联 锁 系 统

学院：信息技术学院

学号：1701111297

姓名：崔宏伟

指导老师：金芝

时间：2017 年 12 月

目 录

一． 系统设置的非形式化描述

本项目是为铁路系统中列车之间的作业指挥设计合适的联锁系统。联锁系统需要包括联锁关系，建立进路，控制道岔的转换和信号灯的转变以及进路解锁，以保证行车安全。

1. 实体描述

系统中具体包括了轨道区段，道岔，信号灯，火车，总的控制中心和联锁表六个实体。轨道区段是指实际的火车轨道，每一段轨段包含轨道电路，并且可以通过传感器探测列车是否在轨道上。

道岔是一列车从一段轨道到另一段轨道之间的通道。在实际过程中，列车进入轨道之前需要锁住相应道岔，列车离开之后道岔才会被解锁。信号灯是指轨道上放置的红绿灯，通过发出信号来指挥列车移动。火车是实际的一列火车，可以发出请求进入轨道的信号和火车已经进入轨道的信号。

控制中心是指可以接受轨道，列车，信号灯，道岔发出的信号，同时负责列车进出轨道的控制器。联锁表是用于确定每一趟列车走哪条进路，开哪些灯，开哪些道岔。

2. 系统的运行场景

本项目设计的联锁系统主要运行在两种实际情况：（1）简单的实际场景，包括一列火车，一个轨道，一个信号灯，一个道岔；（2）复杂的实际场景，包括两列火车，五个轨道，六个信号灯，两个道岔。同时在第一种情况下只需要控制中心即可，第二种情况还需要联锁表指明具体的行路顺序。第二种情况通过假设，设计了实际的铁路图，具体如图 1 所示。图中 S1-S6 代表信号灯，T1-T4 代表轨道，SW1-SW2 代表道岔。

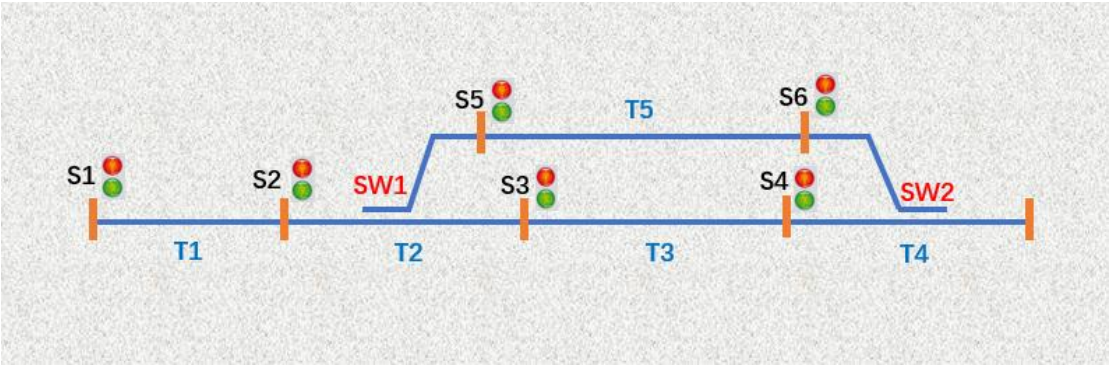


图 1 复杂情况下的假设的铁路图

假设实际的列车进入轨道的运行过程简化为：申请进入轨道->检测轨道是否被占用->如果未被占用点亮绿灯->列车进入轨道->列车离开轨道->轨道回归到未被占用状态->绿灯变为红灯。

二． 系统设置的形式化描述

对于非形式化描述的系统设置，通过确定各个实体的具体属性和符号表示，限制约束等来给出系统的形式化描述。

1. 简单情况下，实体的形式化表示

对于实体火车，符号描述为 Train。每个 Train 能够发出 trainRequest 请求进入轨道的信号，同时能够检测到当前红绿灯的状态，只有当红绿灯处于绿灯 green 时，Train 才能够进入轨道。Train 在进入轨道之后可以向控制系统发出 trainEnter 已经进入的信号和 trainLeave 离开轨道的信号。对于 Train 的限制约束为：在时间上 Train 发出 trainRequest 信号之后到接受红绿灯变为 green 的信号之间的间隔 light_time 应该为 21，同时 Train 在轨道上运行的时间 run_time 应该等于 150，从而保证 Train 不会刚进入 Track 就离开。

对于实体红绿灯，符号描述为 Light。每个 Light 在系统开始时均为红灯，使用变量 light_is 来确定当前红绿灯的状态，即初始化时 light_is=red_on（宏定义）。Light 在系统运行过程中能够接受控制器发出的变绿 dogreen 和变红 dored 的信号，同时在 Light 状态改变之后，一方面更新 light_is 的状态，另一方面发出 green 和 red 的信号来提示火车可以进入。

对于实体轨道，符号描述为 Track。每个 Track 带有表示 Track 有无被占用的变量 y，初始化时 y=UNOCCUPIED，表示未被占用。在系统运行过程中，Track 能够检测到控制中心发出的检测 Track 是否被占用的信号 checkoccupied，通过检测变量 y 来判断当前 Track 有无被占用，同时能够发出占用 occupied 和未占用 unoccupied 的信号。如果 Train 进入 Track，变量 y 是自动设置为 OCCUPIED。

对于实体道岔，符号描述为 Point。每个道岔 Point 初始化时都是未被锁，在设计中使用 point_is_lock=UNLOCK 表示未被锁。在系统运行过程中，Point 能够接受锁道岔 dolock 信号和解锁道岔 dounlock 信号，同时根据接受到的信号的不同，设置 point_is_lock 是 LOCK 或者 UNLOCK。

对于实体控制中心，符号描述为 Control_Center。控制中心用于控制联锁系统中各个实体的状态变化，因此可以收到每个实体发出的信号，也能够发出改变实体的信号。在系统运行过程中，Control_Center 首先接受 trainRequest 火车申请进入的信号，然后发出 checkoccupied 信号，检查 Track 是否被占用，如果被占用则回到初始状态，等待新的 trainRequest 信号。如果未被占用，表示火车可以进入，准备 Train 进入之前的工作。首先发出 dolock 信号，锁住 Point；当 Point 被锁住之后，发出 dogreen 信号，准备亮绿灯；当绿灯亮起之后，火车开始进入 Track。之后当 Train 发出 trainLeave 信号，离开 Track 时，控制中心开始恢复

Track，Point，Light 的状态，首先解锁 Point，然后红绿灯变为 red_on，最后回到初始状态。为了保证系统运行具有一定的真实性和有效性，Control_Center 需要满足部分时间约束：

（1） Control_Center 发出 checkoccupied 信号到根据 occupied 和 unoccupied 信号执行状态转移之间的时间间隔应该小于 4s；

（2） Control_Center 发出 dolock 信号到收到 Point 已经被锁的消息之间的间隔应该小于 4s；

（3） Control_Center 发出 dogreen 信号到检测到红绿灯变为绿灯之间的时间应小于 4s；

（4） Control_Center 从收到 light_is==green_on 到收到 Train 发出的 train Enter 信号之间应该小于 10s；

（5） Control_Center 从收到 trainEnter 信号到收到 Train 发出 trainLeave 的信号之间应该小于 200s；

（6） Control_Center 从发出解锁 Point 的 dounlock 信号到收到 Point 已经处于解锁状态之间的时间间隔应该小于 4s；

（7） Control_Center 从发出 Light 变为红灯的 dored 信号到收到 light_is==red_on 之间的时间间隔应该小于 4s；

2. 复杂情况下，实体的具体表示

复杂情况是指系统中的实体的个数变多，由简单的一个 Track，一个 Train，一个 Light，一个 Point 丰富为两列火车，五个轨道，六个信号灯和两个道岔。同时还增加了联锁表（进路表）来指名具体的在多列火车的情况下的道路选择，并且控制中心的逻辑以需要发生细微的变化。相对于简单情况的实体表示，复杂情况下的表示有以下几点变化：

（1） 为了表示出相同实体之间不同的类型，需要增加辅助变量和信号的个数，属性的个数。辅助变量用于控制当前具体是哪一个实体在实际运作。同时需要同一个实体的设置多个信号和属性，用于表示不同的个体的不同状态。具体的操作是将信号变为信号数组（trainEnter->trainEnter[2]），变量变为变量数组（track_y->track_y[5]）。

（2） 增加联锁表，确定每列火车在运行过程中，走哪条路，开哪些灯，开哪些道岔。根据联锁表，需要能够在系统运行过程中，指派当前应该运行的列车和正在进入的 Track，Light 和 Point，当一个过程结束之后指派下一次的系统参数。具体的对应于图 1 的联锁表为表 1 所示。

（3） 修改控制中心的部分逻辑。考虑到图 1 中的设计的情景，部分 Track 并不存在道岔，因此需要考虑到当前系统运行的情形中是否存在道岔

Point。通过增加参数 use_point==NO_POINT/POINT 来判断有无道岔，如果则与之前一致，否则会跳过锁 Point 和解锁 Point 的两个阶段。

Routes			Signal Lights		Points		Tracks
ID	From	To	Green	Red	Open		
					Up	Down	
R1	S1-S6		S1,S2,S5,S6	S3,S4	SW1,SW2		T1,T2,T5,T4
R2	S1-S4		S1,S2,S3,S4	S5,S6		SW1,SW2	T1,T2,T3,T4

