Towards a Formally Grounded Software Development 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 languages) 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/language 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 and paradigms 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 and two specification approaches, 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. We also support both property-oriented (axiomatic) and model-oriented (constructive) specifications. We are thus providing support for the "building-bricks" tasks of specifying/modelling software artifacts that in our experience are needed for the development process.

Our approach is illustrated with a lift case study, it was also used with other large case studies, and when used on projects by students yielded homogeneous results.

Let us note that it may be used either as itself, e.g., for requirements specification, or in combination with structuring concepts such as the Jackson's problem frames.

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

1 Introduction

1.1 Aims and Scope

One of the goals of software engineering is to provide paradigms, languages, notations, formalisms (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 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 formalism proposed, 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/languages 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 formal specification languages/techniques and some formal specification methods were developed (see [3] for the distinction between method and formalism), 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 [29], e.g., RUP [20] and COMET [14]. Clearly, there is a need to accommodate both worlds, for instance some recent works try to give a precise semantics to UML ([23, 24]), 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, sometimes based on exotic mathematical symbols/very friendly visual notation;

[1] Work partly supported by the Italian National Project SAHARA (Architetture Software per infrastrutture di rete ad accesso eterogeneo).

- 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 a development method which has 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 is also *formally grounded*, shortly FG, that is, its specifications 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 attempt towards such a method formally grounded on the algebraic specification language CASL-LTL [1, 28, 22]. It covers the requirements and the design phases of the development, offering some techniques to specify requirements and designs for very general software systems. Such techniques result in producing specifications having a precise structure at the conceptual level, which can, then, be presented either in a visual way or as a formal CASL-LTL specification.

In our opinion, a relevant result of this work is that the paradigms and the techniques of our method, being originated from the underlying theory are quite different from those supported by the most practical common methods [2]. For example, we do not make use of "use cases", nor of diagrams showing scenarios, as UML sequence and collaboration, and we are not object-oriented [3]. On the other hand we are much more systematic and inherently more rigorous, we use diagrams to visually present the behaviour of an active element (not just its reactions to external stimuli), we explicitly define how the various elements in a system cooperate, we explicitly state which are the elements composing a system, etc.

Our previous experiences suggested that the various activities in a development process are based on the "building-bricks" tasks of specifying [4] software items of different nature at different levels of abstractions. So we propose some methods for developing the basic specification blocks using as underly-

[2] However, we would like to benefit from both worlds, e.g., we use a UML-like notation to illustrate our method, and to establish links to UML is a subject of planned further work.
[3] We do not use the concept of objects.
[4] Notice that "to specify/specifications" are the terms used in the formal community, whereas in the practical world the corresponding ones are "to model/models". For example, we have CASL specifications and UML models.

ing foundation Casl-Ltl, but giving for all of them a corresponding visual presentation. 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).

Thus we present our method to handle the three cases above in Sect. 3, 4 and 5. For each case, we give two specification techniques (*property-oriented*, and *constructive* or model-oriented), giving the abstract structure of the corresponding specifications with the related visual presentation and corresponding formal Casl-Ltl specification.

To introduce a specification method we follow the conceptual schema of [3] that we briefly present in Sect. 2.1; furthermore all property-oriented specification methods presented here are all specializations for particular varieties of items of a general method that we present in Sect. 2.2.

To show that such a method is quite general and powerful, here we present how to use it for the development of systems solving three of the most relevant problems detected by M. Jackson in his work on the problem frames [16, 17], and for the requirement phase for a medium-sized Internet-based system.

The structure of the paper is the following. The remaining of our introduction is devoted to a brief presentation of the Casl and Casl-Ltl specification languages in Sect. 1.2, and to the description of our running example with pointers to its specifications in the paper in Sect. 1.3. In Sect. 2.1 our specification methods framework is described by a conceptual schema, and a general method for property-oriented specification is given. 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 property-oriented specification is developed (the specification of the parts and constituent features, the properties specification using the cell filling technique, an illustration on the lift example, and the Casl/Casl-Ltl view), and last their constructive specification is described (the characteristics, an illustration on the lift example, and the Casl/Casl-Ltl view).

Sect. 6 and 7 are devoted to the applications of our specification methods, respectively to some of the most relevant problem frames, and to present the requirements for an Internet-based lottery system.

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 [5], 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 [19, 1], and various documents are available on CASL, in particular the CASL language summary [28], a complete formal semantics, and the CASL User Manual is being written [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 [6], 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]

spec SPECNAME =
    type *type_name* ::= $constr\_name(argTypes_{constr})$ | ...
    ops $op\_name : argTypes_{op} \rightarrow resType_{op}$
        ...
    preds $pr\_name : argTypes_{pr}$
            ...
    axioms
        formulae

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* ::= $constr\_name(argTypes_{constr})$ | ...
        ...

"In practice, a realistic software specification involves *partial* as well as total functions."[9] Partial operations or constructors are declared with a '?' symbol,

[5] `http://www.brics.dk/Projects/CoFI`
[6] with *strong* equality (Sect. 5.2.1) and a 2-valued logics

and the definedness of a term can be asserted in the axioms.

spec SPECNAME =
  type *type_name* ::= *constr_name*($argTypes_{constr}$)? | ...
  ops *op_name* : $argTypes_{op}$ →? $resType_{op}$
    ...
  axioms
    def(*constr_name*(...)) ⇔ ...

Let use note that special care is needed in specifications involving partial functions [9]. 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* ::= *constr_name*($argTypes_{constr}$) | ...
  ops *op_name* : $argTypes_{op}$ →? $resType_{op}$
    ...
  axioms ...
} end

The following specification is an extension of an instance of the generic specification FINITESET[ELEM] (which is part of the basic specifications library [27]) by INT(again in [27]).

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 [22], which was designed for the dynamic systems specification by giving a CASL view to LTL, the Labelled Transition Logic ([4, 13]).

LTL, and thus CASL-LTL, is based on the idea that a dynamic system is considered as a labelled transition system (lts), and that to specify it one has to specify the labels, the states and the transitions of such system. 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.

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 transition tree associated with *d*. The root of such tree is decorated with *d*, and if the tree has a node decorated with *d1* and $d1 \xrightarrow{l} d2$, then it has a node decorated with *d2*, and an arc from *d1* to *d2* decorated with *l*. 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 [13], 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 property 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 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 $eventually\ \pi_1$
- the path satisfies a condition expressed by another path formula until some point where it satisfies a second condition $\pi_1\ until\ \pi_2$
- the path satisfies a condition expressed by another path formula in any point $always\ \pi_1$.

Path formulae may be also built by combining other path formulae by means of the CASL logical combinators, such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$ and $\exists$.

1.3 Running Example: the Lift System

To illustrate the use of our FG specification methods we specify at different levels a lift system and some of its subparts.

A lift system consists of a *lift plant* (that is the cabin, the motor moving it and the doors at the various floors), some software automatically controlling the lift functioning (the *controller*), and the people using it (the *users*). The controller monitors the lift plant by means of *sensors*, which communicate the status of its various components (e.g., there is a sensor detecting the position of the cabin), and directs its behaviour by means of *orders* (e.g., it can order to open/close the doors).

In Sect. 3.2.3 a property-oriented specification of the lift plant considered as a simple system is developed. The lift plant may communicate the status of some of its components by means of sensors (the position of cabin, of the doors at the floors and the working status of the motor), and can influence its components by means of orders (open/close a door at a given floor, stop/to move up/to move down the motor). Moreover, the lift plant interacts with the external world also when some users enter or leave the cabin.

Sect. 3.3.2 gives a constructive specification of a controller for the lift, considered as a simple system. This specification may be considered as an abstract presentation of a design for such a controller. The controller may send orders to the motor and to the doors, and may receive information on the status of the plant by means of its sensors (status of the motor, and positions of the doors and of the cabin). The users interact with the controller by calling for some floor (i.e., requiring that the cabin goes to a given floor). In Sect. 4.2.3 we develop a property-oriented specification of the lift system considered as a structured system with the lift plant, the controller and the users as subparts, and in Sect. 4.3.2 we give a constructive specification of the lift controller where two parts are distinguished. The floor data structure is specified in a property-oriented way in Sect. 5.2.2, and in a constructive way in Sect. 5.3.2 having an explicit number of floors.

