Temporal Action Language TeAL

Wenbin Li David Brown Jane Hayes
Mirosław Truszczyński

This report introduces the syntax of *TeAL*, the intermediate language we created for bridging the gap between natural language and the low-level target language used for reasoning. The *TeAL* syntax has been changed in several places since last report so that the syntax is closer to natural language. This appendix also introduces *Action Language AL*, *TeAL* is an extension of *AL*. Appendix A contains two *TeAL* system descriptions: *CM1 requirements*, a scenario based on requirements taken from a real project; and *double play*, a scenario based on baseball rules.

Background: *Action Language*

We design *TeAL* as an extension of *Action Language AL*. *AL* is one of the action languages used in knowledge representation domain for specifying the preconditions and effects of actions, and the way in which systems evolve. However, action languages cannot represent temporal information.

AL includes actions and fluents. An action is an event that changes the system. A fluent is a condition that can change. The condition of the system can be described by sets of fluents and each fluent represents an atomic property of the system (statements that are *true* or *false*). Fluents are changed as the effects of actions. A system description D of *AL* consists of expressions of the following form:

State constraint	L **if** P	(1)
Dynamic causal law	A **causes** L **if** P	(2)
Executability condition	**impossible** $A_1, \ldots, A_k$ **if** P	(3)

where L and P are lists of fluents and their negations, and $A, A_1, \ldots, A_k$ are actions. State constraint (1) says that L holds (every fluent and the negation of a fluent in L holds) in every state in which P holds (in the same sense as L). Dynamic causal law (2) describes the effects of actions. Executability condition (3) specifies the prerequisites of actions. For example:

$$\textit{connect}(\textit{serA}, \textit{nodeA}) \textbf{ causes } \textit{connected}(\textit{nodeA}, \textit{serA}) \textbf{ if } \textit{systemOn}$$

says that executing the action $\textit{connect}(\textit{serA}, \textit{nodeA})$ when the system is on results in *nodeA* and *serA* being connected, and

$$\textbf{impossible } \textit{connect}(\textit{serA}, \textit{nodeA}) \textbf{ if } \textit{connected}(\textit{serA}, \textit{nodeA})$$

specifies the prerequisite of the action $\textit{connect}(\textit{serA}, \textit{nodeA})$.

The semantics of a system description D can be modeled as a transition system T_D. The transition system represents all possibles ways a system can evolve. Each transition system contains a set of states that are connected by edges. A state is a collection of fluents and their negations. If a state s contains a fluent *f*, f holds in s. If s contains $\neg$ *f*, f does not hold in s. Each edge is marked by a set of actions a. We define T_D as the set $\langle S, R\rangle$ where:

1. S is the set of all interpretations s such that, for every *state constraint* L **if** P in D, s satisfies L if s satisfies P.

2. R is the set of all triples $\langle s, a, s\rangle$ such that, for every *Dynamic causal law* A **causes** L **if** P in D, s satisfies L if s satisfies P, a contains A, and for every *Executability condition* **impossible** $A_1, \ldots, A_k$ **if** P, if S satisfies P, a does not contain all of $A_1, \ldots, A_k$.

Syntax of *TeAL*

The syntax of *TeAL* contains two parts: *Declaration* and *Statements*. Declaration includes declarations of *sort*, *agent*, *constant*, *variable*, *fluent*, *action*, and *time unit*. *Statements* includes *action description*, *initial constraint*, and *temporal description*.

TeAL Declarations

An *TeAL* signature <S, AG, C, F, AC, T> consists of six sets: a set S of sort names, a set AG of agent names, a set C of constants, a set F of fluent names, a set AC of action names, and a set T of time unit.

A sort declaration is:

sort $s_1, \ldots, s_k$;

For example:

sort *server*, *node*;

declares two types of entities: server and node.

A constant declaration is:

constant s_1 $con_1, \ldots, con_k$;

where s_1 is a sort declared in sort declaration and $con_1, \ldots, con_k$ is a list of constants. For example:

constant *node nodeA*, *nodeB*, *nodeC*;

declares three nodes.

An agent declaration is:

agent $ag_1, \ldots, ag_k$;

where $ag_1, \ldots, ag_k$ are agent names. For example:

agent *nodeA*, *nodeB*, *nodeC*;

declares that three nodes are agents.

