D_H)_1; 0:\sigma^{act} \cup 1:\sigma^{fl}]$ is equivalent to the conjunction of following formulas:

- Formula $FLOW$, which is the rule

$$(1:X_1, 1:X_2, ..., 1:X_n) = (x_1+y_1, \ldots, x_n+y_n) \leftarrow$$
$$(y_1,\ldots,y_n) = \int_0^{\delta} \theta_v dt \ \wedge\ 0:Mode = v \ \wedge\ 0:Dur = \delta$$
$$\wedge\ 0:Wait = \text{TRUE} \ \wedge\ 0:X_1 = x_1 \ \wedge\ 0:X_2 = x_2 \ \wedge\ ...\ 0:X_n = x_n. \quad (5.4)$$

  for each $v \in V$ where $0:X_1, 0:X_2, ..., 0:X_n \in 0:X$ and $1:X_1, 1:X_2, ..., 1:X_n \in 1:X$ for each $v \in V$ and θ_v is the set of $F_i(X)$ for each $X_i \in X$;

- Formula INV, which is the rule

$$\forall t, \mathbf{F}(0 \le t \ \wedge\ t \le \delta \ \wedge\ \mathbf{F} = [0:X_1, 0:X_2, ..., 0:X_n] +$$
$$\int_0^t \theta_v\ dt \ \wedge\ 0:Mode = v \wedge\ 0:Dur = \delta \wedge 0:\ Wait = true \rightarrow \mathsf{Inv}_v(\mathbf{F})) \quad (5.5)$$

  for each $v \in V$ where $0:X_1, 0:X_2, ..., 0:X_n \in 0:X$ and θ_v is the set of $F_i(X)$ for each $X_i \in X$

- Formula $GUARD$, which is the conjunction of

$$\leftarrow 0:\mathsf{hevent}(e) \ \wedge\ 0:\neg\mathsf{Guard}_e(X) \quad (5.6)$$

  for each edge $e \in E$;

- Formula $RESET$, which is the conjunction of

$$\leftarrow \neg\mathsf{Reset}_e(x_0, x_1) \ \wedge\ 1:X = x_1 \ \wedge\ 0:\mathsf{hevent}(e) \ \wedge\ 0:X = x_0 \ \wedge\ 0:Mode = v.$$

  for each edge $e \in E$;

- Formula *MODE*, which is the conjunction of

$$\leftarrow 0 : \mathsf{hevent}(e) \ \wedge \ \neg(0 : Mode = v_1).$$

$$1 : Mode = v_2 \ \leftrightarrow \ (0 : Mode = v_1 \wedge 0 : \mathsf{hevent}(e)) \ \vee \ (1 : Mode = v_2 \wedge 0 : Mode = v_2).$$

  for each edge $e = (v_1, v_2) \in E$;

- Formula *DURATION*, which is the rule

$$0 : Dur = 0 \ \leftarrow \ \bigvee_{e \in E} 0 : \mathsf{hevent}(e).$$

  for each edge $e = (v_1, v_2) \in E$;

- Formula *WAIT*, which is the rule

$$0 : Wait = false \ \leftrightarrow \ \bigvee_{e \in E} 0 : \mathsf{hevent}(e).$$

$$0 : Wait = true \ \leftrightarrow \ \neg\neg 0 : Wait = true.$$

We will show that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1}$ satisfies *FLOW*, *GUARD*, *RESET*, *MODE*, *DURATION*, *WAIT*. From the definition of T_H there are 2 cases for the value of σ:

Case 1: $\sigma = \mathsf{hevent}(e)$ where $e = (v_i, v_{i+1})$. It follows from the construction of p' that $(Dur)^{a_i} = 0$, $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$ and $(Wait)^{a_i} = \text{FALSE}$. Since *FLOW* is trivially satisfied, it is sufficient to consider only *GUARD*, *RESET*, *MODE*, *DURATION*, *WAIT*.

From the fact that:

$$(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$$

is a transition in T_H and that $\sigma_i = \mathsf{hevent}(e)$, it follows from the definition of hybrid transition systems that $\mathsf{Guard}_e(0 : X)$ and $\mathsf{Reset}_e(0 : X, 1 : X)$ is true.

- *GUARD*: It is immediate that $0 : s_i \ \cup \ 1 : s_{i+1} \models_{bg} \mathsf{Guard}_e(0 : X)$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1} \models_{bg} GUARD$

- *RESET*: Note that $s_i \models_{bg} (Mode, X) = (v_i, x_i)$ and $s_{i+1} \models_{bg} (Mode, X) = (v_{i+1}, x_{i+1})$. It is immediate that $0 : s_i \ \cup \ 1 : s_{i+1} \models_{bg} \mathsf{Reset}_e(0 : X, 1 : X), 0 : s_i \models_{bg} 0 : Mode = v_i$, $0 : s_i \models_{bg} 0 : X = x_i$ and $0 : s_i \models_{bg} 1 : X = x_{i+1}$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1} \models_{bg} RESET$

- *MODE*: Note that $s_i \models_{bg} (Mode, X) = (v_i, x_i)$ and $s_i \models_{bg} (Mode, X) = (v_{i+1}, x_{i+1})$. It is immediate that $0 : s_i \models_{bg} 0 : Mode = v_i$ and $1 : s_{i+1} \models_{bg} 1 : Mode = v_{i+1}$. Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1} \models_{bg} MODE$.

- *DURATION*: Since $(Dur)^{a_i} = 0$ and $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, it follows that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1} \models_{bg} DURATION$

- *WAIT*: Since $(\mathsf{hevent}(e))^{a_i} = \text{TRUE}$, and $(Wait)^{a_i} = \text{FALSE}$ it follows that $0 : s_i \ \cup \ 0 : a_i \ \cup \ 1 : s_{i+1} \models_{bg} WAIT$

Case 2: $\sigma_i \in \mathcal{R}_{\geq 0}$. By the construction of p', $Dur^{a_i} = \sigma_i$, $Wait^{a_i} = \text{TRUE}$ and $(\mathsf{hevent}(e))^{a_i} = \text{FALSE}$ for every $e = (v, v') \in E$. Since *GUARD*, *RESET*, *MODE*, *DURATION*, *WAIT* is trivially satisfied, it is sufficient to consider only *FLOW* and *INV*.

From the fact that

$$(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$$

is a transition of T_H and that $\sigma_i \in \mathcal{R}_{\geq 0}$, it follows from the definition of hybrid transition systems that

(a) $v_i = v_{i+1}$, and

(b) there is a differentiable function $f : [0, \sigma_i] \rightarrow \mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma_i] \rightarrow \mathcal{R}^n$ such that: (1) $f(0) = x_i$ and $f(\sigma_i) = x_{i+1}$ and (2) for all reals $\epsilon \in (0, \sigma_i)$, both $\mathsf{Inv}_{\mathsf{v_i}}(\mathsf{f}(\epsilon))$ and $\mathsf{Flow}_{\mathsf{v_i}}(\dot{\mathsf{f}}(\epsilon))$ are true.

By Lemma 13, $f(t) = x_i + \int_0^{\sigma_i} \dot{f}(t)\ dt$ is a differentiable function that satisfies all the conditions in (b).

We check the following:

- *FLOW*: If $\sigma_i \geq 0$, then $Dur^{e^i} \geq 0$. We know that $f(0) = x_i$ and $f(\sigma_i) = x_{i+1} = x + \int_0^{\sigma_i} \dot{f}(t)\ dt$. Since $X^{s_i} = x_i$, $X^{s_{i+1}} = x_{i+1}$ and $\dot{f}$ is basically θ_v, it is clear that $0 : s_i\ \cup\ 0 : a_i\ \cup\ 1 : s_{i+1} \models_{bg} FLOW$.

- *INV*: Since we know that $Dur^{e^i} \geq 0$, $X^{s_i} = x_i$, $f(\epsilon) = x_i + \int_0^{\epsilon} \dot{f}(\epsilon)\ dt$ and $\mathsf{Inv}_{v_i}(f(\epsilon))$ is true for all reals $\epsilon \in (0, \sigma_i)$ then it is clear that $0 : s_i \cup 0 : a_i \cup 1 : s_{i+1} \models_{bg} INV$

∎

Using the lemmas defined and proved above, we prove Lemma 11 as follows:

Proof. By Lemma 14, each s_i is a state of D_H. By Lemma 15, each $\langle s_i, a_i, s_{i+1} \rangle$ is a transition of D_H. So p' is a path in the transition system of D_H. ∎

5.4.2 Proof of Lemma 12

Lemma 12

q' is a path in the transition system of T_H.

Lemma 16

(a) For each $i \geq 0$, (v_i, x_i) is a state in T_H, and

(b) (v_0, x_0) is an initial state in T_H.

Proof.

(a) By definition, we are to show that $\mathsf{Inv}_{v_i}(x_i)$ is true. Since each s_i is a state in the transition system of D_H, by definition,

$$0\!:\!s_i \models_{bg} \text{SM}[(D_H)_0;\emptyset]. \tag{5.7}$$

Note that $\text{SM}[(D_H)_0;\emptyset]$ is equivalent to the conjunction of the formula:

$$0\!:\!\mathsf{Inv}_v(x) \leftarrow 0\!:\!Mode = v \tag{5.8}$$

for each location $v \in V$. Since $(Mode)^{s_i} = v_i$, it follows that $s_i \models_{bg} \mathsf{Inv}_{v_i}(x)$. Since $x^{s_i} = x_i$, it follows that $\mathsf{Inv}_{v_i}(x_i)$ is true.

(b) We are to show that $Init_{v_0}(x_0)$ is true. Since we express $Init_{v_0}(x_0)$ as is in D_H. It is trivially entailed. ∎

Lemma 17 *For each $i \geq 0$, $(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$ is a transition in T_H.*

Proof. From the fact that (s_i, a_i, s_{i+1}) is a transition of D_H, by definition we know that

$$0\!:\!s_i \cup 0\!:\!a_i \cup 1\!:\!s_{i+1} \models_{bg} \text{SM}[(D_H)_1; 0\!:\!\sigma^{act} \cup 1\!:\!\sigma^{fl}]. \tag{5.9}$$

$Comp[(D_H)_1; 0 : \sigma^{act} \cup 1 : \sigma^{fl}]$ is equivalent to the conjunction of $FLOW$, $RESET$, $GUARD$, $DURATION$, $MODE$, $WAIT$.

Consider two cases:

Case 1: There exists an edge $e = (v, v')$ such that $(\mathsf{hevent}(e))^{a_i} = t$.. Since $Mode^{s_i} = v_i$ and $Mode^{s_{i+1}} = v_{i+1}$, it follows that (v, v') must be (v_i, v_{i+1}).As a result, $(\mathsf{hevent}(e))^{a_i} = t$. It follows from the definition that $\sigma_i = \mathsf{hevent}(e)$.

- Since $0 : s_i \cup 0 : a_i \cup 1 : s_{i+1} \models_{bg} \mathit{GUARD}$, $X^{s_{i+1}} = x_{i+1}$ and $X^{s_i} = x_i$, it is immediate that $Guard_e(x_i)$ is true.

- Since $0 : s_i \cup 0 : a_i \cup 1 : s_{i+1} \models_{bg} \mathit{RESET}$, $X^{s_{i+1}} = x_{i+1}$ and $X^{s_i} = x_i$, it is immediate that $Reset_e(x_i, x_{i+1})$ is true.

(b) By Lemma 16, (v_i, x_i) and (v_{i+1}, x_{i+1}) are states.

Case 2: There exists an edge $e = (v, v')$ such that $(\mathsf{hevent}(e))^{a_i} = \text{FALSE}$ for every $(v, v') \in E$. By construction, $(Dur)^{a_i} = \sigma_i$ for some $\sigma_i \in \mathcal{R}_{\geq 0}$. By Lemma 16, (v_i, x_i) and (v_{i+1}, x_{i+1}) are states. From MODE, it follows that $Mode^{s_i} = Mode^{s_{i+1}}$. As a result, $v_i = v_{i+1}$. We are to show that there is a differentiable function $f : [0, \sigma_i] \rightarrow \mathcal{R}^n$, with the first derivative $\dot{f} : [0, \sigma_i] \rightarrow \mathcal{R}^n$ such that: (1) $f(0) = x_i$ and $f(\sigma_i) = x_{i+1}$ and (2) for all reals $\epsilon \in (0, \sigma_i)$, both $\mathsf{Inv}_{v_i}(f(\epsilon))$ and $\mathsf{Flow}_{\mathsf{v_i}}(\mathsf{f}(\epsilon), \dot{\mathsf{f}}(\epsilon))$ are true. Define $f(t) = x_i + \int_0^t \dot{f}(t)\, dt$. We check that f satisfies the above conditions:

- $f(t)$ is differentiable over $[0, \sigma_i]$.

- Since $i : s_i \cup (i+1) : s_{i+1} \models_{bg}$ (5.4.1), it is clear that f is the function being used and that $f(0){=}x_i$ and $f(\sigma){=}x_{i+1}$.

- We know that the θ_v comes directly from the definition of Hybrid Automata and is basically $\dot{f}$. $\mathsf{Flow}(X, \dot{X})$ is true when the set $\dot{X}_i = Formula_i(X)$ is true. To show that $\mathsf{Flow}_v(f(\epsilon), \dot{f}(\epsilon))$ is true for all $\epsilon \in [0, \sigma]$, we must show that $\dot{f}_i(\epsilon) = Formula_i(f(\epsilon))$ is true. Since $i : s_i \cup (i+1) : s_{i+1} \models_{bg}$ (5.4.1), $\mathsf{Flow}_v(f(\epsilon), \dot{f}(\epsilon))$ is true for all $\epsilon \in [0, \sigma]$.

- We check that for any $\epsilon \in (0, \sigma_i)$, $\mathsf{Inv}_{v_i}(f(\epsilon))$ is true. From $i : s_i \cup (i+1) : s_{i+1} \models_{bg}$ (5.4.1), it follows that $\mathsf{Inv}_{v_i}(f(\epsilon))$ is true.

