Proceedings of the
2nd Workshop on Artificial Intelligence and Formal Verification, Logics, Automata and Synthesis (OVERLAY),
September 25, 2020

A Language for Timeline-based Planning*

Giulio Bernardi[1], Amedeo Cesta, Andrea Orlandini, Alessandro Umbrico[2], and Marta Cialdea Mayer[3]

[1]Indipendent Researcher
[2]National Research Council, ISTC-CNR, Rome, Italy
[3]ROMA TRE University, Rome, Italy

Abstract

In order to represent planning problems, Artificial Intelligence planning languages are used to describe environment's conditions and operators which can lead to desired goals by generating a chain of actions based on these conditions and operators. Historically, several languages have been defined according to different planning paradigms. Most of these languages are often strongly related to specific planning systems and lack a clear formal semantics. A recent work defines a formal framework for timeline-based planning and scheduling providing a clear semantics for planning concepts needed to specify timeline-based planning problems. This abstract introduces GHOST, a new specification language for timeline-based planning and scheduling based on such a framework. The main aim of GHOST is to provide a concrete and compact specification language for timeline-based planning and scheduling domains and problems dealing also with uncertainty.

1 Introduction

Among knowledge-based systems, automated planning systems require the definition of domains and problem instances to be solved. This entails suitable planning specification languages to describe environment's conditions and operators which can lead to desired goals by generating a chain of actions based on these conditions and operators. Historically, several planning specification languages have been defined according to different planning paradigms. Most of these languages are often strongly related to specific planning systems and lack a clear formal semantics.

Classical action-based planners usually adopt description languages derived from logic-based formulation such as, e.g., STRIPS [11], the Situation Calculus [21] and ADL [20]. With the aim of encouraging the empirical evaluation of planners performance, and the development of standard sets of problems all in comparable notations, a Planning Domain Description Language (PDDL) [17] was introduced roughly reproducing the expressiveness of Pednault's ADL [20] for propositions and the expressiveness of UMCP [16] for actions. PDDL became the reference language for International Planning Competitions[1] and, with its many extensions [14], it is a sort of standard in the A.I. planning community. Unfortunately, the above languages are seldom used in real world applications of planning and scheduling technologies.

*The CNR authors are partially supported by European Commission under the ShareWork project (G.A. 820807). Giulio Bernardi is now working at Google.

[1]http://www.icaps-conference.org/index.php/Main/Competitions

Indeed, a gap exists between the above languages and application-oriented proposals. The main difference lies in the considered problems and domains: quite simplified in the IPC case, more sophisticated in the applicatiion scenarios.

Attempts to bridge this gap have been done in Planning and Scheduling (P&S) with the aim of dealing with more complex domains while following a principled approach that allows the generalization of results. Following this trend, some P&S languages have been proposed considering other planning approaches. Application oriented planners are usually endowed with their own peculiar description languages, for example the Task Formalism in O-Plan [9], and the specialized languages used in OPIS [22], IxTeT [15] and HSTS [18]. In [6], Cesta and Oddi propose a Domain Description Language (DDL) for ESA mission applications also developing a formal analysis of the representation language. DDL constitutes the main specification language for APSI-TRF [5, 13] and, with some variations [12], for most of its applications [4, 12]. At NASA, the development of the EUROPA planning system [2] fostered the definition of specific description languages, i.e., NDDL[2] and ANML [10].

Unfortunately, most of the above mentioned languages usually lack a reference to a formal conceptual framework and present some limits related to expressive power and syntax. A recent work [8] defines a formal framework for timeline-based planning and scheduling providing a clear semantics for planning concepts and specifying timeline-based planning problems with uncertainty [7]. This paper presents GHOST, a new language for timeline-based planning and scheduling based on such a framework. The main aim of GHOST is to specifically address the above mentioned weaknesses and having readability, maintainability, generality and increased expressive power among its design goals. GHOST also provides a concrete and compact modeling language for timeline-based planning and scheduling with uncertainty.