The declaration of fluent is of the form:

fluent *fluentName*$(s_1, \ldots, s_k)$;

where *fluentName* is a string that represents a fluent name and $s_1, \ldots, s_k$ is a set of sort names. For example:

fluent *connected*$(serA, nodeA)$;

The declaration of action is:

action *actionName*$(s_1, \ldots, s_k)$;

where *actionName* is a string that represents an action name and $s_1, \ldots, s_k$ is a list of sort name, in which s_1 must be a declared agent. For example:

action $estConn(serA, nodeA)$;

declares an action that represents the action *establishing connection.* The action is executed by $serA$.

The declaration of time units is of the form:

timeUnit $unit_1$, , $unit_k$;

For example:

timeUnit *seconds*, *milliseconds*;

declares two time units.

TeAL Statements

A *TeAL* theory is a triple (AD,IC,TC) where AD is the set of *action definitions*, IC is the set of *initial constraints*, and TC is the set of *temporal constraints.*

The *action definitions* specify the causal dependencies among actions. They provide the relationships among fluents, the prerequisites of actions, and action effects. The syntax of AD reuses the syntax of AL.

The second part of *TeAL* theory, IC, contains *initial constraints* that define the initial state of the system. An *initial constraint* is an expression:

initially L

where L is a fluent.

The presence of the component TC in a *TeAL* theory is the key feature that distinguishes *TeAL* from AL. TC specifies temporal information including *temporal constraints* and action durations.

A *duration specification* is an expression:

duration *Action x unit*;

where x is a positive number and *units* refers to the time units such as *millisecond*, *second*, or *minute.*

Temporal constraints describe temporal relationships among the times when events occur. *Temporal conditions* are the basic component of temporal constraints. A temporal condition models the temporal relationship between the occurrence time of two events.

TeAL introduces *prompts* as a component of *temporal conditions.* The general form of a prompt is:

promptOperator [**next** | **previous**] *Action*

TeAL provides two *prompt operators*: **commence** and **terminate**, which represents the previous or next occurrence of starting or successfully finishing an action. Thus, each action *Act* is related to two prompts: **commence** *Act* and **terminate** *Act*. Additionally, one can relate two consecutive occurrences of the same action to each other in *TeAL* statements. To distinguish between them, *TeAL* provides the keywords **previous** and **next**.
For example,

commence previous $connect(serA, nodeA)$

stands for the most recent time in the past of commencing the action $connect(serA, nodeA)$.

The BNF for a temporal condition follows:

$<$ *temporalCondition* $>$ ::= $<$ *timeReference₁* $>$ [$when_1$] | [$when_2$]
$when_1$ ::= $<$ *timeComparator* $><$ *timeModifier* $>$
$when_2$ ::= $<$ *timeComparator* $>$ [$<$ *timeModifier* $>$]
$<$ *timeReference₂* $>$
$<$ *timeComparator* $>$ ::= **earlierthan** | **at** | **laterthan**
| **noearlierthan** | **nolaterthan**
$<$ *timeModifier* $>$ ::= x *units* **before** | x *units* **after**
$<$ *timeReference*$_1$ $>$ ::= $<$ *prompt* $>$ | L
$<$ *timeReference*$_2$ $>$ ::= $<$ *prompt* $>$ | **starttime** | L

where x is an positive value and L is a fluent or its negation.

We use *timeReference₁* and *timeReference₂* to specify time moments. The term **starttime** represents the time moment when the system starts; *prompt* indicates the time moment when the corresponding prompt occurs; L indicates the time moment when the fluent L begins to hold.

We use *timeComparator* to specify the relationship between the time moments indicated by *timeReference₁* and [$<$ *timeModifier* $>$] $<$ *timeReference₂* $>$. If $<$ *timeReference₂* $>$ does not exist in a temporal condition, it means the temporal condition specifies the relationship between the time moments indicated by *timeReference₁* and [$<$ *timeModifier* $>$] $currentTimeMoment$. The term $currentTimeMoment$ can be any time moment that is checked.

The parameter *timeModifier* modifies the time given by the second *timeReference*. Term *units* refers to the time units such as *millisecond*, *second*, and *minute*.

