A Formally Grounded Software Specification Method

Christine Choppy [a] Gianna Reggio [b]

[a] *LIPN, Institut Galilée - Université Paris XIII, France*

[b] *DISI, Università di Genova, Italy*

Abstract

One of the goals of software engineering is to provide what is necessary to write relevant, legible, useful descriptions of the systems to be developed, which will be the basis of successful developments. This goal was addressed both from informal approaches (providing in particular visual notations) and formal ones (providing a formal sound semantic basis). Informal approaches are often driven by a software development method, and, while formal approaches sometimes provide a user method, it is usually aimed at helping to use the proposed formalism when writing a specification. Our goal here is to provide a companion method that helps the user to understand the system to be developed, and to write the corresponding formal specifications. We also aim at supporting visual presentations of formal specifications, so as to "make the best of both formal and informal worlds". We developed this method for the (logical-algebraic) specification languages CASL (Common Algebraic Specification Language, developed within the joint initiative CoFI) and for an extension for dynamic systems CASL-LTL, and we believe it is general enough to be adapted to other paradigms.

Another challenge is that a method that is too general does not encompass the different kinds of systems to be studied, while too many different specialized methods result in partial views that may be difficult to integrate in a single global one. We deal with this issue by providing a limited number of instances of our method, fitted for three different kinds of software items, while keeping a common "meta"-structure and way of thinking. More precisely, we consider here that a software item may be a simple dynamic system, a structured dynamic system, or a data structure, and we show here how to support property-oriented (axiomatic) specifications. We are thus providing support for the "building-bricks" tasks of specifying software artifacts that in our experience are needed for the development process.

Our approach is illustrated with a lift case study.

Key words: Specification method, formal specification, algebraic-logical specification, visual notation, CASL, CASL-LTL

1 Introduction

1.1 Aims and Scope

One of the goals of software engineering is to provide paradigms, languages, notations (together with a companion user method) to write relevant, legible, useful descriptions of the systems to be developed, which will be the basis of successful developments. This goal has been explored both from informal and formal approaches, while informal notations may put emphasis on varieties of attractive graphics, formal approaches offer the serious basis of notations with a formally described semantics. In both cases, one problem may be that the companion user method is not available to start with, and when it is available another problem is that, while it helps to use the proposed notation, it does not always help to understand the system to be developed. Another difficulty when struggling with these issues is that systems under study may be quite different in nature (or may include parts that are so), thus different notations and methods may be needed. To define a homogeneous approach, general enough to encompass different issues, but still carrying meaningful and precise guidelines and concepts, is also a goal.

On the one hand, many formalisms and some formal specification methods were developed (see [3] for the distinction between formalism and method), e.g., algebraic specifications and associated methods [2]. On the other hand, we can witness the success of development methods without or with a very limited grounding in sound formal theories, as those based on UML [25], e.g., RUP [26]. Clearly, there is a need to accommodate both worlds, for instance some recent works try to give a precise semantics to UML ([28, 29]), and the need for UML based rigorous methods has emerged. There are obvious differences between the two kinds of approaches (formal/informal):

- not very friendly notation, sometimes based on exotic mathematical symbols/very friendly visual notation;
- rather rigid with a lot of overhead notation/flexible adaptable notation;
- need time and background to learn the used technique/short time to learn the method;
- mainly simple toy case studies considered/developed having in mind the real common applications;
- user manuals explaining how to use the various constructs are available/ software development methods based on them are available.

Our attempt is to make the *best* of both worlds by trying to propose methods for the basic specification/modelling[1] blocks needed in a development process

[1] Notice that "to specify/specifications" are the terms used in the formal commu-

that have all the good properties of those commonly used (friendly notation based on simple intuitive visual metaphors, easy to understand and to learn, considering real applications, ...), and that are also *formally grounded*, i.e., their specification/model artifacts have a direct counterpart in a formal specification language, and thus a formal semantics, based on well defined underlying formal models.

Here, we present a first proposal for those methods formally grounded on the algebraic specification language CASL and its extension CASL-LTL [9, 24, 27]. Such techniques result in producing specifications/models having a precise structure at the conceptual level, which can, then, be presented either in a visual way or as formal CASL-LTL specifications.

Our previous experiences (see, e.g., [14]) suggested that the various activities in a development process are based on the "building-bricks" tasks of specifying software items of different nature at different levels of abstractions. We assume that a "*software item*" may be either

- a *simple dynamic system* (just a dynamic interacting entity in isolation, e.g., a sequential process) or
- a *structured dynamic system* (a community of mutually interacting entities, simple or in turn structured), or
- a *data structure* (or data type).

For each case, we give a specification method by giving the abstract structure of the relative specifications with the related visual presentation and the corresponding formal CASL-LTL specification. Here, we consider only property oriented specification methods, whereas in [15] we also describe model oriented ones. Our approach is quite systematic, and provides enough guidelines so as to prevent the "empty page syndrome" of someone who would not know where to start with, and also to cover a wide range of features and properties one should not forget.

We could show, see [15], that our specification blocks are general and powerful enough to be used as basis for a development method based on M. Jackson problem frames [21], and we applied them to three of the most relevant problems presented by Jackson, as well as to the requirement phase for a medium-sized Internet-based auction system.

Although in this paper, the target formal language of our methods is CASL-LTL they could be used with other target languages which follow the property oriented paradigm e.g., UML class diagram (with OCL constraints) and/or sequence diagrams and/or activity diagrams.

nity, whereas in the practical world the corresponding ones are "to model/models". For example, we have CASL specifications and UML models.

To introduce a specification method we follow the conceptual schema of [3] that we briefly present in Sect. 2.1; furthermore the specification methods presented here are all specializations for particular varieties of items of a general method that we present in Sect. 2.2. The sections 3, 4 and 5, devoted respectively to simple dynamic systems, structured dynamic systems, and data structures, have the same structure. First, the considered items are described, then, their specification technique is presented, followed by an illustration on an example, and the CASL-LTL or CASL view. In Sect. 6 we draw some conclusions, relate our approach to other ones, and present some future works. The remaining of our introduction is devoted to a brief presentation of the CASL and CASL-LTL specification languages in Sect. 1.2.

1.2 CASL*, the Common Algebraic Specification Language, and* CASL-LTL

"CASL is an expressive language for the formal specification of functional requirements and modular design of software. It has been designed by CoFI [2], the international *Common Framework Initiative for algebraic specification and development.* It is based on a critical selection of features that have already been explored in various contexts, including subsorts, partial functions, first-order logic, and structured and architectural specifications." [1] The CoFI project is presented in [23], and various documents are available on CASL, in particular the CASL Reference [24] including a complete formal semantics, and the CASL User Manual [9]. Thus, only the features of the language that are used in our examples will be shortly presented.

As shown in the example below, a CASL specification may include the declarations of sorts, operations and predicates (together with their arity), and axioms that are first-order formulae [3], respectively introduced by relevant keywords. Some operations play the role of constructors, thus, "datatype declarations may be used to abbreviate declarations of sorts and constructors." [9] Our approach is quite systematic, and provides enough guidelines so as to prevent the "empty page syndrome" of someone who would not know where to start with, and also to cover a wide range of features and properties one should not forget.

spec SPECNAME =
  type $type\_name$ ::= $con\_name(argTypes_{con}) \mid \ldots$
  ops $op\_name : argTypes_{op} \to resType_{op}$
    ...
  preds $pr\_name : argTypes_{pr}$
    ...
  axioms *formulae*

[2] http://www.brics.dk/Projects/CoFI

[3] with *strong* equality (Sect. 5.1) and a 2-valued logics

As shown below, “large and complex specifications are easily built out of simpler ones by means of (a small number of) specification building primitives … Union (keyword ‘**and**’) and extension can be used to structure specifications … Extensions, introduced by the keyword ‘**then**’, may specify new symbols, possibly constrained by some axioms, or merely require further properties of old ones …”[9]

```
spec SpecName =
    SP_1  and … and SP_j  then
    type  type_name  ::=  con_name(argTypes_con) | …
        …
```

“In practice, a realistic software specification involves *partial* as well as total functions.”[9] Partial operations or constructors are declared with a ’?’ symbol, and the definedness of a term can be asserted in the axioms.

```
spec SpecName =
    type  type_name  ::=  con_name(argTypes_con)? | …
    ops op_name : argTypes_op →? resType_op
        …
    axioms
        def(con_name(…))  ⇔  …
```

Let us note that special care is needed in specifications involving partial functions. Functions, even total ones, propagate undefinedness, and predicates do not hold on undefined arguments. Terms containing partial functions may be undefined, i.e., they may not denote any value.

Another helpful feature of Casl is the **free** construct. “Free specifications provide initial semantics and avoid the need for explicit negation … In models of free specifications, it is required that values of terms are distinct except when their equality follows from the specified axioms: the possibility of unintended coincidence between them is prohibited.”[9]

```
spec SpecName =
    SP_1  and … and SP_j  then
    free {  type  type_name  ::=  con_name(argTypes_con) | …
            ops op_name : argTypes_op →? resType_op
            …
            axioms  …   }
```

Generic specifications (also known as parametrized specifications in other specification languages) are very useful for reuse. Their parameter specification is usually very simple, and an instance of a generic specification is obtained by providing an argument specification for each parameter. The following specification is an extension of an instance of the generic specification FiniteSet[Elem] by Int (both are in the basic specifications library [31]).

spec SPECNAME = FINITESET[INT] ... **then** ...

"CASL is the heart of a *family* of languages. Some tools will make use of well-delineated *sub-languages* of CASL ... while *extensions* of CASL are being defined to support various paradigms and applications."[1] One of these extensions is CASL-LTL [27], which was designed for the dynamic systems specification by giving a CASL view to LTL, the Labelled Transition Logic ([4, 17]).

LTL, and thus CASL-LTL, is based on the idea that a dynamic system is considered as a *labelled transition system* (shortly *lts*), and that to specify it one has to specify the labels, the states and the transitions of such system. Recall that an lts is a triple $(State, Label, \rightarrow)$, where $\rightarrow \subseteq State \times Label \times State$. CASL-LTL offers a special construct to declare an lts, by stating that two given sorts correspond respectively to its states and labels, and that a standard arrow predicate corresponds to its transition relation $\rightarrow$.

dsort *St* **label** *Lab* stands for **sorts** *St*, *Lab*
pred $\_\_ \xrightarrow{\_\_} \_\_ : St \times Lab \times St$

The sort *St* is said *dynamic*, because any of its elements, say d, represents a dynamic system, whose behaviour is modelled by the transition tree associated with d. The root of such tree is decorated with d, and if the tree has a node decorated with d and $d \xrightarrow{l} d'$, then it has a node decorated with d', and an arc from d to d' decorated with the label l associated with the transition from from d to d'. Moreover, in such tree the order of the branches is not considered, and two identical decorated subtrees with the same root are considered as a unique subtree.

The CASL formulae built by using the transition predicates allow to express some properties on the behaviour of the dynamic elements, but they are not sufficient. For example, by using them we cannot state liveness properties; whereas they, and other kinds of quite relevant properties, may be expressed by using some temporal logic. Thus, CASL-LTL (as LTL) includes the temporal combinators of the temporal logic of [17], which is many-sorted, first-order, branching-time, CTL-style, and with edge formulae.

