CASL-MDL, modelling dynamic systems with a formal foundation and a UML-like notation (full report)

Christine Choppy[1] and Gianna Reggio[2]

[1] LIPN, UMR CNRS 7030 - Université Paris 13, France
Christine.Choppy@lipn.univ-paris13.fr
[2] DISI, Università di Genova, Italy
gianna.reggio@disi.unige.it

1 Introduction

The starting point of this work is to provide a visual help for writing and reading specifications in CASL-LTL [9], a CASL [7] extension for dynamic systems, that is suitable for specifying different kinds of dynamic systems, and at different levels of abstraction [3, 5]).

The authors already addressed the problems related to use in practice CASL-LTL for specifying/modelling, e.g., by the attempt to provide an alternative graphical syntax, a general purpose development method based on it [5], and some specializations for specific class of problems defined by problem frames [4].

We note the worldwide diffusion of the UML [10] as a modelling notation in many different fields, although it is informal, without a precise semantics, and some of its constructs result from the fact that the UML has been defined by putting together and turning into object-oriented various existing notations (e.g., entity relationships diagrams, state charts, and message sequence charts). However, the UML is visual, there exists a large number of tools for producing its diagrams, and it is quite flexible so as to accommodate most users.

We think that it may be possible to develop a notation that has the nice properties of both the CASL-LTL and the UML without their defective sides, and that moreover has some chances to be used in practice, and thus is the reason for proposing CASL-MDL. We would like to pursue this objective under the general idea of the well-founded methods [2].

We already worked along this direction trying to propose a visual notation as UML-like as possible but with a clear semantics, and pragmatics given by an underlying corresponding CASL-LTL specification. From the UML state machine we have derived the interaction machines (see, e.g., [1] and [6]) which offer a visual presentation to CASL-LTL conditional specifications with initial semantics of simple dynamic systems; but the interaction machines are more general and powerful than UML state machines, and are not restricted to reactive behaviour. It is a rather nice and intuitive visual notation, and its editing/drawing may be achieved with UML tools.

CASL-LTL uses temporal logic formulae to express properties on the dynamic behaviour of simple systems, but this is not an easy to read notation. CASL-MDL has some diagrams similar to the UML sequence diagrams that may used to express some of these properties, and they are more expressive than the UML ones, and much more similar to the live charts of [8].

CASL-MDL has then a type diagram that play the same role of the UML class diagram, but instead of classes allows to introduce the datatypes and the dynamic types need to type the elements of the model.

Summarizing we can say that CASL-MDL is a visual notation strictly corresponding to CASL-LTL specifications whose visual constructs have been borrowed by the UML, and this choice allows also to borrow the professional editors available for the UML, making thus possible just now to easily write CASL-MDL specifications, without having to wait for the development of dedicated tools.

In Sect. 2 we introduce the CASL-MDL models, in Sect. 3 and in Sect. 5 the type diagrams and the sequence diagrams respectively, and finally in the Sect. 9 the conclusions and the future works.

In the paper we will use as a running example the modelling of ASSOC, a case study that requires to describe the functioning of a consortium of associations where associations have boards with a chair and several members, and board meetings take place, to communicate informations or to take decisions via voting. ASSOC has been used as a paradigmatic case study to present a method for the business modelling based on the UML, and thus we think that it may be a good workbench to test the modelling power of CASL-MDL. Fragments of the model of ASSOC will be used to illustrate the various .

2 CASL-MDL Models

A CASL-MDL model represents the modelled item in terms of values and of dynamic systems, and we use the term "*entity*" to denote something that may be a value or a dynamic system; similarly an *entity type* defines a type of entities. In Fig. 1 we present the structure of a CASL-MDL model, by means of its metamodel expressed using the UML[1].

A CASL-MDL model consists of entity type declarations (EntityType), at least one, and any number, also none, of *relationships* between entity types such as extension and subtyping, of *properties* about some of those entities and of *definitions* describing completely some of those entities. In this paper for lack of room we will consider only the highlighted parts.

A CASL-MDL model corresponds to a CASL-LTL specification with at least a sort for each declared entity type, whereas the properties will result in a set of axions and the definitions in subspecifications built by the CASL-LTL "free" construct.

Translation

[1] In the UML the black diamond denotes composition and the big arrow specialization.

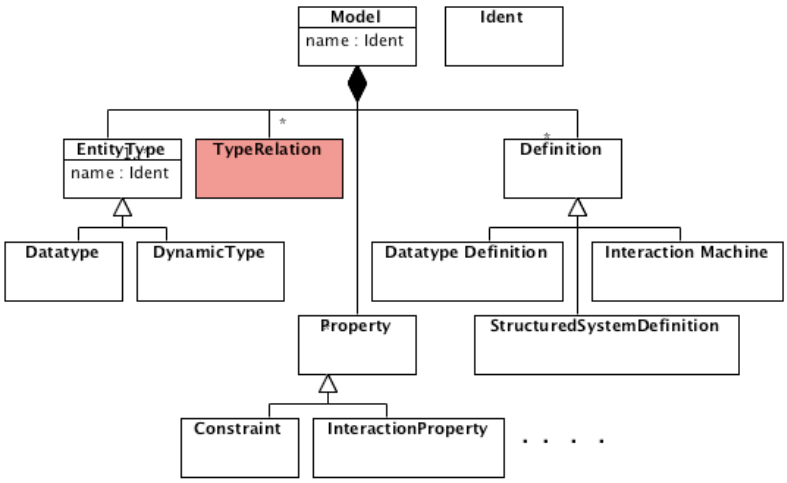


Fig. 1. Structure of the CASL-MDL models (metamodel)

$\mathsf{TModel} : \mathsf{Model} \to$ CASL-LTL-Specification

$\mathsf{TModel}(mod) =$

spec *mod*.name = (T(*mod*.entityType)**then axioms** TProps(*mod*.property))[2]

then free{ T(*mod*.definition) }

The translation of the entity types (at least one must be present in a CASL-MDL model) will result in a CASL-LTL specification declaring all the sorts corresponding to the types, plus some auxiliary sorts, and obviously all the declared operations and predicates.

A property in CASL-MDL corresponds to some CASL-LTL formulas on some of the entities introduced in the model, which will be used to extend the specification resulting from the type declarations. A CASL-MDL model having only properties will in the end correspond to a loose CASL-LTL specification.

A property may be a constraint consisting of a CASL-LTL formula written textually, similarly to the UML constraints expressed using the OCL, but in CASL-MDL constraints are suitable to express also properties on the behaviour of the dynamic systems, whereas OCL roughly corresponds to first-order logic. In CASL-MDL it is also possible to visually present some properties having a specific form, for examples some formulas on the interactions among the parts of a structured system may be expressed visually by diagrams denoted as UML sequence diagrams, and other formulas may be represented by diagrams similar to the UML activity diagrams. In this paper we consider only the properties of kind constraint and interaction properties.

[2] In the UML the name of the target class with low case initial letter is used to navigate along an association, thus *mod*.entityType denotes the set of the elements of class EntityType associated with *mod*

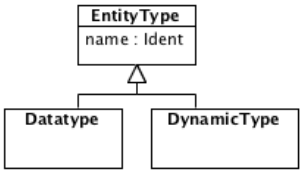


Fig. 2. Structure of the Entity types (metamodel)

A definition in CASL-MDL will define completely a datatype, by fixing its generators and defining its predicates and operations, or a dynamic system again by fixing the structure of its states and labels and defining its transitions.

The translation of the definitions part of a CASL-MDL model will result in a CASL-LTL specification with initial semantics, thus defined using the free CASL construct.

Visually a CASL-MDL model is a set of diagrams including at least a TypeDiagram presenting the entity types together with the associated constraints, and part of the definitions, whereas the other diagrams correspond to the remaining kind of definitions and to the properties having a visual counterpart. In this paper a CASL-MDL model will consists of a type diagram made by entity type declarations and constraints and of a set of sequence diagrams corresponding to theinteraction properties.

The TypeDiagram may become quite large and thus hard to read and to produce, so in CASL-MDL it is possible to split a TypeDiagram in several ones to describe parts of the types and of the constraints. Furthermore some features, as operations and predicates, of a type may be present in one diagram and others in another one. This possibility is like the one offered by the UML with several class diagrams in a model (a class may appear in several of them, and some of its features - operations and attributes - are in one diagram and some in another).

3 Entity Types and Type Diagrams

An entity type (declaration), shown in Fig. 2, defines a datatype or a dynamic type. In Sect. 3.1 we describe the datatypes, and in Sect. 3.2 the dynamic types.

The predefined datatypes of CASL-MDL are those introduced by the CASL libraries and includes the standard parameterized and not datatypes, e.g., NAT, INT, LIST and SET.

Translation

$\mathsf{TETypes} : \mathsf{EntityType}^* \rightarrow$ CASL-LTL-Specification

$\mathsf{TETypes}(et_1 \ldots et_n) =$

LIBRARY **then**

$\mathsf{Basic}(et_1.\text{name})$ **and** $\ldots$ **and** $\mathsf{Basic}(et_n.\text{name})$ **then**

$\mathsf{Detail}(et_1); \ldots \mathsf{Detail}(et_n);$

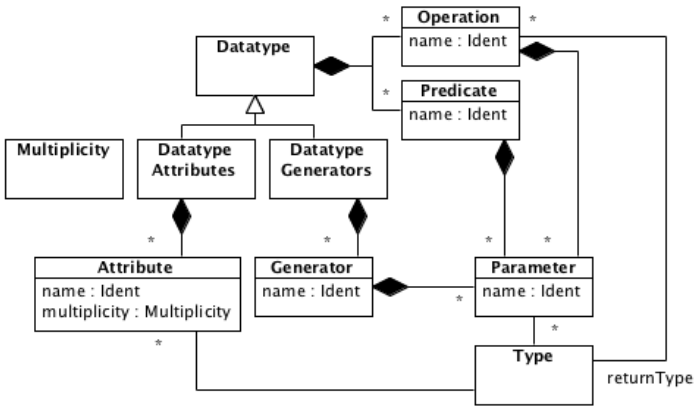


Fig. 3. Datatype Structure (metamodel)

where LIBRARY is a CASL specification corresponding to all the predefined datatypes (parameterized or not) defined by the CASL libraries [7].