2 A specification methods framework

2.1 Specification Methods

To easily present the various specification methods introduced in this paper, we follow the conceptual schema proposed in [3]. In Fig. 1 we report all the

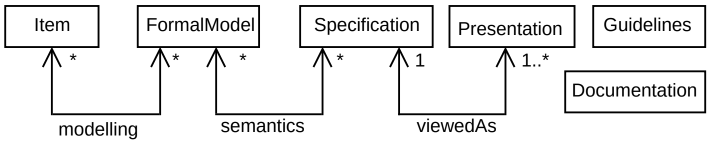


Fig. 1. Conceptual View of a Specification Method following [3]

ingredients of a generic method using an object-oriented visual notation [7], and briefly comment them below.

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.

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.

Modelling A precise and rigorous, but not formal (usually expressed using natural language) description of how the formal models are associated with the items.

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.

Semantics The semantics links a specification with its formal models.

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.

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 (in [15] they were called instead agendas). 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.

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

We assume that the *items* to be developed, and thus specified, should be

- classified in some *variety* (e.g., functional modules/data types, reactive systems, real-time systems, distributed systems, ...), and the items considered by a method should be all of the same variety [8].

[7] Precisely, it is a simple subset of UML 1.3 [29]. Recall that boxes represent classes, and arrows oriented associations.

[8] The association isA from Item to Variety does not associate a unique variety with an item, because the same item may be seen as belonging to different varieties;

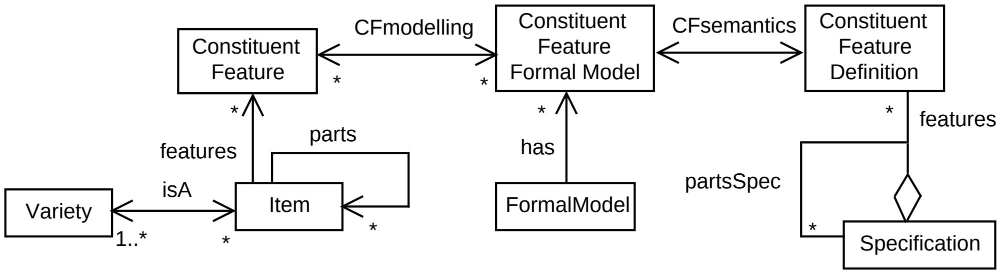


Fig. 2. Items, parts and constituent features

- *structured*, and their subparts are items. Such structure is represented by the association **parts** in Fig. 2[9]. Items associated by **parts** may be of the same variety (homogeneous structure) or of different varieties (e.g., imperative programs made out from procedures).
- 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 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 models of a specification have (as described by the association **has**) all the constituent features.
- the specification language should support the separate specifications of the subparts and should offer means to define the constituent features (**Constituent Feature Definition**).
- the guidelines should take care for finding 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*). We report the peculiar ingredients of a method using property-oriented or a constructive specifications in Fig. 3[10].

Property-oriented (axiomatic) We prefer the term *property-oriented*, as more suggestive than axiomatic. For what concerns the semantics, 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:

functional modules/data types, e.g., a reactive system is also a particular case of a distributed system.

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

[10] The arrow with large head stands for the UML specialization.

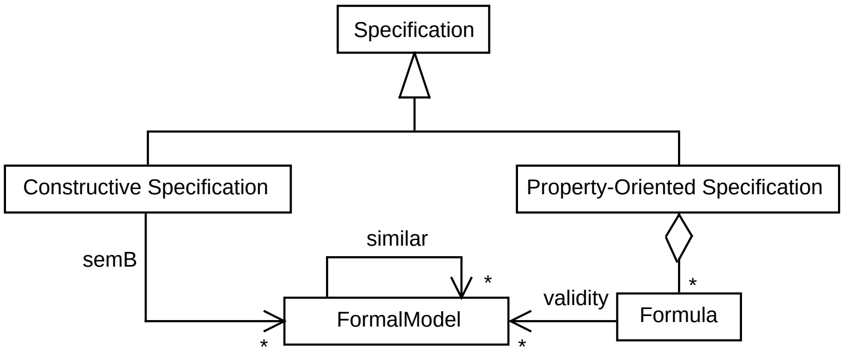


Fig. 3. Property-Oriented and Constructive Specifications

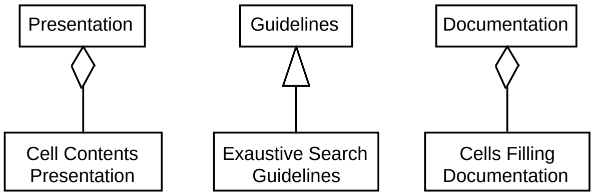


Fig. 4. GPSm

we describe the item at a certain moment in its development by expressing all its "relevant" properties by sentences provided by the formalism (formulae).

Constructive (model-oriented) In this case the semantics is defined as follows: "a model M belongs to the semantics of a specification if and only if there exists another model belonging to the basic semantics (association semB) of the same specification that is similar to M". The methodological ideas supporting this specification style are:
we describe the item at a certain moment in its development by giving a prototype/archetype of it using the specification language; then we say which are the irrelevant features of this archetype by the relation similar. If two models are similar, then they differ from each other for irrelevant details, which can thus be freely fixed later in the development.

The name constructive or construction-oriented means that we specify an item by construction (at the abstraction level supported by the method, that is depending on the formal models and on the specification language); afterwards we would say when another construction may be equivalent.

2.2 A General Property-oriented Specification Method (GPSm)

Now we introduce a General Property-oriented Specification method (GPSm) following the conceptual schema (Fig. 1) introduced in the Sect. 2.1, by specializing and enriching the ingredients of a generic method using property-oriented

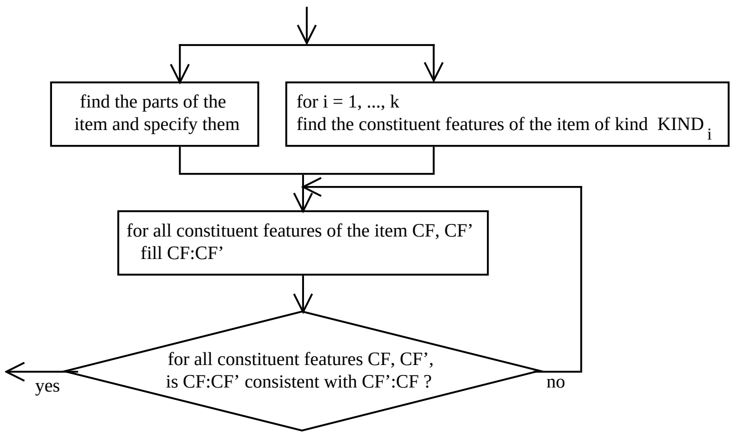


Fig. 5. Exhaustive search guidelines of GPSm

		KIND$_1$				KIND$_k$		
		CF_1^1	.....	CF_{n1}^1	.....	CF_1^k	.....	CF_{nk}^k
KIND$_1$	CF_1^1							
	.....							
	CF_{n1}^1							
	.....							
KIND$_k$	CF_1^k							
	.....							
	CF_{nk}^k							

Fig. 6. Properties Spreadsheet

specifications; these modifications are reported in Fig. 4, and commented below. Let us first introduce the guidelines of GPSm in Fig. 5, where the first steps are to find the parts and to specify them, and to find the constituent features. Then the properties about constituent features should be expressed.

Exhaustive Search Guidelines Our method 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 viewpoint of all its constituent features.

The general idea is to find the properties of a given item by filling the spreadsheet in Fig. 6. The columns and the rows are indexed with the constituent features of this item, where we assume that the kinds of the constituent features of the items of the considered method are KIND$_1$, ..., KIND$_k$, and that for $i = 1, \ldots, k$ the constituent features of kind KIND_i are $\mathrm{CF}^i_1, \ldots, \mathrm{CF}^i_{n_i}$. A cell with index CF^i_j:CF^i_j contains the properties about the constituent feature CF^i_j, and a cell with index CF^i_j:CF^h_m contains the properties expressing the relationships between CF^i_j and CF^h_m.

