CASL-MDL, modelling dynamic systems with a formal foundation and a UML-like notation

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

Abstract. In this paper we present a part of CASL-MDL, a visual modelling notation based on CASL-LTL (an extension for dynamic system of the algebraic specification language CASL). The visual constructs of CASL-MDL have been borrowed from the UML, thus existing editors may be used. 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. We use as a running example a case study that describes the functioning of a consortium of associations.

1 Introduction

The aim of our work is to reshape the formal specification language CASL-LTL [11] (a CASL [7] extension for dynamic systems) as a visual modelling notation, and this requires to provide a visual syntax to the CASL-LTL specifications. This work is motivated by the fact that in our opinion the currently available modelling notations have some problematic aspects, for example the lack of a formal semantics if not of a well-defined syntax.

We decided to attempt this experiment for the following reasons:
- CASL-LTL is very suitable to specify/model different kinds of dynamic systems, and at different levels of abstraction. Indeed, it has been used to specify the use-case based requirements [2], the main features (the domain, the requirements and the machine) for the basic problem frames [3], and recently it has been used to specify the services in the field of SOA (Service Oriented Architecture) [6].
- A modelling (specification) method for CASL-LTL was developed [4], where the modelling is guided by the use of simple ingredients/concepts as data types, dynamic system, elementary interaction, and cooperations between systems.
- CASL-LTL is extremely expressive (it includes a powerful first-order temporal logic), allows different styles of specification/modelling (e.g., property oriented and constructive [4]), while still based on a limited number of constructs,
- it is not object-oriented, and this may be an advantage, whenever the models are not used for developing object-oriented software (e.g., when using SOA),
- and obviously it has a well-defined semantics, and there exists software tools to help the formal verification of its specifications.

However, the textual, quite verbose syntax, may prevent to use CASL-LTL for large specifications, needless to say that this threatens the acceptance by the non-academic modellers. A visual syntax for CASL-LTL will help to keep the dimension of the models quite reasonable, and obviously will ease the understanding and the production of models, even by people without a deep know-how in logics and algebraic specifications.

In [4] we already made a first attempt to give a visual presentation to some CASL-LTL specifications, using ad hoc graphical symbols and icons. The attempt was successful for what concerns the compactness of the specifications and the ease to produce them, but further applications and extensive experimentations were prevented by the lack of supporting software tools, e.g., an editor. Obviously these tools could be developed but to produce high quality tools requires really a large effort. Moreover, the graphics of this first attempt of a visual syntax needed to be improved for a better legibility.

In [10] we find the same concern to ease the use of formal specifications by providing a graphical/visual notation for it, and they use both class diagrams and constraints diagrams to represent Z specifications. In [9], constraint diagrams and VisualOCL are compared as a means to visualize OCL expressions. Clearly, our approach here is not to propose a formal semantics to the UML [12], thus we do not refer to the numerous works of that field [1].

Our present work with CASL-MDL relies on borrowing visual constructs of the UML to build a visual syntax for CASL-LTL. This choice has some advantages:
- the graphical constructs are widely known, and were introduced in the UML by pre-existing notations, and now have been tested by a huge number of users for a long time; thus they are familiar and may be easily understood
- some peculiar characteristics of the UML, as its flexibility and the easiness to define variants of itself, and the very loose static semantics, make it possible to define the concrete visual syntax of CASL-LTL as a variant of the UML (that is as a UML profile)
- software tools for editing the UML models are widely available, and many good ones are free.

However, an issue is that, when looking at a CASL-MDL model, some people may be confused between UML and CASL-MDL, and perhaps may use CASL-MDL with the UML intuition. To overcome this we use the profile mechanism to stress the semantic differences; for example a CASL predicate is depicted as a UML operation without return value stereotyped by <<pred>>. A definitive answer about the fact that the borrowed syntax may lead to confusions can only be given by means of rigorous experiments, as proposed by the empirical software engineering, where people knowing both UML and CASL-MDL would be asked to interpret and to produce some models in a controlled way.

Let us remind of a famous occurrence of "syntax borrowing" that resulted in a big success, i.e., the definition of the Java programming language, where the syntax of the C language was kept on purpose whenever possible even if their semantics are totally different. In this case it seems that no confusion arose and also that the familiar aspects of the new language helped its acceptance.