The translation of a set of entity types consists of a CASL-LTL specification corresponding to the predefined types, enriched with the basic specifications of all the types of the model (defined by the function Basic) and after with the details of each type defined by the Detail function. The Basic function introduces the sort corresponding to the identifier passed as argument. Splitting the translation of a CASL-MDL type allows to have that a type in the type diagram may use all the other types present in the same diagram to define its features, as it is made by the UML for the classes.

3.1 Datatypes

CASL-MDL allows to declare new datatypes using the construct Datatype, and their metamodel is presented in Fig. 3[3].

The datatypes may have predicates and operations, which must have at least an argument typed as the datatype itself, and the operations have a return type.

The structure of a datatype of CASL-MDL may be defined in two different ways, using either *generators* or *attributes*.

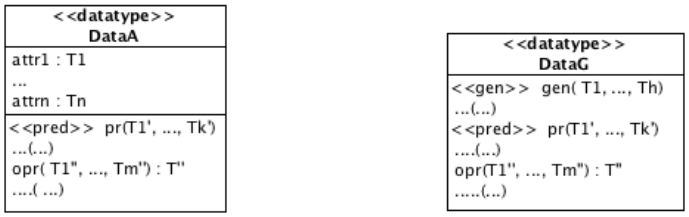
In the first case the datatype values are denoted using generators (as in CASL).[4] The arguments of the generators may be typed using the predefined types (corresponding to those of the CASL library) and the user defined datatypes and dynamic types present in the same TypeDiagram.

[3] Note that for the UML diagrams we follow the convention that a multiplicity equal to 1 is omitted, thus an attribute has exactly one type.

[4] We prefer to use the term generator instead of constructor used in the OO world to make clear that in our notation we have datatypes with values and not classes with objects.

The other possibility is to define the datatype values in terms of attributes, similarly to UML. An attribute attr: T of a datatype D will correspond to a CASL operation _ _ .attr: D → T. In this case there will a standard generator named as the type itself having as many arguments as the attributes, but it will be introduced when defining the datype by an appropriate definition.

Fig. 4 presents the visual notation for the two forms of datatypes by means of two schematic examples, one with attributes and one with generators (<< pred >> marks the predicates and << gen >> the generators).



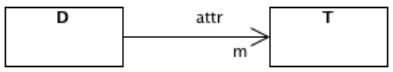
(a) Schematic datatype with attributes (b) Schematic datatype with generators

Fig. 4. Visual notation for datatypes

The attributes may have a multiplicity, (similar to the UML, thus it may be 1, *, 1..*, and $n..m$, with n and m two natural numbers) and its meaning is that the type of the attribute is a set of the associated type and that its values will satisfy an implicit constraint (see Sect. 4) about the size of their set values (e.g., multiplicity 0..1 means that the attribute may be typed by the empty set or by a singleton, * that may be typed by any set also empty, and 1..* by any nonempty set). Multiplicity 1 is omitted and corresponds to type the attribute with the relative type. This construct of the CASL-MDL motivates the implicit definition of the finite sets for each type in the translation of the entity types given in the following.

Obviously anonymous casting operations converting values into singleton sets and vice versa are available.

An attribute attr [m]: T of a dataype D may be also visually presented by means of an oriented association in the following way



.

The modellers are free to use plain attributes or their visual counterpart, but notice that using the arrows will make visible the structuring relationships among the various types.

Notice that it is possible that only the name of the datatype is provided (no generator or attribute, no predicate or operation), and visually it will be represented by a simple box with <<datatype>> and the name of the datatype.

In Sect. 6 we describe how to give a constructive semantics to a datatype by "defining" its operations with a set of conditional rules, and in Sect. 4 how the property oriented semantics of a datatype is given in term of constraints.

Translation

Basic : Datatype → CASL-LTL-Specification

Basic(*dat*) = FINITESET[**sort** *dat*.name]

The basic part of the translation of a datatype is the CASL specification of the finite sets of elements of sort *dat*.name (sort *dat*.name is declared in the specification). The need for an implicit declaration of a finite set type for each datatype (as well as for the dynamic types) is motivated by the possibility to associate a multiplicity to the attributes, which corresponds to implicitly declare their type as a set.

Detail : DatatypeAttributes → CASL-LTL-Specification

Detail(*datA*) = TAttributes(*datA*.attribute, *datA*.name) ;

TPredicates(*datA*.predicate) ; TOperations(*datA*.operation) ;

Below we give part Detail of the translation of the schematic example of datatype with attributes of Fig. 4(a).

op $\_\_.attr1 : DataA \to T1$; %% an operation corresponding to an attribute

...

pred $pr : T1' \times \ldots \times Tk'$; %% a predicate ...

op $opr : T1'' \times \ldots \times Tm'' \to T''$; %% an operation ...

Notice that at this point the standard generator for the sort *DataA* has not been introduced, the type has only some selector like operations corresponding to the attributes (this will allow to refine the datatype with more attributes).

Detail : DatatypeGenerators → CASL-LTL-Specification

Detail(*datG*) = TGenerators(*datG*.generator, *datG*.name) ;

TPredicates(*datG*.predicate) ; TOperations(*datG*.operation) ;

Below we give part Detail of the translation of the schematic example of datatype with generators of Fig. 4(b).

type $DataG ::= gen(T1; \ldots Th) \mid \ldots;$

pred $pr : T1' \times \ldots \times Tk'$; %% a predicate ...

op $opr : T1'' \times \ldots \times Tm'' \to T''$; %% an operation ...

ASSOC Model: Datatypes

Fig. 5 presents a Type Diagram of the CASL-MDL model of ASSOC containing only datatypes. It includes some enumerated types, precisely MeetingStatus and Vote (they are a special case of datatype having only generators without arguments considered as literal see Sect. 6).

Time is a datatype where no detail is given (it will just correspond to introduce the type name). Similarly, no generator is available for BallotRule however a predicate, check, given the votes and the number of voters says if the voting result was positive or not (Int and List are the predefined CASL datatypes for integers and lists). There are some generators for the Item datatype, together

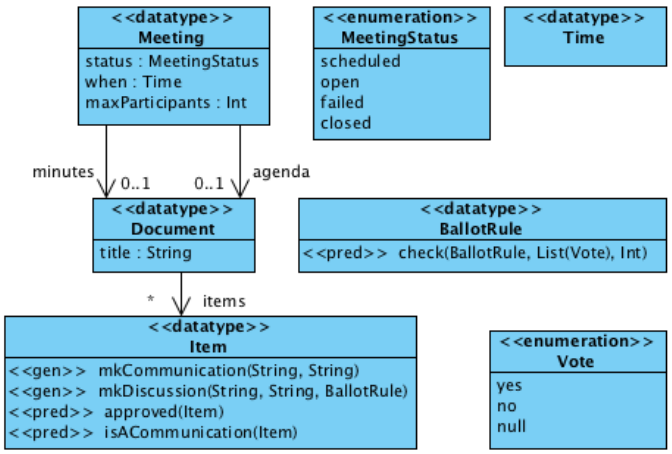


Fig. 5. ASSOC: Type Diagram containing some dataypes

with some predicates. Then there are two examples of datatypes with attributes. A Document has a title and some items (possibly zero), and this is expressed by the textual attribute title typed by the predefined String and by items represented by an arrow. A Meeting always has a status, a date and the maximum number of participants (textual attributes in the picture), and optionally it may have an agenda and/or minutes (visual attributes with multiplicity 0..1).

Here there is the CASL-LTL specification fragment corresponding to part Detail of those types translation.

free type $Vote ::= yes \mid no \mid null$; %% enumerated type
free type $MeetingStatus ::= scheduled \mid open \mid failed \mid closed$;
%% at this stage no generator available for the sort *BallotRule*
pred $check : BallotRule \times List[Vote] \times Int$;
type $Item ::= mkCommunication(String; String)$
$\mid mkDiscussion(String; String; BallotRule)$;
%% An item is a communication or a discussion with a ballot rule
pred $isACommunication : Item$;
pred $approved : Item$;
op $\_\_.status : Meeting \to MeetingStatus$; %% corresponds to an attribute
op $\_\_.when : Meeting \to Time$;
op $\_\_.agenda : Meeting \to Set(Document)$;
op $\_\_.minutes : Meeting \to Set(Document)$;
axiom $\forall\ m : Meeting \bullet size(m.agenda) \leq 1\ \wedge\ size(m.minutes) \leq 1$

Notice that in this part of the translation there is nothing concerning the datatype Time, since the corresponding sort has been already introduced in the basic part of the translation of the types (FINITESET[**sort** *Time*]).

3.2 Dynamic Types

In CASL-LTL and thus in CASL-MDL the dynamic systems represent any kind of dynamic entities, i.e., entities with a dynamic behaviour without making further distinctions (such as reactive, proactive, autonomous, passive behaviour, inner decomposition in subsystems), and are formally considered as labelled transition systems, that we briefly summarize below.

A *labelled transition system* (*lts* for short) is a triple $(State, Label, \rightarrow)$, where *State* denotes the set of states and *Label* the set of transition labels, and $\rightarrow \subseteq State \times Label \times State$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$.

Given an *lts* we can associate with each $s_0 \in State$ a tree (*transition tree*) with root s_0, such that, when it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from n to n'. A dynamic system is thus modelled by a transition tree determined by an *lts* $(State, Label, \rightarrow)$ and an initial state $s_0 \in State$.

CASL-LTL has a special construct **dsort** *state* **label** *label* to declare the two sorts *state* and *label*, and the associated predicate

$\_\text{--}\_\text{-->}\_ : state \times label \times state$

for the transition relation.

Thus a value of a dynamic sort corresponds to a dynamic system, precisely to the labelled transition tree having such value as root, and thus a CASL-LTL specification with a dynamic sort may be truly considered as a dynamic type.

The labels of the transitions of a dynamic system are named in this paper *interactions* and are descriptions of the information flowing in or out the system during the transitions, thus they truly correspond to interactions of the system with the external world[5].

[5] Obviously, a transition may also correspond to some internal activity not requiring any exchange with the external world, in that case the transition is labelled by a special *TAU* value.