We assume that the properties filling the various cells follow particular schemata depending on the kinds of the two indexing elements and on the formulae of the chosen specification language. We need schemata for the cells indexed by:

KIND_i - KIND_i for $i = 1, \ldots, k$ the properties about a constituent feature of kind KIND_i considered by itself, and the properties about the relationships between a constituent feature of kind KIND_i and another one of the same kind

KIND_i - KIND_j for $i, j = 1, \ldots, k$, $i \neq j$ the properties about the relationships between a constituent feature of kind KIND_i and another one of kind KIND_j.

At this point we can 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 general method requires then to check that they are consistent. In the case of a negative answer, we found some inconsistency that must be eliminated. Usually, this activity helps detect some problematic or misunderstood aspects of the specified item.

After having provided the required schemata for filling the cells, we can define the general guidelines of GPSm, which are reported in Fig. 5. Clearly a specialization of this general method may induce a change in the order to follow to perform the tasks inside the various boxes in Fig. 5; moreover finding the parts and the constituent features may also be interleaved. Specializations may also require that any cell CF:CF' must be equivalent to CF':CF, and then to fill only half of the spreadsheet avoiding the cells over the diagonal. Other specializations may add additional equivalence checks in the case some cells share the same property.

Notice 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 changes, and such changes result in adding/removing constituent features, then the properties may be easily modified, in such case we have just to delete/add some specific rows/columns.

Cell Contents Presentation As regards the presentation of the produced specifications, GPSm should provide:

- a nice way to present the properties filling each relevant type of the cells of the spreadsheet;
- a nice way to arrange the contents of the various cells of the spreadsheet; for example by means of a precise sectioning schema with related titles.

The result of this process then needs some presentation work and rearrangement of the found properties to yield a specification nicer to read.

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

2.3 Introducing a Method

To introduce a method in this paper, following the general conceptual framework of Sect. 2.1, we proceed in the following way. We begin by introducing the specified items, the used formal models, and the rationale linking the latter to the first; this is given in a "*section n.1 X Item*" where X may be a simple or structured system, or a data structure, and this part is common for both property oriented and model oriented methods introductions that follow. For each, we adopt the following:

- We show the form of the specification visual presentation. To avoid the obvious problems of precisely presenting a visual notation we follow the UML style (metamodelling), by giving the structure of these artifacts by means of a class diagram presenting all their components, and then saying how to visually depicting them. We accompany this part by the guidelines to produce such artifacts.
- This is then illustrated on an example (the lift).
- Finally we present the corresponding specification artifacts and the specification language/formalism used to write them, in a "*section* CASL-LTL *(or* CASL*) View*".

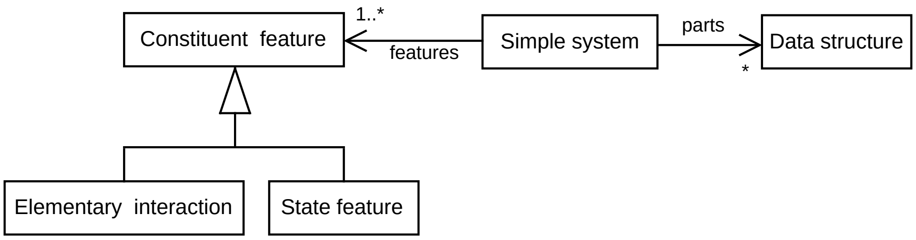


Fig. 7. Simple System Item

3 Specification of Simple Systems

3.1 Simple System Item

Following the framework presented in Sect. 2.1 and in Fig. 2 we describe the simple system items structure. Here the word *system* denotes a dynamic system 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* from now on). Recall that an lts is a triple $(State, Label, \_\_ \xrightarrow{\_\_} \_\_ \subset State \times Label \times State)$. 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., it fully describes the interaction of the system with the external environment during this transition.

To design effective and simple specification methods 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). The above considerations lead us to choose the *elementary interaction types* (just *elementary interactions* from now on) as constituent features of the simple systems.

The form of the states (which are the intermediate situations during the system's life) is also a characterizing feature of simple systems, therefore we need *state constituent features.* However, they are technically different for the property-oriented and the constructive case, and so we will describe them later, when presenting the two methods.

To define the constituent features of a simple system 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 Fig. 7.

3.2 Property-oriented specifications

The property-oriented specification method for simple systems we propose is a specialization of GPSm introduced in Sect. 2.2. According to Fig. 5, we first have to find the parts and constituent features, and then to fill the cells to express the properties.

3.2.1 The specification of parts and constituent features

To keep the specification level abstract, we do not completely describe the states, but we just list what we should be able to observe on them, and thus the state features will correspond to elementary observations on the states (*state observers*). A state observer is characterized by a name, some arguments (elements of some *data structures*), and by the observed value (element of some data structure) [11] .

Fig. 8 shows the structure (by means of a UML class diagram) of a property-oriented specification of a simple system, and Fig. 9 the way to visually depict its parts and the constituent features. type in Fig. 9 stands for a type of values defined by one of the subparts data structures (DATA_{1}, ..., DATA_{j}).

3.2.2 Cell schemata (properties)

All 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

[11] If the observed value is a boolean, then it may be specified with a predicate. For simplicity sake (and by lack of space), this case will not be considered in this paper.

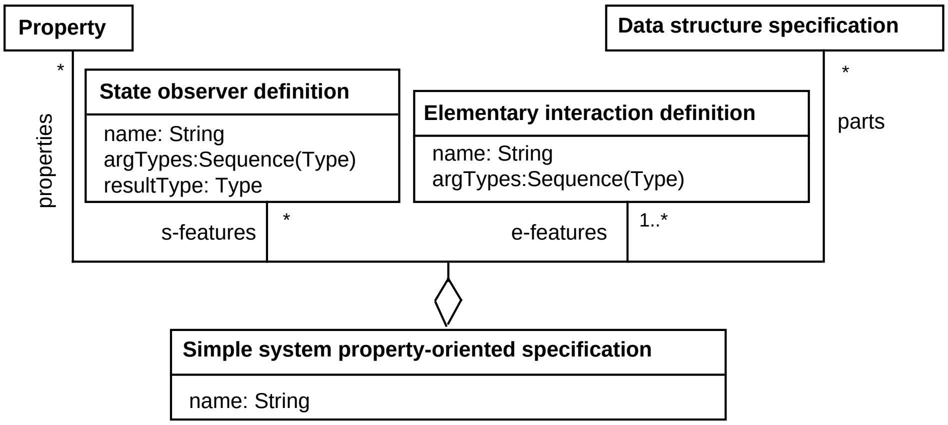


Fig. 8. Simple System Property-OrientedSpecification

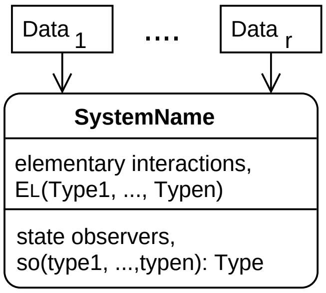


Fig. 9. Visual presentation of a simple system: parts and constituent features

source state, the label and the target state of a transition. Our visual notation offers appropriate ways to present these properties presented below.

Label properties: $ei_1(arg_1)$ **incompatible with** $ei_2(arg_2)$ **if** $cond(arg_1, arg_2)$
where ei_1 and ei_2 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 [12], then they are incompatible, i.e., no label may contain both.

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

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

- **in any case eventually** *eIn* **happen**
  It means that any path (sequence of transitions) leaving the state will contain a transition whose label contains the elementary interaction described by *eIn*.

  These atoms may also be built by **at least in one case** (instead of **in**

[12] Note that it is then not necessary to express that they are different.

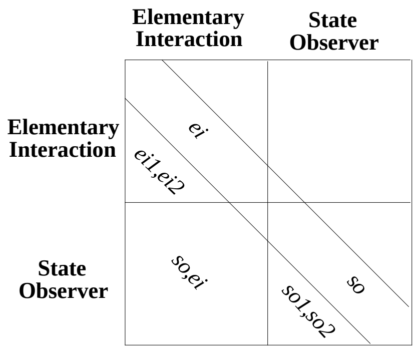


Fig. 10. A generic properties spreadsheet for simple systems

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 ... ”).