表 1 复杂假设下的联锁表

3. 系统中的参数设置和取值范围

（1） 时间约束参数

时间参数	参数含义
light_time	Control_Center 收到 request 消息到列车接收到交通灯回馈的消息之间的时间为 21s
run_time	火车运行时间
check_track_time	Control_Center 发出 checkoccupied 消息到收到 Control_Center 发出反馈消息之间的时间少于 4s
lock_time	Control_Center 发出 dolock 消息到收到 locked 消息之间的时间少于 4s
green_open_time	Control_Center 发出 dogreen 消息到收到 green 消息时间之间的时间少于 4s
trainenter_time	Control_Center 收到 green 消息到收到 trainEnter 消息之间的时间少于 10s
train_run_time	Control_Center 收到 trainEnter 消息到收到 trainLeave 消息之间的时间少于 200s
unlock_time	Control_Center 发出 dounlock 消息到收到 unlocked 消息之间的时间少于 4s
red_open_time	Control_Center 发出 dored 消息到收到 red 消息之间的时间少于 4s

（2） 复杂情况下的辅助参数

参数名称	参数用处
train_n	指明第几辆火车
track_n	指明第几节轨道
light_n	指明第几个信号灯
point_n	指明第几个道岔
use_point	指明需不需要判断道岔
next_step(chan)	用于同步火车经过一次轨道结束到下次火车的开始调度进入

（3） 宏定义参数

宏定义	含义	宏定义	含义	适用变量
red_on	当前红灯亮	green_on	当前绿灯亮	light_is
NO_POINT	轨道无道岔	POINT	轨道有道岔	use_point
OCCUPIED	轨道被占用	UNOCCUPIED	轨道未占用	track_y
LOCK	道岔已锁	UNLOCK	道岔未锁	point_is_lock

三． 系统成分及其自动机建模

1. 实体 Train

（1） 变迁系统模型

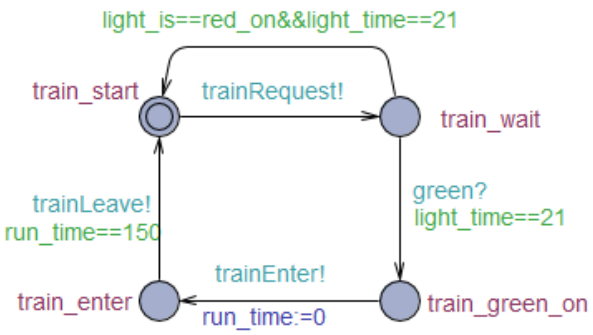
对于实体 Train，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

S	状态的集合，S={train_start, train_wait, train_green_on, train_enter}
I	初始状态集合，I={train_start}
Act	Act={send_enter_request, back_to_start, receive_green_light, train_enter, train_ leave}
AP	原子命题集合，AP={light_is=red_on, light_is=green_on}
L	L(train_start)={light_is=red_on}，L(train_green_on)={light_is=green_on}，L(train_wait) = L(train_start)，L(train_enter)=L(train_green_on)

变迁关系->：

$$train\_start \xrightarrow{send\_enter\_request} train\_wait$$
$$train\_wait \xrightarrow{back\_to\_start} train\_start$$
$$train\_wait \xrightarrow{receive\_green\_light} train\_green\_on$$
$$train\_green\_on \xrightarrow{train\_enter} train\_enter$$
$$train\_enter \xrightarrow{train\_leave} train\_start$$

（2） Uppaal 的表示



Train 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
send_enter_request	trainRequest! ：发送 Train 进入 Track 的请求信号
back_to_start	light_is==red_on&&light_time==21 ：当前信号灯为红色并且要求此时 light_time 为 21s
receive_green _light	<light_time==21><green?> ：light_time 满足为 21s 并且接收到信号灯变为绿的信号
train_enter	<trainEnter!><runtime=0> ：发出 Train 进入 Track 的信号，并且将 train 运行时间初始化为 0
train_leave	<trainLeave!><run_time==150> ：发出 Train 离开 Track 的信号，并且满足了运行时间为 150s

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
Train_start	此状态表示火车准备好，准备发出请求进入 Track 的信号
Train_wait	此状态表示火车等待接受信号灯的信号
Train_green_on	此状态表示火车接收到了信号灯为绿色的信号，准备进入轨道
Train_enter	此状态表示火车已经进入轨道，准备运行之后离开轨道

（4） 非形式化的变迁说明

$train\_start \xrightarrow{send\_enter\_request} train\_wait$

火车初始处于 start 状态，然后发送进入 track 的请求信号，从而进入到 train_wait 的信号

$train\_wait \xrightarrow{back\_to\_start} train\_start$

如果火车在 train_wait 状态时，并且满足在 light_time 到达 21s 时信号灯仍旧为红色，则回到初始状态，重新发送请求信号

$train\_wait \xrightarrow{receive\_green\_light} train\_green\_on$

如果火车在 train_wait 状态时，并且满足在 light_time 到达 21s 时信号灯已经变为绿色，则进入到状态 train_green_on 状态，准备进入 track

$train\_green\_on \xrightarrow{train\_enter} train\_enter$.

当火车处于 train_green_on 状态时，发出进入轨道的信号 trainEnter，进入状态 train_enter 状态

$train\_enter \xrightarrow{train\_leave} train\_start$

当火车处于 train_enter 状态时，发出离开轨道的信号，并且此时火车的运行时间已经达到 150s，火车重新回到初始状态

2. 实体 Track

（1） 变迁系统模型

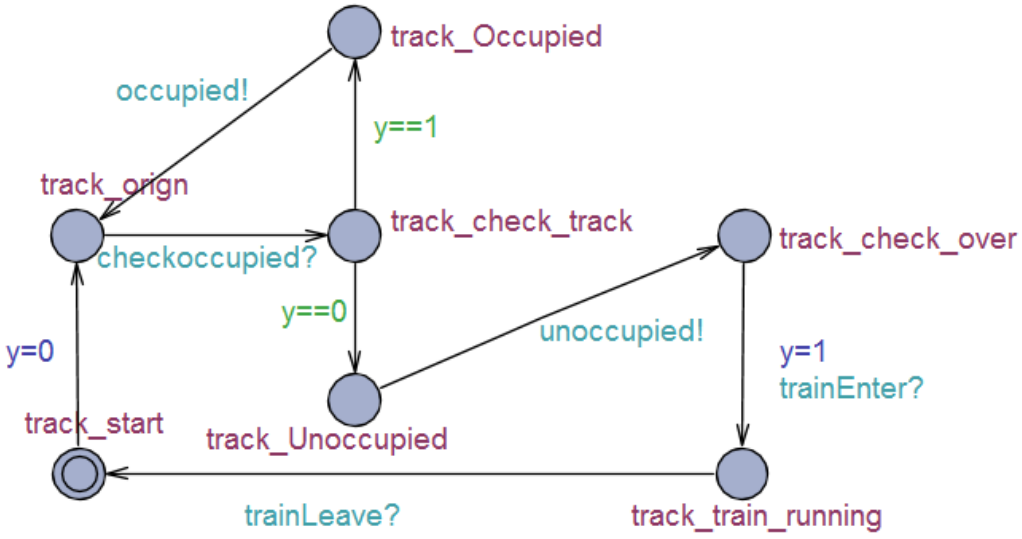
对于实体 Track，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

S	状态的集合，S={track_start, track_orign, track_check, track_Occupied, track_Unoccupied, track_check_over, track_train_running}
I	初始状态集合，I={track_start}
Act	Act={track_init, receive_signal, judge_y_1, judge_y_0, send_occupied, send_ unoccupied, change_y, back_orign}
AP	原子命题集合，AP={y=OCCUPIED, y=UNOCCUPIED }
L	L(track_orign)=L(track_Unoccupied)={y= UNOCCUPIED }, L(track_Occupied)={y= OCCUPIED }, L(else) =∅

变迁关系->：

$$track\_start \xrightarrow{track\_init} track\_start$$
$$track\_orign \xrightarrow{receive\_signal} track\_check$$
$$track\_check \xrightarrow{judge\_y\_1} track\_Occupied$$
$$track\_check \xrightarrow{judge\_y\_0} track\_Unoccupied$$
$$track\_Occupied \xrightarrow{send\_occupied} track\_orign$$
$$track\_Unoccupied \xrightarrow{send\_unoccupied} track\_check\_over$$
$$track\_check\_over \xrightarrow{change\_y} track\_train\_running$$
$$track\_train\_running \xrightarrow{back\_orign} track\_orign$$

（2） Uppaal 的表示



Track 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
track_init	初始化所有的 track 的变量 y 为 0，init_track(track_y)
receive_signal	接受检查 track 的信号， checkoccupied?
judge_y_1	判断当前 track 的 y 是否为 1，y==OCCUPIED
judge_y_0	判断当前 track 的 y 是否为 0，y==UNOCCUPIED
send_occupied	发送 track 被占用的信号，occupied!
send_ unoccupied	发送 track 未被占用的信号，unoccupied!
change_y	改变 track 的 y 的状态，等待 train Enter 信号