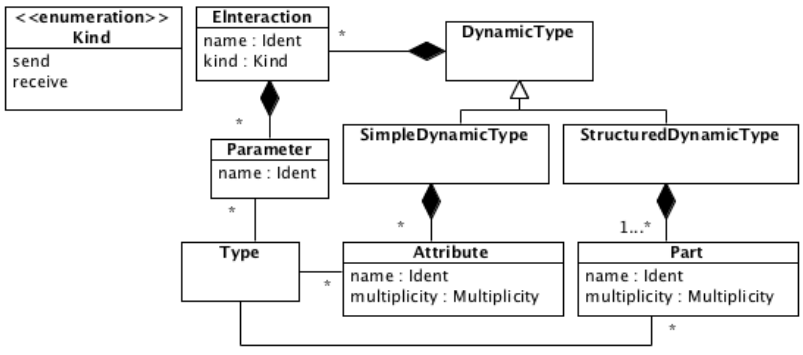


Fig. 6. Dynamic Type Structure (metamodel)

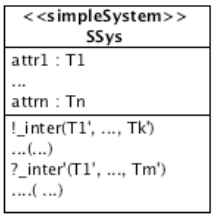


Fig. 7. A schematic Simple Dynamic Type

In Fig. 6 we present the structure of the CASL-MDL declaration of dynamic types (i.e., types of dynamic systems) by means of its metamodel, and later we will detail the two cases of simple and structured dynamic types.

Simple Dynamic Types The simple dynamic systems do not have dynamic subsystems, and in the context of this work, the interactions of the simple systems are either of kind sending or receiving (with a naming convention !_*xx* and ?_*yy*, for sending and receiving interactions resp.) and are characterized by a name and a possibly empty list of typed parameters. These simple interactions correspond to basic acts of either sending out or of receiving something, where the something is defined by the arguments. Obviously, a send act will be matched by a receive act of another simple system and vice versa, and again quite obviously the matching pairs of interactions $!\_xx(v_1, \ldots, v_n)$ and $?\_xx(v_1, \ldots, v_n)$.

The states of simple systems are characterized by a set of typed attributes (precisely the states of the associated labelled transition system), similarly to the case of datatypes with attributes (and, as for each attribute, there is the corresponding operation). A dynamic type DT has also an extra implicit attribute __.id: ident_DT containing the identity of the specific considered instance; the identity values are not further detailed. Obviously the identity is preserved by the transitions and no structured dynamic system will have two subsystems with the same identity. Notice how the treatment of the identity in CASL-MDL is completely different from the one of the UML, where the elements of the type associated with a class are just their identities, because CASL-MDL is not object-oriented.

Fig. 6 shows that a simple dynamic type (i.e., a type of simple systems) is determined by a set of elementary interactions (EInteraction) and by a set of attributes; notice that it has also a name since SimpleDynamicType specializes EntityType, see Fig. 2.

In Fig. 7 we present the visual notation for the simple dynamic types by the help of a schematic example.

Translation

Basic : SimpleDynamicType → CASL-LTL-Specification

Basic($simpDT$) =

FiniteSet[**sort** *simpDT*.name] **and** Ident **with** *ident* ↦ ident_*simpDT*.name

The basic translation of a simple dynamic type includes also the declaration of a datatype for the identity of the dynamic systems having such type.

Detail : SimpleDynamicType → Casl-Ltl-Specification
Detail(*simpDT*) =
dsort *simpDT*.name **label** label_*simpDT*.name
op __.*id* : *simpDT*.name → ident_*simpDT*.name
TAttributes(*simpDT*.attribute, *simpDT*.name);
TEInteractions(*simpDT*.eInteraction, label_*simpDT*.name);

ASSOC Model: Simple Dynamic Types

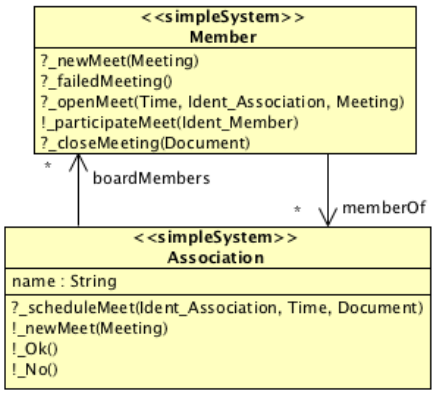


Fig. 8. ASSOC Example: a type diagram including simple dynamic types

Fig. 8 presents a type diagram including two declarations of simple dynamic types. Notice that the type Member has other elementary interactions, e.g., !_vote(Item,Vote,Ident_Member) concerning taking part in a meeting not reported here, they are visible in the complete type diagram, see Appendix A

The simple dynamic type Association models the various associations, characterized by a name and by their members (given by the attributes name and members, the latter represented visually as an arrow). We have used a dynamic system and not a datatype since we are interested in the dynamic behaviour of an association. The elementary interaction ?_scheduleMeeting corresponds to receive a request to schedule a new meeting of the association board, and the last two parameters correspond to the meeting date and agenda, whereas the first, typed by Ident_Association is the identity of the association itself. !_Ok and !_Ko correspond respectively to answer positively and negatively to that request.

Part Detail of the translation of the simple dynamic type Association is as follows.

dsort *Association* **label** label_*Association*

op $\_\_.id : Association \to$ ident_*Association*

op $\_\_.name : Association \to String$

op ?_*scheduleMeeting* : ident_*Association*$\times Time \times Document \to$ label_*Association*

op !_*Ok*, !_*Ko*, *TAU* :$\to$ label_*Association*

TAU is a special implicit element used to label the transitions of the associated transition systems that do not require any exchange of information with the external world, thus without any interaction. Notice that the sorts *Association* and ident_*Association* have been already introduced by the basic part of the type translation.

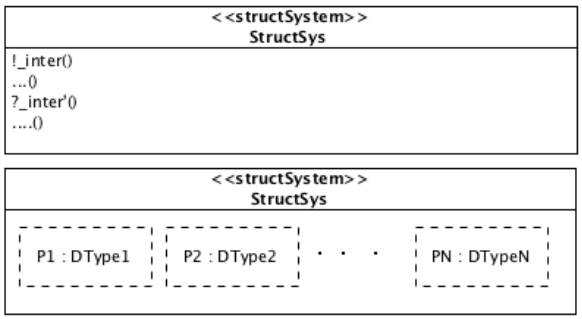


Fig. 9. A schematic Structured Dynamic Type

Structured Dynamic Types We recall that a structured system (cf. Fig. 6) is characterized by its parts, or subsystems (that are in turn other simple or structured dynamic systems), and has its own elementary interactions and name.

In Fig. 9 we present the visual syntax by the above schematic structured dynamic type; its parts are depicted by the dashed boxes (in this case all of them have multiplicity one); DType1, DType2, ..., DTypeN are dynamic types (i.e., types corresponding to dynamic systems, simple or structured, defined in the same model) and P1, P2, ..., PN are the optional names of the parts. At this level we only say that there will be at least those parts, but nothing is said about the way they interact with each other and on the behaviour of the whole system. We use two different boxes for the elementary interactions and the structure in terms of parts to keep the internal structuring encapsulated.

A structured dynamic type has a predefined predicate *isPart* checking if it has a part having a given identity.

Translation

Basic : StructuredDynamicType $\to$ CASL-LTL-Specification

Basic(*structDT*) =

FINITESET[**sort** *structDT*.name] **and**

IDENT **with** *ident* $\mapsto$ ident_*structDT*.name **and** LOCALINTERACTIONS

LocalInteractions specifies the local interactions sets of the structured dynamic systems defined by *structDT*, where a local interaction is a pair consisting of the identity and of an elementary interaction of one of the parts of *structDT*; the local interactions are added to the labels of the associated labelled transition system to record the activities of the parts.

Detail : StructuredDynamicType → Casl-Ltl-Specification
Detail(*structDT*) =
dsort *structDT*.name **label** label_*structDT*.name
op __.*id* : *structDT*.name → ident_*structDT*.name
pred *isPart* : *structDT*.name × *ident_all*
TParts(*structDT*.part, *structDT*.name);
TEInteractionsStruct(*structDT*.eInteraction, label_*structDT*.name,
localInteractions_*structDT*.name);

ident_all is an extra auxiliary sort having as subsorts the identity sorts of all the dynamic systems in the model.

ASSOC Model: Structured Dynamic System

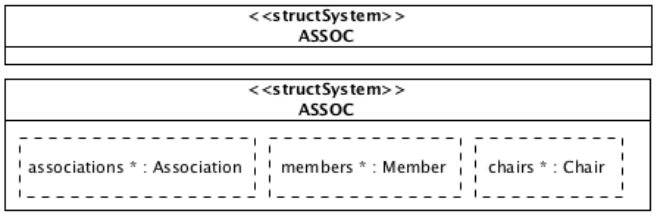


Fig. 10. ASSOC Example: a type diagram including a structured dynamic type

The whole world of ASSOC is modelled as a structured dynamic system ASSOC having as parts the associations, the members and the chairs, any number of them (see the multiplicity * on the three parts). ASSOC is a closed system, that is it does not interact with its external world and so it has no elementary interactions, and all the transitions of the associated labelled system will be labelled by the special null interaction *TAU*.

The Casl-Ltl specification fragment corresponding to the detail part of the translations of the structured dynamic type ASSOC is given below.

dsort *ASSOC* **label** label_*ASSOC*
op __.*id* : *ASSOC* → ident_*ASSOC*
op *associations* : *ASSOC* → *Set*[*Association*]
op *members* : *ASSOC* → *Set*[*Member*]
op *chairs* : *ASSOC* → *Set*[*Chair*]
pred *isPart* : *ASSOC* × *ident_all*
op *TAU* : localInteractions_*ASSOC* → label_*ASSOC*
where LocalInteractions= FiniteSet[LocalInteraction] and

LocalInteraction =
free type = *LocalInteraction* ::=
< _ _ _ _ > (ident_*Association*; label_*Association*) |
< _ _ _ _ > (ident_*Member*; label_*Member*) |
< _ _ _ _ > (ident_*Chair*; label_*Chair*)

Notice that the sorts *structDT*.name and ident_*ASSOC* have been been introduced by the basic part of the type translation.