Not any specifications of CASL-LTL will have a visual counterpart, but only a subset, however large enough to include all the specifications produced following the method of [4].

In CASL-MDL we have a type diagram, introducing the datatypes and the dynamic types, which are types of dynamic system (either simple or structured), used in the model. Constraints allow one to express properties on the introduced types (i.e., on the corresponding values or dynamic systems) using first-order and temporal logics. Interaction diagrams express visually properties on the interactions among the components of a structured dynamic systems using the same constructs as the UML sequence diagrams. The behaviour of the dynamic systems of a given type is modelled by interaction machines, using the same constructs as the UML state machines.

The definition of CASL-MDL is an ongoing work and in this paper we will present only the type diagrams and the interactions diagrams, while in [5] we describe also the interaction machines, the constraints, and the constructive definition of data types. Up to know we have not introduced in CASL-MDL diagrams for modelling the workflow, as the UML activity diagrams or the BPMN process diagrams, which would be very useful for using CASL-MDL for business modelling and the modelling of business processes; we are currently working on that.

In Sect. 2 we introduce the CASL-MDL models, in Sect. 3 and in Sect. 4 the type diagrams and the interaction diagrams respectively, and finally in the Sect. 5 the conclusions and the future works.

In the paper we use as a running example the modelling of ASSOC, a case study that describes 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 CASL-MDL constructs, an organic presentation of this model can be found in [5].

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 "conceptual" metamodel expressed using the UML[1]. The corresponding concrete syntax will be expressed by means of a UML profile, allowing to use the UML tools for editing and for model transformations (e.g., into the corresponding textual CASL-LTL specifications). Thus CASL-MDL has both a conceptual and a con-

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

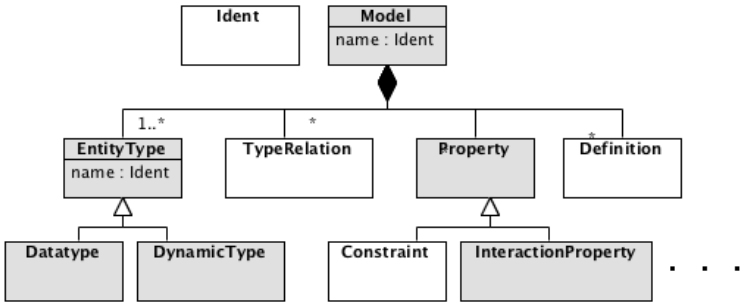


Fig. 1. Structure of the CASL-MDL models ("conceptual" metamodel)

crete metamodel, the first will be used by the human to grasp the notation, and the latter by the computers to produce and elaborate the CASL-MDL models.

A CASL-MDL model consists of *entity type* declarations (EntityType), 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 space we 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 are a set of axioms and the definitions in subspecifications built by the CASL-LTL "free" construct.

Translation

TModel : Model → CASL-LTL-Specification

TModel(*mod*) =

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

The translation of the entity types (at least one must be present in a CASL-MDL model) yields 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

[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*

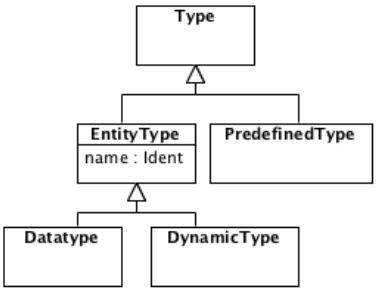


Fig. 2. Structure of Type and the Entity types (metamodel)

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.

Visually a CASL-MDL model is a set of diagrams including at least a Type-Diagram 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 consists of a type diagram made by entity type declarations and constraints and of a set of interaction 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

A type may be either predefined or an entity type (declaration) which, as 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 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** ... **and** $\mathsf{Basic}(et_n.\text{name})$ **then**