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
track_start	Track 的最开始状态，准备初始化 track 的变量 y
track_orign	表示 track 的 y 已经被初始化为未占用，准备接受检查信号
track_check	表示 track 准备检查当前是否被占用
track_Occupied	表示 track 已经被占用，准备发出占用信号
track_ Unoccupied	表示 track 尚未被占用，准备发出未被占用信号
track_check_over	表示 track 已经检查完毕
track_train_running	等待火车离开，track 回到初始状态

（4） 非形式化的变迁说明

$track\_start \xrightarrow{track\_init} track\_start$

Track 当前处于初始状态，然后初始化所有 track 的内部变量 y，进入 track_start 状态

$track\_orign \xrightarrow{receive\_signal} track\_check$

Track 当前处于 orign 状态，当接收到检测 track 是否被占用的信号后，进入状态 check

$track\_check \xrightarrow{judge\_y\_1} track\_Occupied$

Track 当前处于 check 状态，如果 y==OCCUPIED，进入状态 Occupied

$track\_check \xrightarrow{judge\_y\_0} track\_Unoccupied$

Track 当前处于 check 状态，如果 y==UNOCCUPIED，进入状态 Unoccupied

$track\_Occupied \xrightarrow{send\_occupied} track\_orign$

Track 当前处于 Occupied 状态，发送 track 被占用的信号，回到初始状态

$track\_Unoccupied \xrightarrow{send\_unoccupied} track\_check\_over$

Track 当前处于 Unoccupied 状态，发送 track 被未占用的信号，进入检查完毕状态

$track\_check\_over \xrightarrow{change\_y} track\_train\_running$

当前 track 处于检查完毕状态，如果接受火车进入的信号，将轨道 y 置为占用，进入 train_running 状态

$$track\_train\_running \xrightarrow{back\_origin} track\_origin$$

当前 track 处于 train_running 状态，如果接收到火车离开的信号，则回到初始状态

3. 实体 Point

（1） 变迁系统模型

对于实体 Point，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

S	{point_init, point_unlocked, point_locked}
I	{point_init}
Act	{ init, point_lock, point_unlock }
AP	{point_is_lock=LOCK, point_is_lock=UNLOCK}
L	L(point_init)= ∅, L(point_unlocked)={ point_is_lock=UNLOCK } L(point_locked) ={point_is_lock=LOCK}

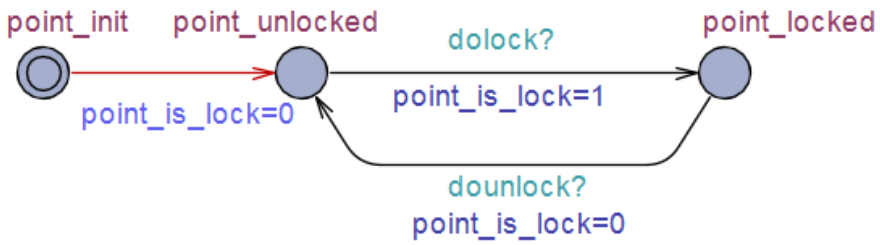
变迁关系->：

$$point\_init \xrightarrow{init} point\_unlocked$$

$$point\_unlocked \xrightarrow{point\_lock} point\_locked$$

$$point\_locked \xrightarrow{point\_unlock} point\_init$$

（2） Uppaal 的表示



point 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
init	初始化 point 为 UNLOCK，init_point(point_is_lock)
point_lock	接受 dolock 信号，锁 point，point_is_lock=LOCK
point_unlock	接受 dounlock 信号，解锁 point，point_is_lock=UNLOCK

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
point_init	Point 的初始状态，准备初始化 Point 为 UNLOCK 状态
point_unlocked	Point 此时处于未锁状态，如果接收到 dolock 信号，跳到 point_locked
point_locked	Point 此时处于锁状态，如果接收到 dounlock 信号，调到 point_unlocked

（4） 非形式化的变迁说明

$point\_init \xrightarrow{init} point\_unlocked$

Point 处于初始状态，然后对所有的 Point 进行初始化，设置为未锁状态

$point\_unlocked \xrightarrow{point\_lock} point\_locked$

Point 处于未锁状态，通过接受锁信号，对 Point 进行锁操作，到达锁状态

$point\_locked \xrightarrow{point\_unlock} point\_init$

Point 处于锁状态，通过接受解锁信号，对 Point 进行解锁操作到达未锁状态

4. 实体 Light

（1） 变迁系统模型

对于实体 Light，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

S	{light_init, light_RED, light_open_green, light_GREEN, light_open_red}
I	{ light_init }
Act	{init, receive_green, turn_green, receive_red, turn_red }
AP	{light_is==red_on, light_is=green_on}
L	L(light_RED)=L(light_open_red)={ light_is=red_on },L(light_init)= $\varnothing$ L(light_open_green) =L(light_GREEN)={ light_is=green_on }

变迁关系->：

$light\_init \xrightarrow{init} light\_RED$

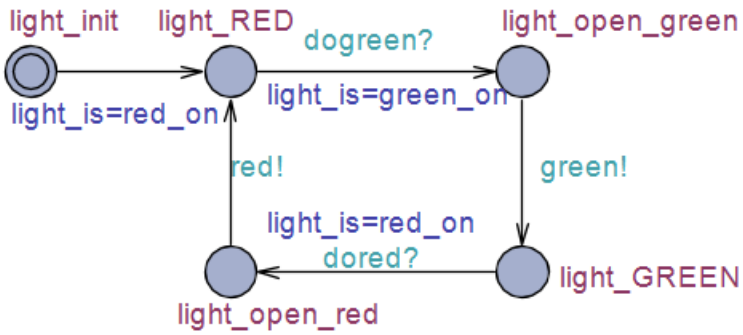
$light\_RED \xrightarrow{receive\_green} light\_open\_green$

$light\_open\_green \xrightarrow{turn\_green} light\_GREEN$

$light\_GREEN \xrightarrow{receive\_red} light\_open\_red$

$light\_open\_red \xrightarrow{turn\_red} light\_RED$

（2） Uppaal 的表示



light 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
init	init_light(light_is)初始化所有的 light 为红灯
receive_green	< dogreen? >< light_is=green_on >接受变绿的信号，改变信号灯的状态
turn_green	green! 发出信号灯为绿的信号
receive_red	< light_is=red_on ><dored?>接受变红的信号，改变信号灯的状态
turn_red	red! 发出信号灯为红的信号

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
light_init	Light 的初始状态，准备初始化所有 light 的状态
light_RED	表示 Light 处于红灯的状态
light_open_green	表示信号灯已经变为绿灯的状态
light_GREEN	表示信号灯处于绿灯的状态
light_open_red	表示信号灯已经变为红灯的状态

（4） 非形式化的变迁说明

$$light\_init \xrightarrow{init} light\_RED$$

Light 当前处于初始状态，然后初始化所有 Light 的内部变量 light_is 为红灯的状态，进入 light_RED 状态

$$light\_RED \xrightarrow{receive\_green} light\_open\_green$$

Light 当前处于红灯状态，通过接受信号灯变绿的信号，改变信号灯的 light_is 为绿灯，进入状态 light_open_green

$$light\_open\_green \xrightarrow{turn\_green} light\_GREEN$$

Light 当前处于刚接收到变绿信号，准备发出已经为绿灯的信号，然后进入 light_Green 状态；

$$light\_GREEN \xrightarrow{receive\_red} light\_open\_red$$

Light 当前处于绿灯状态，通过接受信号灯变红的信号，改变信号灯的 light_is 为红灯，进入状态 light_open_red；

$$light\_open\_red \xrightarrow{turn\_red} light\_RED$$

Light 当前处于刚接收到变红信号，准备发出已经为红灯的信号，然后进入 light_Red 状态。

5. 实体 Control_center

（1） 变迁系统模型

对于实体 Control_center，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

S	{Orign, train_wait, check_track, track_free, point_locking, point_lock_over, green_ing, green_ok, train_enter_track, train_runing, train_leave_track, point_unlocking, point_unlock_over, red_ing}
I	{ Orign }
Act	{recv_req, send_check, recv_occ, recv_unocc, send_lock, wait_lock, send_green, wait_green, recv_enter, wait_run, recv_leave, send_unlock, wait_unlock, send_red, recv_red}
AP	{check_track_time<4, red_open_time<4, light_time<21, point_is_lock=LOCK, lock_time<4, light_is=green_on, trainenter_time<10, train_run_time<200 point_is_lock=UNLOCK, unlock_time<4, green_open_time<4}
L	L(Orign)={ check_track_time<4, red_open_time<4} L(train_wait)=L(point_locking)=L(green_ing)=L(check_track)={light_time<21} L(track_free)={check_track_time<4, light_time<21} L(point_lock_over)={light_time<21, lock_time<4, point_is_lock=LOCK} L(green_ok)={green_open_time<4, light_is=green_on}, L(red_ing)= ∅ L(train_enter_track)={trainEnter_time<10} , L(train_runing)= ∅ L(train_leave_track)={train_run_time<200}, L(point_unlocking)= ∅ L(point_unlock_over)={point_is_lock=UNLOCK, unlock_time<4}