4 Casl-Mdl Constraints

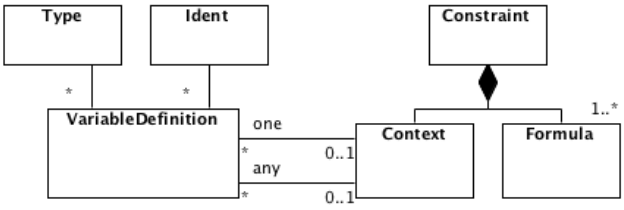


Fig. 11. Casl-Mdl Constraint Structure (metamodel)

A Casl-Mdl constraint expresses a property about generic entities (i.e., datatypes and dynamic systems) introduced by the model by means of a Casl-Ltl formula presented with a slightly simplified syntax; the entities concerned by a constraint are defined by its context, and are distinguished in those universally quantified (any) and in those existentially quantified (one). Such formulas are defined below.

Formula ::=
Data_Atom |
not Formula | Formula => Formula | Formula and Formula | Formula or Formula |
forall Ident • Formula | exists Ident • Formula |
in_any_case Dyn_Exp • Path_Form | in_one_case Dyn_Exp • Path_Form

Data_Atom are the atomic formulas on the values of the datatypes (equations and predicate applications).

The Casl-Ltl logic offers also some branching time CTL* style temporal logic combinators. Dyn_Exp are expression typed by a dynamic type(i.e., a type corresponding to a specific kind of dynamic system). The formula "in_any_case *dexp* • *path_form*" requires that any path starting from the root of the labelled transition system associated with *dexp* satisfies *path_form*, whereas "in_one_case *dexp* • *path_form*" requires that at least one path from the root in the labelled transition system associated with *dexp* satisfies *path_form*.

Path_Form ::=
Interact_Atom | Static_Atom | Local_Interact_Atom |
not Path_Form | Path_Form and Path_Form | Path_Form or Path_Form |
Path_Form => Path_Form |

forall Ident • Path_Form | exists Ident • Path_Form |
eventually Path_Form | always Path_Form | next Path_Form

Interact_Atom ::= InterName(Exp,..., Exp)

Static_Atom ::= AttrName = Exp| PredName(Exp,..., Exp)

InterName and AttrName are names of interactions and attributes respectively of a dynamic type, and Exp are expressions denoting values.

The atomic path formulas express a property on the first state of the path or on the label of the first transition of the path. If $attr = e$ is a static atom, then $attr : T$ is an attribute of the dynamic type. When a static atom of this form holds, it means that in the first state the value of $attr$ is equal to e. If $prid(e_1, \ldots, e_n)$ is a static atom, then $prid(dsort, T_1, \ldots, x_n : T_n)$ is a predicate of the dynamic type. When a static atom of this form holds, it means that in the first state s $prid(s, e_1, \ldots, e_n)$ holds.

If $id(e_1, \ldots, e_n)$ is an interaction atom, then $id(T_1, \ldots, x_n : T_n)$ is an interaction of the dynamic type. When an interaction atom of this form holds, it means that the elementary interaction $id(e_1, \ldots, e_n)$, is the label of the first transition of the path.

Local_Interact_Atom ::= Ident :: InterName(Exp,..., Exp)

Local interactions atoms are special formulas for the structured systems, which correspond to state that a part performs a given elementary interaction. If $pid :: eid(e_1, \ldots, e_n)$ is a local interaction interaction atom, then pid is the identity of a part of the structured system and $eid(T_1, \ldots, T_n)$ is an interaction of that part. When a local interaction atom of this form holds, it means that the part whose identity is pid performs the elementary interaction $id(e_1, \ldots, e_n)$ in the first transition of the path.

next *path_form* holds on a path π if *path_form* holds on the subpath starting from the second state of π. eventually *path_form* holds on a path π if *path_form* holds on the subpath starting from a state of π. always *path_form* holds on a path π if *path_form* holds on all the subpaths starting from any state of π.

Visually the constraints are written in notes attached to all the involved types by dashed lines, such types are those present in the context. The visual presentation of the constraints is part of the TypeDiagram.

Examples of constraints both for datatypes and simple dynamic types can be found in Fig. 12.

Translation

TConstr : Constraint → CASL-LTL-Formula

TConstr(*constr*) = TContext(*constr*.context) • TFormula(*constr*.formula)

The translation in CASL-LTL of the constraints presented in Fig. 12 is given below.

$\forall\ M : Member \bullet$

$in\_any\_case(M, always\ \ <!\_participateMeet(M.id)> \Rightarrow\ \ eventually$

$(<?\_failedMeet()> \ \ \vee\ \exists\ MIN : Document \bullet\ <?\_closeMeet(MIN)>))$

$\forall\ A : Association, M1, M2, Meeting \bullet$

$M1 \in A.boardMeetings\ \wedge\ M2 \in A.boardMeetings\ \wedge\ \neg\ M1 = M2\ \Rightarrow$

$\neg\ M1.time = M2.time$

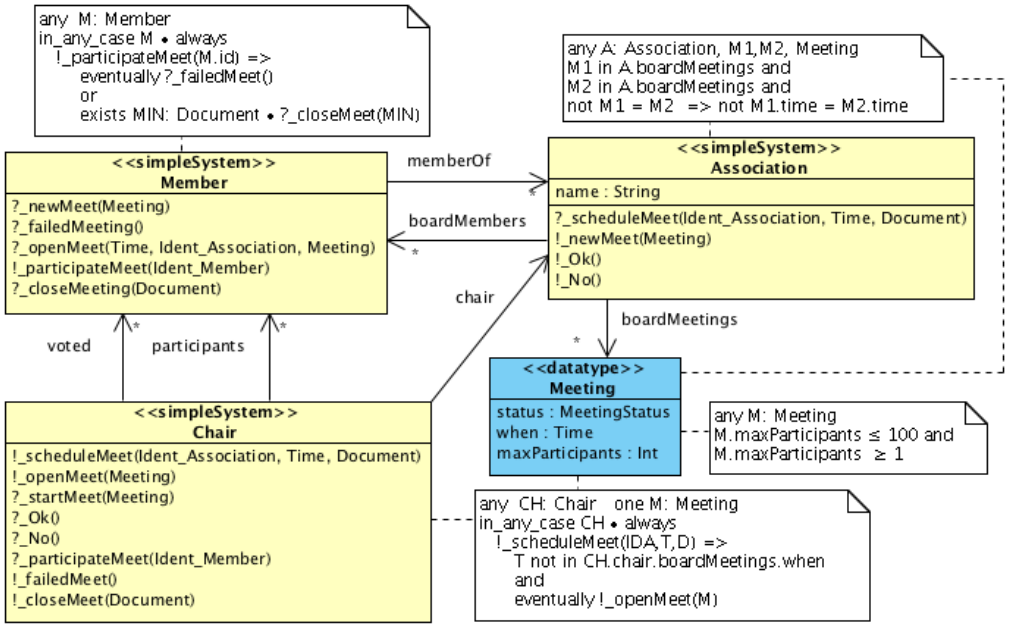


Fig. 12. ASSOC Example: constraints

$\forall\ CH : Chair\ \ exists\ M : Meeting \bullet$
$\quad in\_any\_case(CH, always\ \ <!\_scheduleMeet(IDA, T, D) > \Rightarrow$
$\quad (T \notin CH.chair.boardMeetings.when\ \wedge\ eventually\ \ <!\_openMeet(M) >))$
$\forall\ M : Meeting \bullet M.maxParticipants \leq 100\ \ \wedge\ \ M.maxParticipants \geq 1$

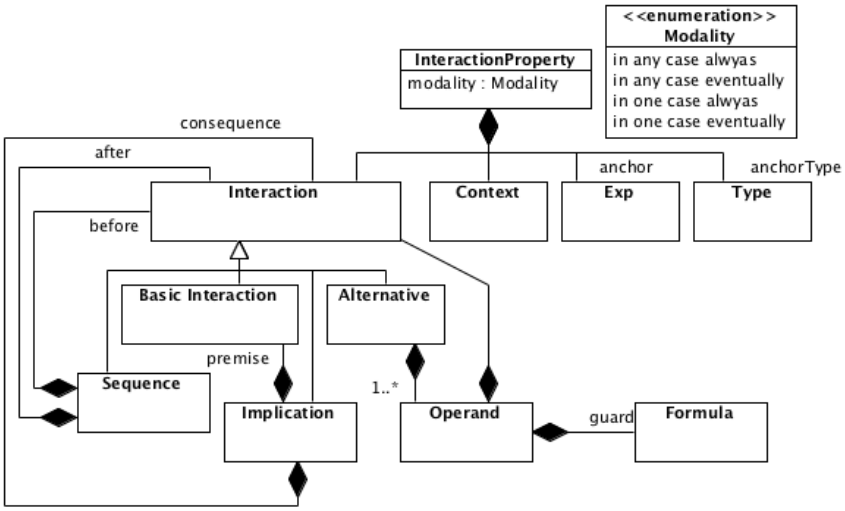
5 Interaction properties

The metamodel of CASL-MDL interaction properties is given in Fig. 13.

An interaction property describes the way parts of a structured dynamic system (that are in turn dynamic systems) interact. Thus, first of all it should be *anchored* to a specific structured dynamic system represented by an expression typed by a structured dynamic type, which may have free variables, corresponding to express a property on more than one dynamic system. [6] Furthermore an interaction property includes a *context* defining the other free variables (universally and existentially quantified) that may appear in it.

In CASL-MDL, contrary to UML sequence diagrams, an interaction property explicitly states if it expresses a property of all possible lives of the anchor, or if there exists at least one life of the anchor satisfying that property. It also states whether the property about the interactions must hold in all possible instants of

[6] We use the word "expression" commonly used in the world of the modelling notations, instead of "term", preferred in the world of the algebraic specifications



The anchor should be an expression whose type is the anchor type, and it a dynamic type corresponding to a structured system

Fig. 13. Interaction Properties structure (metamodel)

that lives, or if eventually there will be an instant in which it will hold. Thus an interaction property has a *modality*, that may assume four values, see Fig. 13.

The Interaction part expresses the required pattern on the interactions among the parts of the anchor and it may be a basic interaction, or a structured interaction built by some combinators (in this paper we consider only alternative, sequential composition and implication).