The temporal formulae of CASL-LTL are anchored to terms of dynamic sort and express some properties about the elements represented by them. Such formulae have the form $in\_any\_case(dt, \pi)$ or $in\_one\_case(dt, \pi)$ stating that any path (at least one path) starting from dt satisfies the condition expressed by the path formula π. A *path* starting form a dynamic element is a sequence of concatenated transitions from such element, and represents one of its possible behaviours. A path formula may require that

- the first state/label of the path satisfies some condition
  $[\ x \bullet cond\ ]$ and $<\ y \bullet cond >$;
- from some point on the path satisfies a condition expressed by another path

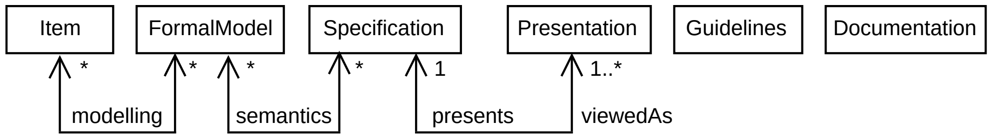
formula *eventually* π_1

- the path satisfies a condition expressed by another path formula (π_1) until some point where it satisfies a second condition (π_2) π_1 *until* π_2
- the path satisfies a condition expressed by another path formula in any point *always* π_1
- the path satisfies a complex condition, by combining other path formulae by means of the Casl first-order combinators, e.g., $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\forall$.

2 A framework for specification methods

2.1 Specification Methods

To easily present the various specification methods introduced in this paper, we follow the conceptual schema proposed in [3]. In the picture below we report all the ingredients of a generic method using an object-oriented visual notation[4], and after briefly present them, using as an example the loose algebraic specification of abstract datatypes [2] (the parts related to the example will be written within [square brackets]).



Items In our opinion a specification method to be effective should consider a quite precise set of *items* to be specified. Such items should be introduced using the natural language, since clearly they cannot be formally defined.
[Datatypes]

Formal models of the items Formal models, intended as mathematical structures, are the formal counterparts of the items, introduced before. Each specification method uses a particular set of formal models.
[Many-sorted Σ-algebras with predicates]

Modelling A precise and rigorous, but not formal, description of how the formal models are associated with the items. [The elements of the carriers model the values of the datatype, whereas the interpretations of the predicates and of the operations model the datatype operations]

Specifications In a very general way a *specification* is a description of an item at some level of abstraction, intended at a given step of the development process. A *specification* is a way to define a class of formal models: all those modelling the item at a given step of the development process.

[4] Precisely, it is a simple subset of UML 1.3 [25]. Recall that boxes represent classes, and arrows oriented associations.

[A specification is a pair consisting of a signature Σ and of a set of first-order formulae over Σ]

Semantics The semantics links a specification with its formal models.

[The semantics of (Σ, AX) is the class of the Σ-algebras satisfying all formulae in AX]

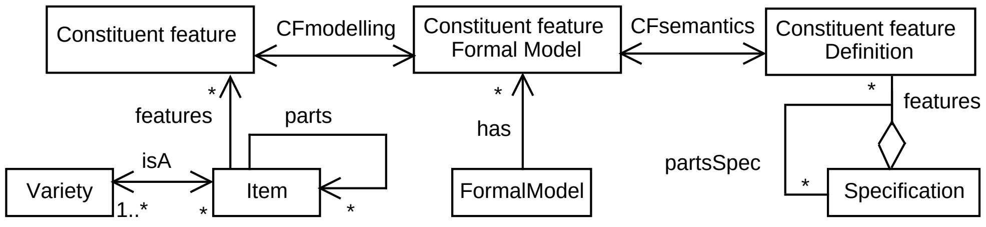
Presentations We mean by *presentation* a way to display a specification artifact for some particular purpose; for example, we can have a presentation for the human users, or using a special notation to be handled by a tool. A specification method may be equipped with different kinds of presentations. Each presentation should be associated with a unique specification.

[A specification (Σ, AX) may presented using the CASL language or visually as suggested in Sect. 5]

Guidelines This part consists of the *guidelines* for steering and helping the task of producing in the best possible way the specifications of the items. The guidelines are understandably driven by the preceding parts of the method, but note the fundamental role played by modelling, if we want seriously to provide professional guidelines. [See Sect. 5]

Documentation We refer to documenting the specification task for use in evolution and maintenance.

We make the following assumptions on the *items*, visually summarized below[5].



- Items are classified in some *variety* (e.g., functional modules/datatypes, reactive systems, real-time systems, distributed systems, ...), and the items considered by a method should be all of the same variety.
- Items are *structured*, and their subparts are items. Such structure is represented by the association parts. Items associated by parts may be of the same variety (homogeneous structure) or of different varieties (e.g., imperative programs made out from procedures).

  [A datatype may be structured, and in this case its parts are other datatypes, e.g., integers are the part of lists of integers]
- Items are characterized by their *constituent features*. We assume that an item is made by various *constituent features*/ingredients that are orthogonal/nonoverlapping, and that may be classified in different *kinds*.

  [The constituent features of a datatype are its predicates and its operations, and thus they are of two kinds]

[5] The white diamond represents the UML aggregation (subobjects containment).

The above assumptions on the items require that

- the "modelling" (that is how the items are associated with the formal models) should be extended to describe how the constituent features of the various kinds correspond to elements/features of the formal models;
- the formal models have (as described by the association has) the formal counterparts of the constituent features;
- the specification language should support the separate specifications of the parts and should offer means to define the constituent features;
- the guidelines should provide help to find the parts and the constituent features of an item.

There are various specification styles. The most quoted distinction is between *property-oriented* (or *axiomatic*) and *constructive* (or *model-oriented*). For what concerns the semantics of property-oriented specifications, the basic way to define it is as follows: "a model belongs to the semantics of a specification if and only if all formulae of the specification are valid on it". The methodological ideas supporting this specification style are:
we describe the item at a certain moment in its development by expressing all its "relevant" properties by sentences provided by the formalism (formulae).

2.2 A General Property-oriented Specification Method (GPSm)

Now we introduce a General Property-oriented Specification method (GPSm) following the conceptual schema introduced in the Sect. 2.1, by specializing and enriching three ingredients (Guidelines, Presentation and Documentation) of a generic method using property-oriented specifications; these modifications are reported and commented below.

Exhaustive Search Guidelines The guidelines for GPSm are as follows. The first steps are to find the parts and to specify them, and to find the constituent features, followed by the search of the properties. GPSm is based on an exhaustive technique for finding all possible relevant properties of an item by examining it from all possible points of view, that is from the viewpoints of all its constituent features. The general idea is to find the properties of a given item by filling the spreadsheet in Fig. 1, whose columns and rows are indexed with the constituent features of that item, say $CF_1, \ldots, CF_n$. A cell with index CF_i:CF_i contains the properties about the constituent feature CF_i, and a cell with index CF_i:CF_j contains the properties expressing the relationships between CF_i and CF_j.

Because the constituent features are of different *kinds*, it is sensible to assume that the properties filling the various cells follow particular schemas depending

	CF_1	.....	CF_n
CF_1			
.....			
CF_n			

Fig. 1. Properties spreadsheet

on the kinds of the two indexing features and on the formulae offered by the chosen specification language. Thus, we need schemas for the cells indexed by:

CF:CF with CF of kind K, for each K, the properties about a constituent feature of kind K considered by itself;

CF:CF′ with CF and CF′ of kind K, for each K, the properties about the relationships between two different constituent features of kind K;

CF:CF′ with CF of kind K and CF′ of kind K′, for each $K \neq K'$, the properties about the relationships between a constituent feature of kind K and another one of kind K′.

Note that the relationships between two different constituent features, say CF and CF′, appear in two different cells (i.e., in CF:CF′ and in CF′:CF), thus we have computed this relationship twice, but in the first case the emphasis/viewpoint is on CF, and in the second case on CF′. The method requires then to check that the contents of the two cells are consistent. In the case of a negative answer, we found some inconsistency in the specification that must be eliminated. Usually, this activity helps detect some problematic or misunderstood aspects of the specified item. In general, depending on the considered particular instance of the GPSm, there may be other overlappings among the content of the cells of the spreadsheet; these repetitions should be made explicit and used for proposing further checks on the consistency of the produced specification.

Note also how the spreadsheet filling technique results in producing a quite structured navigable set of properties, which should be suitable to support evolution. For example, if the ideas about the specified item change, and such changes result in adding/removing constituent features, then the properties may be easily modified, in such case we have just to add/delete some specific rows/columns of the spreadsheet.

Cell Contents Presentation As regards the presentation of the produced specifications, GPSm should provide a nice way to present the various kinds of properties used in the cells of the spreadsheet. The properties found filling the spreadsheet, then need some rearrangement to yield a specification nicer to read (e.g., by dropping the duplicate properties).

Cells Filling Documentation The documentation of the specification process should make recoverable the spreadsheet filling, the justifications of the consistency of the overlapping cells, and a justification for any empty cell.

3 Specification of Simple Systems

3.1 Simple System Item

Following the framework presented in Sect. 2.1 we describe the simple system items structure. Here the word *system* denotes a dynamic entity of whatever kind, and so evolving along the time, without any assumption about other aspects of its behaviour; thus it may be a communicating/nondeterministic/ sequential/... process, a reactive/parallel/concurrent/distributed/... system, but also an agent or an agent system. A *simple system* is a system without any internal components cooperating among them.

In our approach we assume that simple systems are seen formally as *labelled transition systems* (shortly *lts*), see Sect. 1.2. The "modelling" is as follows. The states of an lts modelling a simple system represent the relevant intermediate situations in the life of the system, and each transition $s \xrightarrow{l} s'$ represents the *capability* of the system in the state/situation s of evolving into the state/ situation s'; the label l contains information on the conditions on the external environment for the capability to become effective, and on the transformation induced on this environment by the execution of the transition, i.e., l fully describes the interaction of the system with the external environment during this transition.

To design an effective and simple specification method we assume that the labels have the standard form of a set of *elementary interactions*, where each elementary interaction intuitively corresponds to an elementary (that is not further decomposable) exchange with the external environment. We also assume that the elementary interactions are of different types, and that each type is characterized by a name and by some arguments (elements of some data structures).

To keep the specification level abstract, we do not completely describe the states of the lts modelling the simple system, but we just list what we should be able to observe on them, by means of elementary observations on the states (*state observers*). A state observer is characterized by a name, some arguments and by the observed value (elements of some *data structures*)[6].