变迁关系->：

$$Orign \xrightarrow{recv\_req} train\_wait$$

$$train\_wait \xrightarrow{send\_check} check\_track$$

$$check\_track \xrightarrow{recv\_occ} Orign$$

$$check\_track \xrightarrow{recv\_unocc} track\_free$$

$$track\_free \xrightarrow{send\_lock} point\_locking$$

$$point\_locking \xrightarrow{wait\_lock} point\_lock\_over$$

$$point\_lock\_over \xrightarrow{send\_green} green\_ing$$

$$green\_ing \xrightarrow{wait\_green} green\_ok$$

$$green\_ok \xrightarrow{recv\_enter} train\_enter\_track$$

$$train\_enter\_track \xrightarrow{wait\_run} train\_runing$$

$$train\_runing \xrightarrow{recv\_leave} train\_leave\_track$$

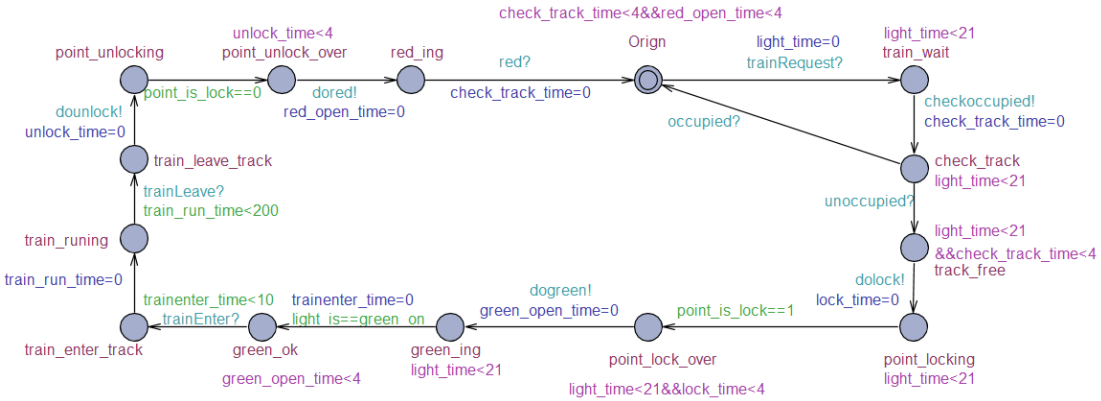
$$train\_leave\_track \xrightarrow{send\_unlock} point\_unlocking$$

$$point\_unlocking \xrightarrow{wait\_unlock} point\_unlock\_over$$

$$point\_unlock\_over \xrightarrow{send\_red} red\_ing$$

$$red\_ing \xrightarrow{recv\_red} Orign$$

（2） Uppaal 的表示



Control_center 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
recv_req	<trainRequest?><light_time=0>接受 train 进入 Track 的请求，同时将 light_time 初始化
send_check	<checkoccupied!><check_track_time=0>发出检查 Track 的信号，同时将 check_track_time 设置为 0
recv_occ	occupied? 接受 occupied 信号
recv_unocc	unoccupied? 接受 unoccupied 信号
send_lock	<dolock!><lock_time=0>发出锁 point 的信号，同时将 lock_time 设置为 0
wait_lock	point_is_lock==1 判断 point 是否已经被锁

send_green	< dogreen!>< green_open_time=0>发出信号灯变绿的消息，同时设置 green_open_time 为 0
wait_green	< light_is==green_on>< trainenter_time=0>判断当前信号灯是否为绿色，同时将 trainenter_time 设置为 0
recv_enter	< trainenter_time<10 >< trainEnter?>接受 trainEnter 的信号，以判断火车是否进入轨道，同时检测此时 trainenter_time 是否<10
wait_run	train_run_time=0 初始化 train_run_time 为 0
recv_leave	< train_run_time<200>< trainLeave?>接受 trainLeave 信号，判断火车是否离开轨道，同时检查列车运行时间是否小于 200s
send_unlock	< dounlock!>< unlock_time=0>发出解锁 point 信号，初始化 unlock_time 为 0
wait_unlock	point_is_lock==UNLOCK 判断 point 是否已经解锁
send_red	< dored!>< red_open_time=0>发出信号灯变红的信号，同时初始化 red_open_time 为 0
recv_red	< red?>< check_track_time=0>接受 red 信号，判断信号灯是否为红色，同时初始化 check_track_time 为 0

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
Orign	Control 的初始状态，准备接受列车发出的请求
train_wait	表示接收到列车进入的请求，此时要求 light_time<21。之后准备接受检查轨道信号
check_track	表示接收到检查轨道信号，此时要求 light_time<21，之后接受检查的结果
track_free	表示接受轨道未被占用的信号，此时要求时间 light_time<21&& Check_track_time<4。之后发出锁 point 信号
point_locking	表示已经发出锁 point 信号，等待 point 被锁，要求时间 light_time<21
point_lock_over	表示 point 已经被锁，此时要求时间 light_time<21&&lock_time<4 之后发出信号灯变绿信号
green_ing	表示等待信号灯变绿，要求时间 light_time<21
green_ok	表示信号灯已经变为绿色，要求时间 green_open_time<4。
train_enter_track	表示已经收到了 trainEnter 的信号，并且之前收到 trainEnter 的时间小于 10
train_runing	表示准备接受 train 发出的 trainLeave 信号
train_leave_track	表示已经接收到 train 离开的信号，并且火车运行时间 train_run_time<200，之后发出解锁 point 的信号
point_unlocking	表示正在等待 point 被解锁

point_unlock_over	表示 point 已经被解锁，并且时间要求 unlock_time<4。之后发出信号灯变红的消息
red_ing	表示正在等待信号灯变红的信号

（4） 非形式化的变迁说明

$Orign \xrightarrow{recv\_req} train\_wait$

当前处于初始化状态，在接收到 Train 进入 Track 的请求之后进入 train_wait 状态，同时在变迁过程中将时间 light_time 置为 0

$train\_wait \xrightarrow{send\_check} check\_track$

当前状态为 train_wait，发出检查 track 是否被占用的信号，同时将时间 check_track_time 设置为 0

$check\_track \xrightarrow{recv\_occ} Orign$

当前状态为 check_track，接受到了 occupied 信号，回到初始状态，重新开始接受信号；

$check\_track \xrightarrow{recv\_unocc} track\_free$

当前状态为 check_track，接收到了 unoccupied 信号，即 track 未被占用，则进入到 track_free 的状态；

$track\_free \xrightarrow{send\_lock} point\_locking$

当前状态为 track_free，发出锁 point 的信号，同时将 lock_time 置为 0，接着进入下一个状态 point_locking；

$point\_locking \xrightarrow{wait\_lock} point\_lock\_over$

当前状态为 point_locking 状态，当检测到 point 已经被锁的状态则进入下一个状态 point_lock_over；

$point\_lock\_over \xrightarrow{send\_green} green\_ing$

当前状态为 point_lock_over 状态，发出信号灯变绿的信号，同时将 green_open_time 设置为 0，进入下一个状态 green_ing；

$green\_ing \xrightarrow{wait\_green} green\_ok$

当前状态为 green_ing，当检测到信号灯已经变为绿色时进入下一个状态 green_ok，同时将 trainenter_time 设置为 0；

$green\_ok \xrightarrow{recv\_enter} train\_enter\_track$

当前状态为 green_ok 状态，当接收到 trainEnter 的信号时，进入 train_enter_track 的状态，同时要求时间 trainenter_time<10;

$train\_enter\_track \xrightarrow{wait\_run} train\_runing$

当前状态为 train_enter_track，设置 train_run_time 为 0，然后进入下一个状态 train_runing；

$train\_runing \xrightarrow{recv\_leave} train\_leave\_track$

当前状态为 train_runing，当接收到 trainLeave 信号时进入到下一个状态 train_leave_track，同时要求时间 train_run_time<200;

$$train\_leave\_track \xrightarrow{send\_unlock} point\_unlocking$$

当前状态为 train_leave_track，发出解锁 point 的信号，同时将时间 unlock_time 设置为 0，然后进入下一个状态 point_unlocking;

$$point\_unlocking \xrightarrow{wait\_unlock} point\_unlock\_over$$

当前状态为 point_unlocking，当检测到 point 已经被解锁时，进入下一个状态 point_unlock_over；

$$point\_unlock\_over \xrightarrow{send\_red} red\_ing$$

当前状态为 point_unlock_over，发出信号灯变红的信号，并且将时间 red_open_time 设置为 0，然后进入下一个状态 red_ing；

$$red\_ing \xrightarrow{recv\_red} Orign$$

当前状态为 red_ing，当收到信号灯变红的信号时跳转到下一个状态 Orign。

四． 系统成分附加说明

对于假设条件下的复杂情况中，设计了 2 个 Train，5 个 Track，6 个 Light，2 个道岔，因此实体之间的同步信号的数量，实体的属性也会增加，同时在系统运行过程中也必须能够指明当前的系统中是在那种情况下运行。