As shown in Fig. 14, an interaction property is visually presented by a *sequence diagram* similar (in form) to the UML sequence diagrams (any v1:T1,. . . ,vn:Tn, one v' 1:T' 1,. . . ,v' m:T' m is the context).

The BasicInteraction, defined in Fig. 15, is the simplest form of Interaction and just corresponds to assert that a series of *elementary interaction occurrences* happen orderly among some generic roles for dynamic systems parts of the anchor (*lifelines*), where an interaction occurrence is the simultaneously performing of a pair of matching input and output elementary interactions by two lifelines.

A lifeline is characterized by a name (just an identifier) and a (dynamic) type and defines a role for a participant to the interaction. An elementary interaction

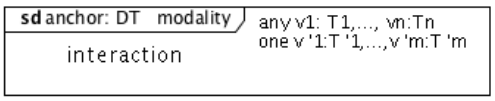


Fig. 14. A generic CASL-MDL sequence diagram

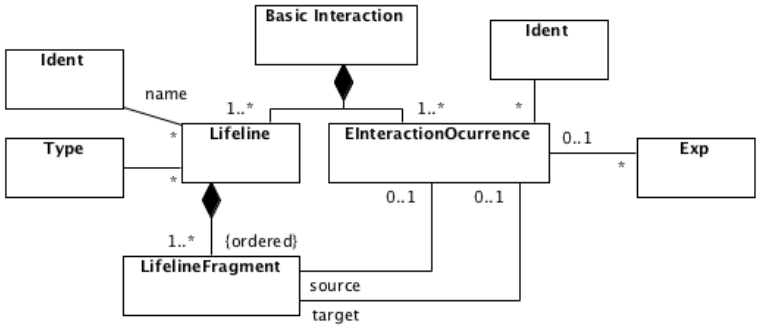


Fig. 15. Structure of Basic Interactions (metamodel)

occurrence connects two lifelines in specific points (represented by the lifeline fragmentsof kind InteractionPoint); the ordering of the interaction points of the various lifelines must determine a partial order on the interaction occurrences. An interaction occurrence is characterized by the name of an elementary interaction s.t. the source type owns it with kind "send" and the target type owns the matching one with the kind "receive", and a set of arguments represented by expressions whose types are in accord with the parameters of the two elementary interactions.

Visually a lifeline is depicted as a box containing its name and type, and by a dashed line summarizing all its fragments, whereas an interaction occurrence is depicted as a horizontal arrow with filled head from the source lifeline to the target one. [7] An elementary interaction occurrence arrow is labelled by inter(exp1...,expn) where !_inter is the send interaction of T1, ?_inter the receive interaction of T2, and exp1...,expn are expressions whose types are in order those of the arguments of !_inter, that are the same of those of ?_inter. Fig. 16 shows a generic case of two lifelines and of an elementary interaction occurrence.

As in the UML the relative distance between two elementary interaction occurrences has no meaning, similarly the only guaranteed ordering is among the the occurrences attached to a single lifeline (due to the ordering of its fragments), whereas in the other cases the visual ordering between two occurrences has no meaning. In Fig. 17 we show two different basic interactions that are, however,

[7] The icon for the elementary interaction occurrence is derived by the synchronous messages of the UML.

Fig. 16. Generic example of elementary interaction occurrence

perfectly equivalent determining both the partial order listed at the bottom; notice that there are many other ones visually different but still equivalent.

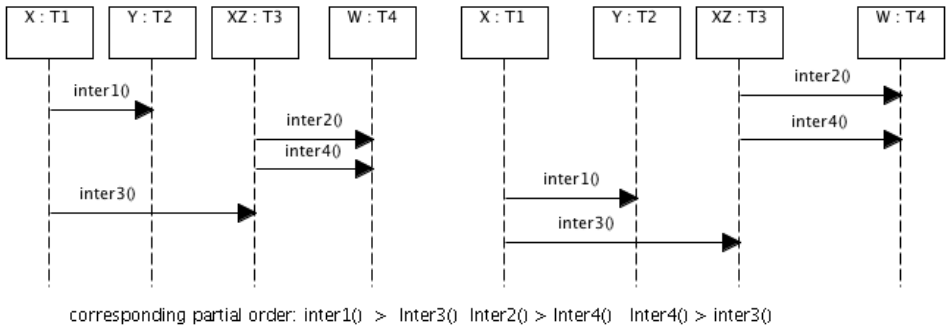


Fig. 17. Two perfectly equivalent basic interactions

A sequence diagram, ie., an interaction property, corresponds to a CASL-LTL formula.

Translation

TIntProp : InteractionProperty → CASL-LTL-Formula

$\mathsf{TIntProp}(iPr) =$

$\forall\ freeVars\, \mathsf{TContext}(iPr.\text{context}) \bullet (\wedge_{x \in iPr.\text{lifeline}}\ isPart(x.id, iPr.\text{anchor}))\ \Rightarrow$

$\mathsf{TModal}(iPr.\text{modality}, iPr.\text{anchor}, \mathsf{TInteract}(iPr.\text{interaction}, true))$

where *freeVars* are all the free variables appearing in the anchor expression and those corresponding to the lifelines.

TModal : Modality × Exp × CASL-LTL-PathFormula → CASL-LTL-Formula

$\mathsf{TModal}(\text{in any case always}, dexp, PF) = in\_any\_case(dexp, always\ PF)$

similarly for the other three cases

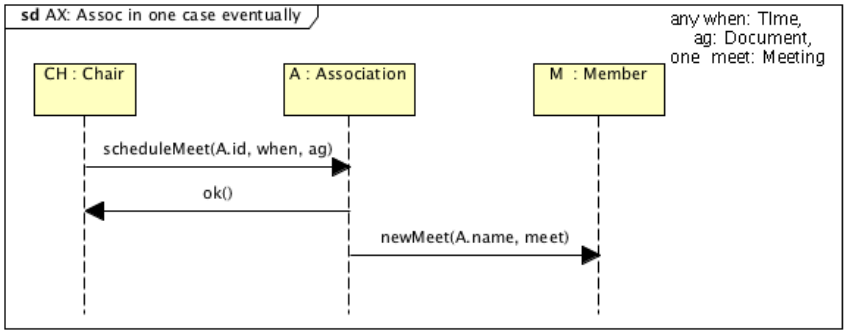


Fig. 18. ASSOC: scheduling a new meeting (successful case)

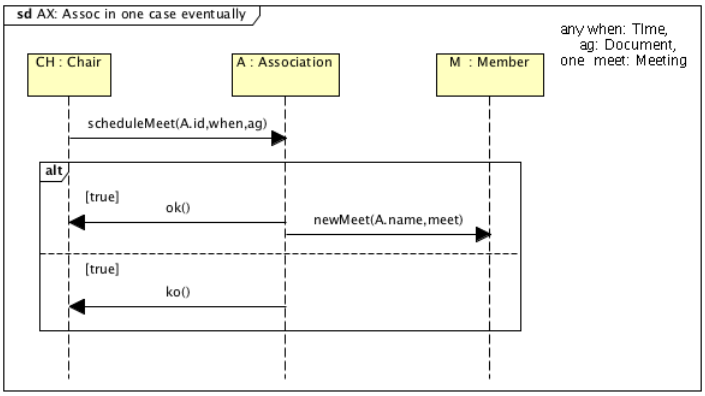


Fig. 19. ASSOC: scheduling a new meeting (sequence and alternative combinator)

$\mathsf{TInteract}: \mathsf{Interaction} \times$ CASL-LTL-PathFormula $\rightarrow$ CASL-LTL-PathFormula

The translation of an interaction is defined by cases, depending on its particular type, and takes as argument a path-formula that will play the role of a continuation; this technical trick allows to correctly translate sequential compositions of interactions.

$\mathsf{TInteract}(basicInt, cont) =$

$\bigvee_{eIOc_{i_1} \ldots eIOc_{i_n} \text{ admissible ordering of } eIOc_1,\ldots,eIOc_n}$

$\mathsf{TIntOcc}(eIOc_{i_1}) \wedge eventually\ (\mathsf{TIntOcc}(eIOc_{i_2}) \wedge (eventually\ \ldots$

$(\mathsf{TIntOcc}(eIOc_{i_n}) \wedge eventually\ cont)\ldots))$

where $basicInt.\text{eInteractionOccurence} = eIOc_1, \ldots, eIOc_n$

$\mathsf{TIntOcc}: \mathsf{InteractionOccurrence} \rightarrow$ CASL-LTL-PathFormula

$\mathsf{TIntOcc}(eIOc) =$

$(x.id{:}!\_inter(exp1, \ldots, expn)\ \wedge\ y.id{:}?\_inter(exp1, \ldots, expn))$[8]

where $eIOc$ has the form in Fig. 16.

Fig. 18 shows a sequence diagram with a basic interaction modelling a successful scheduling a new meeting. This diagram presents a sample of a possible way to execute the successful scheduling of a meeting, precisely the chair asks the association to schedule a new meeting passing the date and the agenda, the association answers ok, and then informs the board members of the new meeting.

In the following we show the CASL-LTL formula corresponding to the sequence diagram of Fig. 18:

$\forall\ AX{:}\ Assoc, when{:}\ Time, ag{:}\ Document, CH{:}\ Chair, A{:}\ Association, M{:}\ Member$

$\exists\ meet{:}\ Meeting \bullet$

$(isPart(CH.id, AX)\ \wedge\ isPart(A.id, AX)\ \wedge\ isPart(M.id, AX))\ \Rightarrow$

$in\_one\_case(AX, eventually$

[8] Recall that $\_\_.id$ is the standard attribute returning the identity of a dynamic system, and that $id{:}\ interact$ is a local interaction atomdefined in Sect. 4.

$(CH.id{:}!\_scheduleMeet(A.id, when, ag) \ \wedge \ A.id{:}?\_scheduleMeet(A.id, when, ag)) \wedge$
$eventually$
$(A.id{:}!\_ok() \ \wedge \ CH.id{:}?\_ok()) \wedge (eventually$
$A.id{:}!\_newMeet(A.name, meet) \ \wedge \ M.id{:}?\_newMeet(A.name, meet))))$

Fig. 13 presents also the structured interactions. We can see that it is possible to express:

– the sequential composition of two interactions, with the intuitive meaning to require that the interaction pattern described by the before argument is followed by the interaction pattern described by the after argument;

– the choice among several guarded alternatives, subsuming conditional and nondeterministic choices; one of the interaction patterns corresponding to the alternatives with the true guard must be performed, if no guards is true it corresponds to require nothing on the interactions;

– the fact that the happening of some elementary interactions matching a given pattern (represented by a basic interaction) must be followed mandatory by some elementary interactions matching another pattern.