For example, *TeAL* can represent "*5 seconds after serA establishes a connection to nodeA, it drops the connection to nodeB*" as:

commence $dropConn(serA, nodeA)$ **at** 5 *seconds* **before**
terminate $estConn(serA, nodeB)$

TeAL can represent "*serA receives a message in 5 seconds after nodeA sends it*" as:

$received(serA, msg, nodeA)$ **nolaterthan** 5 *seconds* **after**
terminate previous $send(nodeA, msg, serA)$

TeAL can represent "*serA sends a message to nodeA in the next 5 seconds*" as:

$send(serA, msg, nodeA)$ **nolaterthan** 5 *seconds* **after**

The basic form of a *temporal constraint* is:

$$\textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_1 \textbf{ or } \ldots \textbf{ or } B_m; \tag{4}$$

where A_1, …, A_k and B_1, …, B_m are *temporal conditions* or their negation. If a temporal constraint is of the form:

$$\textbf{if } (A_{11} \textbf{ or } \ldots \textbf{ or } A_{1n}) \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_1 \textbf{ or } \ldots \textbf{ or } B_m;$$

this temporal constraint is equivalent to the following collection:

$$\begin{gathered}\textbf{if } A_{11} \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_1 \textbf{ or } \ldots \textbf{ or } B_m;\\ \text{or}\\ \ldots\\ \text{or}\\ \textbf{if } A_{1n} \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_1 \textbf{ or } \ldots \textbf{ or } B_m;\end{gathered}$$

Similarly, if a temporal constraint is of the form:

$$\textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_{11} \textbf{ and } \ldots \textbf{ and } B_{1n}) \textbf{ or } \ldots \textbf{ or } B_m;$$

this temporal constraint is equivalent to the following collection:

$$\begin{gathered}\textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_{11} \textbf{ or } \ldots \textbf{ or } B_m;\\ \text{and}\\ \ldots\\ \text{and}\\ \textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_{1n} \textbf{ or } \ldots \textbf{ or } B_m;\end{gathered}$$

It should be noted that if $k = 0$ and $m = 1$, we can view the temporal condition B_1 as a special temporal constraint. In *TeAL*, one can express "*A connected node should re-identify itself to the server within 10 seconds after the connection is established, or the server will drop the connection within 2 seconds*" as:

> **if not terminate** $(nodeA, serA)$ **nolaterthan** 10 *seconds* **after terminate** $connect(serA, nodeA)$,
> **then terminate** $dropConn(serA, nodeA)$ **nolaterthan** 2 *seconds*;

Following is the complete BNF of *TeAL*:

$< declaration > ::= < sortDeclaration > | < constantDeclaration >$
$| < agentDeclaration > | < variableDeclaration >$
$| < fluentDeclaration > | < actionDeclaration >$
$| < unitDeclaration >$
$< sort > ::= string$
$< constant > ::= string$
$< variable > ::= string$
$< agent > ::= < constant >$
$< fluentName > ::= string$
$< actionName > ::= string$
$< unit > ::= string$
$< sorts > ::= < sort > | < sorts >, < sorts >$
$< sortDeclaration > ::=$ **sort** $< sorts >;$
$< constants > ::= < constant > | < constants >, < constants >$
$< constantDeclaration > ::=$ **constant** $< sort > < constants >;$
$< variables > ::= < variable > | < variables >, < variables >$
$< variableDeclaration > ::=$ **variable** $< sort > < variables >;$
$< agents > ::= < agent > | < agents >, < agents >$
$< agentDeclaration > ::=$ **agent** $< agents >;$
$< fluentType > ::= < fluentName > (< sorts >)$
$< fluentDeclaration > ::=$ **fluent** $< fluentType >;$
$< actionType > ::= < actionName > (< agent > [, < sorts >])$
$< actionDeclaration > ::=$ **action** $< actionType >;$
$< units > ::= < unit > | < units >, < units >$
$< unitDeclaration > ::=$ **timeunit** $< units >;$
$< attribute > ::= < constants > | < variable >$
$< attributes > ::= < attribute > | < attributes >, < attributes >$

< *literal* > ::= < *fluentName* > (< *attributes* >)
| **not** < *literal* > | < *specialFluent* >
< *specialFluent* > ::= **inprogress** < *action* >
| **engaged** < *agent* >
< *literals* > ::= < *literal* > | < *literals* > **and** < *literals* >
| < *literals* > **or** < *literals* >
< *action* > ::= < *actionName* > (< *agent* > [, < *attributes* >])
< *actions* > ::= < *action* > | < *actions* > **and** < *actions* >