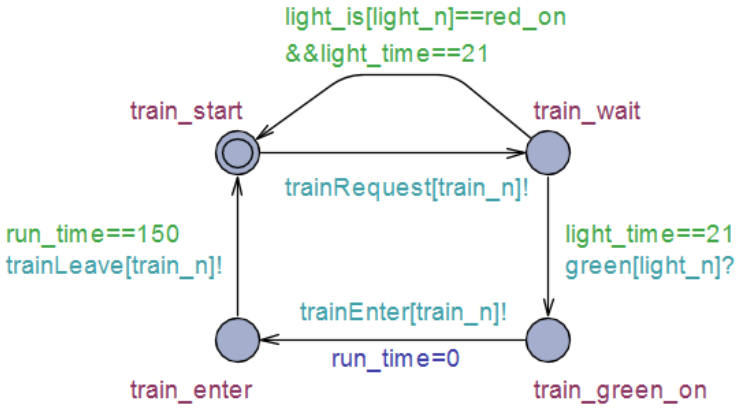
部分三描述的是针对于简单情况下 train，track，light，point 和 control 的设计。因此对于复杂情形下，这些实体都会发生改变，相应的改变信号的个数等参数。同时与简单情况不同，复杂情况下还需要增加实体 RoutTable，以控制系统中参数的变化。

1. 实体 Train 的变化

实体 train 的变迁关系和状态的个数及属性都没有发生变化，主要发生的变化有以下五个：

简单情况下	复杂情况下
light_is	light_is[light_n]
trainLeave	trainLeave[train_n]
trainEnter	trainEnter[train_n]
trainRequest	trainRequest[train_n]
green	green[light_n]

实际的状态图的变化为

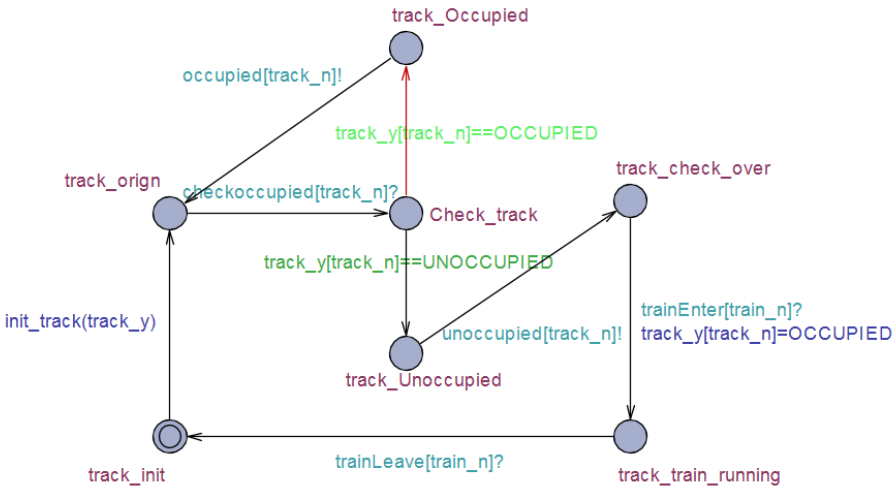


2. 实体 Track 的变化

实体 track 的变迁关系和状态的个数及属性都没有发生变化，主要发生的变化有以下六个：

简单情况下	复杂情况下
y	track_y[track_n]
occupied	occupied[track_n]
unoccupied	unoccupied[track_n]
trainEnter	trainEnter[train_n]
trainLeave	trainLeave[train_n]
y=UNOCCUPIED	init_track(track_y)

实际的状态图的变化为



Init_track 的作用是初始化 track_y 为未被占用状态

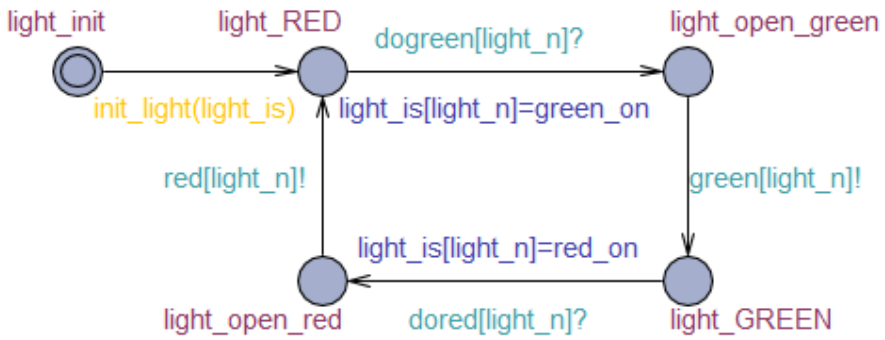
```
Void init_track(int & track_y[5]){
    For(i:int[0,4]){ track_y[i]=UNOCCUPIED;}
}
```

3. 实体 Light 的变化

实体 track 的变迁关系和状态的个数及属性都没有发生变化，主要发生的变化有以下六个：

简单情况下	复杂情况下
dogreen	dogreen[light_n]
light_is	light_is[light_n]
green	green[light_n]
dored	dored[light_n]
red	red[light_n]
light_is=red_on	init_light(light_is)

实际的状态图的变化为



Init_light(light_is)的作用是初始化所有的灯都为红灯

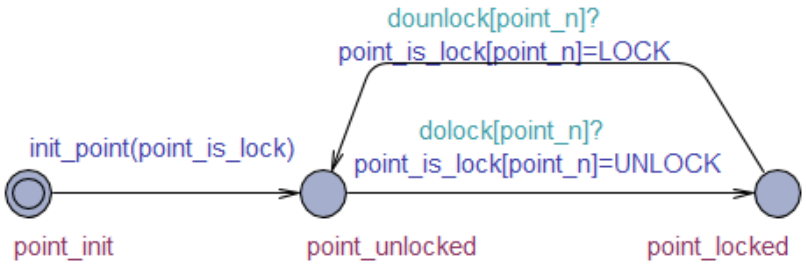
```
Void init_light(int& light_is[6])
{
    For(i:int[0,5]){
        light_is[i]=red_on
    }
}
```

4. 实体 Point 的变化

实体 track 的变迁关系和状态的个数及属性都没有发生变化，主要发生的变化有以下四个：

简单情况下	复杂情况下
dounlock	dounlock[point_n]
dolock	dolock[point_n]
point_is_lock	point_is_lock[point_n]
point_is_lock=0	init_point(point_is_lock)

实际的状态图的变化为



Init_point(point_is_lock)的作用是初始化所有的道岔为未锁状态。

5. 实体 Control_center 的变化

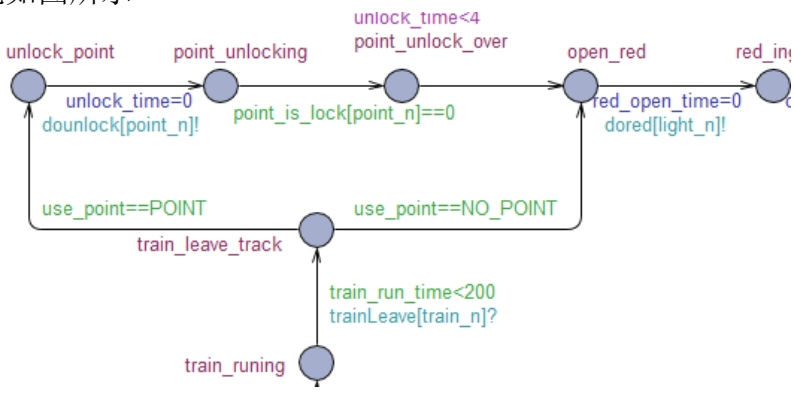
实体 Control_center 主要有两大点方面的变化，一方面是模型结构方面，另一方面则是模型中适用到的变量和同步信号的变化。

（1） 模型结构变化

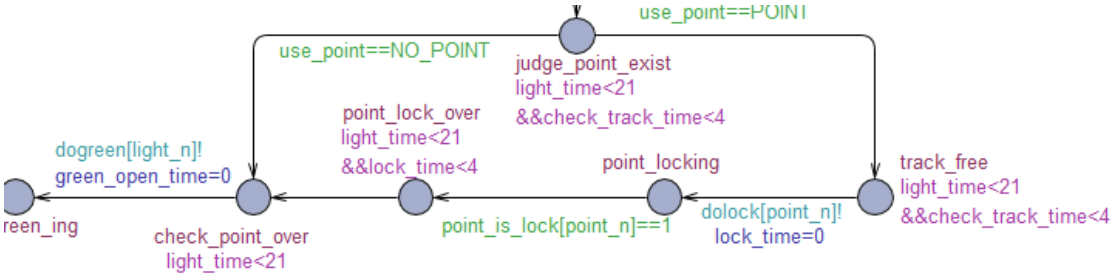
在复杂假设的情境下和简单的情况不相同的地方在于并不是所有的 Track 都会有 Point 作为起始，因此需要对简单情况下设计的每次都锁道岔和解锁道岔的状态变换进行限制。

首先增加全局变量 use_point，取值为{POINT, NOPOINT}。POINT 是指有道岔，NOPOINT 是指没有道岔。

然后在锁道岔和解锁道岔之前使用 use_point 判断当前有没有道岔，如果有道岔则与之前一致，如果没有则直接跳过之前的锁道岔和解锁道岔的状态转移。具体的实现如图所示