[6] If the observed value is a boolean, then a state observer may be specified with a

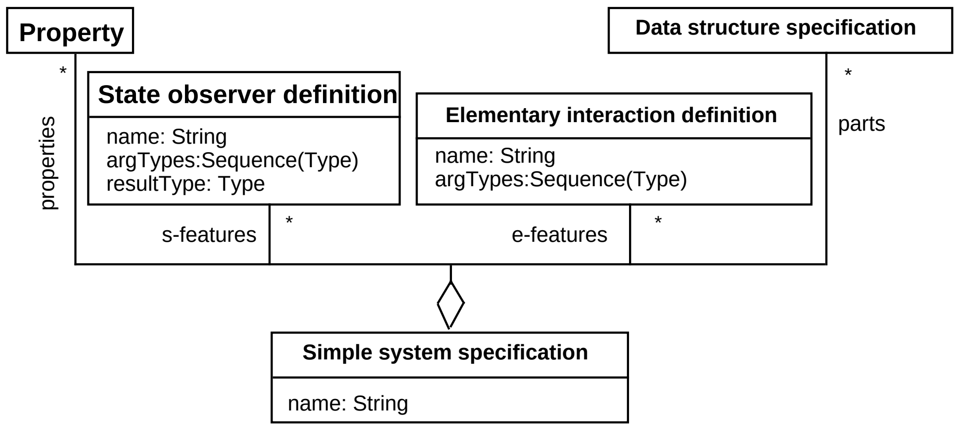


Fig. 2. Simple system specification

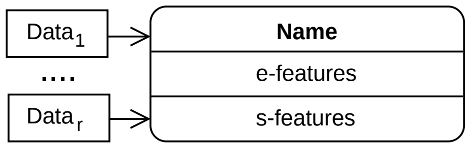
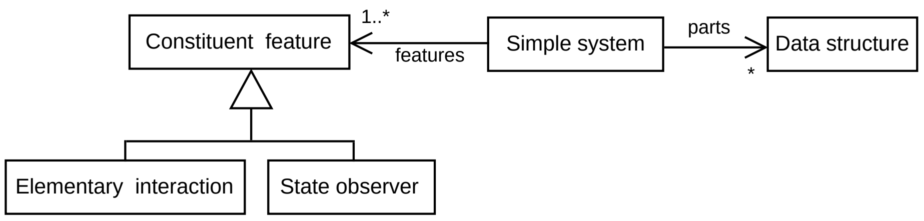


Fig. 3. Visual presentation of a simple system: parts and constituent features

The above considerations lead us to choose as constituent features of the simple systems the elementary interaction types (just *elementary interactions* from now on) and the state observers. Moreover, to define them we use values of various *data structures*; they are the "parts" of the simple systems. We summarize the parts and features of simple systems in the following picture.



3.2 The specification of simple systems

Fig. 2 shows the structure (by means of a UML class diagram) of a specification of a simple system. There Type stands for a type of values defined by one of the parts data structures. Fig. 3 shows how to visually depict the parts and the constituent features of a simple system specification, where $\mathsf{DATA}_1, \ldots, \mathsf{DATA}_r$ are the parts, s-features describe state observers written as name(argTypes): resultType, and e-features describe elementary interactions as NAME(argTypes).

predicate. For simplicity sake (and by lack of space), this case will not be considered in this paper.

All the properties about a simple system correspond to properties on the lts modelling it, and thus on its labels, states and transitions. Recalling our assumptions on the form of the states and labels, these properties may only relate the values observed by the various state observers on a state, express which are the admissible sets of *elementary interactions* building a label, and relate the source state, the label and the target state of a transition. Our method, based on CASL-LTL, offers appropriate ways to present these properties shown below.

Label properties: *ei*(*arg*) **incompatible with** *ei*′(*arg* ′) **if** *cond*(*arg*,*arg* ′)
where *ei* and *ei*′ are two elementary interactions and *cond* is a property of their arguments. It means that under some condition, if the two elementary interactions are different, then they are incompatible, i.e., no label may contain both.[7]

State properties: *cond*
where *cond* is a condition in which some state observers appear. It means that for any state the values returned by those state observers must satisfy this condition.

State properties may include also special atoms, listed below, expressing properties on the *paths* (concatenated sequences of transitions) leaving/ reaching the state, that is on the future/past behaviour of the system from this state.

- **in any case eventually** *eIn* **happen**
  It means that any path starting from the state will contain a transition whose label contains the elementary interaction described by *eIn*.
- **in any case sometime** *eIn* **happened**
  Similarly, it means that any path reaching the state will contain a transition whose label contains the elementary interaction described by *eIn*.

These atoms may also be built by **in one case** (instead of **in any case**, with the meaning there exists at least one path such that ...), or **next** (instead of **eventually**, with the meaning "the label of the first transition of the path contains ..."), or **before** (instead of **sometime**, with the meaning "the label of the last transition of the path contains ...").

Transition properties: *cond*
where *cond* is a condition in which state observers applied to the source and target states (named respectively *source and target state observer*) and atoms of the form "*eIn* **happen**" appear. It means that a transition $tr = x \xrightarrow{l} y$ satisfies *cond*, where source state observers are evaluated on the source state x of *tr*, target state observers are evaluated on the target state y of *tr*, and atoms of the form "*eIn* **happen**" hold iff the elementary interaction described by *eIn* belongs to the label l of *tr*.

The source state observers are denoted by "non primed" *so* identifiers and the target state observers are denoted by "primed" *so* ′ identifiers.

[7] Then, it is not necessary to express that they are different.

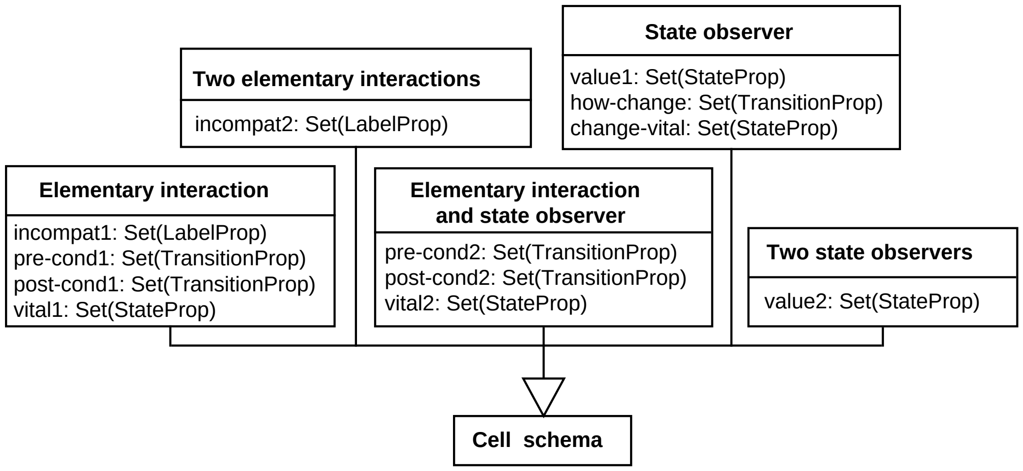

State observer
value1: Set(StateProp)
how-change: Set(TransitionProp)
change-vital: Set(StateProp)
Two elementary interactions
incompat2: Set(LabelProp)
Elementary interaction
incompat1: Set(LabelProp)
pre-cond1: Set(TransitionProp)
post-cond1: Set(TransitionProp)
vital1: Set(StateProp)
Elementary interaction and state observer
pre-cond2: Set(TransitionProp)
post-cond2: Set(TransitionProp)
vital2: Set(StateProp)
Two state observers
value2: Set(StateProp)
Cell schema

Fig. 4. Simple system cell schemas

Since the constituent features of simple systems are of two kinds, elementary interactions and state observers, following Sect. 2.2 we have to consider five kinds of cells:
- properties on an elementary interaction,
- properties on a state observer
- relationship between two elementary interactions,
- relationship between two state observers,
- relationship between an elementary interaction and a state observer.

Fig. 4 describes the schemas for these cells, and we present two schemas in Fig. 5 and 6 (the others are in the Appendix A). There, *arg* stands for generic expressions of the correct types, possibly with free variables, and $cond(arg)$ for a generic condition where the free variables of *arg* may appear.

3.3 Example: Specification of a Lift plant

As an example, we give the specification of a lift plant, considered as a simple system. The lift plant may communicate the status of some of its components by means of sensors (the position of cabin and of the doors at the floors, the working status of the motor), and its components may be influenced by means of orders (open/close the door at a given floor, stopping/making moving up/ down the motor). Moreover, the users may enter or leave the cabin, and a sensor is able to communicate if some user is inside the cabin.

We show the parts and the constituent features of the lift plant in Fig. 7. The elementary interactions (in the upper compartment) model the sensors attached to the plant, and the orders that it can receive, whereas the state observers (in the lower compartment) define the status of its components and how many users are inside its cabin.

incompat1 (label property) Under some conditions, an instantiation of *ei* is incompatible with another elementary interaction, i.e., no label may contain both.

$ei(arg_1)$ **incompatible with** $ei'(arg_2)$ **if** $cond(arg_1, arg_2)$

pre-cond1 (transition property) If the label of a transition contains some instantiation of *ei*, then the source state of the transition must satisfy some condition.

if $ei(arg)$ **happen then** $cond(arg)$
where some source state observers must appear in $cond(arg)$ and the target state observers cannot appear in $cond(arg)$

post-cond1 (transition property) If the label of a transition contains some instantiation of *ei*, then the target state of the transition must satisfy some condition. The condition on the target state may require also the source state to be expressed.

if $ei(arg)$ **happen then** $cond(arg)$
where some target state observers must appear in $cond(arg)$ and the source state observers may appear in $cond(arg)$

vital1 (state property) If a state satisfies some condition, then any path (sequence of transitions) starting from it will eventually contain a transition whose label contains an instantiation of *ei*. Note that in these properties **in any case** may be replaced by **in one case** and **eventually** by **next**.

if $cond(arg)$ **then in any case eventually** $ei(arg)$ **happen**

Fig. 5. Elementary interaction (*ei*) cell schema

value1 (state property) The results of the observation made by *so* on a state must satisfy some conditions.

cond, where *so* must appear in *cond*

how-change (transition property) If the observed value changes during a transition, then some condition on source state, target state, old and new value holds, and some elementary interactions must belong to the transition label.

if $so(arg) = v_1$ **and** $so'(arg) = v_2$ **and** $v_1 \neq v_2$ **then**
    $cond(v_1, v_2, arg)$ **and** $ei_1, \ldots, ei_n$ **happen**

change-vital (state property) If a state satisfies some condition, then the observed value will change in the future. Note that in these properties **in any case** may be replaced by **in one case** and **eventually** by **next**.

if $cond(v_1, v_2, arg)$ **and** $so(arg) = v_1$ **and** $v_1 \neq v_2$ **then**
    in any case eventually $so(arg) = v_2$

Fig. 6. State observer (*so*) cell schema

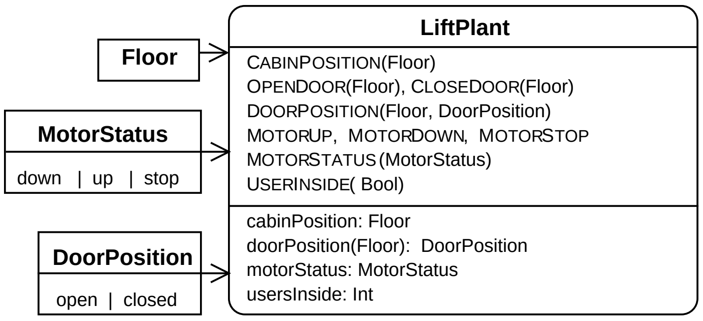


Fig. 7. LiftPlant: parts and constituent features

To define the above constituent features we need some data:
- Floor: the floors among which the cabin is moving (see Sect. 5.2 for its specification),
- MotorStatus: the possible statuses of the motor,
- DoorPosition: the possible positions of the doors at the floors.
MotorStatus and DoorPosition are two simple enumeration data structures, for which we use an ad hoc notation, writing their constructors separated by |.

We followed the cell filling method to find all the relevant properties of the lift plant, see [12] for the complete spreadsheet, but here we show only the content of one cell, precisely the one indexed by MotorUp:MotorUp.

incompat1

MotorUp **incompatible with** MotorStop
MotorUp **incompatible with** MotorDown
No two motor orders may be received simultaneously

pre-cond1

if MotorUp **happen then** $motorStatus = stop$ **and** $cabinPosition \neq top$
The motor up order can be executed only when the motor is stopped and the cabin is not at the top floor

post-cond1

if MotorUp **happen then** $motorStatus' = up$
The motor stop order is always correctly executed

vital1

if $motorStatus = stop$ **and** $cabinPosition \neq top$ **then**
  in one case next MotorUp **happen**
If the motor is stopped and the cabin is not at the top floor, the motor up order can be received

Then, the repeated formulae have been eliminated, after having checked the absence of contradictions, and the others have been slightly rearranged to improve readability. The properties on the motor and the cabin are detailed below, while the others are given in the Appendix B.