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

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

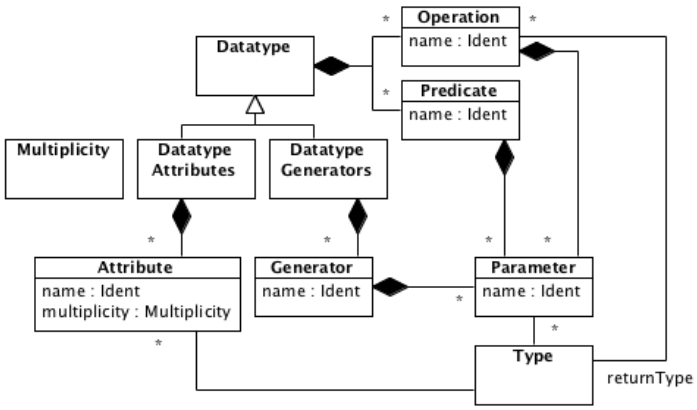


Fig. 3. Datatype Structure (metamodel)

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 one to have that a type in the type diagram may use all the other types present in the same diagram to define its features.

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.

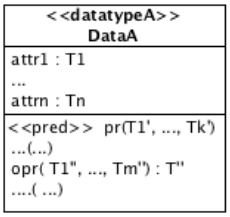
The structure of a datatype of CASL-MDL may be defined in two different ways, using either *generators* or *attributes*.

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.

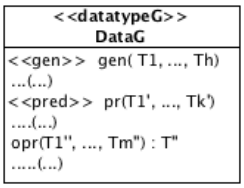
The other possibility is to define the datatype values in terms of attributes, similarly to UML. An attribute attr: T of a datatype D corresponds to a CASL operation _ _ .attr: D → T. In this case there is a standard generator named as the type itself having as many arguments as the attributes, but it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

[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.

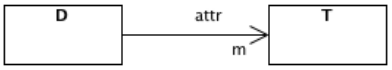
[4] We prefer to use the term generator instead of constructor used in the OO world to make clear that in our notation we have datatypes with values and not classes with objects.



(a) Schematic datatype with attributes



(b) Schematic datatype with generators



(c) Alternative visual presentation of an attribute

Fig. 4. Visual notation for datatypes

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).

The attributes may have a multiplicity, and its meaning is that the type of the attribute is a set of the associated type and that its values satisfy an implicit constraint [5] 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 as in Figure 4(c).

The modellers are free to use plain attributes or their visual counterpart, but notice that using the arrows shows 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 is simply represented by a box including the name of the datatype.

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.

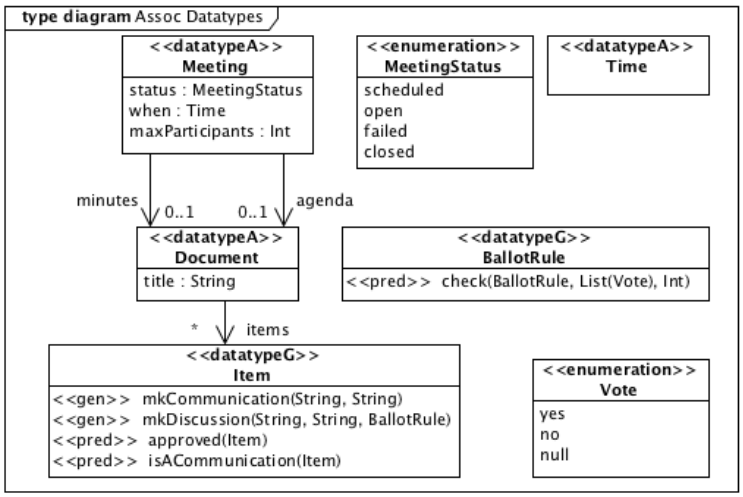


Fig. 5. ASSOC: Type Diagram containing some dataypes

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 allows 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 [5]).

Time is a datatype where no detail is given (it just corresponds to the introduction of 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 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 $\_\_.agenda : Meeting \to Set(Document)$; ...
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 $(State, Label, \to)$, where *State* denotes the set of states and *Label* the set of transition labels, and $\to \subseteq State \times Label \times State$ is the *transition relation*. A triple $(s, l, s') \in \to$ 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