From above, we conclude that (17) is a transition. ∎

Using the lemmas defined and proved above, we prove Lemma 12 as follows:

Proof. By Lemma 16 (a), each (v_i, x_i) is a state in T_H. By Lemma 17, each $(v_i, x_i) \xrightarrow{\sigma_i} (v_{i+1}, x_{i+1})$ is a transition in T_H. So p' is a path in T_H. If $s_0 \models_{bg} init_{v_0}$ then by Lemma 16 (b), (v_0, x_0) is a initial state in T_H. ∎

Chapter 6

IMPLEMENTATION

This chapter introduces the system CPLUS2ASPMT implemented for the purpose of the framework. We briefly go over the system ASPMT2SMT, on which our system is based. We also discuss the syntax and limitations of our system as well as some of the experimental results involved for the same.

6.1 ASPMT2SMT

System ASPMT2SMT (Bartholomew and Lee (2014)) is a prototype implementation of multi-valued propositional formulas under the stable model semantics computed by the SMT solver Z3. This reduction is based on the theorem on completion which describes how to capture the non-monotonic semantics of ASPMT in classical logic.

The implementation first compiles the ASPMT theory into a first-order formula without functions. System F2LP (Lee and Palla (2009)) is used to turn these first-order formulas into normal logic programs. GRINGO is a grounder that is then used to partially ground the logic program. The system then converts the logic program back into an ASPMT theory with functions that is now partially ground. Then, the system computes the completion of the partially ground ASPMT theory, eliminates any remaining variables resulting in a variable-free first order formula with function. Finally, Z3 computes the classical models of this first-order formula, which correspond to the stable models of the original ASPMT theory.

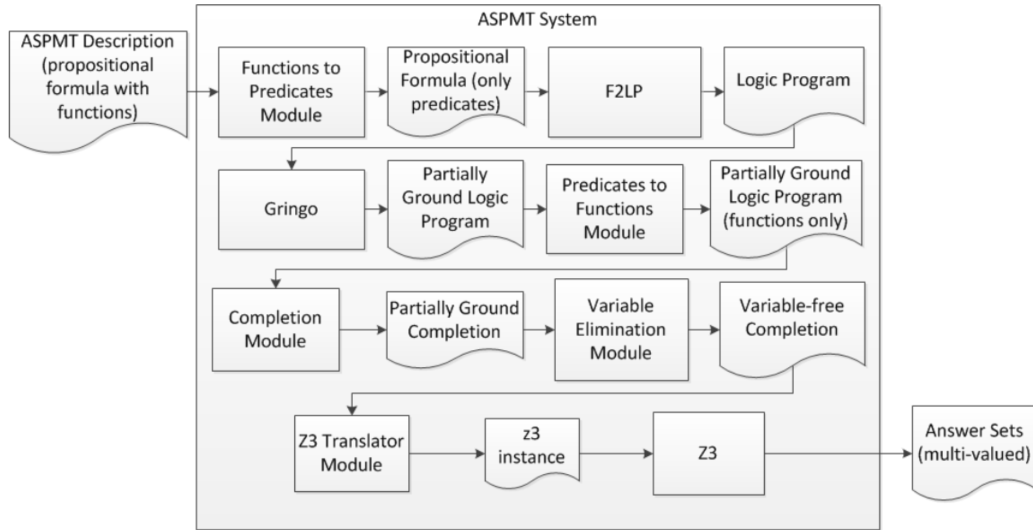

Figure 6.1: Architecture of System ASPMT2SMT as Shown In (Bartholomew and Lee (2014))

6.1.1 Architecture

The architecture of the system is shown in Figure 6.1 The ASPMT2SMT system first converts the ASPMT description to a propositional formula containing only predicates. In addition, this step substitutes auxiliary constants for value variables and necessary preprocessing for F2LP and GRINGO to enable partial grounding of argument variables only. F2LP transforms the propositional formula into a logic program and then GRINGO performs partial grounding on only the argument variables. The ASPMT2SMT system then converts the predicates back to functions and replaces the auxiliary constants with the original expressions. Then the system computes the completion of this partially ground logic program and performs variable elimination on that completion. Finally, the system converts this variable-free description into the language of Z3 and then relies on Z3 to produce models which correspond to stable models of the original ASPMT description.

6.1.2 Syntax Restrictions

System ASPMT2SMT imposes three syntactic restrictions on input ASPMT2SMT theories comprised of rules of the form $H \leftarrow B$ where B is a conjunction of possibly negated literals and H is $\bot$ or $f(\mathbf{t}) = v$: they must have the following properties. The following terms are introduced and defined in (Bartholomew and Lee (2014)).

Variable Isolated

Some SMT solvers do not support variables at all (e.g. iSAT) while others suffer in performance when handling variables (e.g. z3). While we can partially ground the input theories, some variables have large (or infinite) domains and should not (cannot) be grounded. Thus, we consider two types of variables; ASP *variables* - variables which should be grounded - and SMT *variables* - variables which should not be grounded. Eliminating ASP variables is simply done by grounding the original ASPMT theory. Then, we consider the problem of equivalently rewriting the completion of the partially ground ASPMT theory so that the result contains no variables. To ensure that variable elimination can be performed, we impose some syntactic restrictions on ASPMT instances. We first impose that no SMT variable appears in the argument of an uninterpreted function.

av-separated

We call a variable v in a rule an argument variable if it occurs in an argument $\mathbf{t}$ of some uninterpreted function $f(\mathbf{t})$ in the rule. We call a variable v in a rule a value variable if it occurs in

- $f(\mathbf{t}) = v$ for any term where f is an uninterpreted function, or
- $t_1 = t_2$ where t_1, t_2 are terms consisting of interpreted symbols (i.e., from σ^{bg})

and at least one other value variable (different from v) in the rule. A rule is said to be $av - separated$ (argument-value separated) if it contains no variable that is both an argument variable and a value variable.

f-plain

Let f be a function constant. A first-order formula is called f-plain [1] if each atomic formula

- does not contain f, or
- is of the form $f(\mathbf{t}) = u$ where $\mathbf{t}$ is a tuple of terms not containing f, and u is a term not containing f.

For example, $f = 1$ is f-plain, but each of $p(f)$, $g(f) = 1$, and $1 = f$ are not f-plain.

6.2 CPLUS2ASPMT

CPLUS2ASPMT is a system we have developed to handle the proposed framework for representing hybrid transition system. As explained earlier, the input language follows similar syntax and semantics to that of action language C+. The C+ input is then translated into the language of ASPMT. We then make use of an extended version of the system ASPMT2SMT (Bartholomew and Lee (2014)) to solve the translated ASPMT encoding. The ASPMT2SMT system is extended so that it can handle the encoding of ODEs. This is done by using DREAL(ODE based SMT solver) as the SMT solver rather than the native Z3 solver. The architecture of the system can be seen in Figure 6.2.

Downloading and execution instructions as well as tutorials and examples is available at:

[1] The notion of f-plain comes from Lifschitz and Yang (2011).

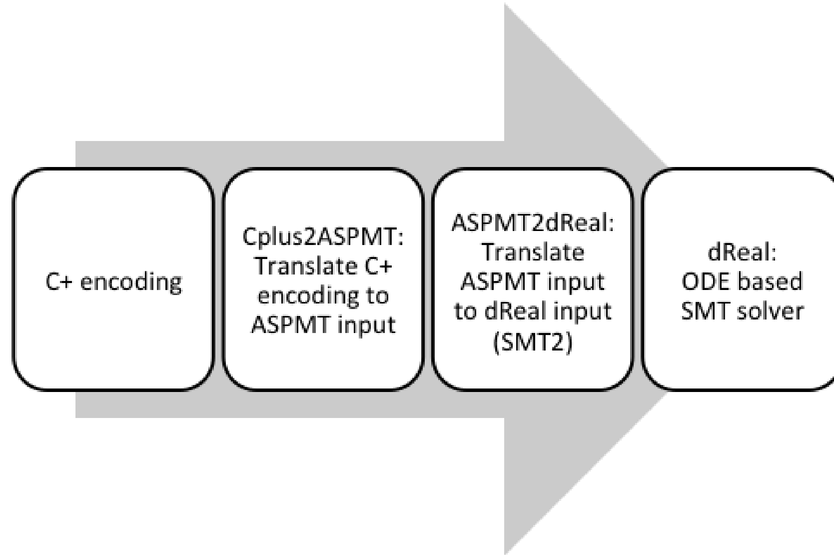

Figure 6.2: Architecture of System CPLUS2ASPMT

`http://reasoning.eas.asu.edu/cplus2aspmt`

Source code of project can be found at

`https://github.com/Nikh13/cplus2ASPMT`

6.3 Syntax of New Constructs

6.3.1 In CPLUS2ASPMT

- Differentiablel Fluent Constant Declaration: All differential fluent constants $X_i \in X$ are declared in the input language as follows:

  $$X_i :: \textbf{continuousFluent}(l..u)$$

  where l, u are lower and upper limits of the constants respectively.

- Rate declarations: Every rate declaration (4.1) is encoded in the input language as

$$d/dt[X_i] = F_i(X) \textbf{ if } mode = v.$$

- Invariant Law:Every invariant law (4.4) is encoded in the input language as

$$\textbf{always_t } F \textbf{ if } mode = v$$

6.3.2 *In Extended* ASPMT2SMT

As mentioned earlier, at the ASPMT level, the system CPLUS2ASPMT extends the system ASPMT2SMT by providing support for evolution of variables based on ODEs. To handle this we add the following constructs:

- t-variables Declaration: DREAL treats a variable c_i_t as the value of a differential constant c between the time points i and $i+1$. We represent such constants at the ASPMT level with the constant declaration:

  $$X_i_t(astep) :: \textbf{real}[l..u]$$

  where l, u are lower and upper limits of the constants respectively. Here *astep* is an integer valued sort describing the transition steps.

- Rate declarations: Every ASPMT rate declaration (4.6) is encoded in the extended input language of ASPMT2SMT as:

  $$d/dt[X_i] = F_i(X) \leftarrow mode = v.$$

  for each $X_i \in X$

- Integral construct: Every ASPMT integral law (4.7) contains the solution $i+1 : [X_1, \ldots, X_n] = i : [X_1, \ldots, X_n] + \int_0^T \theta_v \, dt$ for the ODE system θ_v at $mode = v$.

This solution can be encoded in the extended input language of ASPMT2SMT as the term:

$$int(0, T, [x_1, \ldots, x_n], v)$$

where T,$x_1, \ldots, x_n$ are variables and $[x_1, \ldots, x_n]$ is a term representing a list of terms. Here v is the *mode* for which the corresponding complete ODE system exists from the rate declarations.

- Invariant Law: Every ASPMT invariant law (4.10) is encoded in the extended input language of ASPMT2SMT as:

$$\textbf{always_t}\ F \leftarrow mode = v\ \&\ duration = T$$

6.4 Limitations

The system does not capture all examples of hybrid transition systems. This is due to certain implementation related limitations that are as follows:

1. While defining the ODE system for any given state, it is necessary that the state is defined by a single real valued constant called *mode*. This is due to the fact that the *dReal* system expects an encoding in the form of Hybrid Automata where the states are statically defined as unique modes. This prevents the **if** *clause* in the rate declaration law from being any combination of predicates.

2. Some invariants may need to be expressed using some form of disjunction($A \vee B$, $A \rightarrow B$, $\neg(A \wedge B)$). In most cases disjunctive logical expressions represent non-convex invariants. While the theory of SAT Modulo ODEs is tolerant towards any non-convex invariant, the system DREAL does not currently provide support for invariants using any form of disjunction. Hence, the system CPLUS2ASPMT imposes the restriction that the formula F in the invariant law must be void of any form of disjunction.

3. As mentioned in Chapter 4 we assume that an action description has a unique rate declaration for each X_i and v. While listing complete ODEs could be viewed as a strong condition, dReal only accepts complete ODE systems while using the integral. Hence this system imposes the further restriction that the set of rate declaration laws defined for $mode = v$ must describe a complete ODE system.

In future iterations of the system cplus2ASPMT we hope to resolve these limitations. An idea to resolve (1) would be to allow the **if** clause in the rate declaration law to be any boolean valued formula and later translating each of these formulas to a unique placeholder, *mode*. This *mode* would then be used as required by dReal. The developers of dReal are currently working on adding support for invariant formulas that are using some form of disjunction. We hope to add support for the same once its updated iteration is ready and subsequently resolve the limitation specified in (2). In (Gao *et al.* (2013b)), an extension for lifting the restriction specified in (3) is left for the future work, using new commands `pintegral` and `connect`, but they are still not available in the distributed version. It should be possible to extend the abbreviations to express partial ODEs in accordance with this extension.

6.5 Examples

In this section we will revisit Examples 1 and 3. For each of the examples we will show their cplus2ASPMT input program and visual output of the solution obtained. The intermediate ASPMT as well as SMT translations can be found in Appendix B.