< *statement* > ::= < *actionDescription* >
| < *initialConstraint* > | < *temporalDescription* >
< *actionDescription* > ::= < *stateConstraint* >
| < *dynamicCausalLaw* > | < *executabilityCondition* >
< *stateConstraint* > ::= < *temporalCondition* > **if** < *temporalCondition* >
;
< *dynamicCausalLaw* > ::= < *prompt* > **causes** < *temporalCondition* >
if < *temporalCondition* >;
< *executabilityCondition* > ::= **impossible** < *prompts* >
if < *temporalCondition* >;
< *initialConstraint* > ::= **initially** < *literals* >;
< *temporalDescription* > ::= < *durationSpecification* >
| < *tempConstraint* >
< *durationSpecification* > ::= **duration** < *action* > *integer*
< *unit* >;
< *tempConstraint* > ::= < *temporalCondition* >;
| **if** < *temporalCondition* > **then** < *temporalCondition* >;
| **if** < *temporalCondition* > **then null**;

< *temporalCondition* > ::= < $timeReference_1$ > [$when_1$] | [$when_2$] |
not < *temporalCondition* > |
< *temporalCondition* > **and** < *temporalCondition* > |
< *temporalCondition* > **or** < *temporalCondition* > |
(< *temporalCondition* >)
$when_1$ ::= < *timeComparator* >< *timeModifier* >
$when_2$ ::= < *timeComparator* > [< *timeModifier* >]
< $timeReference_2$ >
< *timeComparator* > ::= **earlierthan** | **at** | **laterthan** |

noearlierthan | **nolaterthan**
$<$ *timeModifier* $>$ ::= x *units* **before** | x *units* **after**
$<$ *timeReference$_1$* $>$ ::= $<$ *prompt* $>$ | L
$<$ *timeReference$_2$* $>$ ::= $<$ *prompt* $>$ | **starttime** | L $<$ *prompt* $>$
::= [$<$ *promptOperator* $>$] [**next** | **previous**] $<$ *action* $>$
$<$ *prompt* $>$ ::= $<$ *prompt* $>$ | $<$ *prompts* $>$ **and** $<$ *prompts* $>$
$<$ *promptOperator* $>$ ::= **commence** | **terminate**

TeAL System Descriptions

Example 1. CM1 requirements
This examples uses requirements in CM1 dataset, a real dataset taken from NASA project.

TeAL theory:
% Comments are marked with
% Declarations:

% States for the simulation and the time horizon

% Sorts used in the simulation
sort node;
sort message;

constant node ccm, icu, scu, icui, scui;
constant message hk, cmd, hbeat, pdu, errorReport, nak;
constant interface icui, scui;
constant endunit icu, scu;
constant controlunit ccm;

variable node Sender, Receiver;
variable message Msg;
variable interface Inter;
variable endunit End;