- Similarly we have atoms to express conditions on the past behaviour of a system before reaching this state

  in any case sometimes *eIn* **happened**

  It means that any path (sequence of transitions) reaching the state will contain a transition whose label contains the elementary interaction described by *eIn*.

  in any case may be replaced by **at least in one case** , and **sometimes** by **before** (with the meaning “the label of the last transition of the path contains ... ”).

Transition properties: *cond*

where *cond* is a property in which state observers on the source and target states (resp. denoted by “non primed” *so* and “primed” *so*’ identifiers, and from now on referred to as “source state observers” and “target state observers”), and atoms of the form “*eIn* **happen**‘” may appear. It means that any transition “$tr= x \xrightarrow{l} y$” satisfies such condition, where source state observers are evaluated on the source state x of tr, the target state observers are evaluated on the target state y of tr, and the atoms of the form “*eIn* **happen**” are true if and only if the elementary interaction described by *eIn*belongs to the label l of tr.

The constituent features of simple systems are of two kinds, elementary interactions and state observers, and so we have to consider five kinds of cells, as shown in Fig. 10. Then we detail the schemata of the “Elementary interaction” and “State observer” cells (the others are detailed in the Appendix A). There, *arg* stands for generic expressions of the correct types, possibly with free variables, and *cond*(xxx) for a generic condition where the free variables of xxx may appear.

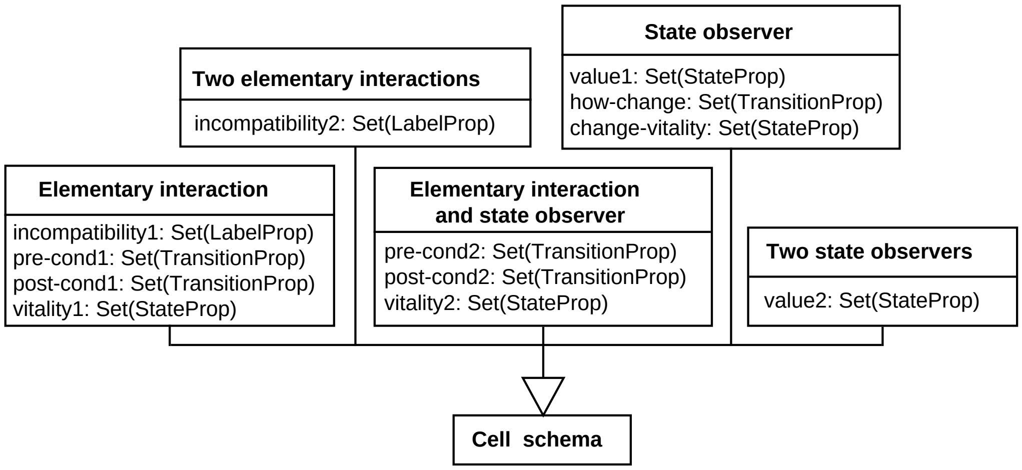


Fig. 11. Simple System Cell Schemata

---

incompatibility1 (label property)

If their arguments satisfy some conditions, then two instantiations of ei are incompatible, 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 source state observers must appear in $cond(arg)$ and target state observers cannot appear

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 target state observers must appear in $cond(arg)$ and source state observers may appear

vitality1 (state property)

If a state satisfies some condition, then any sequence of transitions starting from it will eventually contain a transition whose label contains ei. Note that vitality properties may have also the form "at least in a case" (instead of "in any case") or "next" (instead of "eventually").

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

Fig. 12. Elementary interaction (ei)

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 the source and target state (the old and the 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-vitality (state property) If a state satisfies some condition, then the observed value will change in the future.

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

Note that the properties may have also the form "at least in a case" (instead of "in any case") or "next" (instead of "eventually").

Fig. 13. State observer (*so*)

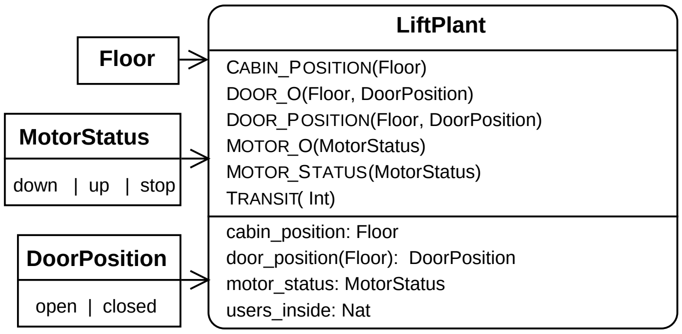


Fig. 14. LiftPlant : Parts and Constituent Features

3.2.3 *Example: a Property-Oriented Specification of a Lift plant*

As an example, we give the property-oriented specification of the lift plant introduced in Sect. 1.3, considered as a simple system. To specify the lift plant should be the first step for developing the lift system, indeed a precise knowledge of the plant is of fundamental importance for developing a good lift system.

Parts and Constituent Features We show the parts and the constituent features of the lift plant in Fig. 14 (the elementary interactions are in upper case letters, the state observers in lower case). The lift plant may communicate the status of some of its components by means of sensors (the position of cabin, of the doors at the floors and the working status of the motor), and can influence its components by means of orders (open/close a door at a given floor - DOOR_O, stop/to move up/to move down the motor - MOTOR_O). Moreover, the lift plant interacts with the external world also when some users enter or leave the cabin (TRANSIT). Let us note that CABIN_POSITION and MOTOR_STATUS are communicated via sensors, and that *cabin_position* and *motor_status* are the corresponding state observers.

To define the above elementary interactions we need some data:
- Floor: the floors among which the cabin is moving (see 5.2.2 for its specification),
- MotorStatus: the possible status of the motor (moving up, moving down or stopped),
- DoorPosition: the possible positions of the doors at the floors (open or closed).

MotorStatus and DoorPosition are two simple enumeration data structures, for which we use an ad hoc notation (just writing the various constructors separated by | in the respective compartments).

To describe the relevant properties of the lift plant we need to know the positions of the doors and of the cabin, the status of the cabin motor and how many people are inside the cabin, and so we introduce the corresponding state observers.

Properties We followed the cell filling methods to find all the relevant properties of the lift plant, but here we dropped repeated formulae, after having checked the absence of contradictions, and slightly rearranged the others to improve readability. The properties on the orders are detailed below, while the others (on the sensors, the cabin, and the users entering/leaving the cabin) are given in the Appendix B. The elementary interactions and the state observers used below are declared in Fig. 14.

On the orders

- Only appropriate groups of orders may be received simultaneously by the lift plant; precisely at most one order for the motor and one for the doors.
  MOTOR_O(ms_1) **incompatible with** MOTOR_O(ms_2)
  DOOR_O(f_1,dps_1) **incompatible with** DOOR_O(f_2,dps_2)
- An order can be received only when its execution is possible; precisely move up (down) only when the motor is stopped and the cabin is not at the top (ground) floor, and open the door at f only when no door is open, the cabin is at floor f and the motor is stopped.
  if MOTOR_O(up) **happen then**
  $motor\_status = stop$ **and** $cabin\_position \neq top$

if MOTOR_O($down$) **happen then**
$motor\_status = stop$ **and** $cabin\_position \neq ground$
if DOOR_O(f_1,$open$) **happen then**
(**for all** f • **if** $f \neq f_1$ **then** $door\_position(f) \neq open$) **and**
$cabin\_position = f_1$ **and** $motor\_status = stop$

– The orders are always correctly executed.
if MOTOR_O(ms) **happen then** $motor\_status' = ms$
if DOOR_O(f,dps) **happen then** $door\_position'(f) = dps$

The complete specification of the lift plant given following our method 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 (the state observers corresponding to sensors are total),
- whereas motor and doors may change status by themselves as result of some failure (no property requires that if the motor changes its status an order has been received), and
- that the plant takes already care of several security checks, such as to avoid that the motor tries to go down when the cabin is at the ground floor ...