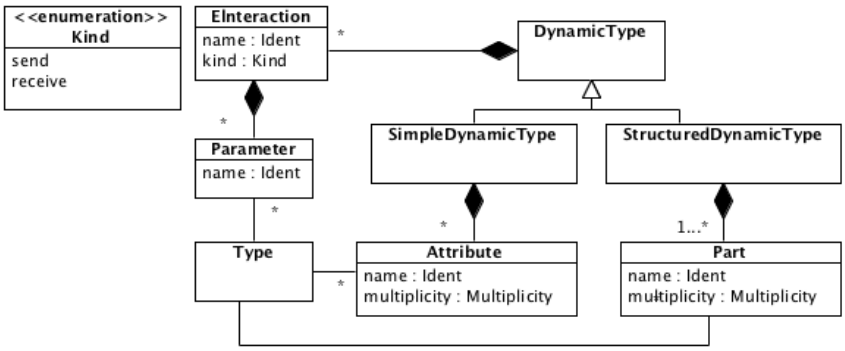


Fig. 6. Dynamic Type Structure (metamodel)

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].

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,[6] 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

[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.

[6] DynamicType is a specialization of Type (see also Fig. 2) which has a link to Part.

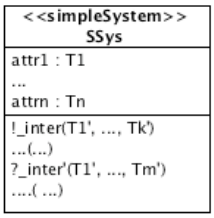


Fig. 7. A schematic Simple Dynamic Type

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** $ident \mapsto$ 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);

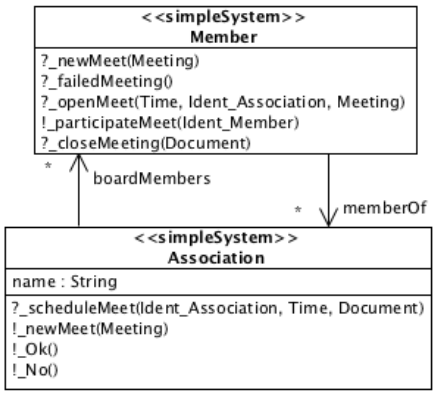


Fig. 8. ASSOC Example: a type diagram including simple dynamic types

ASSOC Model: 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 [5]).

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 type Association is as follows.

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

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

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

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

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

TAU is a special implicit element used to label the transitions 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.

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

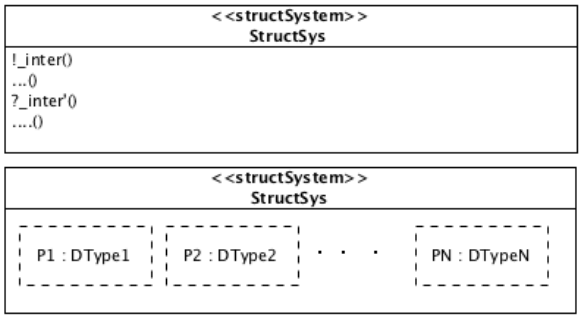


Fig. 9. A schematic Structured Dynamic Type

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 → CASL-LTL-Specification

Basic(*structDT*) =

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

IDENT **with** *ident* ↦ 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

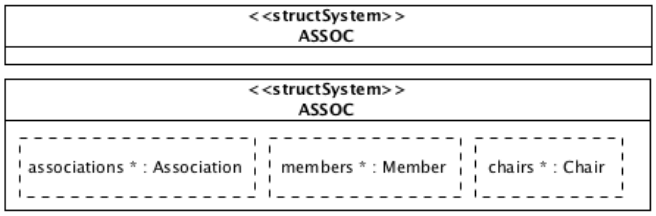


Fig. 10. ASSOC Example: a type diagram including a structured dynamic type

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 : $ASSOC \rightarrow$ ident_$ASSOC$
op $associations$: $ASSOC \rightarrow Set[Association]$
op $members$: $ASSOC \rightarrow Set[Member]$
op $chairs$: $ASSOC \rightarrow Set[Chair]$
pred $isPart$: $ASSOC \times ident\_all$
op TAU : localInteractions_$ASSOC \rightarrow$ label_$ASSOC$

where LOCALINTERACTIONS= FINITESET[LOCALINTERACTION] and
LOCALINTERACTION =
free type = $LocalInteraction$ ::=
$<$ _ _ $>$ (ident_$Association$; label_$Association$) |
$<$ _ _ $>$ (ident_$Member$; label_$Member$) |
$<$ _ _ $>$ (ident_$Chair$; label_$Chair$)

4 Interaction properties

The metamodel of CASL-MDL interaction properties is given in Fig. 11.