– No two motor orders may be received simultaneously.

```
    MotorStop incompatible with MotorDown
    MotorStop incompatible with MotorUp
    MotorDown incompatible with MotorUp
```

– The motor stop order can always be executed, and it is always correctly executed stopping immediately the cabin.

```
if MotorStop happen then
    $motorStatus' = stop$ and $cabinPosition' = cabinPosition$
```

– The motor stop order can always be received.

```
  in one case next MotorStop happen
```

– The motor up/down order can be executed only when the motor is stopped and the cabin is not at the top/ground floor (while the doors may be in any position), and it is always correctly executed.

```
if MotorUp happen then
    $motorStatus = stop$ and $cabinPosition \neq top$ and $motorStatus' = up$
if MotorDown happen then
    $motorStatus = stop$ and $cabinPosition \neq ground$ and $motorStatus' = down$
```

– If the motor is stopped and the cabin is not at the top/ground floor, the motor up/down order can be received.

```
if $motorStatus = stop$ and $cabinPosition \neq top$ then
    in one case next MotorUp happen
if $motorStatus = stop$ and $cabinPosition \neq ground$ then
    in one case next MotorDown happen
```

– The cabin changes position only if the motor is moving up/down.

```
if $cabinPosition \neq cabinPosition'$ then
    ($cabinPosition' = next(cabinPosition)$ and $motorStatus = up$) or
    ($cabinPosition' = previous(cabinPosition)$ and $motorStatus = down$)
```

– If the motor is moving up/down, then the cabin changes position.

```
if $motorStatus = up$ then
    in any case next $cabinPosition = next(cabinPosition)$
if $motorStatus = down$ then
    in any case next $cabinPosition = previous(cabinPosition)$
```

– The motor changes its status only when it receives the corresponding order.

```
if $motorStatus = stop$ and $motorStatus' = up$ then MotorUp happen
if $motorStatus = stop$ and $motorStatus' = down$ then MotorDown happen
if $motorStatus \neq stop$ and $motorStatus' = stop$ then MotorStop happen
```

The complete specification of the lift plant given following our method (see also Appendix B) may seem long, but we think that it is quite complete and it shows all relevant information to build the software for handling it. For example, such specification makes clear that
– sensors never break down,
– motor and doors cannot change status by themselves as a result of some failure, and
– the plant takes care of some security checks, such as to avoid that the motor goes up/down when the cabin is at the top/ground floor.

Here we present the CASL-LTL [27] corresponding version of the specification of simple systems produced following our method introduced in Sect. 3.2. Let *Spec* be a specification of simple systems having the form described in Fig. 2, and assume that

- *Spec*.parts = $\{ds_1, \ldots, ds_j\}$ are the parts, and $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ the corresponding CASL-LTL specifications;
- *Spec*.e-features = $\{ei_1, \ldots, ei_n\}$ are the elementary interactions;
- *Spec*.s-features = $\{so_1, \ldots, so_m\}$ are the state observers.

Below we give the CASL-LTL specification corresponding to *Spec*. Notice that the constructors and the operations may be partial, and this is denoted by a '?', e.g., "so_i.name : $St \times so_i$.argTypes $\rightarrow ?$ so_i.resType".

spec ELEMINTER =
  free type *ElemInter* ::=
    ei_1.name(ei_1.argTypes) | ... | ei_n.name(ei_n.argTypes)
spec *Spec*.name =
  FINITESET[ELEMINTER] **and** DS_1 **and** ... **and** DS_j **then**
  dsort *St* **label** *FinSet*[*ElemInter*]
  ops so_1.name : $St \times so_1$.argTypes $\rightarrow$ so_1.resType
    ...
    so_m.name : $St \times so_m$.argTypes $\rightarrow$ so_m.resType
  axioms
    formulae corresponding to the cell fillings, defined below case by case

Label property: eIn_1 **incompatible with** eIn_2 **if** *cond*
the corresponding formula is
$$\neg\,(eIn_1 = eIn_2) \;\wedge\; cond \;\wedge\; S \xrightarrow{L} S' \;\Rightarrow\; \neg\,(eIn_1 \in L \;\wedge\; eIn_2 \in L)$$
State property: *cond*
the corresponding formula is obtained by adding S (a variable of sort *St*) as extra argument to each state observer appearing in *cond*, and by replacing the special temporal combinators as follows:

in any case ...	$in\_any\_case(S, \ldots)$
in one case ...	$in\_one\_case(S, \ldots)$
eventually *eIn* **happen**	$eventually\ < L \bullet eIn \in L >$
next *eIn* **happen**	$next\ < L \bullet eIn \in L >$
sometime *eIn* **happened**	$sometimes\ < L \bullet eIn \in L >$
before *eIn* **happened**	$before\ < L \bullet eIn \in L >$

Transition property: *cond*
the corresponding formula is $S \xrightarrow{L} S' \;\Rightarrow\; cond'$, where $cond'$ is obtained

from *cond* by adding S as an extra argument to each source state observer, by adding S' as an extra argument to each target state observer, and by replacing each atom of the form "*eIn* **happen**" with "$eIn \in L$".

The Casl-Ltl version of the specification of the lift plant given in Sect. 3.3 is available in [12].

4 Specification of Structured Systems

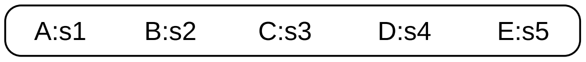
4.1 Structured System Items

A *structured system* item is a specialization of the simple dynamic system of Sect. 3; indeed it is a simple system made by several other dynamic systems, its *subsystems*, which are either simple or in turn structured. We assume that each subsystem is uniquely identified by some identity. A situation during the life of a structured system is fully characterized by the situations of its subsystems, and its (global) moves just consist of the simultaneous executions of (local) moves of some of its subsystems.

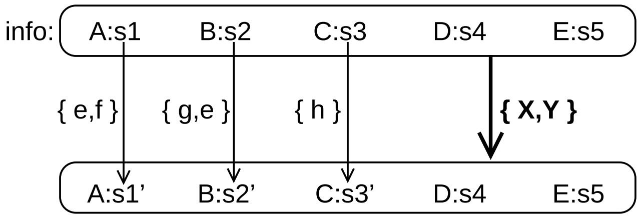
The specification method for structured systems that we present here is a specialization of that for simple systems (see Sect. 3). Thus also structured systems will be modelled by labelled transition systems (lts); but in this case their states will be sets of states of those lts's modelling the subsystems, and their transitions will correspond to simultaneous executions of sets of subsystems transitions (the latter are named their *components*). To represent which are their composing subtransitions, we need to enrich the labelled transitions with an extra part containing such *information*. It is not appropriate to only extend the labels of the transitions with the information about the subsystems moves. Indeed, *labels* should model only the system interaction with the outside world, and in many cases the subsystems moves are completely transparent to outside, as, e.g., two subsystems exchanging a message between themselves. Thus, to describe a given global transition we both need its label (that is a set of elementary interactions visible from outside) and its information part (on the subsystem moves that may not all be visible from outside). For simplicity sake we do not consider here the case of subsystems that may be created and destroyed dynamically, but there are no technical problems to handle them.

Technically, it means that to model structured systems we use *generalized lts*, that are lts specialized by adding an information part to each transition. Thus a generalized lts is a 4-uple $(State, Label, Info, \rightarrow)$, where $\rightarrow \subseteq Info \times State \times Label \times State$, and *Info* is the set of the additional information at-

a sample state of SS (s1, ..., s5 are respectively the states of A, ..., E)



a sample transition of SS (global transition/move) labelled by {X, Y}



- **its composing local transitions/moves**

$$\mathsf{s1} \xrightarrow{\{\,e,f\,\}} \mathsf{s1'} \quad \mathsf{s2} \xrightarrow{\{\,g,e\}} \mathsf{s2'} \quad \mathsf{s3} \xrightarrow{\{\,h\,\}} \mathsf{s3'}$$

- **its local interactions** (the subsystems D and E do not take part in the global move) A.e A.f B.g B.e C.h
- **its (global) elementary interactions** (i.e., of SS towards the outside resulting from the subsystem moves) X Y
- **its (additional) information** info = { (A, e), (A,f), (B,g), (B,e), (C,h) }

Fig. 8. Example of a structured system SS, with five subsystems, A, B, C, D and E

tached to the transitions. A generic transition is usually written *info* : $x \xrightarrow{l} y$. The additional information for the generalized lts modelling the structured systems, which must represent the composing subtransitions, will be sets of pairs made by a subsystem identity (the subsystem performing the subtransition) and by an elementary interaction (belonging to the label of the subtransition). We name these pairs *local elementary interactions*, shortly *local interactions* from now on. We exemplify the concepts introduced so far in Fig. 8.

To take into account the role played by the subsystems in the moves of the structured systems, we consider also the local interactions as their constituent features. Structured systems have also a new kind of parts, the composing subsytems, which may be either simple or in turn structured. Structured systems have special state observers returning the states of the subsystems, which are denoted by the subsystem identities themselves (we do not need to declare them, since they are implicitly determined by the subsystem declarations.) Notice that, however, we need also other state observers. Indeed, our specifications are usually at a quite abstract level and we may want to observe something on the structured system states without knowing which subsystems (and in which way) contribute to this observation. An example may be an observer checking if there is an error in the system, when we do not know anything about the error situations of the single subsystems.

We summarize the parts and the constituent features of structured systems in Fig. 9.