6.5.1 Water Tank Example - Example 1

CPLUS2ASPMT Input Program

```
:- constants
x1 :: simpleFluent(real[0..30]);
x2 :: simpleFluent(real[0..30]);
mode :: inertialFluent(real[1..2]);
e1,e2 :: exogenousAction;
wait :: action;
duration :: exogenousAction(real[0..10]).

:- variables
X11,X21,X10,X20,T.

exogenous x1.
exogenous x2.
e1 causes duration=0.
e2 causes duration=0.

default wait.
e1 causes ~wait.
e2 causes ~wait.

caused false if mode=1 & -(x2>=r2).
caused false if mode=2 & -(x1>=r1).

nonexecutable e1 if -(x2<=r2).
nonexecutable e2 if -(x1<=r1).

nonexecutable e1 if -(mode=1).
nonexecutable e2 if -(mode=2).

constraint (((x1-X10)//T)=w1-v & ((x2-X20)//T)=-v) after x1=X10 &
    x2=X20 & mode=1 & duration=T & wait & T>0.
constraint (x1=X10 & x2=X20) after x1=X10 & x2=X20 & mode=1 &
   duration=0 & wait.
constraint (((x1-X10)//T)=-v & ((x2-X20)//T)=w2-v) after x1=X10 &
    x2=X20 & mode=2 & duration=T & wait & T>0.
constraint (x1=X10 & x2=X20) after x1=X10 & x2=X20 & mode=2 &
   duration=0 & wait.

constraint (x1=X10 & x2=X20) after x1=X10 & x2=X20 & e1.
constraint (x1=X10 & x2=X20) after x1=X10 & x2=X20 & e2.
```

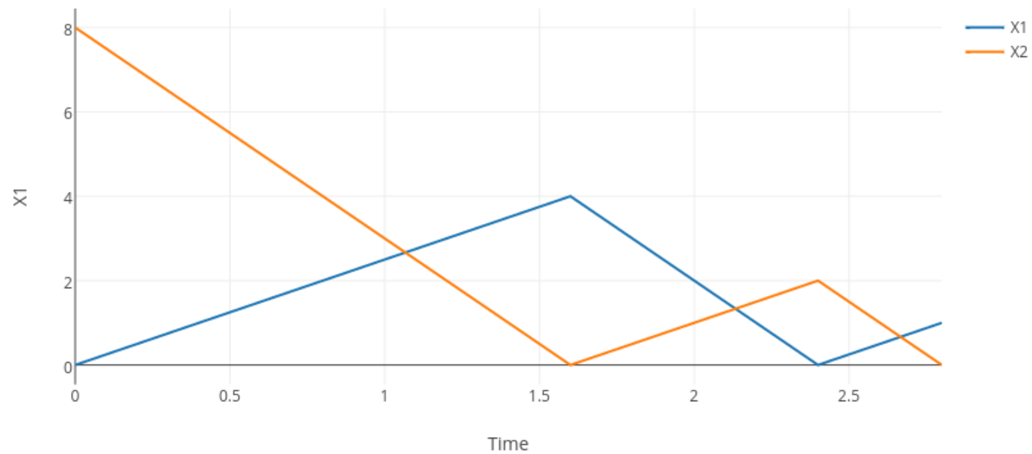

Figure 6.3: Output of Example 1

```
e1 causes mode=2.
e2 causes mode=1.

:- query
label :: init;
maxstep :: 2;
0:mode=1;
0:x1 = 10;
0:x2 = 15;
2:state=2.
```

Output of Example 1 can be seen in Figure 6.3.

6.5.2 Turning Car Example - Example 3

CPLUS2ASPMT **Input Program**

```
:- constants
x :: continuousFluent(real[0..40]);
y :: continuousFluent(real[-50..50]);
theta :: continuousFluent(real[-50..50]);
straighten, turnLeft, turnRight::exogenousAction;
mode :: inertialFluent(real[1..2]);
wait :: action;
duration :: exogenousAction(real[0..10]).
```

```
:- variables
X,X0,X1,X2,D,D1,T,RP,R.

exogenous x.
exogenous y.
exogenous theta.
straighten causes duration=0.
turnLeft causes duration=0.
turnRight causes duration=0.

default wait.
straighten causes ~wait.
turnLeft causes ~wait.
turnRight causes ~wait.

% Rates
derivative of x is cos(theta) if mode=1.
derivative of y is sin(theta) if mode=1.
derivative of theta is 0 if mode=1.
derivative of x is cos(theta) if mode=2.
derivative of y is sin(theta) if mode=2.
derivative of theta is tan(pi/18) if mode=2.
derivative of x is cos(theta) if mode=3.
derivative of y is sin(theta) if mode=3
derivative of theta is tan(-pi/18) if mode=3.

%Invariant
constraint x=X & y=Y & ((X-6)*(X-6) & (Y)*(Y)>9).
always_t (x=X & y=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) if mode=1.
always_t (x=X & y=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) if mode=2.
always_t (x=X & y=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) if mode=3.

constraint x=X & y=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4).
always_t (x=X & y=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) if mode=1.
always_t (x=X & y=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) if mode=2.
always_t (x=X & y=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) if mode=3.

constraint x=X & y=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4).
always_t (x=X & y=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4)) if mode=1.
always_t (x=X & y=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4)) if mode=2.
always_t (x=X & y=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4)) if mode=3.

straighten causes mode=1.
turnLeft causes mode=2.
turnRight causes mode=3.
```

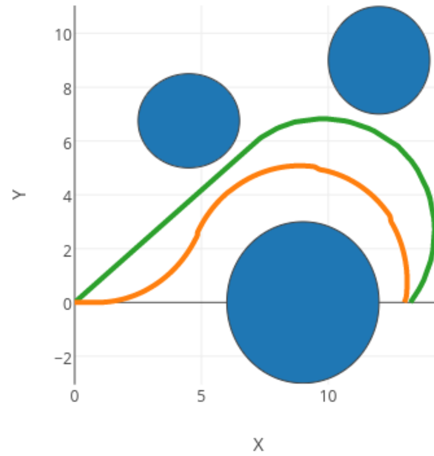

Figure 6.4: Output of Example 3

```
%State Restrictions
nonexecutable straighten if mode=1.
nonexecutable turnLeft if mode=2.
nonexecutable turnRight if mode=3.

%Reset
constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
    after x = D & y=X0 & theta=X1 & turnLeft.
constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
    after x = D & y=X0 & theta=X1 turnRight.
constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
    after x = D & y=X0 & theta=X1 & straighten.

:- query
label :: init;
maxstep :: 5;
0:x=0;
0:y=0;
0:mode=1;
5:x=13;
5:y=0.
```

Output of Example 3 can be seen in Figure 6.4. The different paths are obtained by varying the value of *maxstep*. For example, when $maxstep = 3$, we get the shorter path but when we extend the *maxstep* to 5 we are able to get the longer, more realistic

path.

6.6 Results

We run experiments on the examples described above to gauge the performance of the system in comparison to other Hybrid Automata solvers. We also run experiments to demonstrate the capability of the system in minimizing the effort in encoding by comparing encoding sizes and number of ground atoms with Hybrid Automata and SMT solvers.

Runtime Performance

Steps	dReach	cplus2ASPMT(with path)	cplus2ASPMT(without path)
Thermostat Example			
6	.454	.213	.243
10	0.658	0.645	3.22
12	1.050	1.060	> 10m
20	5.25	9.6076	>10m
Car Example			
3	0.876	1.123	8.735
4	2.312	17.23	3m28
6	5.765	33.533	>10m
7	7.322	2m43	>10m

Table 6.1: Runtime Comparison of Hybrid Automata Solvers - Runtime(s)

We run the thermostat and car example on Hybrid Automata solver dReach and our system and compare the results in Table 6.1. For our system we consider 2 scenarios: (1) Expected path is mentioned in query (2) Only final goal is specified.

This is done to try and effectively compare runtime with dReach as it incrementally solves for a specific transition path [2] . From the table we see that both systems have a similar runtime for smaller number of steps but our system takes a much longer time as steps and domain size increase. While dReach perform better than our system, the focus of this thesis is not to find the most optimum performance solution but to be able to effectively represent hybrid transition systems. dReach falls short here as they are limited to hybrid transition systems described by Hybrid Automata.

Incremental grounding in system Cplus2ASP: As observed, dReach performs better than our system by pruning out the final encoding based on a specific path. A similar approach can be seen in the working of Cplus2ASP, where incremental grounding is performed to optimize performance of the system. This is done by incrementally grounding the encoding up to the maximum number of steps and using a solution at a previous stage to solve further iterations. Since our system is an extension of Cplus2ASP, we could borrow this idea to optimize runtime.

Encoding Size

Example	cplus2ASPMT	dReach	dReal
Water Tank Example(2 steps)	47	33	115
Two Ball Example(4 steps)	45	26	275
Turning Car example(3 steps)	66	37	304

Table 6.2: Encoding Size (Number of Lines) Comparison

For each of the examples described in the earlier section we also compare their encoding size with SMT solver dReal and Hybrid Automata solver dReach in

[2]When solving a Hybrid Automata, system dReach incrementally selects a probable transition path at random and runs a subset of the rules containing only those path variables. This way it solves only a small subset of code and optimizes runtime

Table 6.2. We clearly observe that our system is much more efficient than SMT solvers like dReal in terms of encoding effort. Additionally, it is important to note that in our system, the encoding size remains the size no matter the number of steps required to obtain the solution. This is not the case for its equivalent SMT encoding as SMT input must always be grounded. We notice that dReach has a slight advantage in this metric as well. But once again, we highlight the fact that dReach is not tolerable to hybrid transition systems not defined by Hybrid Automata while cplus2ASPMT is.

Chapter 7

GOING BEYOND HYBRID AUTOMATA

The advantage of using the action language C+ is its expressivity outside of the domain of Hybrid Automata. This can be seen in its ability to handle reasoning about additive fluents and defining more complex relations for the formation of a state.

7.1 Additive Fluents

Additive fluents in C+ help abstracting away complex rules to represent concurrent effects of certain actions on a common fluent constant. Apart from adding support for the new causal rules proposed in this thesis, the CPLUS2ASPMT system also provides support for additive fluents subject to the background theory of reals as in Example 4. Cplus2ASP provided only integer domain support for additive fluents, which seemed to be highly restrictive.

Example 4 *The following formalization as introduced in (Lee and Meng (2013)) describes the level of a water tank that has two taps with different flow rates and possible leaking. The action description for the example can be seen below*

```
:- sorts
taps.

:- objects
t1,t2::taps.

:- constants
on(taps) :: inertialFluent;
tapRate(taps) :: inertialFluent(real[0..10]);
leakRate :: inertialFluent(real[0..10]);
```

```
leaking   :: inertialFluent;
duration  :: exogenousAction(real[0..20]);
turnOn(taps) :: exogenousAction;
turnOff(taps) :: exogenousAction;
level     :: additiveFluent(real[0..30]).

:- variables
Z,X,X1,X2,Y,T,S1,S0,M,D;
TP :: taps.

turnOn(TP) causes on(TP).
caused duration=0 if turnOn(TP).
turnOff(TP) causes ~on(TP).
caused duration=0 if turnOff(TP).

on(TP) increments level by X if duration=T & tapRate(TP)=X1 & X=
   X1*T.
leaking decrements level by X if duration=T & leakRate=X1 & X=X1*
   T.
```

7.2 Complex Definition of States of Hybrid Systems

Additionally, by adding the new constructs for supporting ODE and constraint checking for intermediate time points, the system CPLUS2ASPMT may also be used to represent complex relations that lead to the formation of a state that would otherwise be abstracted away in a Hybrid Automata. This can be seen in Example 5. From the encoding we can see that the action/events are not leading to the state change. The events lead to some changes in specific boolean valued fluents. The state or *mode* at any given time is determined by the relation among some of these fluents. For example *turnPumpOn* sets the value of *pumpOn* to true, but *mode* is set to 1 only if *tapOn* is false at the same time.

Example 5 *Similar to the previous example with minor modifications. When the tap is on, the bucket is filled at some constant rate. When the pump is turned on, the water is drained at an accelerated rate. The action description for the example can*

be seen below.

```
:- constants
tapRate :: inertialFluent(real[0..10]);
tapOn :: inertialFluent;
pumpOn :: inertialFluent;
turnTapOn :: exogenousAction;
turnTapOff :: exogenousAction;
turnPumpOn :: exogenousAction;
turnPumpOff :: exogenousAction;
level  :: continuousFluent(0..30);
leakRate :: continuousFluent(-30..30).

:- variables
Z,X,X1,X2,Y,T,S1,S0,M,D.

default wait.
caused duration=0 if turnPumpOn.
caused duration=0 if turnPumpOff.
caused duration=0 if turnTapOn.
caused duration=0 if turnTapOff.
caused wait=false if turnPumpOn.
caused wait=false if turnPumpOff.
caused wait=false if turnTapOn.
caused wait=false if turnTapOff.

derivative of leakRate is -1 if mode=1.
derivative of level is leakRate if mode=1.

caused mode=1 if pumpOn & -tapOn.
caused mode=2 if tapOn & -pumpOn.
caused mode=3 if -tapOn & -pumpOn.

turnPumpOn causes pumpOn.
turnPumpOff causes ~pumpOn.
turnTapOn causes tapOn.
turnTapOff causes ~tapOn.

constraint level=X after mode=2 & level=X1 & duration=T & wait &
   tapRate=X2 & X=X1+X2*T.
constraint level=X after mode=3 & level=X & duration=T & wait.
constraint (X=X1 & S1=0) & level=X & leakRate=S1 after -(wait) &
   level=X1.
```

Chapter 8

CONCLUSION

Reasoning about hybrid transition system faces several challenges among which are performing defeasible reasoning and efficient computation in the presence of large domains. While, action language $\mathcal{C}+$ helps represent such problems with the help of causal laws, most of its solvers like CPLUS2ASP are restricted to reasoning only over discrete domains. Hybrid Automata is another formalism used to represent hybrid transition systems. However most Hybrid Automata solver do not allow for any complex relations among components of the system that may be ignored or abstracted away while describing a problem in Hybrid Automata.

The research culminated in a framework to represent hybrid automata in action language modulo theories. As the enhanced action language is based on ASPMT, which in turn is founded on the basis of ASP and SMT, it enjoys the development in SMT. SMT is used extensively to represent and perform formal verification on hybrid transition systems, hence developing SMT as an underlying formalism proves to be highly advantageous. The framework also proves to be expressive as we are able to encode hybrid transitions that are not necessarily described by Hybrid Automata. By doing so we do not restrict the encoding to strictly follow Hybrid Automata semantics and allow for more complex relations among components of the system.

The research involved in this thesis culminates in an expressive and effective tool based on an Action Language Modulo ODE to represent and reason about hybrid transition systems. The thesis highlighted the ease of defining the Action Language Modulo ODE on the foundation of ASPMT thus taking advantage of the benefits of both ASP as well as development in SMT. The developed framework presents how

action language modulo ODE lifts the concept of SMT modulo ODEs to the action language level. The key improvement of this framework over other related works is its expressive power. The expressivity was displayed in its ability to represent and reason about hybrid transition systems described using Hybrid Automata as well as those that involve more complex internal relations among components of the system that cannot be expressed in Hybrid Automata. Thus enjoying the benefits of the structure of Hybrid Automata as well as the expressive power of action language $\mathcal{C}+$ in capturing crucial intricate dependencies. This framework is finally bundled as a prototype implementation and presented as system CPLUS2ASPMT.