The visual representation of these structured interactions is illustrated in Fig. 19 and Fig. 20.

To model that the answer of the association may be also negative (elementary interaction ko) we need the structured interactions built with the sequential and alternative combinators, and this corresponds to give just some samples of successful and of failed executions, whereas to represent that after a request of scheduling a new meeting there will be surely an answer by the association we need the implication combinator. Fig. 19 and Fig. 20 presents the various sequence diagrams, with a structured interaction part, corresponding to those cases. In Fig. 19 we have the sequential combination of a basic interaction consisting just of the elementary interaction occurrence scheduleMeet(A.id,when,ag) followed by the alternative among two basic interactions, where the guards are both true corresponding to the pure nondeterministic choice. Again this diagram presents sample of the execution of the scheduling procedure, making explicit that there are two possibilities, a successful one and a failing one; but this diagram does not require that any request to an association will be followed by an answer. Fig. 20 instead shows that an occurrence of the elementary interaction scheduleMeet(A.id,when,ag) will be eventually either followed by an occurrence of ko() or of ok(). Notice that the modality of this sequence diagram is different, it says that whenever the scheduling request occurs it will be followed by an answer.

Translation

TInteract: Interaction × CASL-LTL-PathFormula → CASL-LTL-Formula

$\mathsf{TInteract}(altInt, cont) =$
$\wedge_{J \subseteq \{1,\ldots,n\}} ((\wedge_{j \in J} \ op_j.\text{guard} \ \wedge \wedge_{i \in \{1,\ldots,n\}-J} \neg \ op_i.\text{guard}) \ \Rightarrow$
$\vee_{j \in J} \ \mathsf{TInteract}(op_j.\text{interaction}, cont))$
where $altInt.\text{operand} = op_1, \ldots, op_n$
$\mathsf{TInteract}(seqInt, cont) = \mathsf{TInteract}(seqInt.\text{before}, \mathsf{TInteract}(seqInt.\text{after}, cont))$
$\mathsf{TInteract}(implInt, cont) =$

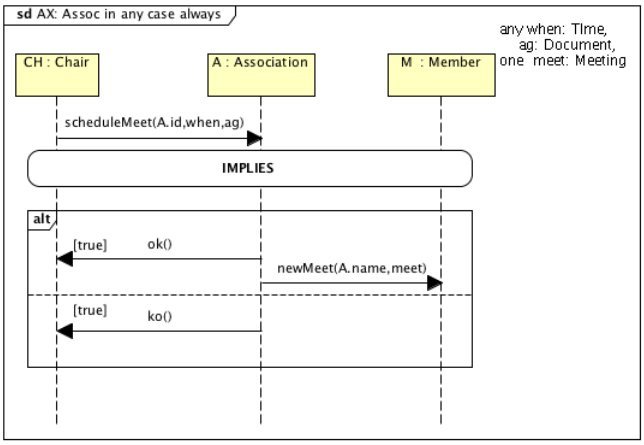


Fig. 20. ASSOC: scheduling a new meeting (implies combinator)

$\bigwedge_{eIOc_{i_1} \ldots eIOc_{i_n} \text{ admissible ordering of } eIOc_1,\ldots,eIOc_n}$
$(\mathsf{TIntOcc}(eIOc_{i_1}) \Rightarrow next\ always\ (\mathsf{TIntOcc}(eIOc_{i_2}) \Rightarrow next\ always\ (\ldots$
$(\mathsf{TIntOcc}(eIOc_{i_n}) \Rightarrow next\ eventually\ \mathsf{TInteract}(implInt.\text{consequence}, cont))\ldots)))$
where $implInt.$premise.eInteractionOccurence $= eIOc_1, \ldots, eIOc_n$

Here there is the CASL-LTL formula corresponding to the sequence diagram of Fig. 19 after some simplifications due to the fact that the guards are both equal to true (e.g., $\neg\ \mathsf{true} \Rightarrow F$ equivalent to true , $F \wedge \mathsf{true}$ equivalent to F and so on):
$\forall\ AX$: $Assoc$, $when$: $Time$, ag: $Document$, CH: $Chair$, A: $Association$, M: $Member$
$\exists\ meet$: $Meeting \bullet$
$(isPart(CH.id, AX) \wedge isPart(A.id, AX) \wedge isPart(M.id, AX)) \Rightarrow$
$in\_one\_case(AX, eventually$
$(CH.id{:}!\_scheduleMeet(A.id, when, ag) \wedge A.id{:}?\_scheduleMeet(A.id, when, ag)) \wedge$
$eventually$
$(\quad (A.id{:}!\_ok() \wedge CH.id{:}?\_ok()) \wedge (eventually$
$A.id{:}!\_newMeet(A.name, meet) \wedge M.id{:}?\_newMeet(A.name, meet))))$
$\vee$
$(\quad (A.id{:}!\_ko() \wedge CH.id{:}?\_ko())$

The CASL-LTL formula corresponding to the sequence diagram of Fig. 20 after some simplifications is instead:
$\forall\ AX$: $Assoc$, $when$: $Time$, ag: $Document$, CH: $Chair$, A: $Association$, M: $Member$
$\exists\ meet$: $Meeting \bullet$
$(isPart(CH.id, AX) \wedge isPart(A.id, AX) \wedge isPart(M.id, AX)) \Rightarrow$
$in\_any\_case(AX, always$
$(CH.id{:}!\_scheduleMeet(A.id, when, ag) \wedge A.id{:}?\_scheduleMeet(A.id, when, ag)) \Rightarrow$
$next\ eventually$

$$
\begin{array}{ll}
(& (A.id{:}!\_ok() \ \wedge \ CH.id{:}?\_ok()) \wedge (eventually \\
 & \quad A.id{:}!\_newMeet(A.name, meet) \ \wedge \ M.id{:}?\_newMeet(A.name, meet)))) \\
\vee & \\
(& (A.id{:}!\_ko() \ \wedge \ CH.id{:}?\_ko())
\end{array}
$$

6 Datatype Definitions

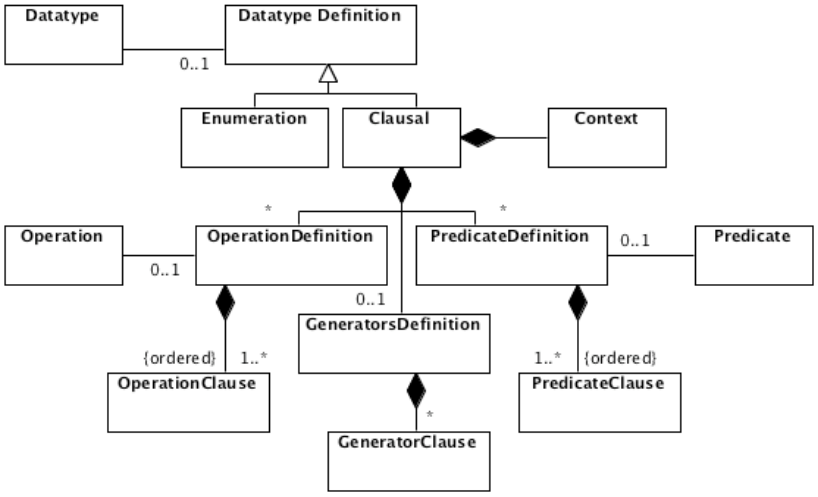


Fig. 21. Datatype Definion Structure

A datatype definition is given by a set of definitions of its operations and predicates (there should be a definition for each of them) and of the its generators (optional), see the corresponding metamodel in Fig. 21.

The context of a datatype definition must have all the variable in the all mode and introduces all the variable that may occur freely in the definition.

The generators are defined by a set of generator clauses that are conditional rules (i.e., conditional axioms) that express in which cases two generators (or the same generator with different arguments) represent the same data value.

$\mathsf{Generator\_Clause} ::= \mathsf{Data\_Atoms} \ \Rightarrow \ \mathsf{Exp} = \mathsf{Exp}$

$\mathsf{Data\_Atoms} ::= \Lambda \mid \mathsf{Data\_Atom} \ \wedge \ \mathsf{Data\_Atoms}$

where the two expressions in the right side of the clause should be built by some generators of the defined dataype.

A predicate is defined by an ordered list of clauses that are conditional rules (i.e., conditional axioms) that express in which cases the predicate holds; the consequence of each clause should be built using the predicate associated with the definition.

$\mathsf{Predicate\_Clause} ::= \mathsf{Data\_Atoms} \ \Rightarrow \ \mathsf{Pr(Exps)}$

$\mathsf{Exps} ::= \Lambda \mid \mathsf{Exp}, \mathsf{Exps}$

A predicate definition for pr consisting of $cl_1, \ldots, cl_m$ determines its truthness in the following way.

$pr(v_1, \ldots, v_n) =$
for $i = 1, ..., m$
{ if consequence of cl_i matches $pr(e_1, \ldots, e_n)$ then
if premises of cl_i where the free variables appearing in $pr(e_1, \ldots, e_n)$ have been replaced by the values determined by the matching procedure holds, then
returns *true*;
} ;
returns *false*;

Notice that above definition asserts that if no clause can be applied to $pr(e_1, \ldots, e_n)$, then the predicate pr does not hold on the values represented by $e_1, \ldots, e_n$, and that it is not needed to explicit write when it is false in the clauses.

An operation is defined by an ordered list of operation clauses that are conditional rules (i.e., conditional axioms) that express which are the values returned by the operation.

$\mathsf{Operation\_Clause} ::= \mathsf{Data\_Atoms} \Rightarrow \mathsf{Op}(\mathsf{Exps}) = \mathsf{Exp}$

An operation definition for op consisting of $cl_1, \ldots, cl_m$ determines its value in the following way.