2 Timeline-based Planning and Scheduling

A timeline-based planning domain contains the characterization of a set of *state variables*, representing the components of a system. A *state variable* x is characterized by the set of values it may assume, denoted by values(x), possible upper and lower bounds on the duration of each value, and rules governing the correct sequencing of such values. A *timeline* for a state variable is made up of a finite sequence of valued intervals, called *tokens*, each of which represents a time slot where the variable assumes a given value. In general, timelines may be *flexible*, i.e., the start and end times of each of its tokens are not necessarily fixed time points, but may range in given intervals. For the sake of generality, temporal instants are taken from an infinite set of non negative numbers $\mathbb{T}$, including 0. The notation $\mathbb{T}^\infty$ will be used to denote $\mathbb{T} \cup \{\infty\}$, where $t < \infty$ for every $t \in \mathbb{T}$.

Tokens in a timeline for the state variable x are denoted by expressions of the form x^i, where the superscript indicates the position of the token in the timeline. Each token x^i is characterized by a value $v_i \in$ values(x), an end time interval $[e_i, e'_i]$ referred to as end_time(x^i), and a duration interval $[d_i, d'_i]$ (as usual, the notation $[x, y]$ denotes the closed interval $\{t \mid x \leq t \leq y\}$). The start time interval start_time(x^i) of the token x^i is $[0, 0]$ if x^i is the first token of the timeline (i.e. $i = 1$), otherwise, if $i > 1$, start_time(x^i) = end_time(x^{i-1}). So, a token has the form $x^i = (v_i, [e_i, e'_i], [d_i, d'_i])$ and a timeline is a finite sequence of tokens $x^1, \ldots, x^k$. The metasymbol FTL (FTL_x) will henceforth be used to denote a timeline (for the state variable x), and **FTL** to denote a set of timelines. Being tokens flexible, their exact start and end times will be decided at execution time. Tokens can be either *controllable* (the controller can decide their end time, i.e. their duration), or *uncontrollable* (their duration depends on the environment's choices). The controllability of tokens start times depends on the controllability of the respective previous token in the timeline. Each token is consequently equipped also with a *controllability tag*, identifying the class it belongs to.

A *scheduled timeline* is a particular case where each token has a singleton $[t, t]$ as its end time, i.e., the end times are all fixed. A *schedule* of a timeline FTL_x is essentially obtained from FTL_x by narrowing down token end times to singletons (time points) in such a way that the duration requirements are fulfilled. In a given timeline-based domain, the behavior of state variables may be restricted by requiring that time intervals with given state variable values satisfy some temporal constraints. Such constraints are stated as a set of *synchronization rules* which relate tokens on possibly different timelines through temporal relations

[2] https://github.com/nasa/europa/wiki/Quick-Start

between intervals or between an interval and a time point. These temporal relations refer to token start or end points, that will henceforth be called *events*. If **FTL** is a set of timelines and tokens(**FTL**) the set of the tokens in **FTL**, then the set $\Upsilon(\mathbf{FTL})$ of the *events* in **FTL** is the set containing all the expressions of the form start_time(x^i) and end_time(x^i) for $x^i \in$ tokens(**FTL**). A *temporal relation* on tokens has one of the following forms: $p \leq_{[lb,ub]} p', p \leq_{[lb,ub]} t, t \leq_{[lb,ub]} p$ where $p, p' \in \Upsilon(\mathbf{FTL})$, $t, lb \in \mathbb{T}$ and $ub \in \mathbb{T}^\infty$ and $lb \leq ub$.

Intuitively, $p \leq_{[lb,ub]} p'$ states that the token start/end point denoted by p occurs from lb to ub time units before that denoted by p'; $p \leq_{[lb,ub]} t$ states that the token start/end point denoted by p occurs from lb to ub time units before the time point t and the third relation that it occurs from lb to ub time units after t. Other relations between tokens ([1]) can be defined in terms of the primitive ones, e.g.: x^i before$_{[lb,ub]}$ y^j is the same as end_time(x^i) $\leq_{[lb,ub]}$ start_time(y^j); x^i during$_{[lb_1,ub_1][lb_2,ub_2]}$ y^j can be defined as start_time(y^j) $\leq_{[lb_1,ub_1]}$ start_time(x^i) and end_time($\mathrm{x^i}$) $\leq_{[lb_2,ub_2]}$ end_time(y^j); a *contains* relation is its converse: x^i contains$_{[lb_1,ub_1][lb_2,ub_2]}$ y^j if and only if y^j during$_{[lb_1,ub_1][lb_2,ub_2]}$ x^i. Temporal relations are also used to state the synchronization rules of the planning domain. Here, it is sufficient to say that such rules allow the modeler to state requirements of the following form: for every token x_0^i where the state variable x_0 assumes the value v_0, there exist tokens $x_1^{i_1}, \ldots, x_n^{i_n}$ where the state variables $x_1, \ldots, x_n$ hold some given specified values, and all these tokens are related one to another by some given temporal relations. Unconditioned synchronization rules are also allowed, and useful for stating domain invariants (initial situation for problems) and planning goals. A *flexible plan* Π is a pair $(\mathbf{FTL}, \mathcal{R})$, where **FTL** is a set of timelines and $\mathcal{R}$ is a set of temporal relations, involving tokens in some timelines in **FTL**. An *instance* of the flexible plan $\Pi = (\mathbf{FTL}, \mathcal{R})$, is any schedule of **FTL** that satisfies every relation in $\mathcal{R}$.