The prototype implementation CPLUS2ASPMT presented in this thesis serve as a proof-of concept for the presented framework by providing the capability to reason and efficiently compute hybrid transition systems whose continuous components are governed by ODEs. The system does not consider composition of hybrid automata in the ODE setting, because the underlying technology of SMT is not yet mature enough to handle this. In (Gao *et al.* (2013b)), such extension is left for the future work, using new commands `pintegral` and `connect`, but they are still not available in the distributed version. It should be possible to extend the abbreviations to express partial ODEs in accordance with this extension. SMT solvers are becoming the key enabling technology in formal verification in hybrid systems. Nonetheless expressing the concept in the language of SMT is non-trivial. Though the framework presented in this thesis, we expect that high level action languages can facilitate reasoning and encoding efforts for hybrid transition systems.

REFERENCES

Akman, V., S. Erdoğan, J. Lee and V. Lifschitz, "A representation of the traffic world in the language of the causal calculator", In *Working Notes of the Fifth Symposium on Formalizations of Commonsense Knowledge* (2001).

Alur, R., T. A. Henzinger, G. Lafferriere, George and J. Pappas, "Discrete abstractions of hybrid systems", in "Proceedings of the IEEE", pp. 971–984 (2000).

Babb, J. and J. Lee, "Cplus2ASP: Computing action language $\mathcal{C}+$ in answer set programming", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 122–134 (2013).

Baral, C., *Knowledge representation, reasoning and declarative problem solving* (Cambridge university press, 2003).

Barrett, C. W., R. Sebastiani, S. A. Seshia and C. Tinelli, "Satisfiability modulo theories", in "Handbook of Satisfiability", edited by A. Biere, M. Heule, H. van Maaren and T. Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 825–885 (IOS Press, 2009).

Bartholomew, M. and J. Lee, "Functional stable model semantics and answer set programming modulo theories", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", (2013).

Bartholomew, M. and J. Lee, "System ASPMT2SMT: Computing aspmt theories by smt solvers", in "Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)", pp. 529–542 (2014).

Bogomolov, S., D. Magazzeni, A. Podelski and M. Wehrle, "Planning as model checking in hybrid domains", in "AAAI'14", (2014).

Boyd, S. and L. Vandenberghe, *Convex optimization* (Cambridge university press, 2004).

Bryce, D., S. Gao, D. Musliner and R. Goldman, "Smt-based nonlinear pddl+ planning", in "Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence", (2015).

Chintabathina, S., "Towards answer set prolog based architectures for intelligent agents.", in "AAAI'08", pp. 1843–1844 (2008).

Coles, A. and A. Coles, "Pddl+ planning with events and linear processes", in "Proceedings of the Twenty- Fourth International Conference on Automated Planning and Scheduling", (2014).

Coles, A., A. Coles, M. Fox and D. Long, "Colin: Planning with continuous linear numeric change", Journal of Artificial Intelligence Research 44 (2012).

Ferraris, P., J. Lee and V. Lifschitz, "Stable models and circumscription", Artificial Intelligence **175**, 236–263 (2011).

Fox, M. and D. Long, "PDDL2.1: An extension to pddl for expressing temporal planning domains", J. Artif. Intell. Res. (JAIR) **20**, 61–124 (2003).

Fox, M. and D. Long, "Modelling mixed discrete-continuous domains for planning", J. Artif. Intell. Res. (JAIR) **27**, 235–297 (2006).

Gao, S., S. Kong and E. Clarke, "Satisfiability modulo odes", arXiv preprint arXiv:1310.8278 (2013a).

Gao, S., S. Kong and E. M. Clarke, "dreal: An smt solver for nonlinear theories over the reals", in "International Conference on Automated Deduction", pp. 208–214 (Springer Berlin Heidelberg, 2013b).

Gebser, M., T. Grote and T. Schaub, "Coala: A compiler from action languages to ASP", in "Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)", pp. 360–364 (2010).

Giunchiglia, E., J. Lee, V. Lifschitz, N. McCain and H. Turner, "Nonmonotonic causal theories", Artificial Intelligence **153(1–2)**, 49–104 (2004).

Henzinger, T. A., "The theory of hybrid automata", in "Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science", pp. 278–292 (1996).

Lee, J., V. Lifschitz and F. Yang, "Action language $\mathcal{BC}$: Preliminary report", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", (2013).

Lee, J. and Y. Meng, "Answer set programming modulo theories and reasoning about continuous changes", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", (2013).

Lee, J. and R. Palla, "System F2LP – computing answer sets of first-order formulas", in "Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 515–521 (2009).

Mellarkod, V. S., M. Gelfond and Y. Zhang, "Integrating answer set programming and constraint logic programming", Annals of Mathematics and Artificial Intelligence **53**, 1-4, 251–287 (2008).

Penna, G. D., D. Magazzeni, F. Mercorio and B. Intrigila, "Upmurphi: A tool for universal planning on pddl+ problems", in "Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling", (2009).

Shin, J.-A. and E. Davis, "Processes and continuous change in a sat-based planner", Artificial Intelligence **166**, 1-2, 194–253 (2005).

APPENDIX A

ABBREVIATIONS IN $\mathcal{C}+$

1. A static law or an action dynamic law of the form

$$\textbf{caused } F \textbf{ if } \top$$

can be written as

$$\textbf{caused } F.$$

2. A fluent dynamic law of the form

$$\textbf{caused } F \textbf{ if } \top \textbf{ after } H$$

can be written as

$$\textbf{caused } F \textbf{ after } H.$$

3. A static law of the form

$$\textbf{caused } \bot \textbf{ if } \neg F$$

can be written as

$$\textbf{constraint } F.$$

4. A fluent dynamic law of the form

$$\textbf{caused } \bot \textbf{ if } \neg F \textbf{ after } G$$

can be written as

$$\textbf{constraint } F \textbf{ after } G.$$

5. An expression of the form

$$\textbf{rigid } c$$

where c is a fluent constant stands for the set of causal laws

$$\textbf{constraint } c{=}v \textbf{ after } c{=}v$$

for all $v \in Dom(c)$.

6. A fluent dynamic law of the form

$$\textbf{caused } \bot \textbf{ after } \neg F$$

can be written as

$$\textbf{always } F.$$

7. A fluent dynamic law of the form

$$\textbf{caused } \bot \textbf{ after } F \wedge G$$

where F is an action formula can be written as

$$\textbf{nonexecutable } F \textbf{ if } G. \tag{A.1}$$

8. An expression of the form

$$F \textbf{ causes } G \textbf{ if } H \tag{A.2}$$

where F is an action formula stands for the fluent dynamic law

$$\textbf{caused } G \textbf{ after } F \wedge H$$

if G is a fluent formula, [1] and for the action dynamic law

$$\textbf{caused } G \textbf{ if } F \wedge H$$

if G is an action formula.

9. An expression of the form

$$\textbf{default } F \textbf{ if } G \tag{A.3}$$

stands for the causal law

$$\textbf{caused } \{F\}^{\text{ch}} \textbf{ if } G.$$

10. An expression of the form

$$\textbf{default } F \textbf{ if } G \textbf{ after } H$$

stands for the fluent dynamic law

$$\textbf{caused } \{F\}^{\text{ch}} \textbf{ if } G \textbf{ after } H.$$

The part

$$\textbf{if } G$$

[1] It is clear that the expression in the previous line is a fluent dynamic law only when G does not contain statically determined fluent constants. Similar remarks can be made in connection with many of the abbreviations introduced below.

in this abbreviation can be dropped if G is $\top$.

11. An expression of the form

$$\textbf{exogenous } c \textbf{ if } G \tag{A.4}$$

where c is a constant stands for the set of causal laws

$$\textbf{default } c{=}v \textbf{ if } G$$

for all $v \in Dom(c)$.

12. An expression of the form

$$F \textbf{ may cause } G \textbf{ if } H \tag{A.5}$$

where F is an action formula stands for the fluent dynamic law

$$\textbf{default } G \textbf{ after } F \wedge H$$

if G is a fluent formula, and for the action dynamic law

$$\textbf{default } G \textbf{ if } F \wedge H$$

if G is an action formula.

13. An expression of the form

$$\textbf{inertial } c \textbf{ if } G \tag{A.6}$$

where c is a fluent constant stands for the set of fluent dynamic laws

$$\textbf{default } c{=}v \textbf{ after } c{=}v \wedge G$$

for all $v \in Dom(c)$.

14. If any of the abbreviations (A.2)–(A.6) ends with

$$\textbf{if } \top$$

then this part of the expression can be dropped.

15. The expression obtained by appending

$$\textbf{unless } c$$

where c is a Boolean statically determined fluent constant to a static law

$$\textbf{caused } F \textbf{ if } G \tag{A.7}$$

stands for the pair of static laws

$$\begin{aligned} &\textbf{caused } F \textbf{ if } G \wedge \sim c, \\ &\textbf{default } \sim c. \end{aligned} \tag{A.8}$$

16. The expression obtained by appending

$$\textbf{unless } c$$

where c is a Boolean action constant to an action dynamic law (A.7) stands for the pair of action dynamic laws (A.8).

17. The expression obtained by appending

$$\textbf{unless } c$$

where c is a Boolean action constant to a fluent dynamic law

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H$$

stands for the pair of dynamic laws

$$\begin{aligned} &\textbf{caused } F \textbf{ if } G \textbf{ after } H \wedge \sim c, \\ &\textbf{default } \sim c. \end{aligned}$$

APPENDIX B

WATER TANK EXAMPLE

As we have already covered the input and output in Chapter 6, we will show only the intermediate ASPMT and SMT translation for Example 1

Intermediate ASPMT Translation

```
:- constants
step:x1 :: real[0..30];
step:x2 :: real[0..30];
step:mode :: real[1..2];
astep:e1,e2 :: boolean;
astep:wait :: boolean;
astep:duration :: real[0..10].

{AS+1:mode=X} <- AS:mode=X.
{ST:x1=X}.
{ST:x2=X}.
{AS:duration=X}.
{AS:e1=X}.
{AS:e2=X}.

%System Rules
%All actions have 0 duration
AS:duration=0 <- AS:e1.
AS:duration=0 <- AS:e2.

{AS:wait=true}.
AS:wait=false <- AS:e1.
AS:wait=false <- AS:e2.

<- ST:mode=1 & -(ST:x2>=r2).
<- ST:mode=2 & -(ST:x1>=r1).

<- AS:e1 & -(AS:x2<=r2).
<- AS:e2 & -(AS:x1<=r1).

<- AS:e1 & -(AS:mode=1).
<- AS:e2 & -(AS:mode=2).
```

```
<- not (((AS+1x1-X10)//T)=w1-v & ((AS+1:x2-X20)//T)=-v) & AS:x1=
   X10 & AS:x2=X20 & AS:mode=1 & AS:duration=T & AS:wait & T>0.
<- not (AS+1:x1=X10 & AS+1:x2=X20) & AS:x1=X10 & AS:x2=X20 & AS:
   mode=1 &  AS:duration=0 & AS:wait.
<- not (((AS+1x1-X10)//T)=-v & ((AS+1:x2-X20)//T)=w2-v) & AS:x1=
   X10 & AS:x2=X20 & AS:mode=2 & AS:duration=T & AS:wait & T>0.
<- not (AS+1:x1=X10 & AS+1:x2=X20) & AS:x1=X10 & AS:x2=X20 & AS:
   mode=2 &  AS:duration=0 & AS:wait.

<- not (AS+1:x1=X10 & AS+1:x2=X20) & AS:x1=X10 & AS:x2=X20 & AS:
   e1.
<- not (AS+1:x1=X10 & AS+1:x2=X20) & AS:x1=X10 & AS:x2=X20 & AS:
   e2.

AS+1:mode=2 <- AS:e1
AS+1:mode=1 <- AS:e2

<- not 0:mode=1.
<- not 0:x1 = 0.
<- not 0:x2 = 8.
<- not 2:mode=2.
```