3.2.4 CASL-LTL *View*

Here we present the CASL-LTL [22] corresponding version of our property-oriented specification of simple systems introduced before in Sect. 3.2, and whose form is abstractly/conceptually presented in Fig. 8 and 11, and whose visual presentation is reported in Fig. 9.

Let *Spec* be a property-oriented specification of simple systems having the form described in Fig. 8, 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*.

spec ELINTERACTION =
free type *elInteraction* ::=
ei_1.name(ei_1.argTypes) | ... | ei_n.name(ei_n.argTypes)

spec *Spec*.name =
FINITESET[ELINTERACTION] **and** DS_1 **and** ... **and** DS_j **then**
dsort *st* **label** *FinSet*[*elInteraction*]
ops so_1.name : $st \times so_1$.argTypes $\to so_1$.resType

...
so_m.name : $st \times so_m$.argTypes $\rightarrow so_m$.resType
axioms
formulae corresponding to the cell fillings, see below

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".

The CASL-LTL formulae corresponding to the cell fillings are given below by considering all kinds of used properties.

- data properties, see Sect. 5
- label properties
  eIn_1 **incompatible with** eIn_2 **if** *cond*
  is expressed by
  $\neg\,(eIn_1 = eIn_2) \;\wedge\; cond \;\Rightarrow\; \neg\,(eIn_1 \in l \;\wedge\; eIn_2 \in l)$
  where l is a variable of sort $FinSet[elInteraction]$
- state properties
  cond is expressed by adding S (a variable of sort st) as extra argument to each source state observer
  The special temporal combinators are expressed as follows:

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

- transition properties
  cond is expressed by $S \xrightarrow{l} S' \;\Rightarrow\; cond'$
  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 "$eIn \in l$".

3.3 Constructive specifications

3.3.1 The specifications characteristics

The constructive specification method for simple system that we present is similar in many respects to the property-oriented one introduced in Sect. 3.2. Indeed, the items (simple systems), their parts and their constituent features

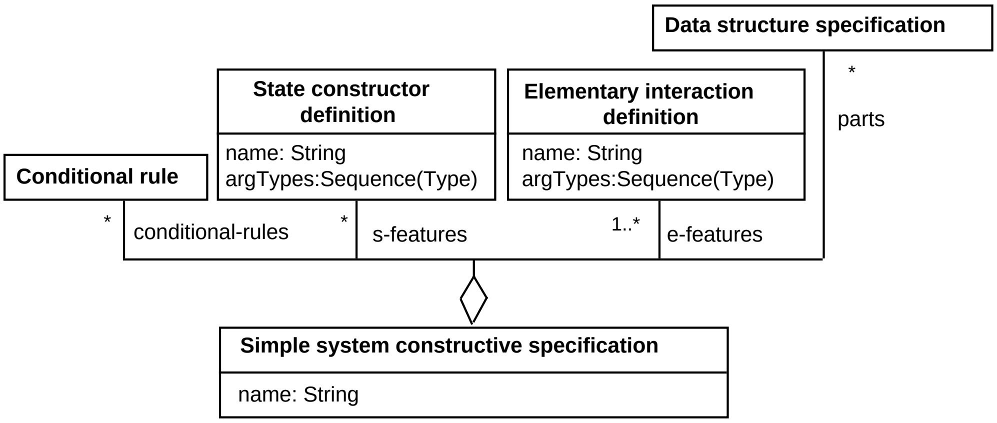


Fig. 15. Simple System Constructive Specification

of kind elementary interactions, the formal models and the modelling are the same. But, the state features are different, since we have here *state constructors* instead of state observers. In this case we have to build the states of the lts modelling a simple system, and we consider that those states may be classified into different categories, according to the different system behaviour. Technically, the states will be a data structure having a constructor with typed arguments (see Sect. 5), for each state category. The state constructors are the state constituent features.

The specification technique is similar to that followed in Sect. 3.2 in that we have also to determine the parts and the constituent features, the only difference is that now we do not give properties but instead we define precisely the states, the labels and the transition of the lts modelling the specified simple system. The definition of the labels and of the states is quite trivial (the labels are sets of elementary interactions, and the states are built by the state constructors). For what concerns the transitions, we define them by means of conditional rules of the following form

(*) $$\textbf{if } \textit{pos-cond}(\textit{arg}_1,\textit{arg}_2,\textit{eiSet}) \textbf{ then } C(\textit{arg}_1) \xrightarrow{\textit{eiSet}} C'(\textit{arg}_2)$$

where

- C and C' are state constructors, resp. of the source and target states
- *eiSet* is a set of elementary interactions
- and $\textit{pos-cond}(\textit{arg}_1,\textit{arg}_2,\textit{eiSet})$ is a conjunction of positive atoms in which the free variables of $\textit{arg}_1$, $\textit{arg}_2$ and *eiSet* may appear.

The form of our constructive specifications of simple system is summarized in Fig. 15.

We present the way to visually depict the parts and the constituent features of a simple system constructive specification in Fig. 16. A conditional rule defining the transitions as in (*) above will be visually represented as the following oriented arc

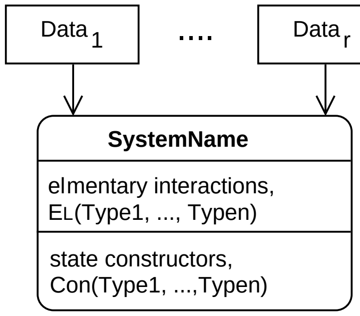
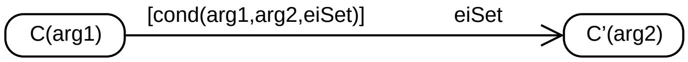
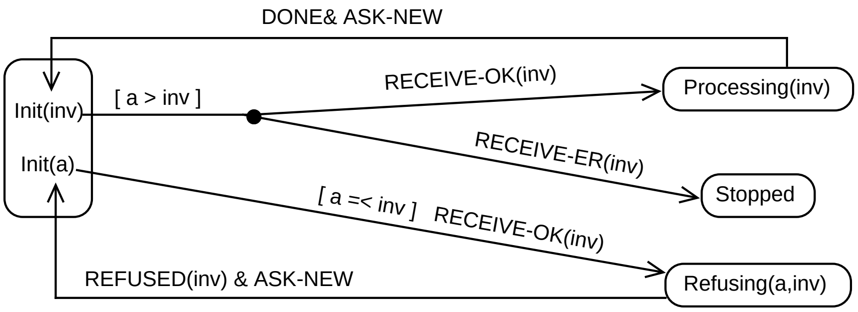


Fig. 16. Simple System Constructive Specification: parts and constituent features



The visual presentations of all the conditional rules may be then put together building an oriented graph, as originally proposed in [26], by collecting all source and target rounded boxes related to states built by the same constructor, and by writing only once repeated constructor instantiations and the repeated conditions. For example, the following diagram



shows the combination of the visual presentations of the following five conditional rules.

if $a > inv$ **then** $Init(a) \xrightarrow{RECEIVE-OK(inv)} Processing(inv)$
if $a > inv$ **then** $Init(a) \xrightarrow{RECEIVE-ER(inv)} Stopped$
if $a \leq inv$ **then** $Init(a) \xrightarrow{RECEIVE-OK(inv)} Refusing(a, inv)$
$Refusing(a, inv) \xrightarrow{\{REFUSED(inv), ASK-NEW\}} Init(a)$
$Processing(inv) \xrightarrow{\{DONE, ASK-NEW\}} Init(inv)$

3.3.2 Example: a Constructive Specification of a Controller for a Lift

As example in this section, we give the constructive specification of a controller for the lift, considered as a simple system. This specification may be considered as an abstract presentation of a design for such a controller.