Resources have been integrated in the theoretical framework in [23], but entering into details is outside the scope of the present work.

3 The GHOST Language

GHOST is a completely new language influenced by a number of popular programming languages and DDL (as GHOST is supposed to replace it). Its name is given after its feature of being "transparent" for engineers, i.e. not influencing the modeling process. This section aims at giving the flavour of the language, showing its expressivity, generality and compactness. As a matter of fact, an important design principle of the language is "Name only what it's meaningful": unused variables, parameters and default values can generally be omitted.

A GHOST file (specifying either a domain or a specific problem, or both) consists of a collection of declarations. The most important ones are the definition of *types*. Beyond usual types such as enumerated or interval ones, types may represent state variables and resources. The declaration of a state variable type contains all needed information: values and their allowed durations and transitions, along with possible synchronization rules governing given values. The following simple example shows the definition of an interval type (`Coord`, which is an integer in a given interval), an enumerated type and a state variable (`sv`) one:

```
type coord = int [-1000,+1000];
type room = enum (A,B);

type Robot = sv (
  uncontr GoingTo(coord x, coord y) [10, 30] -> At(x, y);
  At(coord x, coord y) -> GoingTo;
synchronize:
  GoingTo -> during Camera.PointingAt(0,0);
variable:
  allowed_in: room;
);
```

The initial rows of the description of the `Robot` type define the state variable values and the respective allowed transitions: `GoingTo` (which is uncontrollable) and `At` (controllable, by default), both having a pair of `coord` as parameters; `GoingTo` and `At` can transition to only one possible state. The `GoingTo` value has a duration constraint, expressed by the interval $[10, 30]$, while `At` has none (default), i.e., its duration interval is $[1, \infty]$. The `synchronize` section (when present) introduces synchronization rules. In

this case, it is required that every token with a `GoingTo` value (no need to state its parameters) must occur while the `Camera` state variable assumes the value `PointingAt(0,0)`. The temporal operators used in such rules may be any *a*-la-Allen [1] constraint like, e.g., `meets`, `during` and so on. Like this example shows, state variable values are allowed to have parameters, on which constraints can be expressed both in transitions and synchronization rules.

The definition of a state variable type is allowed to have parameters, specified in the `variable` section (when present). In the above example, every *instance* of the type has a corresponding `room` in which it is allowed to enter. In other terms, variables must be bound to actual values when defining instances of the type (see below). It is worth pointing out that variable types can also be state variables. For instance, one might declare a variable `friend: Robot` (every Robot has another Robot as a friend).

A state variable type can also be defined as either *external* (all its values being uncontrollable) or *planned* (default).

Instances of a type variable (called *components*) are the "actual" state variables, as theoretically defined, i.e., at runtime, timelines for them will be built by the planner. State variables types are useful when problem instances may have multiple components of the same type. When this is not the case, a component can also be defined directly (its "fields" obviously excluding variables). For example, an instance of the state variable type `Robot`, allowed to enter room A, may be defined as follows:

```
comp Robot1 =  Robot [allowed_in=A];
```

or simply, since `Robot` has a single parameter:

```
comp Robot1 =  Robot [A];
```

GHOST also allows one to define resources, both renewable and consumable ones. Like for the case of state variables, resource types can be defined. For instance

```
type tank = resource (0,100);
```

defines a type for the consumable resource `tank`, whose capacity varies from 0 to 100. Resource declarations can be more complicated, and may contain also synchronizations and variable binding, but we do not enter into details here. A resource can be referred to in a state variable `synchronize` section.