SMT Encoding

```
(set-logic QF_NRA_ODE)
(declare-const true_a Bool)
(declare-const false_a Bool)
(declare-const duration_0_ Real)
(declare-const duration_1_ Real)
(declare-const e1_0_ Bool)
(declare-const e1_1_ Bool)
(declare-const e2_0_ Bool)
(declare-const e2_1_ Bool)
(declare-const qlabel_init_ Bool)
(declare-const state_0_ Real)
(declare-const state_1_ Real)
(declare-const state_2_ Real)
(declare-const wait_0_ Bool)
(declare-const wait_1_ Bool)
(declare-const x1_0_ Real)
(declare-const x1_1_ Real)
(declare-const x1_2_ Real)
(declare-const x2_0_ Real)
(declare-const x2_1_ Real)
```

```
(declare-const x2_2_ Real)
(assert true_a)
(assert (not false_a))
(assert (>= duration_0_ 0))
(assert (<= duration_0_ 10))
(assert (>= duration_1_ 0))
(assert (<= duration_1_ 10))
(assert (>= state_0_ 1))
(assert (<= state_0_ 2))
(assert (>= state_1_ 1))
(assert (<= state_1_ 2))
(assert (>= state_2_ 1))
(assert (<= state_2_ 2))
(assert (>= x1_0_ 0))
(assert (<= x1_0_ 30))
(assert (>= x1_1_ 0))
(assert (<= x1_1_ 30))
(assert (>= x1_2_ 0))
(assert (<= x1_2_ 30))
(assert (>= x2_0_ 0))
(assert (<= x2_0_ 30))
(assert (>= x2_1_ 0))
(assert (<= x2_1_ 30))
(assert (>= x2_2_ 0))
(assert (<= x2_2_ 30))
(assert (or (or (and (= wait_1_ true) (= wait_1_ true)) (and (=
   e2_1_ true) (= wait_1_ false))) (and (= e1_1_ true) (= wait_1_
    false))))
(assert (=> (= wait_1_ true) (= wait_1_ true)))
(assert (=> (= e2_1_ true) (= wait_1_ false)))
(assert (=> (= e1_1_ true) (= wait_1_ false)))
(assert (or (or (and (= wait_0_ true) (= wait_0_ true)) (and (=
   e2_0_ true) (= wait_0_ false))) (and (= e1_0_ true) (= wait_0_
    false))))
(assert (=> (= wait_0_ true) (= wait_0_ true)))
(assert (=> (= e2_0_ true) (= wait_0_ false)))
(assert (=> (= e1_0_ true) (= wait_0_ false)))
(assert (or (or (= state_2_ state_1_) (and (= e2_1_ true) (=
   state_2_ 1))) (and (= e1_1_ true) (= state_2_ 2))))
(assert (=> (= e2_1_ true) (= state_2_ 1)))
(assert (=> (= e1_1_ true) (= state_2_ 2)))
(assert (or (or (= state_1_ state_0_) (and (= e2_0_ true) (=
   state_1_ 1))) (and (= e1_0_ true) (= state_1_ 2))))
(assert (=> (= e2_0_ true) (= state_1_ 1)))
(assert (=> (= e1_0_ true) (= state_1_ 2)))
```

```
(assert (= qlabel_init_ true))
(assert (= qlabel_init_ true))
(assert (=> (= e2_1_ true) (= duration_1_ 0)))
(assert (=> (= e1_1_ true) (= duration_1_ 0)))
(assert true_a)
(assert (=> (= e2_0_ true) (= duration_0_ 0)))
(assert (=> (= e1_0_ true) (= duration_0_ 0)))
(assert (not (and (= state_0_ 1) (not (>= x2_0_ 0)))))
(assert (not (and (= state_2_ 1) (not (>= x2_2_ 0)))))
(assert (not (and (= state_1_ 1) (not (>= x2_1_ 0)))))
(assert (not (and (= state_0_ 2) (not (>= x1_0_ 0)))))
(assert (not (and (= state_2_ 2) (not (>= x1_2_ 0)))))
(assert (not (and (= state_1_ 2) (not (>= x1_1_ 0)))))
(assert (not (and (= e1_1_ true) (not (<= x2_1_ 0)))))
(assert (not (and (= e1_0_ true) (not (<= x2_0_ 0)))))
(assert (not (and (= e2_1_ true) (not (<= x1_1_ 0)))))
(assert (not (and (= e2_0_ true) (not (<= x1_0_ 0)))))
(assert (not (and (= e1_1_ true) (not (= state_1_ 1)))))
(assert (not (and (= e1_0_ true) (not (= state_0_ 1)))))
(assert (not (and (= e2_1_ true) (not (= state_1_ 2)))))
(assert (not (and (= e2_0_ true) (not (= state_0_ 2)))))
(assert (not (and (and (and (not (= (/ (- x1_2_ x1_1_)
   duration_1_) (- (/ 75 10) 5))) (> duration_1_ 0)) (= state_1_
   1)) (= wait_1_ true))))
(assert (not (and (and (and (not (= (/ (- x1_1_ x1_0_)
   duration_0_) (- (/ 75 10) 5))) (> duration_0_ 0)) (= state_0_
   1)) (= wait_0_ true))))
(assert (not (and (and (and (not (= (/ (- x2_2_ x2_1_)
   duration_1_) -5)) (> duration_1_ 0)) (= state_1_ 1)) (=
   wait_1_ true))))
(assert (not (and (and (and (not (= (/ (- x2_1_ x2_0_)
   duration_0_) -5)) (> duration_0_ 0)) (= state_0_ 1)) (=
   wait_0_ true))))
(assert (not (and (and (and (not (= x1_2_ x1_1_)) (= wait_1_ true
   )) (= state_1_ 1)) (= duration_1_ 0))))
(assert (not (and (and (and (not (= x1_1_ x1_0_)) (= wait_0_ true
   )) (= state_0_ 1)) (= duration_0_ 0))))
(assert (not (and (and (and (not (= x2_2_ x2_1_)) (= wait_1_ true
   )) (= state_1_ 1)) (= duration_1_ 0))))
(assert (not (and (and (and (not (= x2_1_ x2_0_)) (= wait_0_ true
   )) (= state_0_ 1)) (= duration_0_ 0))))
(assert (not (and (and (and (not (= (/ (- x1_2_ x1_1_)
   duration_1_) -5)) (> duration_1_ 0)) (= state_1_ 2)) (=
   wait_1_ true))))
(assert (not (and (and (and (not (= (/ (- x1_1_ x1_0_)
```

```
   duration_0_) -5)) (> duration_0_ 0)) (= state_0_ 2)) (=
   wait_0_ true))))
(assert (not (and (and (and (not (= (/ (- x2_2_ x2_1_)
   duration_1_) (- (/ 75 10) 5))) (> duration_1_ 0)) (= state_1_
   2)) (= wait_1_ true))))
(assert (not (and (and (and (not (= (/ (- x2_1_ x2_0_)
   duration_0_) (- (/ 75 10) 5))) (> duration_0_ 0)) (= state_0_
   2)) (= wait_0_ true))))
(assert (not (and (and (and (not (= x1_2_ x1_1_)) (= wait_1_ true
   )) (= state_1_ 2)) (= duration_1_ 0))))
(assert (not (and (and (and (not (= x1_1_ x1_0_)) (= wait_0_ true
   )) (= state_0_ 2)) (= duration_0_ 0))))
(assert (not (and (and (and (not (= x2_2_ x2_1_)) (= wait_1_ true
   )) (= state_1_ 2)) (= duration_1_ 0))))
(assert (not (and (and (and (not (= x2_1_ x2_0_)) (= wait_0_ true
   )) (= state_0_ 2)) (= duration_0_ 0))))
(assert (not (and (not (= x1_2_ x1_1_)) (= e1_1_ true))))
(assert (not (and (not (= x1_1_ x1_0_)) (= e1_0_ true))))
(assert (not (and (not (= x2_2_ x2_1_)) (= e1_1_ true))))
(assert (not (and (not (= x2_1_ x2_0_)) (= e1_0_ true))))
(assert (not (and (not (= x1_2_ x1_1_)) (= e2_1_ true))))
(assert (not (and (not (= x1_1_ x1_0_)) (= e2_0_ true))))
(assert (not (and (not (= x2_2_ x2_1_)) (= e2_1_ true))))
(assert (not (and (not (= x2_1_ x2_0_)) (= e2_0_ true))))
(assert (not (not (= state_0_ 1))))
(assert (not (not (= x1_0_ 0))))
(assert (not (not (= x2_0_ 8))))
(assert (not (not (= x2_2_ 0))))




(check-sat)
(exit)
```

APPENDIX C

2 BALL EXAMPLE

Input, Output and all intermediate translations for Example 2.

CPLUS2ASPMT INPUT PROGRAM

```
:- sorts
ball.

:- objects
b1,b2::ball.

:- constants
mode :: inertialFluent(real[0..1]);
height(ball) :: simpleFluent(real[0..50]);
velocity(ball) :: simpleFluent(real[-30..30]);
hitGround(ball) :: exogenousAction;
wait :: action;
duration :: exogenousAction(real[0..100]).

:- variables
E :: events;
B :: ball;
B1 :: ball;
V,V1,V0,D,H,H1,H0,T,R.

%System Rules
%All actions have 0 duration
caused duration=0 if hitGround(B).

default wait.
caused ~wait if hitGround(B).
exogenous height(B).
exogenous velocity(B).

% Flow Translation
constraint height(B)=H1 after velocity(B)=V0 & height(B) = H0 &
H1=H0+V0*T+0.5*-g*T*T & mode=1 & duration = T & wait.
```

```
constraint velocity(B)=V1 after velocity(B) = V0 & V1=V0 + -g*T &
    mode=1 & duration = T & wait.

% Guard Translation
nonexecutable hitGround(B) if -(height(B)=0).

% Hevent Restrictions
nonexecutable hitGround(B) if -(mode=1).

% Invariant Translation
caused false if mode=1 & -(height(B)>=0).

% Reset Translation
constraint (H1=0 & V1=-0.9*V0) & height(b2)=H1 & velocity(b2)=V1
   after height(b2)=H0 & velocity(b2)=V0 & hitGround(b2).
constraint (H1=0 & V1=-0.8*V0) & height(b1)=H1 & velocity(b1)=V1
   after height(b1)=H0 & velocity(b1)=V0 & hitGround(b1).
constraint (H1=H0 & V1=V0) & height(B)=H1 & velocity(B)=V1 after
   height(B)=H0 & velocity(B)=V0 & -(B=B1) & hitGround(B1).

% Planning
:- query
label :: init;
maxstep :: 7;
0:mode=1;
0:height(b1)=2;
0:height(b2)=3;
0:velocity(b1)=0;
0:velocity(b2)=0.
```

Intermediate ASPMT translation

```
:- sorts
events,ball;astep;step.

:- objects
0..maxstep :: step;
0..maxstep-1 :: astep;
b1,b2::ball.

:- variables
ST :: step;
AS :: astep.

:- constants
```

```
step:mode :: integer[0..5];
step:height(ball) :: real[0..50];
step:velocity(ball) :: real[-30..30];
astep:hitGround(ball)::boolean;
astep:wait :: boolean;
astep:duration :: real[0..100].

:- variables
E :: events;
B :: ball;
B1 :: ball.

{AS+1:mode=X} <- AS:mode=X.
{ST:height(B)=X}.
{ST:velocity(B)=X}.
{AS:duration=X}.
{AS:hitGround(B)=X}.

%System Rules
%All actions have 0 duration
AS:duration=0 <- AS:hitGround(B).

{wait(AS)=true}.
AS:wait=false <- AS:hitGround(B).

% Flow Translation
<- not (AS+1:height(B)=H1) & AS:velocity(B)=V0 & AS:height(B) =
    H0 & H1=H0+V0*T+0.5*-g*T*T & AS:mode=1 & AS:duration = T & AS:
    wait.
<- not (AS+1:velocity(B)=V1) & AS:velocity(B) = V0 & V1=V0 + -g*T
     & AS:mode=1 & AS:duration = T & AS:wait.

% Guard Translation
<- AS:hitGround(B) & -(AS:height(B)=0).

% Hevent Restrictions
<- AS:hitGround(B) & -(AS:mode=1).

% Invariant Translation
<- not (ST:height(B)>=0)

% Reset Translation
<- not (H1=0 & V1=-0.9*V0) & AS+1:height(b2)=H1 & AS+1:velocity(
    b2)=V1 &
AS:height(b2)=H0 & AS:velocity(b2)=V0 & AS:hitGround(b2) .
```

```
<- not (H1=0 & V1=-0.8*V0) & AS+1:height(b1)=H1 & AS+1:velocity(
    b1)=V1 &
AS:height(b1)=H0 & AS:velocity(b1)=V0 & AS:hitGround(b1).

<- not (H1=H0 & V1=V0) & AS+1:height(B)=H1 & AS+1:velocity(B)=V1
    &
AS:height(B)=H0 & AS:velocity(B)=V0 & -(B=B1) & AS:hitGround(B).

% Planning
0:mode=1;
0:height(b1)=2;
0:height(b2)=3;
0:velocity(b1)=0;
0:velocity(b2)=0.
```