解锁道岔



锁道岔

（2） 内部参数变化

复杂情况下的 control_center 在参数方面主要有以下变化

简单情况下	复杂情况下
trainRequest	trainRequest[train_n]
checkoccupied	checkoccupied[track_n]
occupied	occupied[track_n]
unoccupied	unoccupied[track_n]
point_is_lock	point_is_lock[point_n]
dolock	dolock[point_n]
dogreen	dogreen[light_n]
trainEnter	trainEnter[train_n]
light_is	light_is[light_n]
trainLeave	trainLeave[train_n]
dounlock	dounlock[point_n]
dored	dored[light_n]
red	red[light_n]

6. 增加实体 RoutTable

（1） 变迁系统模型

对于实体 Routtable，设计其变迁系统为六元组 TS=(S, Act, ->, I, AP, L)，其中每个部分的具体解释为：

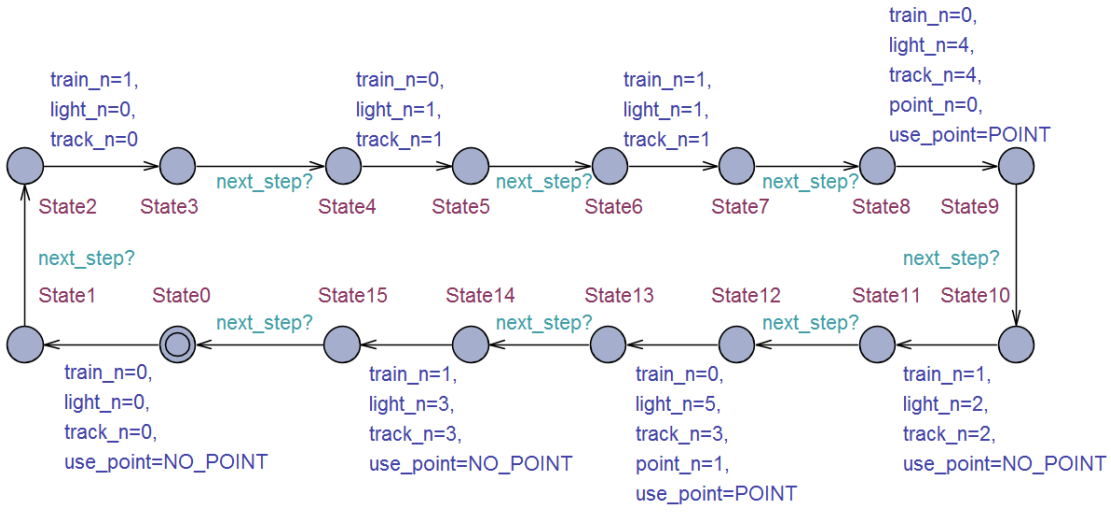
S	{state0, state1, state2, state3, state4, state5, state6, state7, state8, state9, state10, state11, state12, state13, state14, state15}
I	{state0}
Act	{change_state, set_step1, set_step2, set_step3, set_step4, set_step5, set_step6, set_step7, set_step8}
AP	{train_n=1,train_n=0,light_n=0,light_n=1,light_n=2,light_n=3,light_n=4, Light_n=5,track_n=0,track_n=1,track_n=2,track_n=3,track_n=4,point_n=0, point_n=1, use_point=POINT, use_point=NO_POINT }
L	L(state0)=L(state2)= L(state4)= L(state6)= L(state8)= L(state10)= L(state12)= ∅ L(state14) =∅ L(state1) = {train_n=0, light_n=0, track_n=0, use_point=NO_POINT} L(state3) = {train_n=1, light_n=0, track_n=0, use_point=NO_POINT}

	L(state5) = {train_n=0, light_n=1, track_n=1} L(state7) = {train_n=1, light_n=1, track_n=1} L(state9) = {train_n=0, light_n=4, track_n=4, point_n=0, use_point=POINT} L(state11) = {train_n=1, light_n=2, track_n=2, use_point=NO_POINT} L(state13) = {train_n=0, light_n=5, track_n=3, point_n=1, use_point=POINT} L(state15) = {train_n=1, light_n=3, track_n=3, use_point=NO_POINT}

变迁关系->：

$$state0 \xrightarrow{set\_step1} state1$$
$$state1 \xrightarrow{change\_state} state2$$
$$state2 \xrightarrow{set\_step2} state3$$
$$state3 \xrightarrow{change\_state} state4$$
$$state4 \xrightarrow{set\_step3} state5$$
$$state5 \xrightarrow{change\_state} state6$$
$$state6 \xrightarrow{set\_step4} state7$$
$$state7 \xrightarrow{change\_state} state8$$
$$state8 \xrightarrow{set\_step5} state9$$
$$state9 \xrightarrow{change\_state} state10$$
$$state10 \xrightarrow{set\_step6} state11$$
$$state11 \xrightarrow{change\_state} state12$$
$$state12 \xrightarrow{set\_step7} state13$$
$$state13 \xrightarrow{change\_state} state14$$
$$state14 \xrightarrow{set\_step8} state15$$
$$state15 \xrightarrow{change\_state} state0$$

（2） Uppaal 的表示



routtable 的变迁系统的图形表示

（3） 图形表示的具体解释

对于行为 Act 集合中每个具体的行为的解释为：

行为名称	行为具体的形式
change_state	next_step? 用于控制系统执行进入下一个阶段
set_step1	train_n=0, light_n=0, track_n=0, use_point=NO_POINT 设置第一阶段执行时的参数，包括列车号，轨道号，信号灯号，有无道岔
set_step2	train_n=1,light_n=0,track_n=0 设置第二阶段的运行参数
set_step3	train_n=0,light_n=1,track_n=1 设置第三阶段的运行参数
set_step4	train_n=1,light_n=1,track_n=1 设置第四阶段的运行参数
set_step5	train_n=0,light_n=4,track_n=4,point_n=0,use_point=POINT
set_step6	train_n=1,light_n=2,track_n=2,use_point=NO_POINT
set_step7	train_n=0,light_n=5,track_n=3,point_n=1,use_point=POINT
set_step8	train_n=1,light_n=3,track_n=3,use_point=NO_POINT

对于变迁系统中的每个状态的具体解释为：

状态名称	状态解释
state0	初始状态
state1	设置火车 0，信号灯 0，轨道 0，没有道岔，开始执行火车 0 进入轨道 0
state2	火车 0 已经进入轨道 0 结束
state3	设置火车 1，信号灯 0，轨道 0，没有道岔，开始执行火车 1 进入轨道 0
state4	火车 1 已经进入轨道 0 结束
state5	设置火车 0，信号灯 1，轨道 1，没有道岔，开始执行火车 0 进入轨道 1
state6	火车 0 已经进入轨道 1 结束
state7	设置火车 1，信号灯 1，轨道 1，没有道岔，开始执行火车 1 进入轨道 1
state8	火车 1 已经进入轨道 1 结束
state9	设置火车 0，信号灯 4，轨道 4，有道岔 0，开始执行火车 0，通过道岔 0，进入轨道 4
state10	火车 0，通过道岔 0，已经进入轨道 4 结束
state11	设置火车 1，信号灯 2，轨道 2，没有道岔，开始执行火车 1 进入轨道 2
state12	火车 1 已经进入轨道 2 结束
state13	设置火车 0，信号灯 5，轨道 3，有道岔 1，开始执行火车 0，通过道岔 1，进入轨道 3
state14	火车 0，通过道岔 1，已经进入轨道 3 结束
State15	设置火车 1，信号灯 3，轨道 3，没有道岔，开始执行火车 1 进入轨道 3

五. 系统定义

1. 系统的时间约束的实现

针对于系统中的每个时间约束，通过设置时间变量进行约束。同时复杂情况和简单情况下的时间约束的实际一致，并无差别。

约束：Center 模型收到 request 消息到列车接收到交通灯回馈的消息之间的时间为 21s

实现：控制中心 control_center 收到 trainRequest 时，设置 light_time 为 0，并且通过设置状态的不变量以约束这些状态需要在 light_time<21 的时间内执行，需要设置的状态包括：train_wait，check_track，track_free，point_locking，point_lock_over。之后 control 则会发出信号灯变量的信号，当 train 收到 green 信号时，也会检查当前的 light_time 是否为 21。

约束：Center 模型发出 checkoccupied 消息到收到 Center 模型发出反馈消息之间的时间少于 4s

实现：控制中心在发出 checkoccupied 信号时，将 check_track_time 设置为 0，然后将在 orign 和 track_free 两个状态设置不变量，以保证到达此状态时，check_track_time<4。

约束：Center 模型发出 dolock 消息到收到 locked 消息之间的时间少于 4s

实现：控制中心在发出 dolock 信号时，将时间 lock_time 设置为 0，并且在状态 point_lock_over 时设置不变量，以保证 lock_time<4。

约束：Center 模型发出 dogreen 消息到收到 green 消息时间之间的时间少于 4s

实现：控制中心发出 dogreen 信号时，将 green_open_time 设置为 0，并且在状态 green_ok 中设置不变量，以保证 green_open_time<4。

约束：Center 模型收到 green 消息到收到 trainEnter 消息之间的时间少于 10s

实现：控制中心在检测到 light_is==green_on 时，将 trainenter_time 设置为 0，并且在收到 trainEnter 信号时，检查 trainenter_time 是否小于 10