Our controller may send orders to the motor and to the doors, and may receive information on the status of the plant by means of its sensors (status of the

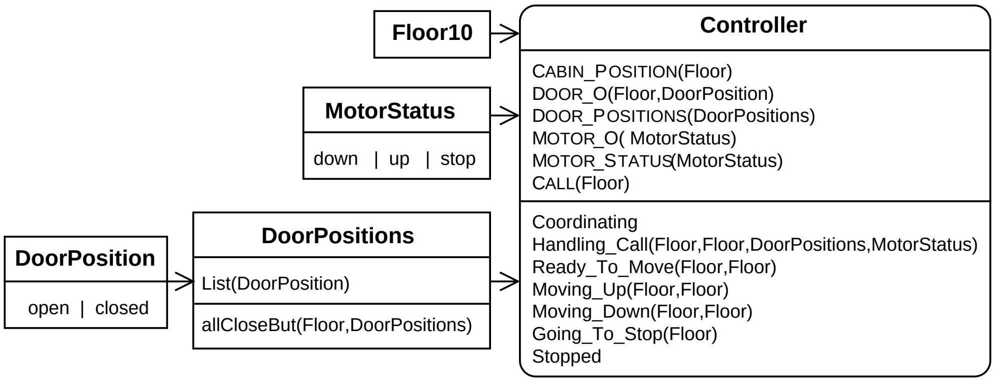


Fig. 17. Lift Controller: Parts and Constituent Features

motor, and positions of the doors and of the cabin). The users interact with the controller by calling the floors, that it by requiring that the cabin goes to a given floor.

Following our method we first determine the elementary interactions, the used data structures, and finally define the behaviour of the controller. In this case the elementary interactions (see Fig. 17) correspond to sending the orders, and to receiving signals from the sensors and calls from the users.

To needed data structures are Floor10 (providing an *ord* operation yielding the floor number), defined in Sect. 5.3.2, MotorStatus and DoorPositions. The latter is defined by using the predefined parametric type List [27]. DoorPositions also has a derived predicate *all_Close_But* defined by the following conditional rules. Recall that *select* given a list and a natural number returns the element in such position, and that we assume that the positions of the various doors are listed one after the other starting from *ground* till to *top*.

if *select*(*ord*(*ground*),*dposs*) = *open* **and**
  select(*ord*(*second*),*dposs*) = *closed* **and** ... **and**
  select(*ord*(*top*),*dposs*) = *closed* **then**
  all_Close_But(*ground*,*dposs*)
...
if *select*(*ord*(*top*),*dposs*) = *open* **and**
  select(*ord*(*ground*),*dposs*) = *closed* **and** ... **and**
  select(*ord*(*ninth*),*dposs*) = *closed* **then**
all_Close_But(*top*,*dposs*)

The behaviour of the controller is shown in Fig. 18. This controller waits for a call from the users (Coordinating), then if the call may be satisfied, closes the door (DOOR_O(f1, close)) and orders to the motor to move in the correct versus, till the cabin reaches the required floor (Stopping(f1)). Then it opens the door and waits for the next call. It is also able to detect some failures (an order to the motor has not been executed), and in such cases it stops to work.

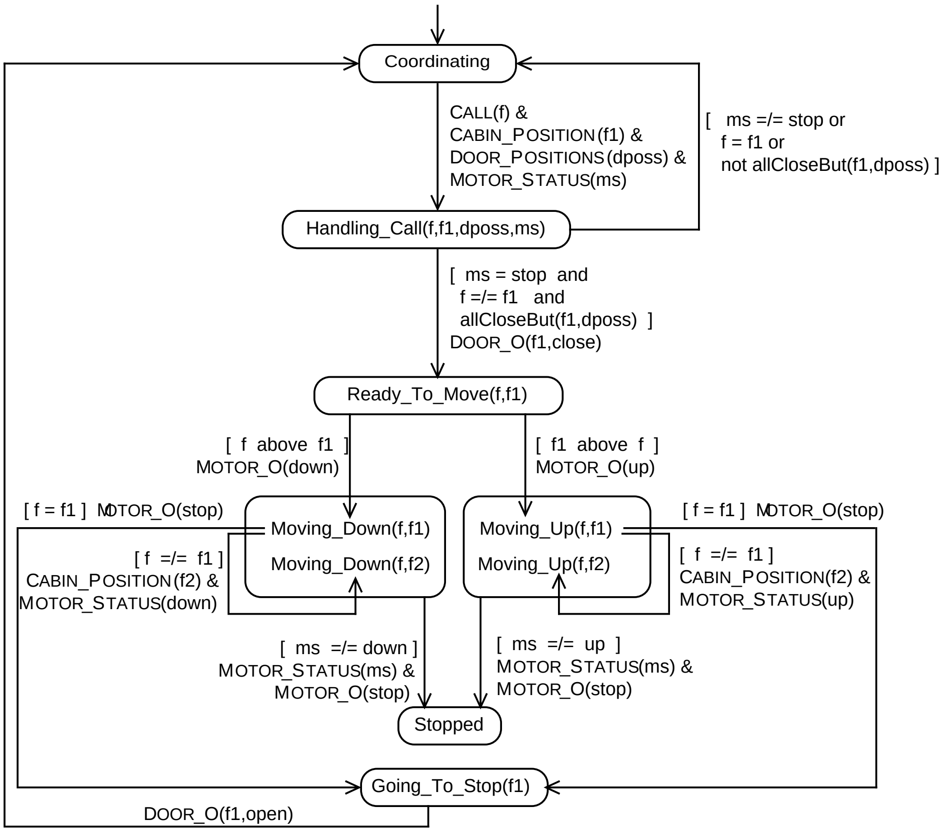


Fig. 18. Lift Controller Behaviour

3.3.3 Casl-Ltl *View*

Here we present the Casl-Ltl [22] corresponding version of our constructive specifications of simple systems introduced before in Sect. 3.3.

Let *conSpec* be a constructive specification of simple systems having the form described in Fig. 15, and assume that

- *conSpec*.parts $= \{ds_1, \ldots, ds_j\}$ are the parts, and $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ the corresponding Casl-Ltl specifications;
- *conSpec*.e-features $= \{ei_1, \ldots, ei_n\}$ are the elementary interactions;
- *conSpec*.s-features $= \{sCon_1, \ldots, sCon_m\}$ are the state constructors.

Below we give the Casl-Ltl specification corresponding to *conSpec*.

spec ElInteraction =
  free type *elInteraction* ::=
    ei_1.name(ei_1.argTypes) | ... | ei_n.name(ei_n.argTypes)

spec *conSpec*.name =
  FiniteSet[ElInteraction] **and** DS_1 **and** ... **and** DS_j **then**
free {
  dsort *st* **label** *FinSet*[*elInteraction*]

ops $sCon_1$.name : $sCon_1$.argTypes $\rightarrow st$

...

$sCon_m$.name : $sCon_m$.argTypes $\rightarrow st$

axioms

formulae corresponding to conditional rules

} **end**

A conditional rule

if $\textit{pos-cond}(arg_1, arg_2, eiSet)$ **then** $C(arg_1) \xrightarrow{eiSet} C'(arg_2)$

is expressed by the CASL-LTL formulae

$\textit{pos-cond}(arg_1, arg_2, eiSet) \Rightarrow C(arg_1) \xrightarrow{eiSet} C'(arg_2)$.

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/concurrent executions of (local) moves of some of its subsystems.

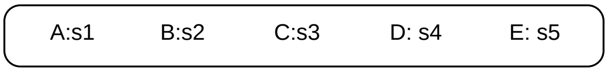
The specification methods for structured systems (property-oriented and constructive), which we present here, are specializations of those for simple systems (see Sect. 3). Thus also structured systems will be modelled by labelled transition systems (lts); but in this case their states will be sets of states of those lts's modelling the subsystems, and their transitions will correspond to concurrent/simultaneous executions of sets of transitions of the subsystems (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 extend the labels of the transitions with the information about the subsystems moves. Indeed, labels should model only the interaction of the system with the outside world, and in many cases the moves of the subsystems are completely transparent to outside; think, e.g., of two subsystems exchanging a message between themselves.