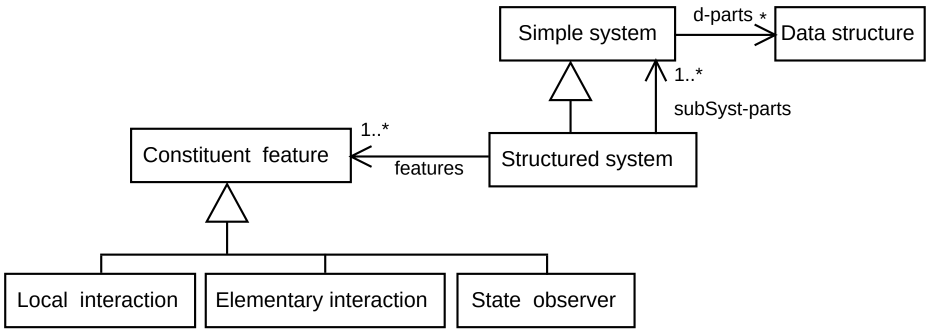


Fig. 9. Structured system item

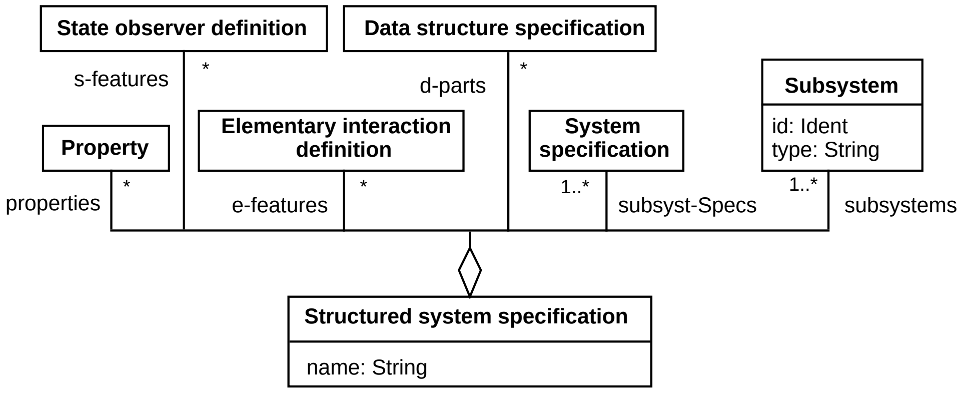


Fig. 10. Structured system specification

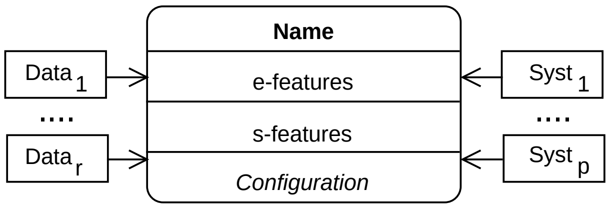
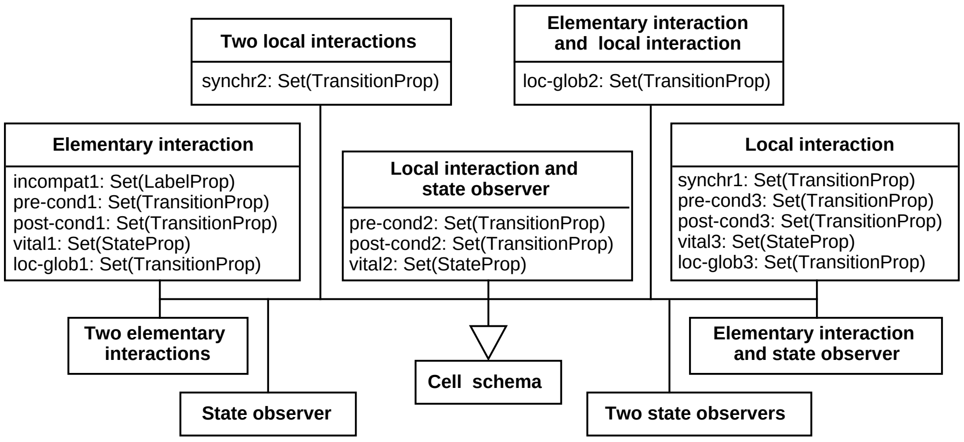


Fig. 11. Visual presentation of a structured system: parts and constituent features

4.2 Specification of structured systems

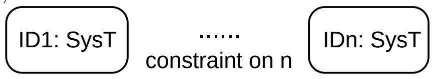
We assume that a structured system may have many subsystems of the same type (i.e., whose specification is the same), and that they are identified by elements of a special data structure IDENT (standard identifiers). Thus to specify the subsystem parts it is sufficient to give the subsystem specifications, and for any subsystem its identity and its *type*, i.e., the name of its specification. The local interactions are implicitly determined after we have given the subsystems, and so they do not need to be explicitly specified. The form of a specification of a structured system is then summarized in Fig. 10.

Fig. 11 presents how to visually depict the parts and the constituent features of a structured system specification. In this picture Syst_1, ..., Syst_p are the names of the subsystem specifications, given apart, and Configuration is a visual



Two local interactions
synchr2: Set(TransitionProp)
Elementary interaction and local interaction
loc-glob2: Set(TransitionProp)
Elementary interaction
incompat1: Set(LabelProp)
pre-cond1: Set(TransitionProp)
post-cond1: Set(TransitionProp)
vital1: Set(StateProp)
loc-glob1: Set(TransitionProp)
Local interaction and state observer
pre-cond2: Set(TransitionProp)
post-cond2: Set(TransitionProp)
vital2: Set(StateProp)
Local interaction
synchr1: Set(TransitionProp)
pre-cond3: Set(TransitionProp)
post-cond3: Set(TransitionProp)
vital3: Set(StateProp)
loc-glob3: Set(TransitionProp)
Two elementary interactions
State observer
Cell schema
Elementary interaction and state observer
Two state observers

Fig. 12. Structured system cell schemas

presentation of which are the subsystems[8]. A subsystem is represented by a rounded box containing its identity and type, that is the name of the corresponding specification. We use the notation 

to represent a set of subsystems of type SysT made by n elements with n satisfying some constraint. In the particular case where there is just a unique element of a type we can drop the subsystem identity and write only the type name; thus the subsystem will be named as the type.

Structured systems have a new kind of constituent features, the local interactions, so we have new types of cells to be filled; moreover local interactions should be considered also when defining the schemas for the cells already used for simple systems. The state observers corresponding to subsystem states should be considered as the others, with the corresponding cells.

To model structured systems, we upgraded lts's to generalized lts, which differ for the additional information part of the transitions (the set of the local interactions). Now, we consequently upgrade the properties on the transitions (see Sect. 3.2) with new atoms "*locIn* **happen**" (where *locIn* is a local interaction) which express that *locIn* belongs to the set of the local interactions of that transition. More precisely, *locIn* **happen** holds on a transition of a generalized lts "$inf\colon x \xrightarrow{l} y$" iff $locIn \in inf$. The new properties will allow us to take into account the local interactions when expressing the properties of the various cells.

In Fig. 12 we present the schemas for the cells needed to specify a structured system; there the undetailed "boxes" refer to Fig. 4 of Sect. 3.2, as well as the slots that are not redefined here. Clearly, for the parts already defined in Sect. 3.2, here we must consider also the local interactions together with

[8] As in Fig. 3 of Sect. 3.2 DATA_i are the parts, s-features are state observers descriptions, and e-features elementary interactions descriptions.

synchr1 (transition property) Under some condition, an instantiation of *sid.ei* is/ is not synchronized (i.e., executed simultaneously) with another local interaction, i.e., one is a component of a global transition iff the other also is/is not so; clearly the two local interactions are performed by different subsystems.

if *sid.ei*(*arg*) **happen and** *cond*(*arg*,*locIn*) **then** *locIn* **happen**
or
if *sid.ei*(*arg*) **happen and** *cond*(*arg*,*locIn*) **then not** *locIn* **happen**

loc-glob3 (transition property) If an instantiation of *sid.ei* is a component of a global transition, then, under some condition, the label of this global transition must contain some elementary interaction, or vice versa.

if *sid.ei*(*arg*) **happen and** *cond*(*arg*,*eIn*) **then** *eIn* **happen**
or
if *eIn* **happen and** *cond*(*arg*,*eIn*) **then** *sid.ei*(*arg*) **happen**

pre-cond3*, *post-cond3*, *vital3 defined as the homonymous slots for simple system, but where the elementary interaction is replaced by the local interaction.

Fig. 13. Local interaction (*sid.ei*) cell schema

synchr2 (transition property) Under some condition, an instantiation of $sid_1.ei_1$ is/is not synchronized with an instantiation of $sid_2.ei_2$, i.e., one is a component of a global transition iff the other also is/is not so; clearly the two instantiations are performed by different subsystems.

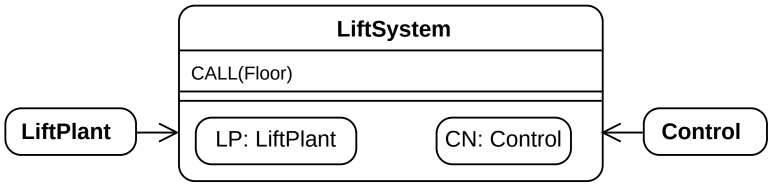
if $sid_1.ei_1(arg_1)$ **happen and** $cond(arg_1,arg_2)$ **then** $sid_2.ei_2(arg_2)$ **happen**
or
if $sid_1.ei_1(arg_1)$ **happen and** $cond(arg_1,arg_2)$ **then**
  not $sid_2.ei_2(arg_2)$ **happen**

Fig. 14. Two local interactions ($sid_1.ei_1$,$sid_2.ei_2$) cell schema

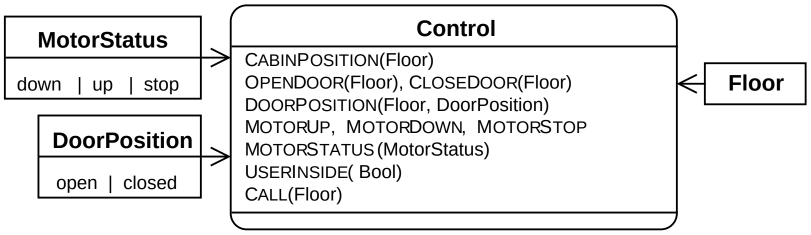
elementary interactions in the state and transition properties. The schemas for the new cells are reported in Fig. 13 and 14, and in Appendix C.

4.3 Example: Specification of a Lift System

The lift system consists of the lift plant, see Sect. 3.3 and of the automated software controller; and, thus it is a structured system. Here we use our method to express its relevant properties, which are mainly about how its subparts influence each other. The produced specification may be considered as a precise definition of the requirements on the controller, stating precisely how it will affect and interact with the lift plant.



The above picture shows the parts and the constituent features of the lift system. The subsystems are the plant and the controller, and both of them are simple systems; whereas the used data structures are those of the subsystems and so we do not repeat them. The specification of the lift plant, LiftPlant, has been given in Sect. 3.3 and that of the controller is here; the elementary interaction Call corresponds to receive a call for a floor.


MotorStatus
down | up | stop
DoorPosition
open | closed
Control
CABINPOSITION(Floor)
OPENDOOR(Floor), CLOSEDOOR(Floor)
DOORPOSITION(Floor, DoorPosition)
MOTORUP, MOTORDOWN, MOTORSTOP
MOTORSTATUS (MotorStatus)
USERINSIDE(Bool)
CALL(Floor)
Floor

Notice that this specification has no properties, because in this case the requirements concern only the effects of the controller on the lift plant, and not its precise behaviour.

The lift system interacts with its outside world only by receiving from the users calls for a given floor; thus it has a unique elementary interaction Call. No state observer different from those observing the states of the two subsystems is needed, and so the other compartment is empty. We followed the cell filling method to find all the relevant properties of the lift system reported below, see [12] for the complete spreadsheet.

The calls for a floor are received by the controller.
Call(f) **happen iff** *CN*.Call(f) **happen**
Any received call for a given floor will be eventually satisfied in any case
if Call(f) **happen then in any case eventually**
$LP.cabinPosition = f$ **and** $LP.motorStatus = stop$ **and**
$LP.doorPosition(f) = open$
In any case eventually the lift system will be able to receive a call for a given floor
in any case eventually in one case Call(f) **happen**

All the remaining properties state that local interactions with the same name of the lift plant and of the controller are synchronized. We give just an example of such properties, the one concerning stopping the motor.

LP.MotorStop **happen iff** *CN*.MotorStop **happen**

4.4 CASL-LTL *View*

Here we present the CASL-LTL corresponding version of our specification of structured systems introduced before in Sect. 4.2. The only difference with the case of the simple system of Sect. 3.4 is that now we use generalized lts, however CASL-LTL offers also a special construct to declare that three sorts correspond to the states, the labels and the additional information of a generalized lts together with a standard arrow predicate corresponding to the transition relation.

dsort *St* **label** *Lab* **info** *I* stands for **sorts** *St*, *Lab*, *I*
pred $\_\_ : \_\_ \xrightarrow{\_\_} \_\_ : I \times St \times Lab \times St$

Let *Spec* be a specification of structured systems having the form described in Fig. 10, and assume that

- *Spec*.d-parts = $\{ds_1, \ldots, ds_j\}$ are the data parts, and $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ the corresponding CASL-LTL specifications;
- *Spec*.subsyst-Specs = $\{ssp_1, \ldots, ssp_k\}$ are the subsystem specifications, $\mathrm{SSP}_1, \ldots, \mathrm{SSP}_k$ are the corresponding CASL-LTL specifications, and $\mathrm{ELEMINTER}_1, \ldots, \mathrm{ELEMINTER}_k$ are the specifications of their elementary interactions;
- *Spec*.e-features = $\{ei_1, \ldots, ei_n\}$ are the (global) elementary interactions;
- *Spec*.s-features = $\{so_1, \ldots, so_m\}$ are the state observers;
- *Spec*.subsystems = $\{ss_1, \ldots, ss_r\}$ are the subsystems.