A specific initialization section in a GHOST file describes the situation of a specific scenario, and is usually written in the problem file. It contains the description of known facts about the initial state of the system and desired goals, along with other problem-specific parameters, such as a start time (the first available time instant) and the planning horizon (all of them obviously having default values).

Here follows a simple example of a complete GHOST specification file (where singleton intervals of the form `[t,t]` are abbreviated by `t`).

```
domain TrafficLightDomain ;

//State Variable types
type TrafficLight = sv (
       Red 30 -> Green ;
       Green 20 -> Yellow ;
       Yellow 10 -> Red ;
synchronize :
       Green -> starts other.Red ;
variable :
       other : TrafficLight ;
);

//Components
comp TL1 : TrafficLight [ TL2 ];
comp TL2 : TrafficLight [ TL1 ];

//Facts, goals and temporal parameters
init (
     var horizon = 200;
     var resolution = 300;
     fact TL1.Green at 0;
     fact TL2.Red at 0;
     goal TL2.Yellow ;
)
```

4 Conclusions

A recent work defines a formal framework for timeline-based planning and scheduling providing a clear semantics for planning concepts needed to specify timeline-based planning problems. This abstract briefly presented GHOST, a new specification language for timeline-based planning and scheduling based on such a framework. The main aim of GHOST is to provide a concrete and compact specification language for timeline-based planning and scheduling domains and problems dealing also with uncertainty. The definition of GHOST is part of a more general initiative addressing Knowledge Engineering issues in timeline-based P&S. Indeed, GHOST is defined as supporting language for KEEN, a Knowledge Engineering Environment for developing Timeline-based Planning applications [19].

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, Nov. 1983.

[2] T. Bedrax-Weiss, C. McGann, A. Bachmann, W. Edgington, and M. Iatauro. EUROPA2: User and contributor guide. Technical report, Technical report, NASA Ames Research Center, 2005.

[3] A. Ceballos, S. Bensalem, A. Cesta, L. de Silva, S. Fratini, F. Ingrand, J. Ocón, A. Orlandini, F. Py, K. Rajan, R. Rasconi, and M. van Winnendael. A Goal-Oriented Autonomous Controller for space exploration. In *ASTRA 2011, 11th Symposium on Advanced Space Technologies in Robotics and Automation*, 2011.

[4] A. Cesta, G. Cortellessa, S. Fratini, and A. Oddi. MrSPOCK: Steps in Developing an End-to-End Space Application. *Computational Intelligence*, 27(1), 2011.

[5] A. Cesta and S. Fratini. The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. In *27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2008)*, 2008.

[6] A. Cesta and A. Oddi. DDL.1: a formal description of a constraint representation language for physical domains. In M. Ghallab and A. Milani, editors, *New directions in AI planning*, pages 341–352. IOS Press, 1996.

[7] M. Cialdea Mayer and A. Orlandini. An executable semantics of flexible plans in terms of timed game automata. In F. Grandi, M. Lange, and A. Lomuscio, editors, *The 22nd International Symposium on Temporal Representation and Reasoning (TIME)*. IEEE, 2015.

[8] M. Cialdea Mayer, A. Orlandini, and A. Umbrico. Planning and execution with flexible timelines: a formal account. *Acta Informatica*, 53(6):649–680, 2016.

[9] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49 – 86, 1991.

[10] W. C. David E. Smith, Jeremy Frank. The ANML language. In *Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2008.

[11] R. E. Fikes and N. J. Nilsson. STRIPS, a retrospective. *Artificial Intelligence*, 59(1):227 – 232, 1993.

[12] S. Fratini, A. Cesta, R. De Benidictis, A. Orlandini, and R. Rasconi. APSI-based deliberation in Goal Oriented Autonomous Controllers. In *ASTRA 2011, 11th Symposium on Advanced Space Technologies in Robotics and Automation*, 2011.

[13] S. Fratini, G. Cortellessa, A. Oddi, and A. Cesta. *Software User Manual – APSI Framework*. European Space Agency, issue 3, revision 2 edition, 2010.

[14] A. Gerevini and D. Long. Preferences and soft constraints in pddl3. In A. Gerevini and D. Long, editors, *ICAPS workshop on Planning with Preferences and Soft Constraints*, pages 46–53, 2006.