约束：Center 模型收到 trainEnter 消息到收到 trainLeave 消息之间的时间少于 200s

实现：控制中心在收到 trainEnter 信号之后，将 train_run_time 设置为 0，并且在收到 trainLeave 信号时，检查 train_run_time 是否小于 200

约束：Center 模型发出 dounlock 消息到收到 unlocked 消息之间的时间少于 4s

实现：控制中心在发出 dounlock 信号时，将时间 unlock_time 初始化为 0，并且在状态 point_unlock_over 状态设置不变量，以保证此时 unlock_time<4

约束：Center 模型发出 dored 消息到收到 red 消息之间的时间少于 4s

实现：控制中心在发出 dored 信号时，将 red_open_time 设置为 0，并且在状态 orign 设置不变量，以保证此状态的 red_open_time<4

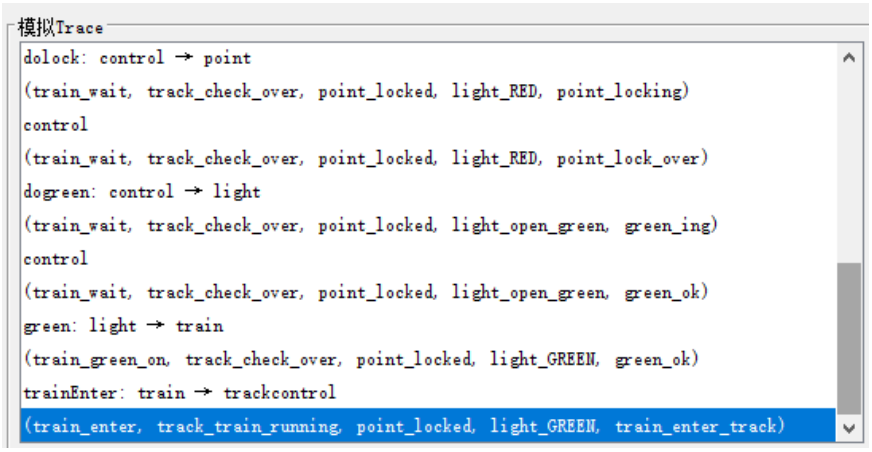
2. 系统并发实现

为了实现多个实体之间的并发执行，在系统建模过程中，设置了一组通道变量 chan，来达到实体之间并发的作用，具体的 chan 变量的含义和用处为：

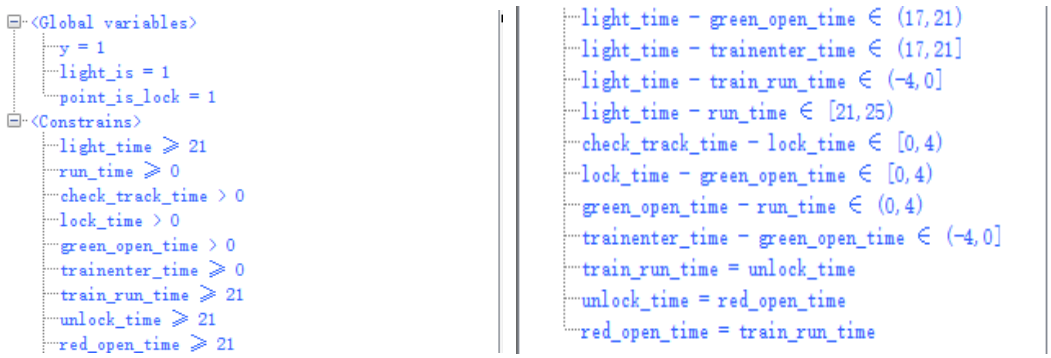
信号	发出方	接收方	信号含义
trainEnter	Train	Control/Track	表示 train 即将进入 track
trainLeave	Train	Control/Track	表示 train 即将离开 track
trainRequest	Train	Control	表示 train 请求进入 track
red	Light	Control	表示信号灯已经变为红色
green	Light	Train	表示信号灯已经变为绿色
dogreen	Control	Light	表示信号灯需要变为绿色
dored	Control	Light	表示信号灯需要变为红色
checkoccupied	Control	Track	表示需要检查 track 是否被占用
occupied	Track	Control	表示 track 正在被占用
unoccupied	Track	Control	表示 track 未被占用
dolock	Control	Point	表示需要将 point 锁住
dounlock	Control	Point	表示需要解锁 point
next_step	Control	RoutTable	表示此阶段结束，进入下一阶段