Below we give the CASL-LTL specification corresponding to *Spec*, where ELEMINTER has been defined as in Sect. 3.4.

```
spec LOCALINTER =
    ELEMINTER_1 and ... and ELEMINTER_k and IDENT then
    free type SubElemInter ::= _(ElemInter_1) | ... | _(ElemInter_k)
    free type LocalInter ::= < _, _ > (Ident, SubElemInter)

spec Spec.name =
    FINITESET[ELEMINTER] and FINITESET[LOCALINTER] and
    DS_1 and ... and DS_j and SSP_1 and ... and SSP_k then
    dsort St label FinSet[ElemInter] info FinSet[LocalInter]
    ops so_1.name : St × so_1.argTypes → so_1.resType  %% state observers
          ...
          so_m.name : St × so_m.argTypes → so_m.resType
          ss_1.id : St → ss_1.type  %% observers of the subsystem states
          ...
          ss_r.id : St → ss_r.type
    axioms
          formulae corresponding to the cell fillings, see below
```

For the properties on structured systems we have used a new kind of transition properties, and so here we give how to transform them in CASL-LTL. Similarly to what was done in Sect. 3.4, a transition property *cond* is transformed into $inf\colon S \xrightarrow{L} S' \ \Rightarrow\ cond'$, where *cond'* is obtained from *cond* by adding S as an extra argument to each source state observer, by adding S' as an extra argument to each target state observer, and by replacing each atom of the form "*eIn* **happen**" with $eIn \in L$, and each atom of the form "*locIn* **happen**" with $locIn \in inf$.

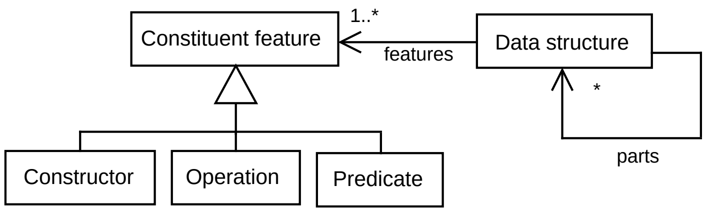
The CASL-LTL version of the specification of the lift system given in Sect. 4.3 can be found in [12].

5 Specification of Data Structures

5.1 Data Structure Items and Specifications

A *data structure* consists of a set of values, some constructors for denoting them, some operations and predicates. The constructors, the operations and the predicates may also have arguments of other types, thus a data structure may have other data structures as subparts. Constructors and operations may be total (always defined), or partial. Constructors and operations may be constants (considered as 0-ary operations), and constants are always total.

In our setting the data structures are seen formally as *many sorted algebras*, or *structures*, and the modelling is quite trivial: the carriers model the set of values, and functions (of course of different kinds) model constructors, operations and predicates. Thus, data structures may be characterized by their constructors, operations and predicates, and so they will have three corresponding kinds of constituent features. Below we summarize the constituent features and parts of the data structures.



The specification method for data structures we propose is a specialization of GPSm introduced in Sect. 2.2. The form of the resulting specifications is reported in Fig. 15. The parts and the constituent features are visually presented as shown in Fig. 16, where $\mathsf{DATA}_1, \ldots, \mathsf{DATA}_r$ are the parts, c-features describe constructors as name(argTypes) (name(argTypes)? if partial), p-features

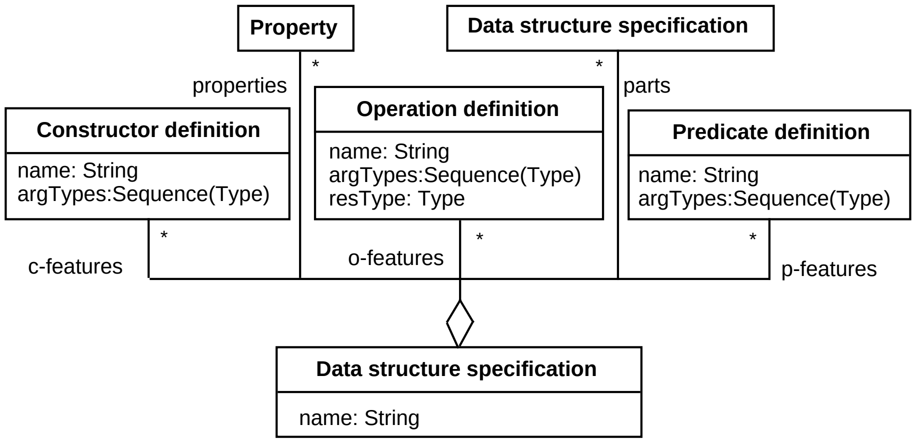


Fig. 15. Data Structure Specification

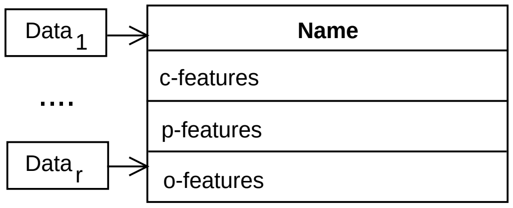


Fig. 16. Visual presentation of a data structure: parts and constituent features

describe predicates as name(argTypes), and o-features describe operations as name(argTypes): resType (name(argTypes):? resType if partial).

The properties correspond to first-order formulae and are determined using the cell filling approach. The constituents of data structures are of three kinds, constructors, predicates and operations, and so we have to consider nine kinds of cells; and we present their schemas in Fig. 17, and the details in Fig. 18 and 19, and in Appendix D. Let us note that, as regards constructors and operations, the properties to be described should in particular address both definedness and the values denoted/returned.

In CASL, "=" is the *strong equality*, characterized by the fact that $t = t'$ iff either both terms are defined and denote the same value or both are undefined. Thus a property $t = t'$ in the case t is defined implicitly requires also that t' must be defined. In order to avoid the undefined case, the premises of many properties used in the cell schemas require the definedness of all the elements involved in the property, thus their form is

if ($\textbf{and}_{t \text{ is a term appearing in } cond}\ \mathbf{def}(t)$) **then** *cond*.

Because properties having the above form may be quite long, they are usually written in a more compact way as:

when all defined *cond*

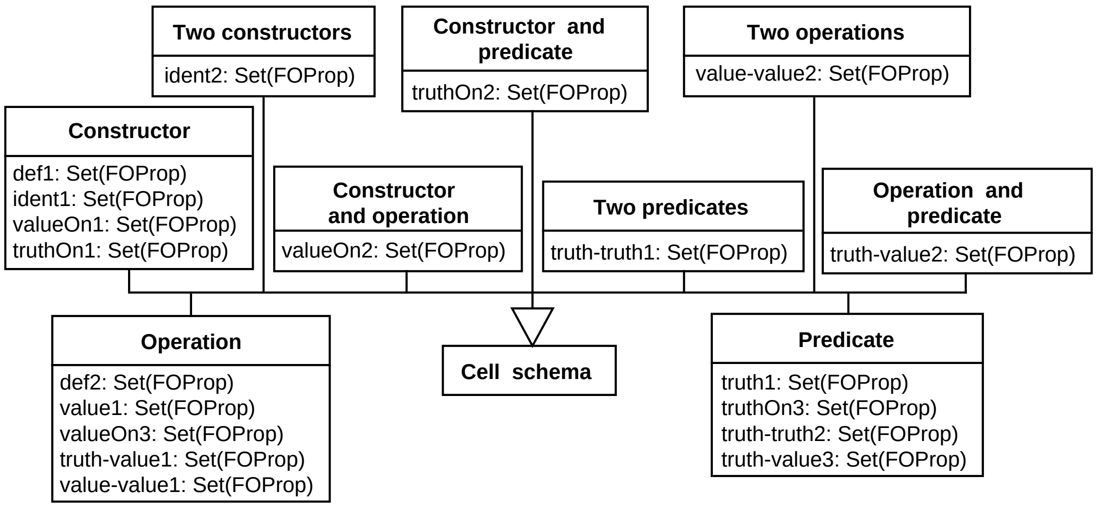


Fig. 17. Data structure: cell schemas

ident2 Values represented by con_1 are/are not identified with values represented by con_2

when all defined *cond*

where *cond* includes atoms of the form $con_1(arg_1) = con_2(arg_2)$ or **not** $con_1(arg_1) = con_2(arg_2)$

Fig. 18. Two constructors (con_1,con_2) cell schema

truth-value2 Relationships between the truth of *pr* and the values returned by *op*

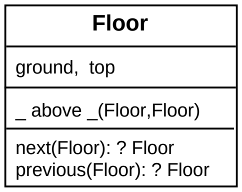
when all defined *cond*

where *cond* includes atoms of the form $pr(arg_1)$ and of the form $op(arg_2)$

Fig. 19. Operation (*op*) and predicate (*pr*) cell schema

5.2 Example: Specification of Floor

We specify the Floor data structure used in the lift related examples (Sect. 3.3 and 4.3).



The above picture shows that the constructors are *ground* and *top*, the predicate is *above*, and the (partial) operations are *next* and *previous*. Moreover, Floor does not use any other data structure, thus there are no parts. The properties given below were worked out using our cell filling approach, then redundant properties were removed and the result was reorganized, see [12] for the complete presentation of the cell contents.

- There exists a ground and a top floor, and they are different.
  $\mathit{ground} \neq \mathit{top}$
- *above* is total order over the floors with *top* as maximum and *ground* as minimum.
  top above ground
  not exists f **s.t.** *ground above* f **or** f *above top*
  $f_1 = f_2$ **or** f_1 *above* f_2 **or** f_2 *above* f_1
  not f *above* f
  if f_1 *above* f_2 **then not** f_2 *above* f_1
  if f_1 *above* f_2 **and** f_2 *above* f_3 **then** f_1 *above* f_3
- *next* returns the floor immediately above a given one, if it exists, i.e., there is no floor between f and $\mathit{next}(f)$.
  not def$(\mathit{next}(\mathit{top}))$
  def$(\mathit{next}(\mathit{ground}))$ **and** $\mathit{next}(\mathit{ground}) \neq \mathit{ground}$
  def$(\mathit{next}(f))$ **iff** *top above* f
  when all defined $\mathit{next}(f)$ *above* f **and**
  (**not exists** $f_1 \bullet (\mathit{next}(f)$ *above* f_1 **and** f_1 *above* $f)$) **and**
  $\mathit{next}(\mathit{previous}(f)) = \mathit{previous}(\mathit{next}(f)) = f$
- Properties on *previous* are similar to those of *next*, and are given in Appendix E.

5.3 CASL *View*

Here we present the CASL [9] corresponding version of our specification of data structures introduced in Sect. 5.1.