[15] M. Ghallab and H. Laruelle. Representation and control in IXTET, a temporal planner. In *Proceedings of the Second Intl. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*, pages 61–67, 1994.

[16] D. S. N. Kutluhan Erol, James Hendler. UMCP: A sound and complete procedure for hierarchical task-network planning. In *2nd International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 249–254, 1994.

[17] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[18] N. Muscettola. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., editor, *Intelligent Scheduling*. Morgan Kauffmann, 1994.

[19] A. Orlandini, G. Bernardi, A. Cesta, and A. Finzi. Planning meets verification and validation in a knowledge engineering environment. 8(1):87–100, 2014.

[20] E. P. D. Pednault. ADL: exploring the middle ground between strips and the situation calculus. In *1st international conference on Principles of knowledge representation and reasoning*, pages 324–332, 1989.

[21] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[22] S. F. Smith. The OPIS framework for modeling manufacturing systems. Technical report, The Robotics Institute, Carnegie Mellon University, Pittsburgh. Tech. Rep. CMU-RI-TR-89-30, 1989.

[23] A. Umbrico, A. Cesta, M. Cialdea Mayer, and A. Orlandini. Integrating resource management and timeline-based planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, pages 264–272, 2018.

A A Ghost Domain Example

This section contains some complete code examples written in GHOST language. The domain was originally defined in DDL for the GOAC project [3]. The code shown below has been translated to GHOST language from the original DDL definition, maintaining planner-specific information by the means of annotations.

```
domain GOAC_Domain;

type coordinate = int [-1000, +1000];
type angle = int [-360, 360];
type file_id = int [0, 100];

comp RobotBase : sv (
  GoingTo(coordinate x, coordinate y) [10, 30] -> At(x, y);
  At(coordinate x, coordinate y) [1, +INF] -> GoingTo;
  StuckAt(coordinate x, coordinate y) [1, +INF] -> GoingTo;
synchronize:
  GoingTo -> during Platine.PointingAt(0,0);
);

comp Platine : sv (
  MovingTo(angle pan, angle tilt) [1, +INF] -> PointingAt(pan, tilt);
  PointingAt(angle pan, angle tilt) [10, 20] -> MovingTo;
);

comp Camera : sv (
  CamIdle [1, +INF] -> TakingPicture;
  TakingPicture(file_id, coordinate, coordinate, angle pan, angle tilt) 10 -> CamIdle;
synchronize:
  TakingPicture(_, x, y, pan, tilt) -> (
    during RobotBase.At(x, y);
    during Platine.PointingAt(pan, tilt);
  );
);

comp Communication : sv (
  CommIdle [1, +INF] -> Communicating;
  Communicating(file_id) [10, 20] -> CommIdle;
synchronize:
  Communicating -> (
    during RobotBase.At;
    @(?) during CommunicationVW.Visible;
  );
);

external comp CommunicationVW : sv(None -> Visible, Visible -> None);

comp MissionTimeline : sv (
  Idle [1, +INF] -> (TakingPicture, Communicating, At);
  TakingPicture(file_id, coordinate, coordinate, angle, angle) 10 -> Idle;
  Communicating(file_id) [10, 20] -> Idle;
  At(coordinate, coordinate) [1, +INF] -> Idle;
synchronize:
  TakingPicture(file_id, x, y, pan, tilt) -> (
    @(!) var cd1 = Camera.TakingPicture(file_id, x, y, pan, tilt);
    @(!) var cd5 = Communication.Communicating(file_id);
    @(!) meets MissionTimeline.Idle;
    contains cd1;
    contains(_,0) cd5;
    cd1 < cd5;
  );
  At(x, y) -> equals RobotBase.At(x, y);
);
```