$op(v_1, \ldots, v_n) =$
for $i = 1, ..., m$
{ if right hand-side of the consequence of cl_i matches $op(e_1, \ldots, e_n)$ then
if premises of cl_i where the free variables appearing in $op(e_1, \ldots, e_n)$ have been replaced by the values determined by the matching procedure holds, then
returns left hand-side of the consequence of cl_i where the free variables appearing in $op(e_1, \ldots, e_n)$ have been replaced by the values determined by the matching procedure;
} ;
error;

If no clause can be applied to $op(e_1, \ldots, e_n)$ or the premises of all the applicable clauses do not hold, then it is considered an error; the modeller should take a good care to ensure that a clause is always applicable.

A datatype definition is visually presented by means of a note attached to the datatype icon, containing the context, the generators definitions if present, and then the predicate and operation definitions. The premises of the last clause of an operation (predicate) definition may be simply "else", and it stands for the negation of the premises of all the preceding clauses.

There is a special notation for the definition of enumeration datatypes, similar to the UML one, and Fig. 22 presents a generic example of it. For readability this constructive view is usually represented in the TypeDiagram.

Translation

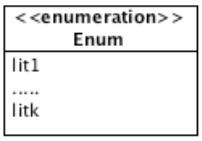


Fig. 22. A generic enumeration datatype

TDatDef: DatatypeDefinition → CASL-LTL-Specification
TDatDef(*datDef*) =
free type *datDef*.datatype.name ::= Generators(*datDef.datatype*);
axioms
TGensDef(*datDef*.generatorsDefinition);
TOpDef(*datDef*.operationDefinition);
TPredDef(*datDef*.predicateDefinition);

where Generators returns either the user declared generators, in the case of the definition of datatype with generators or the standard generator named as the type itself and having as many arguments as the attributes of the datatype itself, otherwise.

TGensDef: GeneratorsDefinition → CASL-LTL-Formula*
TGensDef($genCl_1 \dots genCl_n$) =
$genCl_1$.premise $\Rightarrow$ $genCl_1$.consequence
$genCl_2$.premise $\Rightarrow$ $genCl_2$.consequence
...
$genCl_n$.premise $\Rightarrow$ $genCl_n$.consequence

The above particular form of the axioms corresponding to an operations/ predicate definition guarantees that the various clauses are applied in order.

TOpDef: OperationDefinition → CASL-LTL-Formula*
TOpDef($opCl_1 \dots opCl_n$) =
$opCl_1$.premise $\Rightarrow$ $opCl_1$.consequence
$(\neg\ opCl_1$.premise$)\ \wedge\ opCl_2$.premise $\Rightarrow$ $opCl_2$.consequence
...
$(\neg\ opCl_1$.premise$)\ \wedge\ (\neg\ opCl_2$.premise$) \dots\ \wedge\ opCl_n$.premise $\Rightarrow$ $opCl_n$.consequence
The definition of TPredDef is similar.

Below we present the specification associated with the enumeration datatype of Fig. 22, where we use the free construct to assert that all the values are different.

free type *Enum* ::= *lit1* | ... | *litk*;

7 Interaction machines

The dynamics of the simple systems represented by a simple dynamic type may be presented in a constructive way by means of an interaction machine [1].

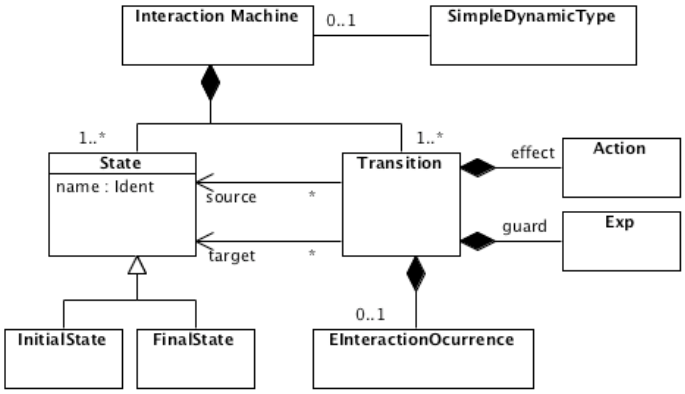
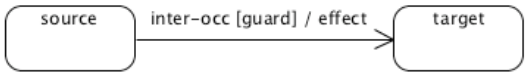
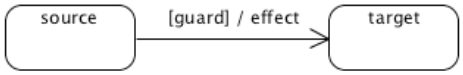


Fig. 23. The structure of an Interaction machine

An *interaction machine* associated with a simple dynamic type is an oriented graph, whose nodes represent its intermediate states and whose arcs represent its possible transitions. Fig. 23 shows the structures of the interaction machines.

The states should be considered as interaction states and correspond to the possible stages of the activity modelled by the machine (they include the special initial and final states). The transitions are decorated by an interaction occurrence, a guard, and an effect, and have the following forms:





The interaction occurrence inter-occ (interaction occurrence specification to be more precise) may have the following forms: "!_inter(e_1,...,e_n)", where !_inter is an elementary interaction and e_1, ..., e_n are ground expressions built over the attributes, or "?_inter(X_1,...,X_n)", where ?_inter is an elementary interaction and X_1, ..., X_n are variables. A transition without interaction occurence corresponds to some internal activity. guard is a boolean expression built over the simple dynamic type attributes, effect is an action over those attributes, the free variables, if any, of the interaction occurrence may appear in effect[9]. Here, we restrict the form of the effect to a sequence of assignments to the attributes.

An interaction machine must have a unique initial state, while it may have any number (also none) of final states. Obviously, no transition may enter in the initial state and no transition may leave a final state. At least a transition must leave a non final state.

[9] Notice that differently from the UML, in CASL-MDL the free variables cannot appear in the guard

The behaviour of an interaction machine may be defined in terms of run-to-completion-steps, similarly to UML state machines. At the beginning the initial state is active; at each time exactly one state is active, and the behaviour ends when a final state becomes active.

A run-to-completion-step is defined as follows. All transitions leaving the active state are collected (this collection cannot be empty). The guards of those transitions are evaluated, if no guard is true the step is terminated. Then the enabled transitions, i.e., those with a true guard are collected and the arguments of the output interactions are evaluated. If an enabled transition has no interaction occurrence, or its interaction occurrence is matched by a complementary one of the context, it is executed (if there are several ones in this situation, one of them is nondeterministically picked). If none is in this situation, the machine will wait in the active state making available to the context the interaction occurrences of all the enabled transitions, till one of them is matched.
To execute a transition corresponds to do the following: if the corresponding interaction occurrence is of kind receive, then the values of the matching send interaction occurrence are assigned to the corresponding variable arguments; then in bothe cases the effect is executed, after the target state becomes active, the source state (if different from the target) is no more active, and the step is terminated.

The guard of the form true may be omitted, similarly the effect corresponding to zero assignments.

Translation

T : InteractionMachine $\rightarrow$ CASL-LTL

FINITESET[**sort** *simpDT*.name] **and with** *ident* $\mapsto$ ident_*simpDT.name*

The states of a simple dynamic system when defined by an interaction machine are completely determined by the attributes by a generator named as the type itself having an argument for each attribute (also for the standard implicit attribute id).

$\mathsf{T}(interM) =$

free {

generated type $control\_simType ::= state_1 \mid \ldots \mid state_h$

generated type $simType ::= SimType(\_id : \text{ident\_}simType, control\_simType, a_1 : T_1, \ldots, a_n : T_n) \mid Final \mid Initial$

generated type $\text{label\_}simType ::= inter_1 \mid \ldots \mid inter_k$

axioms

$\mathsf{T}(trans_1) \ldots \mathsf{T}(trans_m)$

}

where $simType = interM.$SimpleDynamicType.name, $a_1 : T_1, \ldots, a_n : T_n$ are its attributes, $inter_1, \ldots, inter_k$ are its elementary interactions, $trans_1, \ldots, trans_m$ are the transitions of the interaction machine and $state_1, \ldots, state_h$ are the control states of the state machine.

T : Transition $\rightarrow$ CASL-LTL

Not connected with initial o final state

$\mathsf{T}(trns) = \mathsf{T}(trns.guard) \Rightarrow$

$SimType(id, trns.source, va_1, \ldots, va_n) \xrightarrow{\mathsf{T}(trns.EInteractionOccurence)} SimType(id, trns.target, \mathsf{T}(trns.effect,$
Transition from the initial state
$\mathsf{T}(trns) =$
$Inital \xrightarrow{\mathsf{T}(trns.EInteractionOccurence)} SimType(id, trns.target, \mathsf{T}(trns.effect, x, \ldots, x))$
Transition into final state $\mathsf{T}(trns) = \mathsf{T}(trns.guard) \Rightarrow$
$SimType(id, trns.source, va_1, \ldots, va_n) \xrightarrow{\mathsf{T}(trns.EInteractionOccurence)} Final$
dsort *simpDT*.name **label** label_*simpDT*.name
op _ *id* : *simpDT.name* → ident_*simpDT.name*
T(*simpDT*.attribute, *simpDT*.name);
T(*simpDT*.eInteraction, label_*simpDT*.name);

8 Structured Dynamic Type Definitions

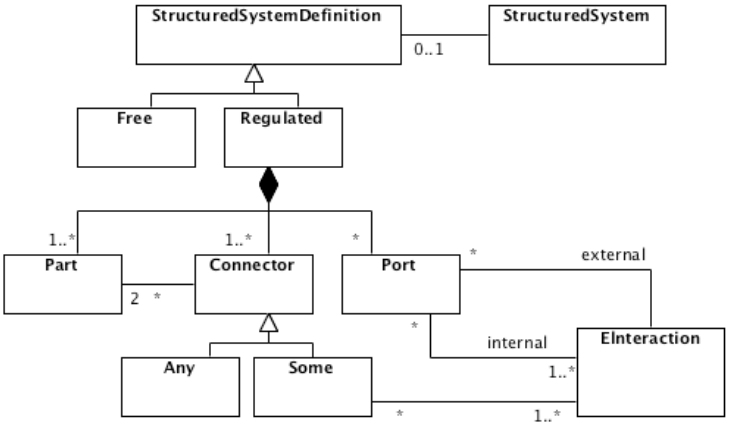


Fig. 24. The structure of the Structured Dynamic Type Definition

A structured dynamic type defines exactly which are the parts of the system and how they interact each other making clear also what will be the result of such interactions in term of interaction of the whole system with the external (to it) world. Thus a structured typedefinition define both the inner architecture of the system and its behaviour.