% Agents, the entities in the simulation

```
% that can take actions
agent node;

% Fluents; state values for the simulation
fluent received(scui,pdu,scu);
fluent received(icui,cmd,ccm);
fluent received(scui,cmd,ccm);
fluent received(icui,hbeat,icu);
fluent received(ccm,hbeat,icui);
fluent received(icu,cmd,icui);
fluent received(scu,cmd,scui);
fluent error(cmd);
fluent correct(cmd);

% Actions for the simulation
action produce(ccm,hk);
action process(ccm,cmd);
action send(scui,pdu,scu);
action send(icui,cmd,ccm);
action send(scui,cmd,ccm);
action send(icui,hbeat,icu);
action send(ccm,hbeat,icui);
action send(icu,cmd,icui);
action send(scu,cmd,scui);
action verify(icui,cmd);
action discard(icui,cmd);
action send(icui,errorReport,ccm);
action send(icui,nak,icu);

% Time unit for the simulation
timeunit unit;

% Duration
duration produce(ccm,hk) 1 unit;
duration process(ccm,cmd) 1 unit;
duration send(scui,pdu,scu) 1 unit;
duration send(icui,cmd,ccm) 1 unit;
duration send(scui,cmd,ccm) 1 unit;
duration send(icui,hbeat,icu) 1 unit;
duration send(ccm,hbeat,icui) 1 unit;
duration send(icu,cmd,icui) 1 unit;
duration send(scu,cmd,scui) 1 unit;
duration verify(icui,Msg) 1 unit;
duration discard(icui,Msg) 1 unit;
duration send(icui,errorReport,ccm) 1 unit;
duration send(icui,nak,icu) 1 unit;

% Statements of simulation

% The Instrument Control Unit shall send real-time commands
% to the Interface of Control Component every B milliseconds.
% Default value: B=10
commence send(icu,cmd,icui) nolaterthan B unit after starttime;
if terminate send(icu,cmd,icui) then commence next send(icu,cmd,icui) at B
unit after;
impossible send(icu,cmd,icui) if not terminate previous send(icu,cmd,icui) at
B unit before and not nolaterthan B unit after starttime;

% Once a message is sent, it is received within Z units.
% Default value: Z=10
if terminate send(Sender,Msg,Receiver) then received(Receiver,Msg,Sender)
nolaterthan Z unit after;
impossible received(Receiver,Msg,Sender) if not terminate send(Sender,Msg,Receiver)
noearlierthan Z unit before;

% The Control Component shall send the heart beat message
% to the Interface of Instrument Control Unit at an interval
% of E milliseconds. The interface will send the message to
% the Instrument Control Unit.
% Default value: E=15
commence send(ccm,hbeat,icui) nolaterthan E unit after starttime;
if terminate send(ccm,hbeat,icui) then commence next send(ccm,hbeat,icui)
at E unit after;
impossible send(ccm,hbeat,icui) if not terminate previous send(ccm,hbeat,icui)
at E unit before and not nolaterthan E unit after starttime;
if received(icui,hbeat,ccm) then commence send(icui,hbeat,icu);
```

impossible send(icui,hbeat,icu) if not received(icui,hbeat,ccm);

% The Interface of Spacecraft Control Unit shall send
% one Program Data Update message to the Spacecraft
% Control Unit every H unit.
% Default value: H=10
commence send(scui,pdu,scu) nolaterthan H unit after starttime;
if terminate send(scui,pdu,scu) then commence next send(scui,pdu,scu) at H unit after;
impossible send(scui,pdu,scu) if not terminate previous send(scui,pdu,scu) at H unit before and not nolaterthan H unit after starttime;

% The Control Component shall produce housekeeping messages
% at a rate of X milliseconds.
% Default value: X=20
commence produce(ccm,hk) nolaterthan X unit after starttime;
if terminate produce(ccm,hk) then commence next produce(ccm,hk) at X unit after;
impossible commence produce(ccm,hk) if not terminate previous produce(ccm,hk) at X unit before and not nolaterthan X unit after starttime;

% The Interface of Spacecraft Control Unit shall be
% capable of receiving a telecommand from the Spacecraft
% Control Unit every G milliseconds and forward it to the
% Control Component.
% Default value: G=20
commence send(scu,cmd,scui) nolaterthan G unit after starttime;
if terminate send(scu,cmd,scui) then commence next send(scu,cmd,scui) at G unit after;
impossible send(scu,cmd,scui) if not terminate previous send(scu,cmd,scui) at E unit before and not nolaterthan G unit after starttime;
if received(scui,cmd,scu) then commence send(scui,cmd,ccm);
impossible send(scui,cmd,ccm) if not received(scui,cmd,scu);

% The Control Component shall process commands within
% F milliseconds of receipt from the Interface of
% Instrument Control Unit or the Spacecraft Control Unit.
% Default value: F=10

```
if received(ccm,cmd,Inter) then commence process(ccm,cmd) nolaterthan F
unit after;
impossible process(ccm,cmd) if not received(ccm,cmd,Inter) noearlierthan F
unit before;
```

Example 2. Double Play
This program simulates a double play in the sport of baseball. The batter starts at home plate with a runner on first base. First, the batter must hit the ball, which brings it in to play. Then the runners can run and the defense attempts to field the ball; once fielded, the players can throw the ball between bases and tag out the runners.