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. 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

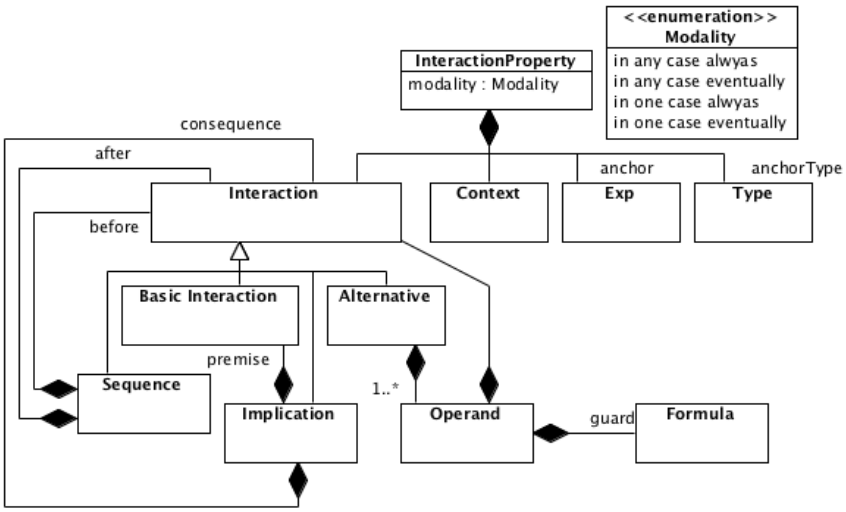


Fig. 11. Interaction Properties structure (metamodel)

whether the property about the interactions must hold in all possible instants of those 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. 11.

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. 12, an interaction property is visually presented by reusing 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. 13, is the simplest form of Interaction and just corresponds to assert that a series of *elementary interaction occurrences* happen in some order among some generic roles for dynamic systems parts of the anchor (*lifelines*), where an interaction occurrence is the simultaneous 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 occurrence connects two lifelines in specific points (represented by the lifeline

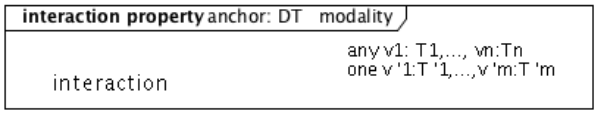


Fig. 12. Visual presentation of a generic CASL-MDL interaction property

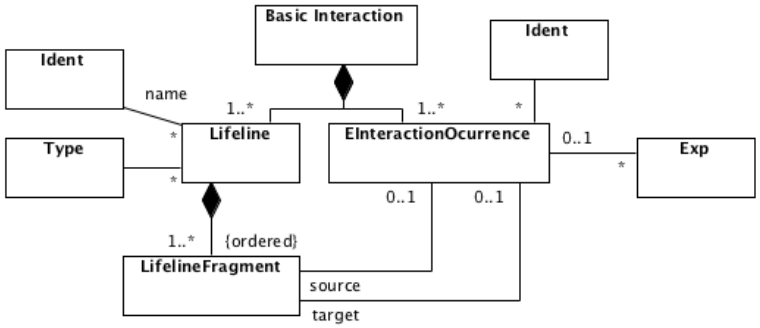


Fig. 13. Structure of Basic Interactions (metamodel)

fragments); 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. 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. 14 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. 15 we show two different basic interactions that are, however, perfectly equivalent determining both the partial order listed at the bottom; notice that there are many other ones visually different but still equivalent.

An interaction property corresponds to a CASL-LTL formula.