3. 实际的运行效果

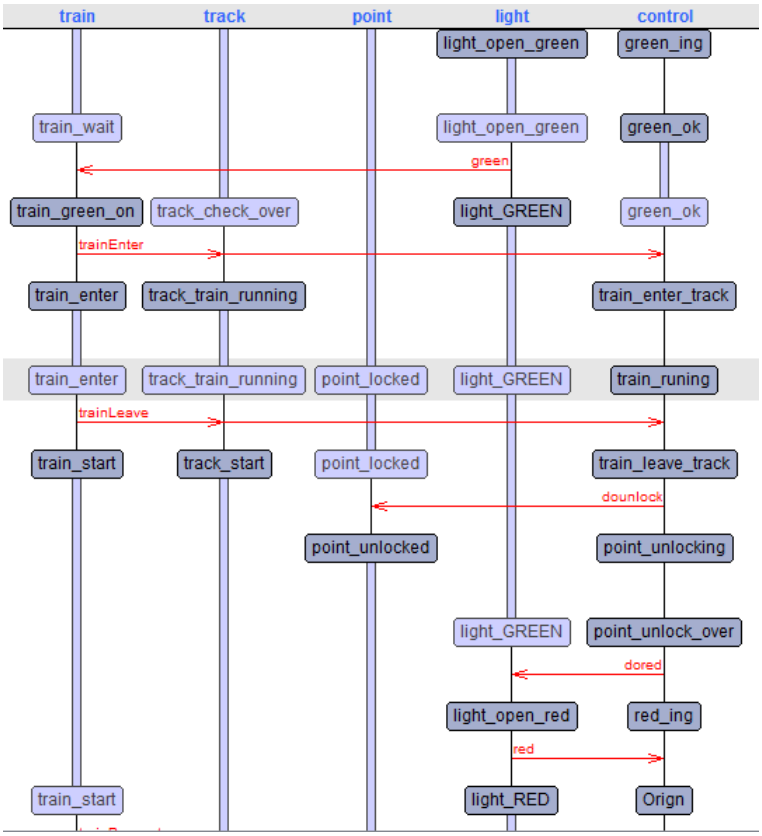
（1） 简单情况下运行效果图



简单情境下状态迁移图

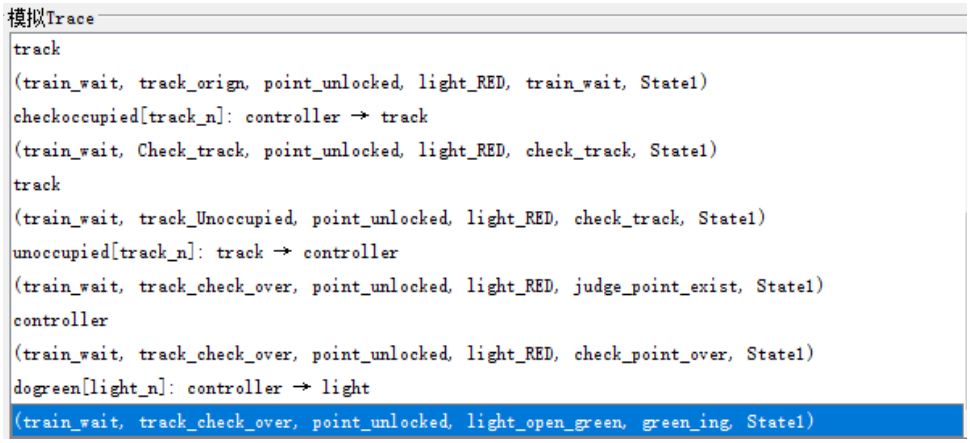


简单情景下运行中的变量值

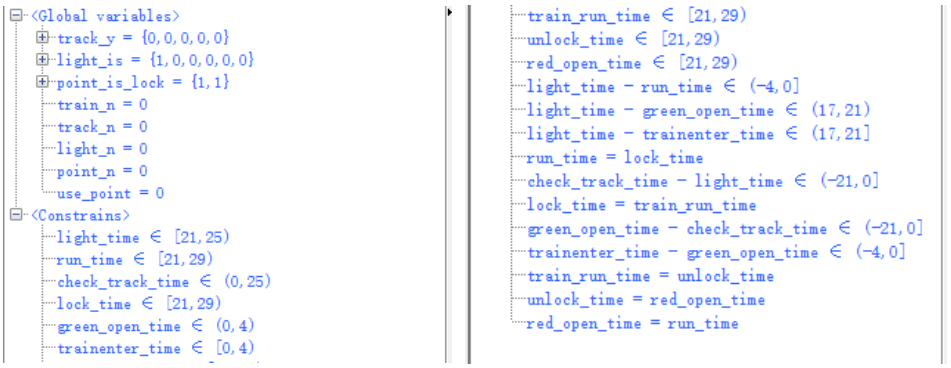


简单情景下并发执行图

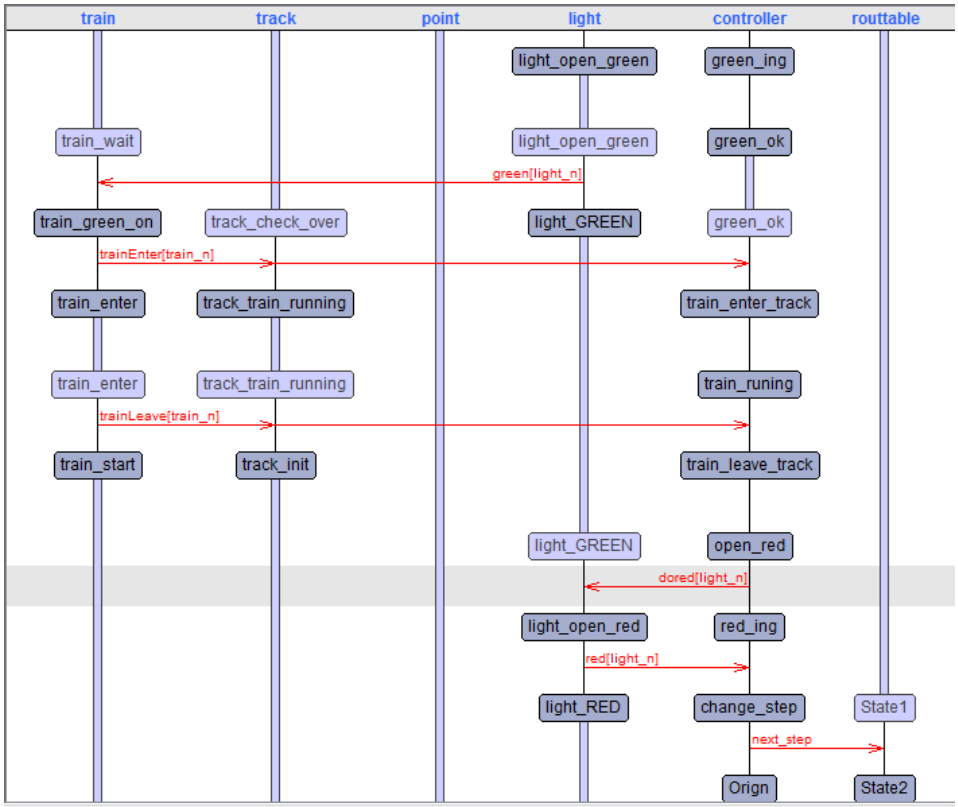
（2） 复杂情况下运行效果图



复杂情境下状态迁移图



复杂情景下运行中的变量值



复杂情景下并发执行图

六. （附加）系统性质定义和性质验证

1. 简单情境下的性质定义和验证

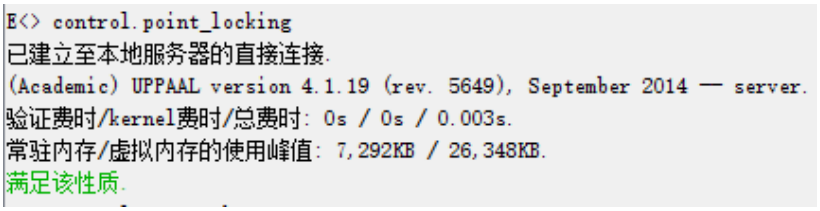
（1） A[] not deadlock

性质：系统有没有死锁

验证结果：由于系统中包含 broadcast 的 chan 变量，并且系统在声明时规定了实体之间的优先级，所以暂时无法验证。(train, track <point <light < control)

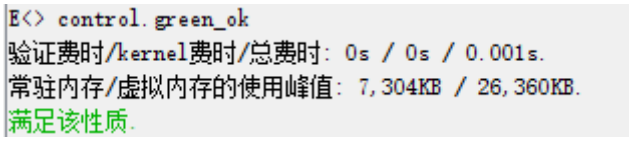
（2） E<> control.point_locking

性质：只要列车请求进入，就会存在一条路径会使得道岔被解锁



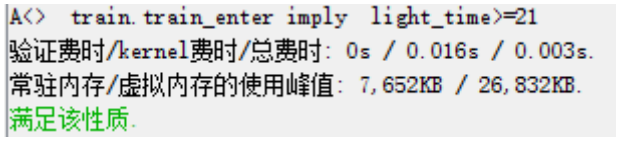
（3） E<> control.green_ok

性质：只要列车发出进入请求，就会存在一条路径能够让绿灯亮起



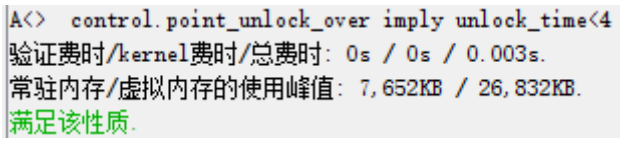
（4） A<> train.train_enter imply light_time>=21

性质：列车只要申请进入轨道，最终所有的可能的路径都可以到达火车进入轨道的状态，并且从申请开始到进入的时间大于等于 21s



（5） A<> control.point_unlock_over imply unlock_time<4

性质：控制器的最终都会到达解锁 point 的状态，并且此时道岔已经解锁，且解锁花费的时间小于 4s



2. 复杂情境下的性质定义和验证

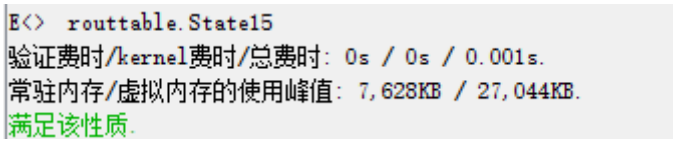
（1） A[] not deadlock

性质：系统有没有死锁

验证结果：由于系统中包含 broadcast 的 chan 变量，并且系统在声明时规定了实体之间的优先级，所以暂时无法验证。(train, track<point<light<controller<routtable)

（2） E<> routtable.State15

性质：联锁表在控制系统执行时，可以到达最后一步的控制状态，即完整执行联锁表



（3） E<> controller.point_lock_over imply lock_time<4

性质：系统的每次运行都有可能进入到锁道岔的阶段，并且上锁操作小于 4s

（4） E<> controller.point_unlock_over imply unlock_time<4

性质：系统的每次运行都有可能进入到解锁道岔的阶段，并且解锁操作小于 4s

（5） E<> controller.change_step

性质：系统的每次运行控制器都可以结束运行，并且发出 next_step 的信号

七. 实验总结与感想

这次实验的过程中，遇到了很多的问题，包括开始时 uppaal 软件的建模方式，时间约束的实现等问题。因为网上关于 uppaal 软件进行时间自动机建模的案例非常少，官网上给出的简单的介绍文档之中的案例也是非常的简单，并没有给出很大的帮助。所以实验过程中在 uppaal 的软件使用方面主要都是通过自身摸索以及同学之间的讨论。

在设计时间约束的时候，自己在课上理解的不是很明白，并且也不知道 uppaal 中的具体实现。开始的时候只是简单的以为系统在运行过程中会有一个时钟在自己一步一步增加，我们只需要在边上加入 time<4s 的判断条件就行，结果就出现了状态跳转不明确，造成了死锁。通过同学之间的交流和讨论，最终自己的理解是，边上的例如 time<4s 的约束的具体含义是：当执行这条边进行状态跳转时，time 会是小于 4s 的。而且时间也只有在这样的判断语句或者是赋值语

句之后才会发生改变。在实现过程中的一个问题就是要求发出火车进入的信号到接收到信号灯变绿时时间应该为 21s，此时就应该在火车收到 green？信号的时候检测 time==21，同时还应在控制中心所有在发出 dogreen？信号的状态之前，加入不变量约束 time<21，表示这些状态应该在 time 未到 21s 之前到达。如果不加入这些不变量约束，系统将可以选择直接执行 train 实体中的状态转移，然后时间 time 变为不小于 21s，因为系统认为此时这个状态并没有收到限制，是可以执行的。

在实验过程中的另一个问题是实验二的设计，在开始的时候并不知道如何设计能够让系统有多个火车同时随机的根据铁路的状态自主选择执行，因此后来自己选择了一种最为简单直接的方法，人为规定当前应该是哪一个火车在哪一个铁轨上，判断哪一个信号灯，然后执行。但是我认为这种方式仍然存在问题，人为规定的因素太多，没有普遍性，需要针对每一列火车都要人为的设置，太过于麻烦。这也是之后需要改进的地方。

在完成报告的过程中，自己遇到的另外一个问题是变迁模型六元组中的原子命题 AP。在开始的时候我不知道应该不知道怎么设计或者描述 AP，通过翻阅课件，我看到了 AP 是指所关注的特征，因此我便将每个状态需要满足的性质作为 AP 的集合，例如某个状态应该满足当前是红灯，我就将 light_is=red_on 加入到 AP 中，但是我还是不太确定对不对。

经过了这次课程设计，自己确实加深了课上的理论知识，也着实锻炼了自己解决实际问题的能力和独立思考的能力。虽然问题还存在一些，但是自身收获的已经很多了。最后谢谢金芝老师，真的非常和蔼！