Final SMT Encoding

```
(set-logic QF_NRA_ODE)
(declare-const true_a Bool)
(declare-const false_a Bool)
(declare-const duration_0_ Real)
(declare-const duration_1_ Real)
(declare-const duration_2_ Real)
(declare-const duration_3_ Real)
(declare-const height_b1_0_ Real)
(declare-const height_b1_1_ Real)
(declare-const height_b1_2_ Real)
(declare-const height_b1_3_ Real)
(declare-const height_b1_4_ Real)
(declare-const height_b2_0_ Real)
(declare-const height_b2_1_ Real)
(declare-const height_b2_2_ Real)
(declare-const height_b2_3_ Real)
(declare-const height_b2_4_ Real)
(declare-const hevent_hitGround_b1__0_ Bool)
(declare-const hevent_hitGround_b1__1_ Bool)
(declare-const hevent_hitGround_b1__2_ Bool)
(declare-const hevent_hitGround_b1__3_ Bool)
(declare-const hevent_hitGround_b2__0_ Bool)
(declare-const hevent_hitGround_b2__1_ Bool)
(declare-const hevent_hitGround_b2__2_ Bool)
(declare-const hevent_hitGround_b2__3_ Bool)
(declare-const qlabel_init_ Bool)
(declare-const s0_0_ Bool)
(declare-const s0_1_ Bool)
```

```
(declare-const s0_2_ Bool)
(declare-const s0_3_ Bool)
(declare-const s0_4_ Bool)
(declare-const velocity_b1_0_ Real)
(declare-const velocity_b1_1_ Real)
(declare-const velocity_b1_2_ Real)
(declare-const velocity_b1_3_ Real)
(declare-const velocity_b1_4_ Real)
(declare-const velocity_b2_0_ Real)
(declare-const velocity_b2_1_ Real)
(declare-const velocity_b2_2_ Real)
(declare-const velocity_b2_3_ Real)
(declare-const velocity_b2_4_ Real)
(declare-const wait_0_ Bool)
(declare-const wait_1_ Bool)
(declare-const wait_2_ Bool)
(declare-const wait_3_ Bool)
(assert true_a)
(assert (not false_a))
(assert (>= duration_0_ 0))
(assert (<= duration_0_ 100))
(assert (>= duration_1_ 0))
(assert (<= duration_1_ 100))
(assert (>= duration_2_ 0))
(assert (<= duration_2_ 100))
(assert (>= duration_3_ 0))
(assert (<= duration_3_ 100))
(assert (>= height_b1_0_ 0))
(assert (<= height_b1_0_ 50))
(assert (>= height_b1_1_ 0))
(assert (<= height_b1_1_ 50))
(assert (>= height_b1_2_ 0))
(assert (<= height_b1_2_ 50))
(assert (>= height_b1_3_ 0))
(assert (<= height_b1_3_ 50))
(assert (>= height_b1_4_ 0))
(assert (<= height_b1_4_ 50))
(assert (>= height_b2_0_ 0))
(assert (<= height_b2_0_ 50))
(assert (>= height_b2_1_ 0))
(assert (<= height_b2_1_ 50))
(assert (>= height_b2_2_ 0))
(assert (<= height_b2_2_ 50))
(assert (>= height_b2_3_ 0))
(assert (<= height_b2_3_ 50))
```

```
(assert (>= height_b2_4_ 0))
(assert (<= height_b2_4_ 50))
(assert (>= velocity_b1_0_ -30))
(assert (<= velocity_b1_0_ 30))
(assert (>= velocity_b1_1_ -30))
(assert (<= velocity_b1_1_ 30))
(assert (>= velocity_b1_2_ -30))
(assert (<= velocity_b1_2_ 30))
(assert (>= velocity_b1_3_ -30))
(assert (<= velocity_b1_3_ 30))
(assert (>= velocity_b1_4_ -30))
(assert (<= velocity_b1_4_ 30))
(assert (>= velocity_b2_0_ -30))
(assert (<= velocity_b2_0_ 30))
(assert (>= velocity_b2_1_ -30))
(assert (<= velocity_b2_1_ 30))
(assert (>= velocity_b2_2_ -30))
(assert (<= velocity_b2_2_ 30))
(assert (>= velocity_b2_3_ -30))
(assert (<= velocity_b2_3_ 30))
(assert (>= velocity_b2_4_ -30))
(assert (<= velocity_b2_4_ 30))
(assert (or (or (and (= wait_3_ true) (= wait_3_ true)) (and (=
   hevent_hitGround_b2__3_ true) (= wait_3_ false))) (and (=
   hevent_hitGround_b1__3_ true) (= wait_3_ false))))
(assert (=> (= wait_3_ true) (= wait_3_ true)))
(assert (=> (= hevent_hitGround_b2__3_ true) (= wait_3_ false)))
(assert (=> (= hevent_hitGround_b1__3_ true) (= wait_3_ false)))
(assert (or (or (and (= wait_2_ true) (= wait_2_ true)) (and (=
   hevent_hitGround_b2__2_ true) (= wait_2_ false))) (and (=
   hevent_hitGround_b1__2_ true) (= wait_2_ false))))
(assert (=> (= wait_2_ true) (= wait_2_ true)))
(assert (=> (= hevent_hitGround_b2__2_ true) (= wait_2_ false)))
(assert (=> (= hevent_hitGround_b1__2_ true) (= wait_2_ false)))
(assert (or (or (and (= wait_1_ true) (= wait_1_ true)) (and (=
   hevent_hitGround_b2__1_ true) (= wait_1_ false))) (and (=
   hevent_hitGround_b1__1_ true) (= wait_1_ false))))
(assert (=> (= wait_1_ true) (= wait_1_ true)))
(assert (=> (= hevent_hitGround_b2__1_ true) (= wait_1_ false)))
(assert (=> (= hevent_hitGround_b1__1_ true) (= wait_1_ false)))
(assert (or (or (and (= wait_0_ true) (= wait_0_ true)) (and (=
   hevent_hitGround_b2__0_ true) (= wait_0_ false))) (and (=
   hevent_hitGround_b1__0_ true) (= wait_0_ false))))
(assert (=> (= wait_0_ true) (= wait_0_ true)))
(assert (=> (= hevent_hitGround_b2__0_ true) (= wait_0_ false)))
```

```
(assert (=> (= hevent_hitGround_b1__0_ true) (= wait_0_ false)))
(assert (= s0_4_ s0_3_))
(assert (= s0_3_ s0_2_))
(assert (= s0_2_ s0_1_))
(assert (= s0_1_ s0_0_))
(assert (= qlabel_init_ true))
(assert (= qlabel_init_ true))
(assert (=> (= hevent_hitGround_b2__3_ true) (= duration_3_ 0)))
(assert (=> (= hevent_hitGround_b1__3_ true) (= duration_3_ 0)))
(assert (=> (= hevent_hitGround_b2__2_ true) (= duration_2_ 0)))
(assert (=> (= hevent_hitGround_b1__2_ true) (= duration_2_ 0)))
(assert (=> (= hevent_hitGround_b2__1_ true) (= duration_1_ 0)))
(assert (=> (= hevent_hitGround_b1__1_ true) (= duration_1_ 0)))
(assert true_a)
(assert (=> (= hevent_hitGround_b2__0_ true) (= duration_0_ 0)))
(assert (=> (= hevent_hitGround_b1__0_ true) (= duration_0_ 0)))
(assert (not (and (and (not (= height_b2_1_ (+ (+ height_b2_0_ (*
    velocity_b2_0_ duration_0_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_0_) duration_0_)))) (= s0_0_ true)) (= wait_0_ true))
   ))
(assert (not (and (and (not (= height_b2_2_ (+ (+ height_b2_1_ (*
    velocity_b2_1_ duration_1_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_1_) duration_1_)))) (= s0_1_ true)) (= wait_1_ true))
   ))
(assert (not (and (and (not (= height_b2_3_ (+ (+ height_b2_2_ (*
    velocity_b2_2_ duration_2_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_2_) duration_2_)))) (= s0_2_ true)) (= wait_2_ true))
   ))
(assert (not (and (and (not (= height_b2_4_ (+ (+ height_b2_3_ (*
    velocity_b2_3_ duration_3_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_3_) duration_3_)))) (= s0_3_ true)) (= wait_3_ true))
   ))
(assert (not (and (and (not (= height_b1_1_ (+ (+ height_b1_0_ (*
    velocity_b1_0_ duration_0_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_0_) duration_0_)))) (= s0_0_ true)) (= wait_0_ true))
   ))
(assert (not (and (and (not (= height_b1_2_ (+ (+ height_b1_1_ (*
    velocity_b1_1_ duration_1_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_1_) duration_1_)))) (= s0_1_ true)) (= wait_1_ true))
   ))
(assert (not (and (and (not (= height_b1_3_ (+ (+ height_b1_2_ (*
    velocity_b1_2_ duration_2_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_2_) duration_2_)))) (= s0_2_ true)) (= wait_2_ true))
   ))
(assert (not (and (and (not (= height_b1_4_ (+ (+ height_b1_3_ (*
```

```
    velocity_b1_3_ duration_3_)) (* (* (/ (* (/ 5 10) -98) 10)
   duration_3_) duration_3_)))) (= s0_3_ true)) (= wait_3_ true))
   ))
(assert (not (and (and (not (= velocity_b2_1_ (+ velocity_b2_0_
   (* (/ -98 10) duration_0_)))) (= s0_0_ true)) (= wait_0_ true)
   )))
(assert (not (and (and (not (= velocity_b2_2_ (+ velocity_b2_1_
   (* (/ -98 10) duration_1_)))) (= s0_1_ true)) (= wait_1_ true)
   )))
(assert (not (and (and (not (= velocity_b2_3_ (+ velocity_b2_2_
   (* (/ -98 10) duration_2_)))) (= s0_2_ true)) (= wait_2_ true)
   )))
(assert (not (and (and (not (= velocity_b2_4_ (+ velocity_b2_3_
   (* (/ -98 10) duration_3_)))) (= s0_3_ true)) (= wait_3_ true)
   )))
(assert (not (and (and (not (= velocity_b1_1_ (+ velocity_b1_0_
   (* (/ -98 10) duration_0_)))) (= s0_0_ true)) (= wait_0_ true)
   )))
(assert (not (and (and (not (= velocity_b1_2_ (+ velocity_b1_1_
   (* (/ -98 10) duration_1_)))) (= s0_1_ true)) (= wait_1_ true)
   )))
(assert (not (and (and (not (= velocity_b1_3_ (+ velocity_b1_2_
   (* (/ -98 10) duration_2_)))) (= s0_2_ true)) (= wait_2_ true)
   )))
(assert (not (and (and (not (= velocity_b1_4_ (+ velocity_b1_3_
   (* (/ -98 10) duration_3_)))) (= s0_3_ true)) (= wait_3_ true)
   )))
(assert (not (and (= hevent_hitGround_b2__0_ true) (not (=
   height_b2_0_ 0)))))
(assert (not (and (= hevent_hitGround_b2__1_ true) (not (=
   height_b2_1_ 0)))))
(assert (not (and (= hevent_hitGround_b2__2_ true) (not (=
   height_b2_2_ 0)))))
(assert (not (and (= hevent_hitGround_b2__3_ true) (not (=
   height_b2_3_ 0)))))
(assert (not (and (= hevent_hitGround_b1__0_ true) (not (=
   height_b1_0_ 0)))))
(assert (not (and (= hevent_hitGround_b1__1_ true) (not (=
   height_b1_1_ 0)))))
(assert (not (and (= hevent_hitGround_b1__2_ true) (not (=
   height_b1_2_ 0)))))
(assert (not (and (= hevent_hitGround_b1__3_ true) (not (=
   height_b1_3_ 0)))))
(assert (not (and (= hevent_hitGround_b2__0_ true) (= s0_0_ false
   ))))
```

```
(assert (not (and (= hevent_hitGround_b2__1_ true) (= s0_1_ false
    ))))
(assert (not (and (= hevent_hitGround_b2__2_ true) (= s0_2_ false
    ))))
(assert (not (and (= hevent_hitGround_b2__3_ true) (= s0_3_ false
    ))))
(assert (not (and (= hevent_hitGround_b1__0_ true) (= s0_0_ false
    ))))
(assert (not (and (= hevent_hitGround_b1__1_ true) (= s0_1_ false
    ))))
(assert (not (and (= hevent_hitGround_b1__2_ true) (= s0_2_ false
    ))))
(assert (not (and (= hevent_hitGround_b1__3_ true) (= s0_3_ false
    ))))
(assert (not (and (= s0_0_ true) (not (>= height_b2_0_ 0)))))
(assert (not (and (= s0_0_ true) (not (>= height_b1_0_ 0)))))
(assert (not (and (= s0_1_ true) (not (>= height_b2_1_ 0)))))
(assert (not (and (= s0_2_ true) (not (>= height_b2_2_ 0)))))
(assert (not (and (= s0_3_ true) (not (>= height_b2_3_ 0)))))
(assert (not (and (= s0_4_ true) (not (>= height_b2_4_ 0)))))
(assert (not (and (= s0_1_ true) (not (>= height_b1_1_ 0)))))
(assert (not (and (= s0_2_ true) (not (>= height_b1_2_ 0)))))
(assert (not (and (= s0_3_ true) (not (>= height_b1_3_ 0)))))
(assert (not (and (= s0_4_ true) (not (>= height_b1_4_ 0)))))
(assert (not (and (not (= height_b1_1_ 0)) (=
    hevent_hitGround_b1__0_ true))))
(assert (not (and (not (= height_b1_2_ 0)) (=
    hevent_hitGround_b1__1_ true))))
(assert (not (and (not (= height_b1_3_ 0)) (=
    hevent_hitGround_b1__2_ true))))
(assert (not (and (not (= height_b1_4_ 0)) (=
    hevent_hitGround_b1__3_ true))))
(assert (not (and (not (= velocity_b1_1_ (* (/ -8 10)
    velocity_b1_0_))) (= hevent_hitGround_b1__0_ true))))
(assert (not (and (not (= velocity_b1_2_ (* (/ -8 10)
    velocity_b1_1_))) (= hevent_hitGround_b1__1_ true))))
(assert (not (and (not (= velocity_b1_3_ (* (/ -8 10)
    velocity_b1_2_))) (= hevent_hitGround_b1__2_ true))))
(assert (not (and (not (= velocity_b1_4_ (* (/ -8 10)
    velocity_b1_3_))) (= hevent_hitGround_b1__3_ true))))
(assert (not (and (not (= height_b2_1_ 0)) (=
    hevent_hitGround_b2__0_ true))))
(assert (not (and (not (= height_b2_2_ 0)) (=
    hevent_hitGround_b2__1_ true))))
(assert (not (and (not (= height_b2_3_ 0)) (=
```

```
   hevent_hitGround_b2__2_ true))))
(assert (not (and (not (= height_b2_4_ 0)) (=
   hevent_hitGround_b2__3_ true))))
(assert (not (and (not (= velocity_b2_1_ (* (/ -9 10)
   velocity_b2_0_))) (= hevent_hitGround_b2__0_ true))))
(assert (not (and (not (= velocity_b2_2_ (* (/ -9 10)
   velocity_b2_1_))) (= hevent_hitGround_b2__1_ true))))
(assert (not (and (not (= velocity_b2_3_ (* (/ -9 10)
   velocity_b2_2_))) (= hevent_hitGround_b2__2_ true))))
(assert (not (and (not (= velocity_b2_4_ (* (/ -9 10)
   velocity_b2_3_))) (= hevent_hitGround_b2__3_ true))))
(assert (not (and (not (= height_b2_1_ height_b2_0_)) (=
   hevent_hitGround_b1__0_ true))))
(assert (not (and (not (= height_b2_2_ height_b2_1_)) (=
   hevent_hitGround_b1__1_ true))))
(assert (not (and (not (= height_b2_3_ height_b2_2_)) (=
   hevent_hitGround_b1__2_ true))))
(assert (not (and (not (= height_b2_4_ height_b2_3_)) (=
   hevent_hitGround_b1__3_ true))))
(assert (not (and (not (= height_b1_1_ height_b1_0_)) (=
   hevent_hitGround_b2__0_ true))))
(assert (not (and (not (= height_b1_2_ height_b1_1_)) (=
   hevent_hitGround_b2__1_ true))))
(assert (not (and (not (= height_b1_3_ height_b1_2_)) (=
   hevent_hitGround_b2__2_ true))))
(assert (not (and (not (= height_b1_4_ height_b1_3_)) (=
   hevent_hitGround_b2__3_ true))))
(assert (not (and (not (= velocity_b2_1_ velocity_b2_0_)) (=
   hevent_hitGround_b1__0_ true))))
(assert (not (and (not (= velocity_b2_2_ velocity_b2_1_)) (=
   hevent_hitGround_b1__1_ true))))
(assert (not (and (not (= velocity_b2_3_ velocity_b2_2_)) (=
   hevent_hitGround_b1__2_ true))))
(assert (not (and (not (= velocity_b2_4_ velocity_b2_3_)) (=
   hevent_hitGround_b1__3_ true))))
(assert (not (and (not (= velocity_b1_1_ velocity_b1_0_)) (=
   hevent_hitGround_b2__0_ true))))
(assert (not (and (not (= velocity_b1_2_ velocity_b1_1_)) (=
   hevent_hitGround_b2__1_ true))))
(assert (not (and (not (= velocity_b1_3_ velocity_b1_2_)) (=
   hevent_hitGround_b2__2_ true))))
(assert (not (and (not (= velocity_b1_4_ velocity_b1_3_)) (=
   hevent_hitGround_b2__3_ true))))
(assert (not (not (= s0_0_ true))))
(assert (not (not (= height_b1_0_ 2))))
```

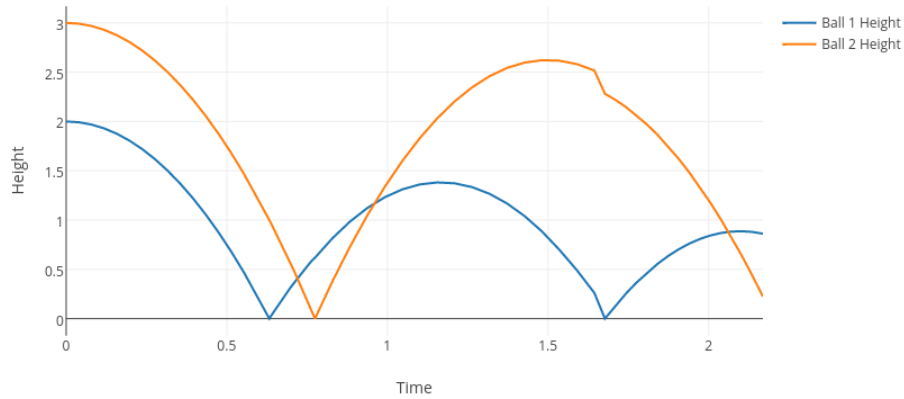

Figure C.1: Output of Example 2

```
(assert (not (not (= height_b2_0_ 3))))
(assert (not (not (= velocity_b1_0_ 0))))
(assert (not (not (= velocity_b2_0_ 0))))
(assert (not (not (= hevent_hitGround_b1__1_ true))))
(assert (not (not (= hevent_hitGround_b2__3_ true))))
(assert (not true))
```

```
(check-sat)
(exit)
```

Output of Example 2 can be seen in Figure C.1.