Translation

TIntProp: InteractionProperty → CASL-LTL-Formula

TIntProp(iPr) =

Fig. 14. Generic example of elementary interaction occurrence

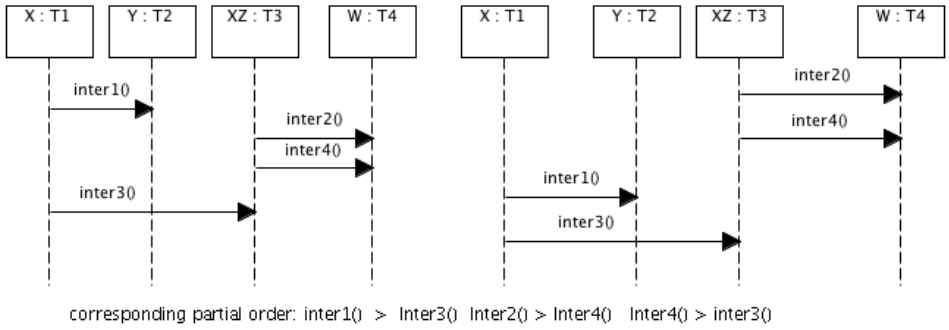


Fig. 15. Two perfectly equivalent basic interactions

$\forall\ \mathit{freeVars}\,\mathsf{TContext}(\mathit{iPr}.\text{context}) \bullet (\wedge_{x \in \mathit{iPr}.\text{lifeline}}\ \mathit{isPart}(x.\mathit{id}, \mathit{iPr}.\text{anchor})) \Rightarrow$
$\mathsf{TModal}(\mathit{iPr}.\text{modality}, \mathit{iPr}.\text{anchor}, \mathsf{TInteract}(\mathit{iPr}.\text{interaction}, \mathit{true}))$
where *freeVars* are all the free variables appearing in the anchor expression and those corresponding to the lifelines.

$\mathsf{TModal} : \mathsf{Modality} \times \mathsf{Exp} \times$ CASL-LTL-PathFormula $\rightarrow$ CASL-LTL-Formula
$\mathsf{TModal}(\text{in any case always}, \mathit{dexp}, \mathit{PF}) = \mathit{in\_any\_case}(\mathit{dexp}, \mathit{always}\ \mathit{PF})$
similarly for the other three cases

$\mathsf{TInteract} : \mathsf{Interaction} \times$ 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.

$\mathsf{TInteract}(\mathit{basicInt}, \mathit{cont}) =$
$\vee_{\mathit{eIOc}_{i_1} \ldots \mathit{eIOc}_{i_n} \text{ admissible ordering of } \mathit{eIOc}_1, \ldots, \mathit{eIOc}_n}$
$\mathsf{TIntOcc}(\mathit{eIOc}_{i_1}) \wedge \mathit{eventually}\ (\mathsf{TIntOcc}(\mathit{eIOc}_{i_2}) \wedge (\mathit{eventually}\ \ldots$
$(\mathsf{TIntOcc}(\mathit{eIOc}_{i_n}) \wedge \mathit{eventually}\ \mathit{cont}) \ldots))$

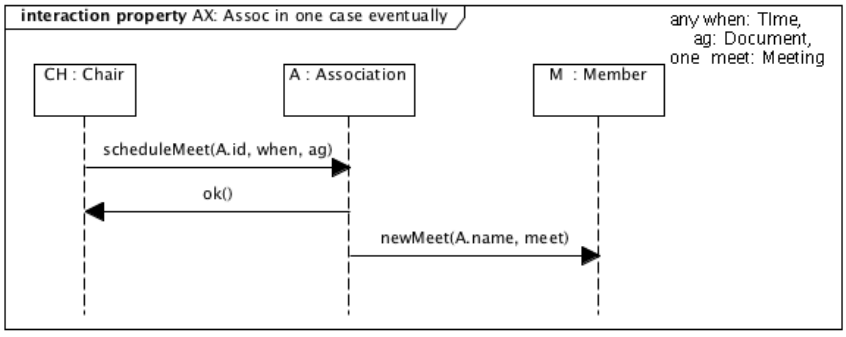


Fig. 16. ASSOC: scheduling a new meeting (successful case)

where $basicInt$.eInteractionOccurence = $eIOc_1, \ldots, eIOc_n$

TIntOcc: InteractionOccurrence → CASL-LTL-PathFormula

TIntOcc($eIOc$) =

$(x.id{:}!\_inter(exp1, \ldots, expn) \ \wedge \ y.id{:}?\_inter(exp1, \ldots, expn))$[7]

where $eIOc$ has the form in Fig. 14.

Fig. 16 shows an interaction property 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.

Fig. 11 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. 17 and Fig. 18.

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. 17 and Fig. 18 presents the interaction properties, with a structured interaction part, corresponding to those cases. In Fig. 17 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. 18 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 interaction property is different, it says that whenever the scheduling request occurs it will be followed by an answer.