Let *Spec* be a specification of data structures having the form described in Fig. 15, and assume that

- *Spec*.parts $= \{ds_1, \ldots, ds_j\}$ are the parts, and $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ the corresponding CASL specifications;
- *Spec*.c-features $= \{con_1, \ldots, con_n\}$ are the constructors;
- *Spec*.o-features $= \{op_1, \ldots, op_m\}$ are the operations;
- *Spec*.p-features $= \{pr_1, \ldots, pr_p\}$ are the predicates.

Below we give the CASL specification corresponding to *Spec* (some constructors and operations may be partial, which is denoted by adding a '?', cf. Sect. 1.2).

[9] Here we do not need to use the CASL-LTL extension.

spec *Spec*.name =
    DS$_1$ **and** ... **and** DS$_j$ **then**
    type *Spec*.name ::= *con$_1$*.name(*con$_1$*.argTypes) | ... | *con$_n$*.name(*con$_n$*.argTypes)
    ops *op$_1$*.name : *op$_1$*.argTypes → *op$_1$*.resType
        ...
        op$_m$.name : *op$_m$*.argTypes → *op$_m$*.resType
    preds *pr$_1$*.name : *pr$_1$*.argTypes
        ...
        pr$_p$.name : *pr$_p$*.argTypes
    axioms
        formulae corresponding to the cell fillings

The Casl formulae corresponding to the cell fillings for data structures are quite obvious, since their abstract structure is the same, the only difference is in the concrete syntax.

The Casl version of the specification of the floor data structure given in Sect. 5.2 can be found in [12].

6 Conclusions, Related and Further Work

In this paper we have presented an attempt to design a basis for software development methods that are formally grounded, shortly FG, by giving the FG methods for the basic modelling/specification tasks. By *formally grounded* we mean methods

– which have all the good properties of those commonly used (using friendly notation based on simple intuitive visual metaphors, easy to understand and to learn, relevant for real applications, ...),
– but where any used model/specification has a direct formal semantics (not to be shown to the users) based on well defined underlying formal models,
– and also where the pragmatic characteristics of the first point have been determined by the underlying formal foundations.

Notice that by formally-grounded we intend more than just to have a formal semantics. We mean that the underlying concepts are reflected in the method and used as such (although they are distilled to the potential user through precise methodological guidelines and nice visual notations).

As a formal basis for grounding our methods we have chosen the algebraic specification language Casl [24] and its extension for behavioural/dynamic specifications Casl-Ltl [27]. Reasons for this choice are that from works on algebraic specifications, "foundations have been laid down for a neat formal treatment of requirement and design specifications, including neat semantics"

[6]. Then, the CASL language, resulting from a common effort of the scientific community in this area, "encompasses all previously designed algebraic specification languages, has a clean, perfectly designed semantics" [6].

Our intention was to investigate if this idea is feasible, and so we proceeded in a quite systematic way, so as to handle any possible case and to exhibit how to produce the specifications (we do not just to give some sample FG specifications). Our previous experiences suggested that the various activities in a development process are based on the "building-bricks" tasks of specifying/modelling software artifacts of different nature at different levels of abstractions. So we designed methods for specifying/modelling data structures, simple systems (just dynamic interacting entities in isolation, e.g., sequential processes), and structured systems (communities of mutually interacting entities, simple or in turn structured). We also addressed two kinds of specifications, the more abstract property-oriented ones, presented here, and the more concrete constructive ones presented in [15]. To present our specification methods for these different cases, we have followed the conceptual schema of [3], where the distinction between the chosen specification formalism and all the other ingredients are explicitly presented.

To try to evaluate the strength and the applicability of our proposal we have used three of the M. Jackson problem frames [21] as a kind of benchmark (see [14, 15]). The result of this experiment is that all the specifications required to cope with these problem frames (i.e., specifications concerning the problem domain, the requirements and the design) can be given using our method. For each case, all relevant aspects of the frame may be satisfactorily expressed, through user friendly presentations, while the corresponding underlying formal specifications, suitable for possible formal analysis, are available.

We have made another experiment concerning the specification of the requirements for an application for running Internet based lotteries [15]. The same case study has been used by one of the authors to present a UML-based *precise* method [5], quite different from the RUP [26].

In this paper we propose some methods grounded in a formal notation, with the aim of having some of the benefits of using formal methods available within practical usual development methods, trying to reduce the impact of all the well-know disadvantages of their use (as exotic notation, and hard underlying formal concepts based on complex mathematics). This approach is quite new and so there are not, for what we know, similar approaches, except for works by the authors, as the JTN (a formally grounded visual notation for the design of Java targeted applications see [16]); see also [6] for further considerations on our view of the relationships between formal and practical used methods. However, we would like to mention works that address issues complementary to ours, e.g., how to write readable specifications in CASL [32], avoiding semantic pitfalls (also addressed in the CASL reference manual [9]), how to use/combine observability concepts for writing specifications [8], guidelines for the iterative and incremental development of specification [10]. As regards the combination of static and dynamic views in a specification, [18] distinguishes several layers, [13] elaborate on how to combine these views for complex systems.

Most of the work in the literature concerning the combination of formal methods with practical ones follows different approaches. A lot of approaches match the following pattern "take some practical more or less precise notation, e.g., UML, select a subset (usually small) of it, give this subset a formal semantics either directly or by translation into some formal notation". In many cases the final aim is to allow to use the good verification/validation tools associated with the chosen formalism. For example, for what concerns UML this pattern may be found instantiated with a large variety of formalism (we just cite one nice paradigmatic example [22], for more references look at [30]). A more recent pattern is the following "select a subset of the specifications given using some formalism and show that they correspond/can be presented as particular UML diagrams" (e.g., see [7]). The main differences of these approaches with ours is that they usually handle a particular kind of specifications applicable to particular problems to be able to use tools to automatically do some checks on the specifications.

M. Heisel introduced the interesting concept of specification development *agendas* that provide a list of tasks to be done, together with validation conditions to be used to check what is achieved [19].

In [20], M. Heisel and J. Souquières use this agenda concept together with a requirement elicitation approach, using Z for formal specification, and they also use a lift example. Their approach requires to (i) introduce the domain vocabulary, (ii) identify relevant operations and events, (iii) state facts, assumptions and requirements, (iv) then formalise them as constraints on events traces. Their approach shows how to incrementally develop the specification, by taking into account the requirements one by one. We can observe that the properties they express come from the facts, assumptions and requirements, that may be described at a detailed level, while in the work we present here guidelines are provided to find the relevant properties through the tableau cell description. We also introduce some help in structuration by distinguishing simple and structured systems, and datatypes.

On another side, many aspects of our FG specifications methods are quite general and not strictly related to CASL, CASL-LTL and in general to the algebraic specifications, as, for instance, the general GPSm method for property oriented specifications. So we would like to investigate whether it is possible to build other FG specification methods starting from different formal basis. We think that this can be done if we choose some formalism based on other formal models as stream processing functions (instead of labelled transition systems) as the one in [11]. For what concerns the general GPSm method for property oriented specifications, we are working to see if it can be to adapted also to produce UML models, or models on a (quite substantial) UML subset to which a formal semantics may be given.

Clearly, to be able to promote the use our proposed FG methods we need to develop supporting software tools. Such tools should consist of a graphical editor helping to prepare the visual specifications, of a type checker signalling all static errors, and of wizards implementing the proposed guidelines, this will be really important for the GPSm method, and obviously of a part offering the possibility to generate the underlying corresponding formal specifications. Such tools do not pose any particular problem, and can be developed using the current technology, only given the necessary material resources. Instead, we do not plan the development of any specific tool for verification and or validation, the existing tools for the underlying specifications may be used.

Acknowledgements We would like to thank Jeanine Souquières and Fernando Orejas for their careful reading and fruitful remarks.

References

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL : the Common Algebraic Specification Language. *T.C.S.*, 286(2):153–196, 2002.

[2] E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification.* Springer Verlag, 1999.

[3] E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2):3–34, 2000.

[4] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12):831–879, 2001.

[5] E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. SEKE 2002*, pages 143–150, New York, Usa, 2002. ACM Press.

[6] 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, 10th Anniversary Colloquium of UNU/IIST the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers.*, number 2757 in Lecture Notes in Computer Science, pages 132 – 150. Springer Verlag, Berlin, 2003.

[7] V. D. Bianco, L. Lavazza, M. Mauri, and G. Occorso. Towards UML-based Formal Specifications of Component Based Real-Time Software. In

M.Pezzè, editor, *Proc. FASE 2003*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003.

[8] M. Bidoit, R. Hennicker, and A. Kurz. On the Integration of Observability and Reachability Concepts. In *Proc. FOSSACS'2002*, Lecture Notes in Computer Science, pages 21–36. Springer Verlag, Berlin, 2003.

[9] M. Bidoit and P. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language.* Number 2900 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[10] B. Blanc. *Prise en compte de principes architecturaux lors de la formalisation des besoins - Proposition d'une extension en CASL et d'un guide méthodologique associé.* PhD thesis, 2002.

[11] M. Broy and G. Stefanescu. The Algebra of Stream Processing Functions. *T.C.S.*, 258(1/2):99–129, 2001.

[12] C. Choppy and G. Reggio. A Formally Grounded Specification of a Lift System. WEB document, available at `http://www.disi.unige.it/person/ReggioG/FG/LiftSystem.html`, 2004.

[13] C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In T. Rus, editor, *Proc. Amast 2000*, number 1816 in Lecture Notes in Computer Science, pages 165–180. Springer-Verlag, Berlin, 2000.

[14] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 14th International Workshop WADT'99*, number 1827 in Lecture Notes in Computer Science, pages 106–125. Springer Verlag, Berlin, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[15] C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.

[16] E. Coscia and G. Reggio. JTN: A Java-targeted Graphic Formal Notation for Reactive and Concurrent Systems. In F. J.-P., editor, *Proc. FASE 99*, number 1577 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.

[17] G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.

[18] H. Ehrig and F.Orejas. Integration and Classification of Data Type and Process Specification Techniques. Technical report, TU Berlin, 1998.

[19] M. Heisel. Agendas – A Concept to Guide Software Development Activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32. Chapman & Hall London, 1998.

[20] M. Heisel and J. Souquières. De l'élicitation des besoins à la spécification formelle. *Technique et science informatiques*, 18(7):777–801, 1999.

[21] M. Jackson. *Problem Frames: Analyzing and Structuring Software Devel-*

opment Problems. Addison-Wesley, 2001.

[22] J. Lillius and I. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *Proc. UML'99*, number 1723 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.

[23] P. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 115–137. Springer Verlag, Berlin, 1997.

[24] P. Mosses, editor. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. Number 2960 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[25] OMG. *UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.

[26] Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Tp 165, 8/01, 2001.

[27] 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, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/ person/ReggioG/ReggioEtAll03b.ps`.

[28] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[29] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[30] G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. W. (editors). Dynamic Behaviour in UML Models: Semantic Questions. Technical report, Ludwig-Maximilian University, Munich (Germany), 2000. `http://www.disi.unige.it/person/ReggioG/UMLWORKSHOP/ ACCEPTED.html`.

[31] M. Roggenbach and T. Mossakovski. Basic Datatypes in CASL. CoFI Note L-12 version 0.4.1. Technical report, 2000. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .

[32] M. Roggenbach and T. Mossakowski. What is a good CASL specification. In M. Wirsing, D. Pattinson and R. Hennicker editors, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th International Workshop WADT'02*, number 2755 in Lecture Notes in Computer Science. Springer Verlag, 2003.