APPENDIX D

CAR EXAMPLE

As we have already covered the input and output in Chapter 6, we will show only the intermediate ASPMT and SMT translations for Example 3.

Intermediate ASPMT translation

```
:- sorts
astep;step.

:- objects
0..maxstep :: step;
0..maxstep-1 :: astep.

:- constants
astep:wait :: boolean;
0:x :: real[0..30];
astep:x_t :: real[0..30];
0:y :: real[0..30];
astep:y_t :: real[0..30];
0:theta :: real[0..30];
astep:theta_t :: real[0..30];
astep:duration :: action(real[0..30]);
astep:straighten, turnLeft, turnRight :: action;
step:mode :: simpleFluent(real[0..100]).

:- variables
E::events.

%Extra implicit rules
{AS:x_t=X}.
{AS:y_t=X}.
{AS:theta_t=X}.
{AS:straighten=X}.
{AS:turnLeft=X}.
{AS:turnRight=X}.
{AS:duration=X}.
AS:duration = 0 <- AS:straighten.
```

```
AS:duration = 0 <- AS:turnLeft.
AS:duration = 0 <- AS:turnRight.

{AS+1:wait}<-AS:wait.
{AS:wait=true}.
{AS:wait=false} <- AS:straighten.
{AS:wait=false} <- AS:turnLeft.
{AS:wait=false} <- AS:turnRight.

% Rates
d/dt[x](1)=cos(theta).
d/dt[y](1)=sin(theta).
d/dt[theta](1)=0.
d/dt[x](2)=cos(theta).
d/dt[y](2)=sin(theta).
d/dt[theta](2)=tan(3.14/18).
d/dt[x](3)=cos(theta).
d/dt[y](3)=sin(theta).
d/dt[theta](3)=tan(-3.14/18).


%Flow
0:[x_t, y_t, theta_t] =D<- 0:x = D1 & 0:y=X0 & 0:theta=X1 & D =
    int(0,T,[D1,X0,X1],d/dt(1)) & 0:mode=1 & 0:duration = T & 0:
    wait.
AS:[x_t, y_t, theta_t] =D<- AS-1:x_t = D1 & AS-1:y_t=X0 & AS-1:
    theta_t=X1 & D = int(0,T,[D1,X0,X1],d/dt(1)) & AS:mode=1 & AS:
    duration = T & AS:wait.
0:[x_t, y_t, theta_t] =D<- 0:x = D1 & 0:y=X0 & 0:theta=X1 & D =
    int(0,T,[D1,X0,X1],d/dt(2)) & 0:mode=2 & 0:duration = T & 0:
    wait.
AS:[x_t, y_t, theta_t] =D<- AS-1:x_t = D1 & AS-1:y_t=X0 & AS-1:
    theta_t=X1 & D = int(0,T,[D1,X0,X1],d/dt(1)) & AS:mode=2 & AS:
    duration = T & AS:wait.
0:[x_t, y_t, theta_t] =D<- 0:x = D1 & 0:y=X0 & 0:theta=X1 & D =
    int(0,T,[D1,X0,X1],d/dt(2)) & 0:mode=3 & 0:duration = T & 0:
    wait.
AS:[x_t, y_t, theta_t] =D<- AS-1:x_t = D1 & AS-1:y_t=X0 & AS-1:
    theta_t=X1 & D = int(0,T,[D1,X0,X1],d/dt(1)) & AS:mode=3 & AS:
    duration = T & AS:wait.

%Invariant
<- not (AS:x_t=X & AS:y_t=Y & ((X-6)*(X-6) & (Y)*(Y)>9)).
always_t (AS:x_t=X & AS:y_t=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) & AS:
    mode=1 & AS:duration=D.
```

```
always_t (AS:x_t=X & AS:y_t=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) & AS:
    mode=2 & AS:duration=D.
always_t (AS:x_t=X & AS:y_t=Y & ((X-6)*(X-6) & (Y)*(Y)>9)) & AS:
    mode=3 & AS:duration=D.

<- not (AS:x_t=X & AS:y_t=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)).
always_t (AS:x_t=X & AS:y_t=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) &
    AS:mode=1 & AS:duration=D.
always_t (AS:x_t=X & AS:y_t=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) &
    AS:mode=2 & AS:duration=D.
always_t (AS:x_t=X & AS:y_t=Y & ((X-5)*(X-5) & (Y-7)*(Y-7)>4)) &
    AS:mode=3 & AS:duration=D.

<- not (AS:x_t=X & AS:y_t=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4)).
always_t (AS:x_t=X & AS:y_t=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4))
    & AS:mode=1 & AS:duration=D.
always_t (AS:x_t=X & AS:y_t=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>4))
    & AS:mode=2 & AS:duration=D.
always_t (AS:x_t=X & AS:y_t=Y & ((X-12)*(X-12) & (Y-9)*(Y-9)>9))
    & AS:mode=3 & AS:duration=D.

AS+1:mode=1<-AS:straighten.
AS+1:mode=2<-AS:turnLeft.
AS+1:mode=3<-AS:turnRight.

%State Restrictions
<- AS:straighten & AS:mode=1.
<- AS:turnLeft & AS:mode=2.
<- AS:turnRight & AS:mode=3.

%Reset
<- (RP=D & X2=X0 & D1=X2) & 0:x_t = RP & 0:y_t=X2 & 0:theta_t=D1
     & 0:x = D & 0:y=X0 & 0:theta=X1 & 0:turnLeft.
<- (RP=D & X2=X0 & D1=X2) & AS:x_t = RP & AS:y_t=X2 & AS:theta_t=
    D1  & AS-1:x = D & AS-1:y=X0 & AS-1:theta=X1 & AS:turnLeft.
<- (RP=D & X2=X0 & D1=X2) & 0:x_t = RP & 0:y_t=X2 & 0:theta_t=D1
    & 0:x = D & 0:y=X0 & 0:theta=X1 & 0:turnRight.
<- (RP=D & X2=X0 & D1=X2) & AS:x_t = RP & AS:y_t=X2 & AS:theta_t=
    D1  & AS-1:x = D & AS-1:y=X0 & AS-1:theta=X1 & AS:turnRight.
<- (RP=D & X2=X0 & D1=X2) & 0:x_t = RP & 0:y_t=X2 & 0:theta_t=D1
    & 0:x = D & 0:y=X0 & 0:theta=X1 & 0:straighten.
<- (RP=D & X2=X0 & D1=X2) & AS:x_t = RP & AS:y_t=X2 & AS:theta_t=
    D1 & AS-1:x = D & AS-1:y=X0 & AS-1:theta=X1 & AS:straighten.

0:x=0;
```

```
0:y=0;
0:theta=0.698132;
0:mode=1;
3:x=13;
3:y=0.
```