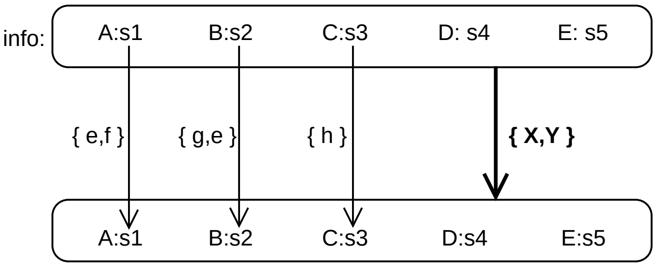
Technically, it means that to model structured systems we use *generalized lts*, that are lts specialized by adding an information part to each transition. Thus

structured system SS, with five subsystems, identified respectively by A, B, C, D and E

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



sample transition of SS (global transition/move)



composing local transitions/moves

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

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

(gobal) elementary interactions of SS towards the outside resulting from the subsystem moves

X Y

additional information

info = { A.e, A.f , B.g , B.e , C.h }

Fig. 19. Structured system examples

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 attached to the transitions. A generic transition is usually written $i : x \xrightarrow{l} y$.

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) an by an elementary interaction (belonging to the label of the subtransition). We name these pairs, subsystem identity and elementary interaction, *local elementary interactions*, shorty *local interactions* from now on.

We exemplify the concepts introduced so far in Fig. 19.

To take into account the role played by the subsystems in the moves of the structured systems, we consider as their constituent features also the local interactions. Structured systems have also a new kind of parts, the composing subsytems, which may be either simple or in turn structured. We summarize the parts and constituent features of structured systems in Fig. 20.

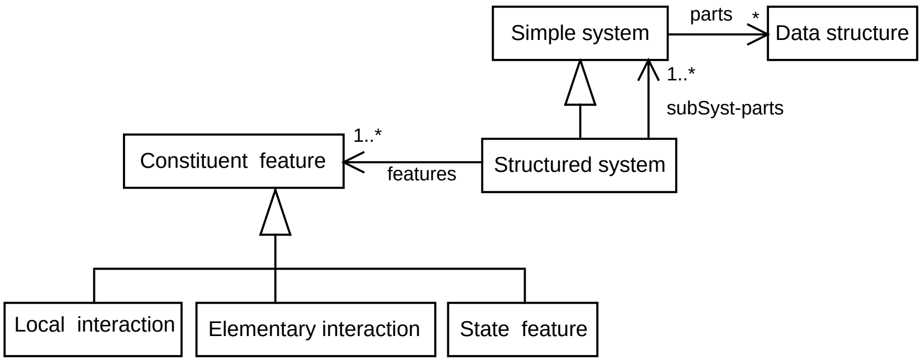


Fig. 20. Structured System Item

Furthermore, for simplicity sake we do not consider the case of subsystems that may be created and destroyed dynamically, but there are no technical problems to handle them.

4.2 Property-oriented specifications

The method for property-oriented specifications of structured systems, which we present here, is a specialization of the one for the simple systems of Sect. 3.

4.2.1 Specification of parts and constituent features

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, property-oriented 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.

Here, we assume that a structured system may have many subsystems of the same kind (i.e., whose specification is the same), and that they are identified by elements of a special data structure IDENT (just standard identifiers). Thus to specify the subsystem parts it is sufficient to give the subsystem specifications and the identity and the name of the corresponding specification of the various subsystems. The local interactions are implicitly determined after we have given the subsystems and their specifications, and so they do not need to be explicitly specified.

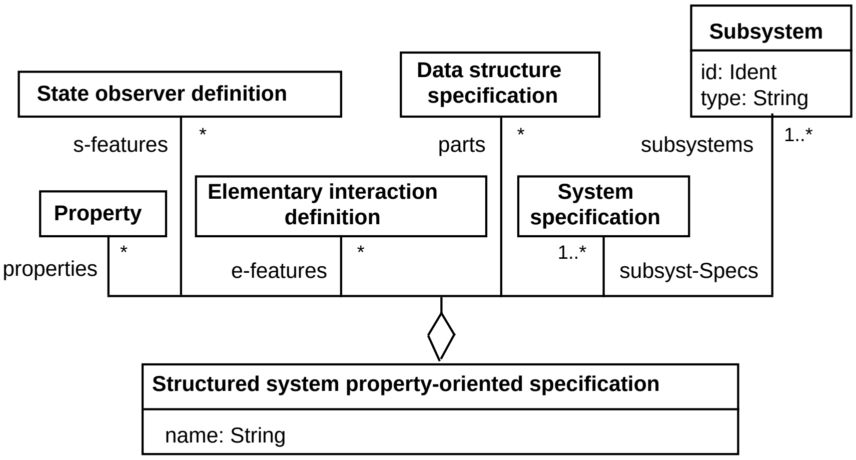


Fig. 21. Structured System Property-Oriented Specification

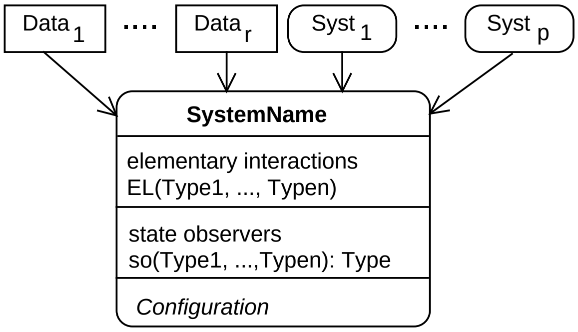
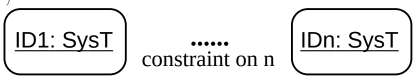


Fig. 22. Visual presentation of a structured system: parts and constituent features

The structure of a property-oriented specification of a structured system is then summarized in Fig. 21.

Fig. 22 presents the way to visually depict the parts and the constituent features of a property-oriented structured system specification. In this picture Syst$_1$, ..., Syst$_p$ are the names of the subsystem specifications, given apart, and *Configuration* is a visual presentation of which are the subsystems with their types and identities. A subsystem is represented by a rounded box containing its identity and type, that is the name of the corresponding specification. We use dot 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 (underlined) type name; thus the subsystem will be named as the type. In Fig. 23 we show an example of a configuration; the structured system with that configuration has one subsystem of type Sys1, named Sys1, two subsystems of type Sys2 named respectively A and B, and n, with $1 < n < 10$ subsystems of type Sys named respectively C1, ..., Cn.

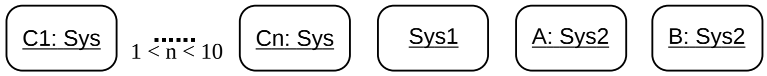


Fig. 23. Configuration example

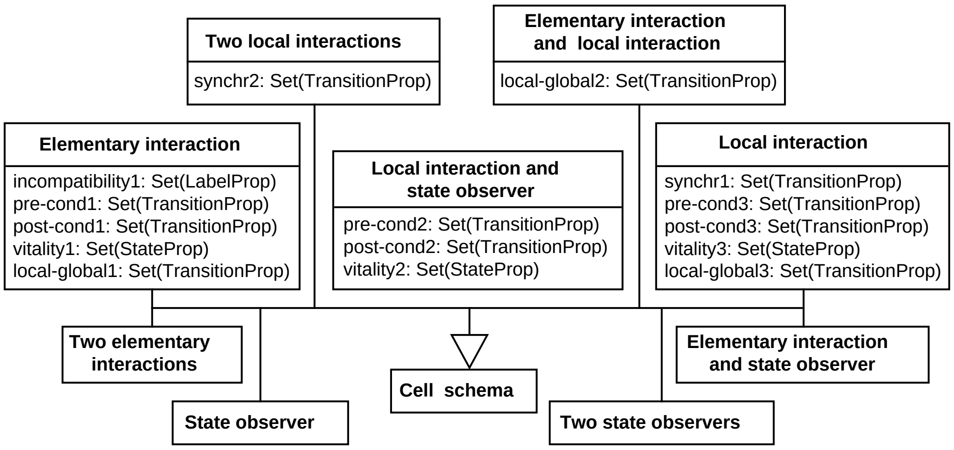


Fig. 24. Structured System Cell Schemata

4.2.2 Cell fillings (properties)

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 schemata 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 had upgraded lts's to generalized lts, which differ for the addtional information part of te transitions (just the set of the local interactiosn). Now, we consequently upgrade the properties on the transitions (see Sect. 3.2.2) with new atoms "*lIn* **happen**", with *lIn* local interaction, which state that *lIn* belongs to the set of local interactions of the transition. More precisely, *lIn* **happen** holds on a transition of a generalized lts "$inf : x \xrightarrow{l} y$" iff $lIn \in inf$.