A Simple system, cell schemas

incompat2 (label property) If their arguments satisfy some conditions, then an instantiation of ei_1 and one of ei_2 are incompatible, i.e., no label of a transition may contain both

$ei_1(arg_1)$ **incompatible with** $ei_2(arg_2)$ **if** $cond(arg_1,arg_2)$

Two elementary interactions (ei_1,ei_2) cell schema

value3 (state property) The results of the observation made by so_1 and so_2 on a state must satisfy some conditions

cond, where both so_1 and so_2 must appear in *cond*

Two state observers (so_1,so_2) cell schema

pre-cond2 (transition property) If the label of a transition contains some instantiation of *ei*, then the result of the observation made by *so* on the source state of the transition must satisfy some condition.

if $ei(arg)$ **happen then** $cond(arg)$
where source state observer *so* must appear in $cond(arg)$ and the target state observers cannot appear in $cond(arg)$

post-cond2 (transition property) If the label of a transition contains some instantiation of *ei*, then the result of the observation made by *so* on the target state of the transition must satisfy some condition.

if $ei(arg)$ **happen then** $cond(arg)$
where the target state observer so' must appear in $cond(arg)$ and the source state observers may appear in $cond(arg)$

vital2 (state property) If the result of the observation made by *so* on a state satisfies some condition, then any path (sequence of transition) starting from it will eventually contain a transition whose label contains an instantiation of *ei*. Note that in these properties **in any case** may be replaced by **in one case** and **eventually** by **next**.

if $cond(arg)$ **then in any case eventually** $ei(arg)$ **happen**
where *so* must appear in $cond(arg)$

Elementary interaction (*ei*) and state observer (*so*) cell schema

B Example: Fragment of a Specification of a Lift plant

On the sensors

The sensors cannot communicate two different data simultaneously;
recall that in the property "X **incompatible with** Y" it is implicitly assumed that "X" is different from "Y".
$\textsc{CabinPosition}(f_1)$ **incompatible with** $\textsc{CabinPosition}(f_2)$
$\textsc{DoorPosition}(f, dps_1)$ **incompatible with** $\textsc{DoorPosition}(f, dps_2)$
$\textsc{MotorStatus}(ms_1)$ **incompatible with** $\textsc{MotorStatus}(ms_2)$
$\textsc{UserInside}(b)$ **incompatible with** $\textsc{UserInside}(b')$

The sensors always communicate the correct data.
if $\textsc{CabinPosition}(f)$ **happen then** $cabinPosition = f$
if $\textsc{DoorPosition}(f, dps)$ **happen then** $doorPosition(f) = dps$
if $\textsc{MotorStatus}(ms)$ **happen then** $motorStatus = ms$
if $\textsc{UserInside}(b)$ **happen then** $b = (usersInside \neq 0)$

The sensors never break down,
thus they are always able to communicate the correct data.
in one case next $\textsc{CabinPosition}(cabinPosition)$ **happen**
in one case next $\textsc{DoorPosition}(f, doorPosition(f))$ **happen**
in one case next $\textsc{MotorStatus}(motorStatus)$ **happen**
in one case next $\textsc{UserInside}(usersInside \neq 0)$ **happen**

The sensors and the devices for receiving the orders are independent,
that is they can send their data and receive the orders also simultaneously.

On the doors

No two door orders may be received simultaneously.
$\textsc{OpenDoor}(f_1)$ **incompatible with** $\textsc{OpenDoor}(f_2)$
$\textsc{OpenDoor}(f_1)$ **incompatible with** $\textsc{CloseDoor}(f_2)$
$\textsc{CloseDoor}(f_1)$ **incompatible with** $\textsc{CloseDoor}(f_2)$

The open door at floor f order can be executed only when the motor is stopped, the cabin is at floor f, and the doors at all the other floors are closed
(thus also if the door at f is already open), and it is always correctly executed.
if $\textsc{OpenDoor}(f)$ **happen then**
  $motorStatus = stop$ **and** $cabinPosition = f$ **and**
  (**for all** f' • **if** $f \neq f'$ **then** $doorPosition(f') = closed$) **and**
  $doorPosition'(f) = open$

Whenever the open door order may be executed, it can be received
if $motorStatus = stop$ **and** $cabinPosition = f$ **and**
  (**for all** f' • **if** $f \neq f'$ **then** $doorPosition(f') = closed$) **then**
  in one case next $\textsc{OpenDoor}(f)$ **happen**

The close door at floor f order can be executed only when the motor is stopped and the cabin is at floor f (thus also if the door at f is already open),
and it is always correctly executed.
if $\textsc{CloseDoor}(f)$ **happen then**
    $\mathit{motorStatus} = \mathit{stop}$ **and** $\mathit{cabinPosition} = f$ **and** $\mathit{doorPosition}'(f) = \mathit{closed}$

Whenever the close door order may be executed, it can be received.
if $\mathit{motorStatus} = \mathit{stop}$ **and** $\mathit{cabinPosition} = f$ **then**
    in one case next $\textsc{CloseDoor}(f)$ **happen**

The door at floor f can be open only when the cabin is at f.
if $\mathit{doorPosition}(f) = \mathit{open}$ **then** $\mathit{cabinPosition} = f$

The door at f becomes closed/open only if the corresponding order is executed.
if $\mathit{doorPosition}(f) = \mathit{open}$ **and** $\mathit{doorPosition}'(f) = \mathit{closed}$ **then**
    $\textsc{CloseDoor}(f)$ **happen**
if $\mathit{doorPosition}(f) = \mathit{closed}$ **and** $\mathit{doorPosition}'(f) = \mathit{open}$ **then**
    $\textsc{OpenDoor}(f)$ **happen**

On the users inside the cabin

The physical limits of the cabin is five persons.
$\mathit{usersInside} \geq 0$ **and** $\mathit{usersInside} \leq 5$

User may enter/leaving the cabin only when the cabin is stopped at a floor
with open doors.
if $\mathit{usersInside} \neq \mathit{usersInside}' =$ **then**
    exists f **s.t.** $\mathit{doorPosition}(f) = \mathit{open}$ **and**
    $\mathit{cabinPosition} = f$ **and** $\mathit{motorStatus} = \mathit{stop}$

It is assumed that in any case eventually the users will leave the cabin;
this information helps understand the typical user behaviour.
if $\mathit{usersInside} \neq 0$ **then in any case eventually** $\mathit{usersInside} = 0$

C Structured system, cell schemas

loc-glob1 (transition property) If a global transition is composed of some local interactions, then, under some condition, an instantiation of *ei* belongs to the label of this global transition; or vice versa, i.e., if an instantiation of *ei* belongs to the label of a global transition, then, under some condition, this global transition is composed of some local interactions.

if $locIn_1, \ldots, locIn_n$ **happen and** $cond(arg, locIn_1, \ldots, locIn_n)$ **then**
    $ei(arg)$ **happen**
or
if $ei(arg)$ **happen and** $cond(arg, locIn_1, \ldots, locIn_n)$ **then**
    $locIn_1, \ldots, locIn_n$ **happen**

incompat1*, *pre-cond1*, *post-cond1*, *vital1 defined as in Sect. 3.2

Elementary interaction (*ei*) cell schema

loc-glob2 (transition property) If an instantiation of *sid.ei* is a component of a global transition, then, under some condition, the label of this global transition must contain an instantiation of ei_1, or vice versa.

if $sid.ei(arg)$ **happen and** $cond(arg, arg_1)$ **then** $ei_1(arg_1)$ **happen**
or
if $ei_1(arg_1)$ **happen and** $cond(arg, arg_1)$ **then** $sid.ei(arg)$ **happen**

Elementary interaction (ei_1) and local interaction (*sid.ei*) cell schema

pre-cond2*, *post-cond2*, *vital2 defined as the homonymous slots for simple system, but where the elementary interaction is replaced by the local interaction.

Local interaction (*sid.ei*) and state observer (*so*) cell schema

D Data structures, cell schemas

def1 Conditions on the definedness of *con* (required only for partial constructors)

cond, where *cond* includes atoms of the form $\mathbf{def}(con(arg))$

ident1 Values represented by *con* are/are not identified with those represented by other constructors

when all defined *cond*,

where *cond* includes atoms of the form $con(arg) \ = con'(arg')$ or **not** $con(arg) = con'(arg')$, for some constructor con'

valueOn1 Conditions on the values returned by the application of operations to values represented by *con*

when all defined *cond*,

where *cond* includes terms of the form $op(con(arg))$, for some operation *op*

truthOn1 Conditions on the truth of predicates over the values represented by *con*

when all defined *cond*,

where *cond* includes atoms of the form $pr(con(arg))$, for some predicate *pr*

Constructor (*con*) cell schema

valueOn2 Conditions on the values returned by the application of *op* to values represented by *con*

when all defined *cond*, where *cond* includes terms of the form $op(con(arg))$

Constructor (*con*) and operation (*op*) cell schema

truthOn2 Conditions on the truth of *pr* over values represented by *con*

when all defined *cond*, where *cond* includes atoms of the form $pr(con(arg))$

Constructor (*con*) and predicate (*pr*) cell schema

value-value2 Relationships between the value returned by op_1 and those returned by op_2

when all defined *cond*

where *cond* includes atoms of the form $op_1(arg_1)$ and of the form $op_2(arg_2)$

Two operations (op_1,op_2) cell schema

def2 Conditions on the definedness of *op* (required only for partial operations)

cond, where *cond* includes atoms of the form **def**($op(arg)$)

value1 Conditions on the values returned by *op*

when all defined *cond*, where *cond* includes terms of the form $op(arg)$

valueOn3 Conditions on the values returned by the application of *op* to values represented by constructors

when all defined *cond*,

where *cond* includes terms of the form $op(con(arg))$, for some constructor *con*

truth-value1 Relationships between the value returned by *op* and the truth of predicates

when all defined *cond*

where *cond* includes atoms of the form $op(arg_1)$ and of the form $pr(arg_2)$ for some predicate *pr*

value-value1 Relationships between the value returned by *op* and those returned by other operations

when all defined *cond*

where *cond* includes atoms of the form $op(arg_1)$ and of the form $op'(arg_2)$ for some opertion op'

Operation (*op*) cell schema

truth-truth2 Relationships between the truth of pr_1 and that of pr_2

when all defined *cond*

where *cond* includes atoms of the form $pr_1(arg_1)$ and $pr_2(arg_2)$

Two predicates (pr_1,pr_2) cell schema

truth1 Conditions on the truth of *pr*

when all defined *cond*, where *cond* includes atoms of the form $pr(arg)$

truthOn3 Conditions on the truth of *pr* over the values represented by some constructor

when all defined *cond*,

where *cond* includes atoms of the form $pr(con(arg))$, for some constructor *con*

truth-truth2 Relationships between the truth of *pr* and that of other predicates

when all defined *cond*

where *cond* includes atoms of the form $pr(arg)$ and $pr'(arg')$, for some predicate pr'

truth-value3 Relationships between the truth of *pr* and the values returned by the operations

when all defined *cond*

where *cond* includes atoms of the form $pr(arg)$ and of the form $op(arg')$, for some operation *op*

Predicate (pr) cell schema

E Example: Fragment of a Specification of **Floor**

This example is given in Sect. 5.2 where the properties of *previous* given below were skipped. *previous* returns the floor immediately below a given one, if it exists, i.e., there is no floor between $previous(f)$ and f.

not def$(previous(ground))$
def$(previous(top))$ **and** $previous(top) \neq top$
def$(previous(f))$ **iff** f *above ground*
when all defined f *above* $previous(f)$ **and**
(**not exists** $f_1 \bullet (f_1$ *above* $previous(f)$ **and** f *above* $f_1))$ **and**
$next(previous(f)) = previous(next(f)) = f$