Final SMT Encoding

```
(set-logic QF_NRA_ODE)
(declare-const true_a Bool)
(declare-const false_a Bool)
(declare-const theta  Real)
(declare-const theta_0  Real)
(declare-const theta_0_t  Real)
(declare-const theta_1_t  Real)
(declare-const theta_2_t  Real)
(declare-const x  Real)
(declare-const x_0  Real)
(declare-const x_0_t  Real)
(declare-const x_1_t  Real)
(declare-const x_2_t  Real)
(declare-const duration_0  Real)
(declare-const duration_1  Real)
(declare-const duration_2  Real)
(declare-const turnLeft_0_ Bool)
(declare-const turnLeft_1_ Bool)
(declare-const turnLeft_2_ Bool)
(declare-const turnRight_0_ Bool)
(declare-const turnRight_1_ Bool)
(declare-const turnRight_2_ Bool)
(declare-const straighten_0_ Bool)
(declare-const straighten_1_ Bool)
(declare-const straighten_2_ Bool)
(declare-const qlabel_init_ Bool)
(declare-const mode_0_ Real)
(declare-const mode_1_ Real)
(declare-const mode_2_ Real)
(declare-const mode_3_ Real)
(declare-const y  Real)
(declare-const y_0  Real)
(declare-const y_0_t  Real)
(declare-const y_1_t  Real)
(declare-const y_2_t  Real)
(declare-const wait_0_ Bool)
(declare-const wait_1_ Bool)
```

```
(declare-const wait_2_ Bool)

; Flow declaration(must implicitly convert to single line
   declarations)
; movingStraight state
; d/dt[x](movingStraight)=cos(theta).
; d/dt[y](movingStraight)=sin(theta).
; d/dt[theta](movingStraight)=0.
(define-ode flow_1 ((= d/dt[x] (cos theta))(= d/dt[y] (sin theta)
   )(= d/dt[theta] 0)))

; turningLeft state
;d/dt[x](turningLeft)=cos(theta).
;d/dt[y](turningLeft)=sin(theta).
;d/dt[theta](turningLeft)=tan(pi/8).
(define-ode flow_2 ((= d/dt[x] (cos theta))(= d/dt[y] (sin theta)
   )(= d/dt[theta] (tan 0.226893))))

; turningRight state
;d/dt[x](turningRight)=cos(theta).
;d/dt[y](turningRight)=sin(theta).
;d/dt[theta](turningRight)=tan(pi/8).
(define-ode flow_3 ((= d/dt[x] (cos theta))(= d/dt[y] (sin theta)
   )(= d/dt[theta] (tan -0.226893))))

(assert true_a)
(assert (not false_a))
(assert (>= theta_0 -50))
(assert (<= theta_0 50))
(assert (>= theta_0_t -50))
(assert (<= theta_0_t 50))
(assert (>= theta_1_t -50))
(assert (<= theta_1_t 50))
(assert (>= theta_2_t -50))
(assert (<= theta_2_t 50))
(assert (>= x_0 -40))
(assert (<= x_0 40))
(assert (>= x_0_t -40))
(assert (<= x_0_t 40))
(assert (>= x_1_t -40))
(assert (<= x_1_t 40))
(assert (>= x_2_t -40))
(assert (<= x_2_t 40))
(assert (>= duration_0 0))
(assert (<= duration_0 40))
```

```
(assert (>= duration_1 0))
(assert (<= duration_1 40))
(assert (>= duration_2 0))
(assert (<= duration_2 40))
(assert (>= y_0 -50))
(assert (<= y_0 50))
(assert (>= y_0_t -50))
(assert (<= y_0_t 50))
(assert (>= y_1_t -50))
(assert (<= y_1_t 50))
(assert (>= y_2_t -50))
(assert (<= y_2_t 50))

; Completion for wait
(assert (or (or (or (and (= wait_2_ true) (= wait_2_ true)) (and
   (= straighten_2_ true) (= wait_2_ false))) (and (= turnLeft_2_
    true) (= wait_1_ false)))(and (= turnRight_2_ true) (=
   wait_2_ false))))
(assert (or (or (or (and (= wait_1_ true) (= wait_1_ true)) (and
   (= straighten_1_ true) (= wait_1_ false))) (and (= turnLeft_1_
    true) (= wait_1_ false)))(and (= turnRight_1_ true) (=
   wait_1_ false))))
(assert (or (or (or (and (= wait_0_ true) (= wait_0_ true)) (and
   (= straighten_0_ true) (= wait_0_ false))) (and (= turnLeft_0_
    true) (= wait_0_ false)))(and (= turnRight_0_ true) (=
   wait_0_ false))))

; default wait=true
(assert (=> (= wait_2_ true) (= wait_2_ true)))
(assert (=> (= wait_1_ true) (= wait_1_ true)))
(assert (=> (= wait_0_ true) (= wait_0_ true)))

; hevent(E) causes wait=false
(assert (=> (= straighten_2_ true) (= wait_2_ false)))
(assert (=> (= turnLeft_2_ true) (= wait_2_ false)))
(assert (=> (= turnRight_2_ true) (= wait_2_ false)))
(assert (=> (= straighten_1_ true) (= wait_1_ false)))
(assert (=> (= turnLeft_1_ true) (= wait_1_ false)))
(assert (=> (= turnRight_1_ true) (= wait_1_ false)))
(assert (=> (= straighten_0_ true) (= wait_0_ false)))
(assert (=> (= turnLeft_0_ true) (= wait_0_ false)))
(assert (=> (= turnRight_0_ true) (= wait_0_ false)))

; Completion for mode
(assert (or (or (or (= mode_3_ mode_2_) (and (= straighten_2_
```

```
   true) (= mode_3_ 1)))(and (= turnLeft_2_ true) (= mode_3_ 2)))
   (and (= turnRight_2_ true) (= mode_3_ 3))))
(assert (or (or (or (= mode_2_ mode_1_) (and (= straighten_1_
   true) (= mode_2_ 1)))(and (= turnLeft_1_ true) (= mode_2_ 2)))
   (and (= turnRight_1_ true) (= mode_2_ 3))))
(assert (or (or (or (= mode_1_ mode_0_) (and (= straighten_0_
   true) (= mode_1_ 1)))(and (= turnLeft_0_ true) (= mode_1_ 2)))
   (and (= turnRight_0_ true) (= mode_1_ 3))))

; hevent(straighten) causes mode=1.
; hevent(turnLeft) causes mode=2.
; hevent(turnRight) causes mode=3.
(assert (=> (= straighten_2_ true) (= mode_3_ 1)))
(assert (=> (= turnLeft_2_ true) (= mode_3_ 2)))
(assert (=> (= turnRight_2_ true) (= mode_3_ 3)))
(assert (=> (= straighten_1_ true) (= mode_2_ 1)))
(assert (=> (= turnLeft_1_ true) (= mode_2_ 2)))
(assert (=> (= turnRight_1_ true) (= mode_2_ 3)))
(assert (=> (= straighten_0_ true) (= mode_1_ 1)))
(assert (=> (= turnLeft_0_ true) (= mode_1_ 2)))
(assert (=> (= turnRight_0_ true) (= mode_1_ 3)))

; label:init
(assert (= qlabel_init_ true))
(assert (= qlabel_init_ true))

; hevent(E) causes duration=0
(assert (=> (= straighten_2_ true) (= duration_2 0)))
(assert (=> (= turnLeft_2_ true) (= duration_2 0)))
(assert (=> (= turnRight_2_ true) (= duration_2 0)))
(assert (=> (= straighten_1_ true) (= duration_1 0)))
(assert (=> (= turnLeft_1_ true) (= duration_1 0)))
(assert (=> (= turnRight_1_ true) (= duration_1 0)))
(assert (=> (= straighten_0_ true) (= duration_0 0)))
(assert (=> (= turnLeft_0_ true) (= duration_0 0)))
(assert (=> (= turnRight_0_ true) (= duration_0 0)))

(assert true_a)

; Flow(turningLeft).
(assert (=> (and (= mode_0_ 2) (= wait_0_ true)) (= [x_0_t y_0_t
   theta_0_t] (integral 0. duration_0 [x_0 y_0 theta_0] flow_2)))
   )
(assert (=> (and (= mode_1_ 2) (= wait_1_ true)) (= [x_1_t y_1_t
   theta_1_t] (integral 0. duration_1 [x_0_t y_0_t theta_0_t]
```

```
   flow_2))))
(assert (=> (and (= mode_2_ 2) (= wait_2_ true)) (= [x_2_t y_2_t
   theta_2_t] (integral 0. duration_2 [x_1_t y_1_t theta_1_t]
   flow_2))))

; Flow(movingStraight).
(assert (=> (and (= mode_0_ 1) (= wait_0_ true)) (= [x_0_t y_0_t
   theta_0_t] (integral 0. duration_0 [x_0 y_0 theta_0] flow_1)))
   )
(assert (=> (and (= mode_1_ 1) (= wait_1_ true)) (= [x_1_t y_1_t
   theta_1_t] (integral 0. duration_1 [x_0_t y_0_t theta_0_t]
   flow_1))))
(assert (=> (and (= mode_2_ 1) (= wait_2_ true)) (= [x_2_t y_2_t
   theta_2_t] (integral 0. duration_2 [x_1_t y_1_t theta_1_t]
   flow_1))))

; Flow(turningRight).
(assert (=> (and (= mode_0_ 3) (= wait_0_ true)) (= [x_0_t y_0_t
   theta_0_t] (integral 0. duration_0 [x_0 y_0 theta_0] flow_3)))
   )
(assert (=> (and (= mode_1_ 3) (= wait_1_ true)) (= [x_1_t y_1_t
   theta_1_t] (integral 0. duration_1 [x_0_t y_0_t theta_0_t]
   flow_3))))
(assert (=> (and (= mode_2_ 3) (= wait_2_ true)) (= [x_2_t y_2_t
   theta_2_t] (integral 0. duration_2 [x_1_t y_1_t theta_1_t]
   flow_3))))

(assert (> (+ (* (- x_0 9)(- x_0 9))(* y_0 y_0)) 9))
(assert (> (+ (* (- x_0_t 9)(- x_0_t 9))(* y_0_t y_0_t)) 9))
(assert (> (+ (* (- x_1_t 9)(- x_1_t 9))(* y_1_t y_1_t)) 9))
(assert (> (+ (* (- x_2_t 9)(- x_2_t 9))(* y_2_t y_2_t)) 9))

(assert (forall_t 1 [0 duration_0] (> (+ (* (- x_0_t 9)(- x_0_t
   9))(* y_0_t y_0_t)) 9)))
(assert (forall_t 1 [0 duration_1] (> (+ (* (- x_1_t 9)(- x_1_t
   9))(* y_1_t y_1_t)) 9)))
(assert (forall_t 1 [0 duration_2] (> (+ (* (- x_2_t 9)(- x_2_t
   9))(* y_2_t y_2_t)) 9)))

(assert (forall_t 2 [0 duration_0] (> (+ (* (- x_0_t 9)(- x_0_t
   9))(* y_0_t y_0_t)) 9)))
(assert (forall_t 2 [0 duration_1] (> (+ (* (- x_1_t 9)(- x_1_t
   9))(* y_1_t y_1_t)) 9)))
(assert (forall_t 2 [0 duration_2] (> (+ (* (- x_2_t 9)(- x_2_t
   9))(* y_2_t y_2_t)) 9)))
```

```
(assert (forall_t 3 [0 duration_0] (> (+ (* (- x_0_t 9)(- x_0_t
   9))(* y_0_t y_0_t)) 9)))
(assert (forall_t 3 [0 duration_1] (> (+ (* (- x_1_t 9)(- x_1_t
   9))(* y_1_t y_1_t)) 9)))
(assert (forall_t 3 [0 duration_2] (> (+ (* (- x_2_t 9)(- x_2_t
   9))(* y_2_t y_2_t)) 9)))

(assert (> (+ (* (- x_0 5)(- x_0 5))(* (- y_0 7) (- y_0 7))) 4))
(assert (> (+ (* (- x_0_t 5)(- x_0_t 5))(* (- y_0_t 7) (- y_0_t
   7))) 4))
(assert (> (+ (* (- x_1_t 5)(- x_1_t 5))(* (- y_1_t 7) (- y_1_t
   7))) 4))
(assert (> (+ (* (- x_2_t 5)(- x_2_t 5))(* (- y_2_t 7) (- y_2_t
   7))) 4))

(assert (forall_t 1 [0 duration_0] (> (+ (* (- x_0_t 5)(- x_0_t
   5))(* (- y_0_t 7) (- y_0_t 7))) 4)))
(assert (forall_t 1 [0 duration_1] (> (+ (* (- x_1_t 5)(- x_1_t
   5))(* (- y_1_t 7) (- y_1_t 7))) 4)))
(assert (forall_t 1 [0 duration_2] (> (+ (* (- x_2_t 5)(- x_2_t
   5))(* (- y_2_t 7) (- y_2_t 7))) 4)))

(assert (forall_t 2 [0 duration_0] (> (+ (* (- x_0_t 5)(- x_0_t
   5))(* (- y_0_t 7) (- y_0_t 7))) 4)))
(assert (forall_t 2 [0 duration_1] (> (+ (* (- x_1_t 5)(- x_1_t
   5))(* (- y_1_t 7) (- y_1_t 7))) 4)))
(assert (forall_t 2 [0 duration_2] (> (+ (* (- x_2_t 5)(- x_2_t
   5))(* (- y_2_t 7) (- y_2_t 7))) 4)))

(assert (forall_t 3 [0 duration_0] (> (+ (* (- x_0_t 5)(- x_0_t
   5))(* (- y_0_t 7) (- y_0_t 7))) 4)))
(assert (forall_t 3 [0 duration_1] (> (+ (* (- x_1_t 5)(- x_1_t
   5))(* (- y_1_t 7) (- y_1_t 7))) 4)))
(assert (forall_t 3 [0 duration_2] (> (+ (* (- x_2_t 5)(- x_2_t
   5))(* (- y_2_t 7) (- y_2_t 7))) 4)))

(assert (> (+ (* (- x_0 12)(- x_0 12))(* (- y_0 9) (- y_0 9))) 4)
   )
(assert (> (+ (* (- x_0_t 12)(- x_0_t 12))(* (- y_0_t 9) (- y_0_t
    9))) 4))
(assert (> (+ (* (- x_1_t 12)(- x_1_t 12))(* (- y_1_t 9) (- y_1_t
    9))) 4))
(assert (> (+ (* (- x_2_t 12)(- x_2_t 12))(* (- y_2_t 9) (- y_2_t
    9))) 4))
```

```
(assert (forall_t 1 [0 duration_0] (> (+ (* (- x_0_t 12)(- x_0_t
   12))(* (- y_0_t 9) (- y_0_t 9))) 4)))
(assert (forall_t 1 [0 duration_1] (> (+ (* (- x_1_t 12)(- x_1_t
   12))(* (- y_1_t 9) (- y_1_t 9))) 4)))
(assert (forall_t 1 [0 duration_2] (> (+ (* (- x_2_t 12)(- x_2_t
   12))(* (- y_2_t 9) (- y_2_t 9))) 4)))

(assert (forall_t 2 [0 duration_0] (> (+ (* (- x_0_t 12)(- x_0_t
   12))(* (- y_0_t 9) (- y_0_t 9))) 4)))
(assert (forall_t 2 [0 duration_1] (> (+ (* (- x_1_t 12)(- x_1_t
   12))(* (- y_1_t 9) (- y_1_t 9))) 4)))
(assert (forall_t 2 [0 duration_2] (> (+ (* (- x_2_t 12)(- x_2_t
   12))(* (- y_2_t 9) (- y_2_t 9))) 4)))

(assert (forall_t 3 [0 duration_0] (> (+ (* (- x_0_t 12)(- x_0_t
   12))(* (- y_0_t 9) (- y_0_t 9))) 4)))
(assert (forall_t 3 [0 duration_1] (> (+ (* (- x_1_t 12)(- x_1_t
   12))(* (- y_1_t 9) (- y_1_t 9))) 4)))
(assert (forall_t 3 [0 duration_2] (> (+ (* (- x_2_t 12)(- x_2_t
   12))(* (- y_2_t 9) (- y_2_t 9))) 4)))

; nonexecutable hevent(straighten) if mode=1.
; nonexecutable hevent(turnLeft) if mode=2.
; nonexecutable hevent(turnRight) if mode=3.
(assert (not (and (= straighten_2_ true) (= mode_2_ 1))))
(assert (not (and (= straighten_1_ true) (= mode_1_ 1))))
(assert (not (and (= straighten_0_ true) (= mode_0_ 1))))
(assert (not (and (= turnLeft_2_ true) (= mode_2_ 2))))
(assert (not (and (= turnLeft_1_ true) (= mode_1_ 2))))
(assert (not (and (= turnLeft_0_ true) (= mode_0_ 2))))
(assert (not (and (= turnLeft_2_ true) (= mode_2_ 3))))
(assert (not (and (= turnLeft_1_ true) (= mode_1_ 3))))
(assert (not (and (= turnLeft_0_ true) (= mode_0_ 3))))

; Reset
; constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
   after hevent(turnLeft) & x = D & y=X0 & theta=X1.
; constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
   after hevent(turnRight) & x = D & y=X0 & theta=X1.
; constraint (RP=D & X2=X0 & D1=X2) & x = RP & y=X2 & theta=D1
   after hevent(straighten) & x = D & y=X0 & theta=X1.
(assert (not (and (not (= x_2_t x_1_t)) (= turnLeft_2_ true))))
(assert (not (and (not (= x_1_t x_0_t)) (= turnLeft_1_ true))))
(assert (not (and (not (= x_0_t x_0)) (= turnLeft_0_ true))))
```

```
(assert (not (and (not (= y_2_t y_1_t)) (= turnLeft_2_ true))))
(assert (not (and (not (= y_1_t y_0_t)) (= turnLeft_1_ true))))
(assert (not (and (not (= y_0_t y_0)) (= turnLeft_0_ true))))
(assert (not (and (not (= theta_2_t theta_1_t)) (= turnLeft_2_
   true))))
(assert (not (and (not (= theta_1_t theta_0_t)) (= turnLeft_1_
   true))))
(assert (not (and (not (= theta_0_t theta_0)) (= turnLeft_0_ true
   ))))
(assert (not (and (not (= x_2_t x_1_t)) (= turnRight_2_ true))))
(assert (not (and (not (= x_1_t x_0_t)) (= turnRight_1_ true))))
(assert (not (and (not (= x_0_t x_0)) (= turnRight_0_ true))))
(assert (not (and (not (= y_2_t y_1_t)) (= turnRight_2_ true))))
(assert (not (and (not (= y_1_t y_0_t)) (= turnRight_1_ true))))
(assert (not (and (not (= y_0_t y_0)) (= turnRight_0_ true))))
(assert (not (and (not (= theta_2_t theta_1_t)) (= turnRight_2_
   true))))
(assert (not (and (not (= theta_1_t theta_0_t)) (= turnRight_1_
   true))))
(assert (not (and (not (= theta_0_t theta_0)) (= turnRight_0_
   true))))
(assert (not (and (not (= x_2_t x_1_t)) (= straighten_2_ true))))
(assert (not (and (not (= x_1_t x_0_t)) (= straighten_1_ true))))
(assert (not (and (not (= x_0_t x_0)) (= straighten_0_ true))))
(assert (not (and (not (= y_2_t y_1_t)) (= straighten_2_ true))))
(assert (not (and (not (= y_1_t y_0_t)) (= straighten_1_ true))))
(assert (not (and (not (= y_0_t y_0)) (= straighten_0_ true))))
(assert (not (and (not (= theta_2_t theta_1_t)) (= straighten_2_
   true))))
(assert (not (and (not (= theta_1_t theta_0_t)) (= straighten_1_
   true))))
(assert (not (and (not (= theta_0_t theta_0)) (= straighten_0_
   true))))

; Query
(assert (= x_0 0))
(assert (= y_0 0))
(assert (= mode_0_ 1))

(assert (not (not (= x_2_t 13))))
(assert (not (not (= y_2_t 0))))


(check-sat)
(exit)
```