Listing 1: The GOAC Domain in GHOST

B The GHOST Language Grammar

This is the definition of the GHOST language in EBNF form.

```
(* <WS>, <ML_COMMENT>, <SL_COMMENT>, <SL_ANNOTATION>,
<BR_ANNOTATION>, and <DIRECTIVE> may occur anywhere *)

<Ghost> ::=
    [(<DomainDecl> | <ProblemDecl>) <SEP>]
    {<ImportDecl> <SEP>}
    [<TopLevelDeclaration> {<SEP> <TopLevelDeclaration>} {<SEP>}];

<DomainDecl> ::= 'domain' <ID>;
<ProblemDecl> ::= 'problem' <ID>;
<ImportDecl> ::= 'import' <ID>;

<TopLevelDeclaration> ::= <TypeDecl> | <CompDecl> | <ConstDecl> | <InitSection>;
<TypeDecl> ::= <SimpleType> | <ComponentType>;

<SimpleType> ::= <IntDecl> | <EnumDecl>;
<ComponentType> ::= <SvDecl> | <ResourceDecl>;
<IntDecl> ::=  [<Externality>] 'type'<ID> '=' 'int' <Interval>;
<EnumDecl> ::= [<Externality>] 'type'<ID> '='
    'enum' '(' [<EnumLiteral> {<SEP> <EnumLiteral>}] ')';
<SvDecl> ::=  [<Externality>] 'type'<ID> '=' 'sv' [<ID>] <SvBody>;
<ResourceDecl> ::=  [<Externality>] 'type'<ID> '=' 'resource' [<ID>] <ResourceBody>;

<SvBody> ::= ['('
    <UnnamedTransitionSection>
    {<TransitionSection> |
    <SynchronizeSection> |
    <VariableSection>}
')'];

<TransitionSection> ::= 'transition:' [<TransConstraint> {<SEP> <TransConstraint>} {<SEP>}];
<SynchronizeSection> ::= 'synchronize:'
    [<Synchronization> {<SEP> <Synchronization>} {<SEP>}];
<VariableSection> ::= 'variable:' [<ObjVarDecl> {<SEP> <ObjVarDecl>} {<SEP>}];
<UnnamedTransitionSection> ::= [<TransConstraint> {<SEP> <TransConstraint>} {<SEP>}];

<TransConstraint> ::= [<Controllability>] <ValueDecl>
    [<IntvOrDflt>] ['->' <TransConstrBody>];

<ValueDecl> ::= <ID> [<FormalParList>];
<SimpleInstVal> ::= <ID> [<ArgList>];
<QualifInstVal> ::= <QUALID> [<ArgList>];
<InstVal> ::= 'this' | <QualifInstVal> | <ResConstr>;

<TransConstrBody> ::= (<SingleTransConstr>) | ('('
    [<SingleTransConstr> {<SEP> <SingleTransConstr>} {<SEP>}] ')');
<SingleTransConstr> ::= 'inherited' | <LocVarDecl> | <GenericExpression>;

<Synchronization> ::= <TriggerType> '->' <SyncBody> {'or' <SyncBody>};
<SyncBody> ::= (<SingleSyncConstr>) | ('('
    [<SingleSyncConstr> {<SEP> <SingleSyncConstr>} {<SEP>}] ')');
<SingleSyncConstr> ::=  (<TemporalExp>) | <LocVarDecl> | <GenericExpression>;

<TriggerType> ::= <ResSimpleInstVal> | <SimpleInstVal>;

<ResConstr> ::= <ResourceAction> <QualifInstVal>;

<ObjVarDecl> ::= <ID> ':' <ID>;

<ResourceBody> ::= ['('[<ConstExpr> [<SEP> <ConstExpr>] {<SEP>}]
    {<SynchronizeSection> | <VariableSection>}
')'];

<ResSimpleInstVal> ::= <ResourceAction> '(' <ID_> ')';

<FormalParList> ::= '(' [<ID> [<ID_>] {<SEP> <ID> [<ID_>] } {<SEP>} ] ')';
<ArgList> ::= '(' [<GenericExpression> {<SEP> <GenericExpression>} {<SEP>} ] ')';
<BindList> ::= '['[<ID> '='] <ID> {<SEP> [<ID> '='] <ID> } {<SEP>} ']';

<LocVarDecl> ::= 'var' <ID> '=' <RValue>;
<RValue> ::= (<TemporalExp>) | <GenericExpression>;