TeAL theory:

```
% Comments are marked with % % Declarations:

% States for the simulation and the time horizon
parameter numStates = 12;
parameter horizon = 300;

% Sorts used in the simulation
sort offense;
sort base;
sort defense;

constant offense batter,runner;
constant base home,first,second,outfield;

% Note - this program considers the ball part of the defense;
% this is so the output will track what happens to the ball
% (and so that only one thing can happen to the ball at a time).
constant defense ball;

% Agents, the entities in the simulation
% that can take actions
agent offense;
agent defense;

% Fluents; state values for the simulation
```

```
fluent onBase(offense,base);
fluent inPlay(defense);
fluent ballFielded(defense);
fluent ballAt(defense, base);

% Actions tracked by the simulation
show action run(offense,base,base);
show action hit(offense);
show action field(defense);
show action throw(defense,base,base);
show action tag(defense,base);

% Durations
parameter RunTime = 270;
parameter TagTime = 10;
parameter HitTime = 20;
parameter FieldTime = 60;
parameter Grab = 20;
parameter LongThrow = 120;
parameter ShortThrow = 80;

duration run(batter,home,first) RunTime unit;
duration run(batter,home,second) RunTime unit;
duration run(batter,first,home) RunTime unit;
duration run(batter,first,second) RunTime unit;
duration run(batter,second,home) RunTime unit;
duration run(batter,second,first) RunTime unit;
duration run(runner,home,first) RunTime unit;
duration run(runner,home,second) RunTime unit;
duration run(runner,first,home) RunTime unit;
duration run(runner,first,second) RunTime unit;
duration run(runner,second,home) RunTime unit;
duration run(runner,second,first) RunTime unit;
duration tag(ball,second) TagTime unit;
duration tag(ball,first) TagTime unit;
duration hit(batter) HitTime unit;
duration field(ball) FieldTime unit;
duration throw(ball,outfield,first) LongThrow unit;
duration throw(ball,outfield,second) ShortThrow unit;
duration throw(ball,first,second) ShortThrow unit;
duration throw(ball,second,first) ShortThrow unit;
duration throw(ball,first,outfield) LongThrow unit;
duration throw(ball,second,outfield) ShortThrow unit;
duration throw(ball,outfield,home) LongThrow unit;
duration throw(ball,first,home) ShortThrow unit;
duration throw(ball,second,home) LongThrow unit;
duration throw(ball,home,first) ShortThrow unit;
duration throw(ball,home,second) LongThrow unit;

% Initial state of simulation

% Batter at home plate, runner at first
initially onBase(batter,home);
initially onBase(runner,first);
initially ballAt(ball, outfield);

% Rules for running

% Running moves the player from one base to another
terminate run(P,L1,L2) causes not onBase(P,L1);
terminate run(P,L1,L2) causes onBase(P,L2);

% The ball must be in play to run
impossible commence run(P,L1,L2) if not inPlay(ball);

% The bases must be run in order
impossible commence run(P,L1,L2) if not onBase(P,L1);
impossible commence run(P,L1,second) if not onBase(P,first);
impossible commence run(P,L1,first) if not onBase(P,home);

% You can only hit the ball from home
impossible commence hit(batter) if not onBase(batter,home);

% Hitting the ball causes it to be in play
terminate hit(batter) causes inPlay(ball);
```

```
% You can't field the ball if it's not in play
% or it has already been fielded
impossible commence field(ball) if not inPlay(ball);
impossible commence field(ball) if ballFielded(ball);

% Fielding the ball causes it to be fielded
terminate field(ball) causes ballFielded(ball);

% The ball has to be at a base to tag that base
impossible commence tag(ball,B) if not ballAt(ball,B);
impossible commence tag(ball,B) if not inPlay(ball);

% The ball has to have been fielded (i.e., retrieved by the fielder)
% to be thrown; it also has to be where it's being thrown from
% to be thrown...
impossible commence throw(ball,B1,B2) if not ballAt(ball,B1);
impossible commence throw(ball,B1,B2) if not ballFielded(ball);

% Throwing the defense causes it to cease to be at its initial
% position and be at the base it was thrown to
terminate throw(ball,I,B) causes not ballAt(ball,I);
terminate throw(ball,I,B) causes ballAt(ball,B);

% Force the runners to get to base shortly after the ball is hit
if terminate hit(batter) then terminate run(batter,home,first) nolaterthan
280 unit after;
if terminate hit(batter) then terminate run(runner,first,second) nolaterthan
280 unit after;

% And give the defense a time limit to complete both tags
terminate tag(ball,second) nolaterthan 290 unit after starttime;
terminate tag(ball,first) nolaterthan 290 unit after starttime;
```

References

[1] C. Baral and M. Gelfond, “Reasoning agents in dynamic domains,” *Logic- based artificial intelligence*, pp. 257-279, 2000.