Fig. 24 shows the two possible forms of a structured dynamic type definition,

Let us to consider first the case of the Regulated systems. We assume that the parts/subsystems of a structured systems interact by performing pairwise matching elementary interactions, and the connector construct allows to express which matching pairs of interactions may be performed. A connector of kind Any depicts the case where any pairs of transitions of the two connected parts whose elementary interactions match may be performed simultaneously; whereas

a connector of kind Some allows to be performed simultaneously only the associated elementary interactions. Obviously if there is a connector with elementary interaction e between part A of type Sys1 and part B of type Sys2, thus either $!\_e$ is an elementary interaction of Sys1 and $?\_e$ is an elementary interaction of Sys2 or vice versa.

The port connector allows instead to propagate outside the system unmatched transitions, precisely if a part connected to a port performs a transition whose elementary interactions is among those on the line connecting to the port, then such transition may be performed without the need of a matching transition of another part, and will result in a transition of the whole structured system with the elementary transition attached to the port symbol.

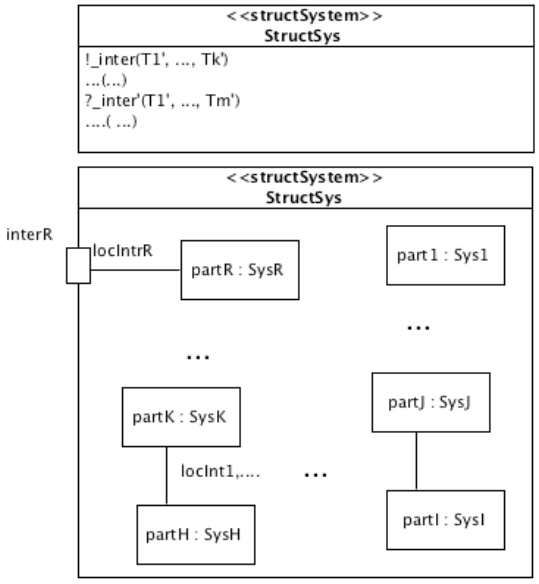


Fig. 25. A generic example of Structured Dynamic Type Definition

In Fig. 25 we present visually a generic structured dynamic type definition of the kind "regulated".

A very frequent case of structured system consists of a collection of any number of dynamic systems of some given (dynamic) types freely interacting each other (obviously only by performing matching elementary interactions); if we represent it using our notation it will result in a complete graph where the nodes are the parts and the arcs are the "any" connectors; thus we offer a special simplified notation for this case.

In Fig. 26 we show on the left the compact form for this case and on the right the completely equivalent expanded form.

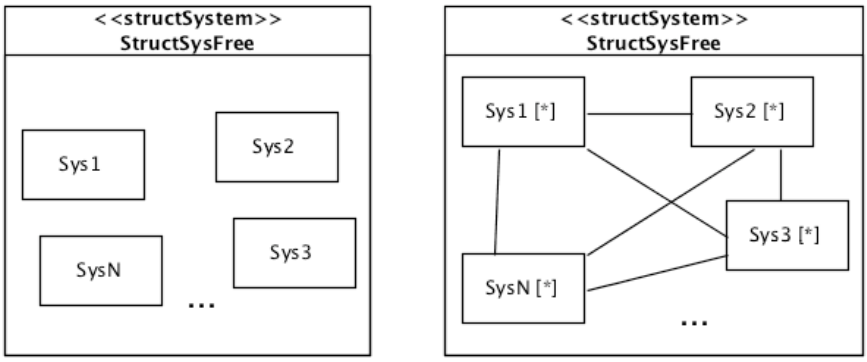


Fig. 26. A generic example of Free Structured Dynamic type

9 Conclusions and future work

In this paper we have presented a part of CASL-MDL, a visual modelling notation based on CASL-LTL (the extension for dynamic system of the algebraic specification language CASL developed by the COFI initiative). The visual constructs of CASL-MDL have been borrowed by the UML, so as to use professional visual editors; in this paper for example we used Visual Paradigm for UML[10].

A CASL-MDL model is a set of diagrams but it corresponds to a CASL-LTL specification, thus CASL-MDL is a suitable means to easily read and write large and complex CASL-LTL specifications; furthermore the quite mature technologies for UML model transformation may be used to automatize the transformation of the CASL-MDL models into the corresponding CASL-LTL specifications. Notice that what we have done is different from designing a new notation and then give it a semantics using CASL-LTL.

CASL-MDL may be used by people familiar with CASL-LTL to produce in an easier way specifications written with it with the help of an editor. However, the corresponding specifications are readable and can be modified directly, for example if there is the need of to do fine tuning for automatic verification.

We plan to investigate whether CASL-MDL may be presented directly as a visual modelling notation to be used for the various modelling tasks that occur in the software development processes, producing a user manual that following the "well-founded" approach fully hides from the users the formal foundation.

We are currently working out the relationships among the types, and consider the introduction of workflow-like diagrams similar to the UML activity diagrams to visualize the behaviour formula of groups of dynamic systems.

CASL-MDL shares many similar features with the UML, the visual notation being the most relevant one, but it is quite different, first of all because it is not object-oriented and has a simple immediate formal semantics; some constructs

[10] `http://www.visual-paradigm.com/product/vpuml/`

of CASL-MDL have been inspired by those of the UML, but are not exactly the same. Consider for example the sequence diagrams, in this case the CASL-MDL sequence diagrams allow also to express implications among the interactions (message exchanges in the UML), thus they are more powerful, and closer to the live charts of Harel and Damm [8]. We think that a careful investigation of the differences and relationships between CASL-MDL and UML may have as a result a better understanding of some of the UML constructs and perhaps some suggestions for possible evolutions. In some cases the two notations offer different but equally expressive ways to achieve the same modelling capacity, for example UML offers the objects and CASL-MDL the dynamic systems, in these case the best way to assess the relative merits of the two proposals is to organize some empirical experiment, trying to assess which one is easier to learn, use or allows to build models of better quality.

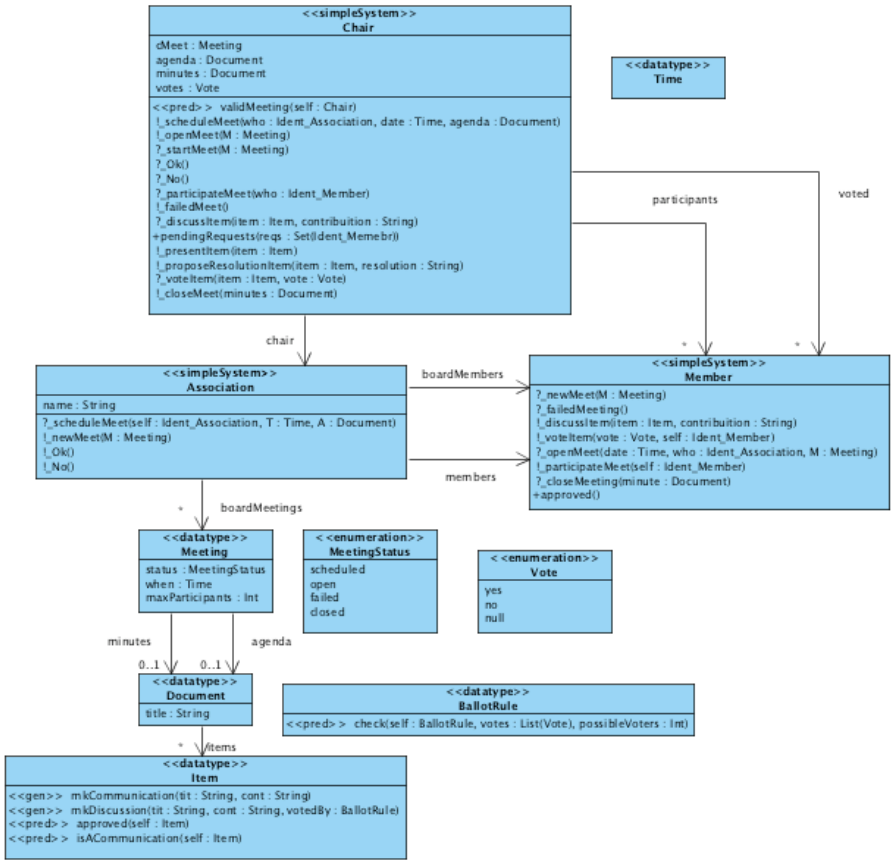
As regards the relationships between the UML and CASL-MDL let us note that CASL-MDL is not a semantics of the UML expressed in CASL-LTL, and CASL-MDL is not even a UML profile. However we plan to try to add the profiling mechanism in the CASL-MDL, since we found it quite valuable in using the UML.

Acknowledgement We warmly thank Maura Cerioli for a careful reading of a draft of this paper, and for her valuable comments.

A The Assoc case study: the complete model

A.1 The TypeDiagram

Here we show the "pure" TypeDiagram of the Assoc case study, where only the constructs introducing the various datatypes and dynamic systems are shown.



References

1. E. Astesiano and G. Reggio. From Conditional Specifications to Interaction Charts. In *Formal Methods in Software and Systems Modeling, Essays*, LNCS 3393, pages 167–189. Springer, 2005.
2. E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, LNCS 2757, pages 132 – 150. 2003.
3. C. Choppy and G. Reggio. Improving use case based requirements using formally grounded specifications. In *Fundamental Approaches to Software Engineering*, LNCS 2984, pages 244–260, 2004.

4. C. Choppy and G. Reggio. A UML-Based Approach for Problem Frame Oriented Software Development. *Journal of Information and Software Technology*, 47:929–954, 2005.
5. C. Choppy and G. Reggio. A formally grounded software specification method. *Journal of Logic and Algebraic Programming*, 67(1-2):52–86, 2006.
6. C. Choppy and G. Reggio. Service Modelling with Casl4Soa: A Well-Founded Approach - Part 1 (Service in isolation). In *Symposium on Applied Computing*, pages 2444–2451. ACM, 2010.
7. CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
9. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0–Summary. Technical Report DISI-TR-03-36, 2003.
10. UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.