<BasicExp> ::= '(' <EqExp> ')' | '_' | <TimePointOp> | <InstVal> | <NumAndUnit>;
<Term> ::= <BasicExp>  {('*' | '/' | '%') <BasicExp>};
<SumExp> ::= <Term>  {( '+' | '-' ) <Term>};
```

```
<CompExp> ::= <SumExp>  [('<' | '<=' | '>' | '>=' ) <SumExp>];
<EqExp> ::= <CompExp>  [('=' | '!=') <CompExp>];

<TemporalExp> ::= [<SumExp>] <TemporalRelation> <SumExp>;

<TemporalRelation> ::=
    '=' | '!=' | 'equals' | '|=' | 'meets' | '<' | '>' | 'starts' | 'finishes' |
    (('before' | 'after') ['(' [<IntvOrDflt>] ')'] ) |
    ( ('contains'|'during') ['(' [<IntvOrDflt> [<SEP> <IntvOrDflt>]] ')'] );

<GenericExpression> ::= <EqExp>;

<CBasicExp> ::= '(' <CSumExp> ')' | '_' | <ID> | ((<NumAndUnit>) | (<Interval>));
<CTerm> ::= <CBasicExp>  {('*' | '/' | '%') <CBasicExp>};
<CSumExp> ::= <CTerm>  {('+' | '-' ) <CTerm>};

<ConstExpr> ::= <CSumExp>;

<Externality> ::= 'planned' | 'external';
<Controllability> ::= 'contr' | 'uncontr';
<ResourceAction> ::= 'require' | 'produce' | 'consume';

<CompDecl> ::= <NamedCompDecl> | <AnonSVDecl> | <AnonResDecl>;
<NamedCompDecl> ::= [<Externality>] 'comp' <ID> ':' <ID> <CompBody>;
<AnonSVDecl> ::= [<Externality>] 'comp' <ID> ':' 'sv' <CompSVBody>;
<AnonResDecl> ::= [<Externality>] 'comp' <ID> ':' 'resource' <CompResBody>;

<CompBody> ::= ( <CompResBody>) | <CompSVBody>;

<CompSVBody> ::= [<BindList>]['('
    <UnnamedTransitionSection>
    {<TransitionSection> | <SynchronizeSection> }
')'];

<CompResBody> ::= [<BindList>]['(' [<ConstExpr> [<SEP> <ConstExpr>] {<SEP>}]
    {<SynchronizeSection>}
')'];


<InitSection> ::= 'init' (<InitConstr> | ('('
    [<InitConstr> {<SEP> <InitConstr>} {<SEP>}] ')'));
<InitConstr> ::= <LocVarDecl> | <FactGoal>;
<FactGoal> ::= ('fact'|'goal') <InstVal> ['at' <AtParams>];

<ConstDecl> ::= 'const' <ID> '=' <ConstExpr>;

<AtParams> ::= <AtParamsNamed> | ( <AtParamsPos>);
<AtParamsNamed> ::= 3 * [ ('start' | 'duration' | 'end') '=' <IntvOrDflt>];
<AtParamsPos> ::= [<IntvOrDflt> [<IntvOrDflt> [<IntvOrDflt>]]];

<TimePointOp> ::= ( ('start'|'end') '(' <GenericExpression> ')');

<EnumLiteral> ::= <ID>;
<IntvOrDflt> ::= ('_' | <Interval>);
<Interval> ::= ( '[' <NumAndUnit> <SEP> <NumAndUnit>']') | <NumAndUnit>;

<NumAndUnit> ::= (<Number> [<ID>]);
<Number> ::= <PosNumber> | <NegNumber>;
<PosNumber> ::= ['+'] (<INT> |'INF');
<NegNumber> ::= '-' (<INT> |'INF');

<QUALID> ::= <ID>{'.'<ID>};
<ID_> ::= '_' | <ID>;

(* Terminals *)

<SL_ANNOTATION> ::= '@' { <ANYCHAR> - ('\n'|'\r') } [['\r'] '\n'];
<BR_ANNOTATION> ::= ? '@(' -> ')'; ?
<DIRECTIVE> ::= '$' { <ANYCHAR> - ('\n'|'\r') } [['\r'] '\n'];

<SEP> ::= (',' | ';');
<ID> ::= ('a'..'z'|'A'..'Z') {'a'..'z'|'A'..'Z'|'0'..'9'|'_'};
<INT> ::= ('0'..'9') {'0'..'9'|'_'};
<ML_COMMENT> ::= ? '/*' -> '*/'; ?
<SL_COMMENT> ::= '//' { <ANYCHAR> - ('\n'|'\r') } [['\r'] '\n'];

<WS> ::= (' '|'\t'|'\r'|'\n') {' '|'\t'|'\r'|'\n'};

<ANYCHAR> ::= ? any ASCII character ?
```