Translation

[7] Recall that $\_\_.id$ is the standard attribute returning the identity of a dynamic system, and that id: $interact$ is a local interaction atom.

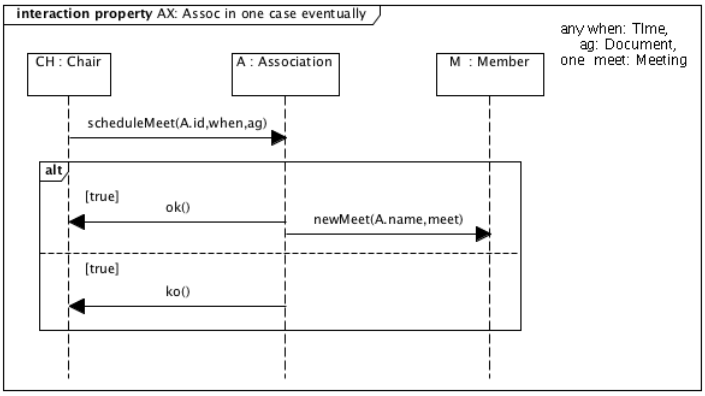


Fig. 17. ASSOC: scheduling a new meeting (sequence and alternative combinator)

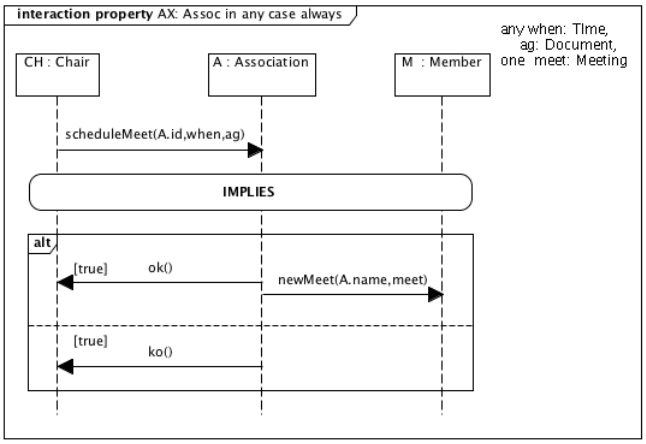


Fig. 18. ASSOC: scheduling a new meeting (implies combinator)

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

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

$\quad \wedge_{J \subseteq \{1,\ldots,n\}} ((\wedge_{j \in J}\ op_j.\text{guard}\ \wedge \wedge_{i \in \{1,\ldots,n\}-J} \neg\ op_i.\text{guard})\ \Rightarrow$

$\qquad \vee_{j \in J}\ \mathsf{TInteract}(op_j.\text{interaction}, cont))$

where $altInt.\text{operand} = op_1, \ldots, op_n$

$\mathsf{TInteract}(seqInt, cont) = \mathsf{TInteract}(seqInt.\text{before}, \mathsf{TInteract}(seqInt.\text{after}, cont))$

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

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

$\quad (\mathsf{TIntOcc}(eIOc_{i_1})\ \Rightarrow\ next\ always\ (\mathsf{TIntOcc}(eIOc_{i_2})\ \Rightarrow\ next\ always\ (\ldots$

$\qquad (\mathsf{TIntOcc}(eIOc_{i_n})\ \Rightarrow\ next\ eventually\ \mathsf{TInteract}(implInt.\text{consequence}, cont))\ldots)))$

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

Here there is the CASL-LTL formula corresponding to the interaction property of Fig. 17 after some simplifications due to the fact that the guards are both equal to true:

$$
\begin{array}{l}
\forall\ AX\text{: }Assoc, when\text{: }Time, ag\text{: }Document, CH\text{: }Chair, A\text{: }Association, M\text{: }Member \\
\exists\ meet\text{: }Meeting \bullet \\
(isPart(CH.id, AX)\ \wedge\ isPart(A.id, AX)\ \wedge\ isPart(M.id, AX))\ \Rightarrow \\
\quad in\_one\_case(AX, eventually \\
\quad (CH.id{:}!\_scheduleMeet(A.id, when, ag)\ \wedge\ A.id{:}?\_scheduleMeet(A.id, when, ag) \wedge \\
\quad (eventually \\
\qquad (A.id{:}!\_ok()\ \wedge\ CH.id{:}?\_ok() \wedge eventually \\
\qquad\quad (A.id{:}!\_newMeet(A.name, meet)\ \wedge\ M.id{:}?\_newMeet(A.name, meet))) \\
\qquad \vee\ (A.id{:}!\_ko()\ \wedge\ CH.id{:}?\_ko())))
\end{array}
$$

The CASL-LTL formula corresponding to the interaction property of Fig. 18 can be found in [5].

5 Conclusions and future work

In this paper we present 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 to the UML, so as to use professional visual editors; in this paper for example we used Visual Paradigm for UML[8].

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.

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 fine tuning for automatic verification.

We present here a part of CASL-MDL, the type diagram and the interaction diagrams, [5] presents also constraints, definitions for datatypes (which make precise their structure and the meaning of their operations and predicates), definitions of structured dynamic types, which fix their structures and the way their parts interact among them, and interaction machines, which are diagrams visually similar to the UML state machines, modelling the behaviour of the simple dynamic types.

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 formulas on the behaviour of groups of dynamic systems.

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

UML is the most relevant visual modelling notation, thus it is important to asses the common aspects and the differences with CASL-MDL.

CASL-MDL and UML are visually alike, but they are quite different, first of all because CASL-MDL is not object-oriented and has a simple "native" formal semantics, and because the semantics of syntactically similar constructs is not exactly the same. Consider for example the CASL-MDL interaction diagrams visually similar to the UML sequence diagrams; the interaction diagrams allow also to express implications among the interactions (message exchanges in the UML), thus they are more powerful than the UML sequence diagrams, and closer to the live charts of Harel and Damm [8]. The appendix compares in a tabular form the features of CASL-MDL and of UML.

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.

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 that it is not true that a CASL-MDL model may be transformed into an equivalent UML model

Acknowledgements: We warmly thank Maura Cerioli for a careful reading of a draft of this paper, and for her valuable comments. We would also like to thank Hubert Baumeister and the anonymous referees for their helpful comments.

References

1. M. V. Cengarle, A. Knapp, and H. Mühlberger. Interactions. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 205–248. John Wiley & Sons, 2009.
2. 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.
3. 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.
4. C. Choppy and G. Reggio. A formally grounded software specification method. *Journal of Logic and Algebraic Programming*, 67(1-2):52–86, 2006.
5. C. Choppy and G. Reggio. CASL-MDL, modelling dynamic systems with a formal foundation and a UML-like notation (full report). Technical report, Universite Paris 13, and Universita di Genova, 2010. `http://www-lipn.univ-paris13.fr/~choppy/REPORTS/casl-mdl-report.pdf`.
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. A. Fish, J. Howse, G. Taentzer, and J. Winkleman. Two visualisations of OCL: A comparison. Technical Report VMG.05.1, University of Brighton, 2005.

10. S.-K. Kim and D. Carrington. Visualization of formal specifications. In *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, APSEC '99, pages 38–45. IEEE Computer Society, 1999.
11. 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.
12. UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.

A Comparison between Casl-Mdl and UML

	CASL-MDL	UML
datatypes		
user defined attribute style	+	+
user defined with constructors a la ML	+	
explicit predicates	+	
partial operations	+	exceptions? OCL ?
property oriented definition	invariants, pre-post conditions on operations plus any kind of first order formulas about operations, constructors and attributes	invariants pre-post conditions on operations
constructive definition	rule-based definitions of operations and constructors	methods associated with operations
dynamic entities		
	dynamic systems	active objects
communication mechanism	execution of groups of matching elementary interactions	operation call and signal sending
property oriented definition	branching time temporal logic formulas (e.g., invariants, safety and liveness)	invariants, pre-post conditions on operations
constructive definition	interaction machine (reactive, proactive, passive and internal behaviour)	state machine (reactive behaviour)
objects	as a special kind of passive dynamic systems	native objects
structured dynamic entities	structured dynamic systems	standard community of all objects, structured classes
specification of the interaction among components of structured entities	interaction properties (possibility of expressing liveness and safety properties)	sequence diagrams (samples of message exchanges)
workflow	under development	activity diagrams
.....		