The new properties will allow us to take into account the local interactions hen expressing the properties the various cells.

In Fig. 24 we present the schemata for the new cells and the updates of those already used for simple systems, where undetailed "boxes" refer to Fig. 11 as well as the slots that are not redefined here. Clearly, for the parts already defined in Sect. 3.2.2 here we must consider also the local interactions together with elementary interactions in the state and transition properties. The schemata for the various cells are then reported respectively in Fig. 25 to 29.

---

local-global1 (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 $lIn_1, \ldots, lIn_n$ **happen and** $cond(arg, lIn_1, \ldots, lIn_n)$ **then** $ei(arg)$ **happen**
or
if $ei(arg)$ **happen and** $cond(arg, lIn_1, \ldots, lIn_n)$ **then** $lIn_1, \ldots, lIn_n$ **happen**

Fig. 25. Elementary interaction (*ei*)

---

local-global2 (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 $cond(arg, arg_1)$ **and** $sid.ei(arg)$ **happen then** $ei_1(arg_1)$ **happen**
or
if $cond(arg, arg_1)$ **and** $ei_1(arg_1)$ **happen then** $sid.ei(arg)$ **happen**

Fig. 26. Elementary interaction and local interaction (ei_1:*sid.ei*)

4.2.3 Example: a Property-Oriented Specification of a Lift System

The lift system consists of the lift plant (the cabin, the motor moving it, the doors at the floors and the user entering/leaving the cabin), the automated software controller and the user; and, following Sect. 4.1, it is a structured system. Here we use our method to express the relevant properties on it, 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 its context.

We present in Fig. 30 the parts and the constituent features of the lift system. The subsystems are the controller, the plant, and the users (a unique system modelling all the lift users), and all of them are simple systems; whereas the used data structures are those of the subsystems and so we do not repeat them.

We specify in a property-oriented way the three subsystems. The specification of the lift plant, LiftPlant, has been given in Sect. 3.2.3 and that of the users is shown in Fig. 31. Notice that it has no properties, because in this case we do not know anything on the behaviour of the users, for instance we do no assume

synchr1 (transition property)

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

if *cond*(*arg*,arg_1) **and** *sid.ei*(*arg*) **happen then** $sid_1.ei(arg_1)$ **happen**
or
if *cond*(*arg*,arg_1) **and** *sid.ei*(*arg*) **happen then not** $sid_1.ei(arg_1)$ **happen**

local-global3 (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** *eIn* **happen**

pre-cond3*,*post-cond3*,*vitality3 defined as the homonymous slots for simple system but where the elementary interaction is replaced by the local interaction.

Fig. 27. Local interaction (*sid.ei*)

synchr2 (transition property)

Under some condition, an instantiation of $sid_1.ei_1$ is synchronized/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 so/is not so; clearly the two instantiations are performed by different subsystems.

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

Fig. 28. Two local interactions ($sid_1.ei_1$:$sid_2.ei_2$)

pre-cond2*,*post-cond2*,*vitality2 defined as the homonymous slots for simple system but where the elementary interaction is replaced by the local interaction.

Fig. 29. Local interaction and state observer (*sid.ei*:*so*)

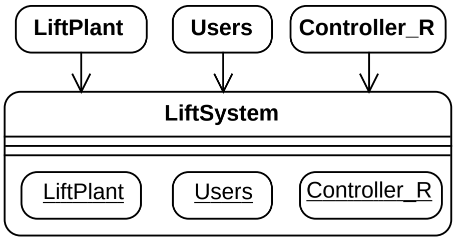


Fig. 30. LiftSystem: Parts and Constituent Features

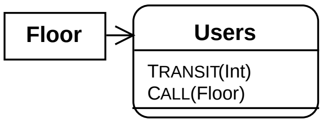


Fig. 31. Users Specification

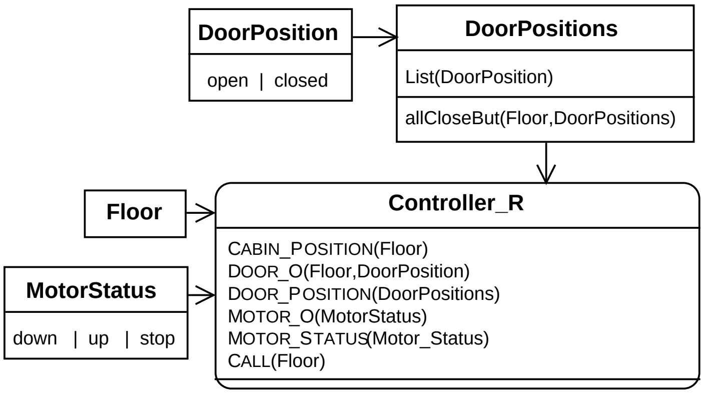


Fig. 32. Controller_R Specification

that a user that enters the lift will eventually leave it. The specification of Controller_R is in Fig. 32. Also in this case we do not have properties, because the requirements concern only its effect on the context, and not its precise behaviour.

The lift system is closed, in the sense that it does not interact with its outside world, or better possibly interactions do not concern the requirement that we have to describe; thus the elementary interaction compartment is empty. No state observer different from those observing the states of the subsystems is needed, and so also the other compartment is empty. Thus, the only constituent features of the lift systems are of kind local interactions, so to give its properties we have just to fill cells of the form "Local interaction" and "Two local interactions", whose ***local-global*** parts will be always empty.

All the local interactions with the same name of different subsystems are synchronized. We give just an example of such properties, the one concerning calling the cabin.

Users.CALL(f) **synchronized with** Controller_R.CALL(f)

Instead the local interaction DOOR_POSITIONS is synchronized with many of the kind Door_Position

LiftPlant.Door_Position($ground$,dps_1, ... LiftPlant.Door_Position(top,dps_{10}),
synchronized with Controller_R.Door_Positions(dps_1 :: ... :: dps_{10})

The last property is a post condition concerning the local interaction Users.Call(f)

if Users.Call(f) **happen then in any case eventually**
LiftPlant.$cabin\_position(f)$ **and**
LiftPlant.$motor\_status(stop)$ **and**
LiftPlant.$door\_position(f)$ $= open$

4.2.4 Casl-Ltl *View*

Here we present the Casl-Ltl [22] corresponding version of our property-oriented specification of structured systems introduced before in Sect. 4.2.1, and whose form is abstractly/conceptually presented in Fig. 21 and 24, and whose visual presentation is reported in Fig. 22. The only difference with the case of the simple system of Sect. 3.2.4 is that now we use generalized lts, however Casl-Ltl offers 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** inf stands for **sorts** st, lab, inf
pred $\_\_ : \_\_ \xrightarrow{\_\_} \_\_ : inf \times st \times lab \times st$

Let *Spec* be a property-oriented specification of structured systems having the form described in Fig. 21, 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*.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 ElInteraction$_1$, ..., ElInteraction$_k$ are the specifications of their elementary interactions;
- *Spec*.e-features $= \{ei_1, \ldots, ei_n\}$ are the 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*.

spec LocalInteraction =
ElInteraction$_1$ **and** ... **and** ElInteraction$_k$ **and** Ident **then**
free type $subElInteraction ::= \_(elInteraction_1) \mid \ldots \mid \_(elInteraction_k)$
%% disjoint union of the elementary interaction types of the subsystems

free type *localInteraction* ::= $<\_,\_>$ (*ident*, *subElInteraction*)

spec *Spec*.name =
FINITESET[ELINTERACTION] **and** FINITESET[LOCALINTERACTION] **and**
DS_1 **and** ... **and** DS_j **and** SSP_1 **and** ... **and** SSP_k **then**
dsort *st* **label** *FinSet*[*elInteraction*] **info** *FinSet*[*localInteraction*]
ops so_1.name : $st \times so_1$.argTypes $\rightarrow so_1$.resType %% state observers
...
so_m.name : $st \times so_m$.argTypes $\rightarrow so_m$.resType
ss_1.id : $st \rightarrow ss_1$.type %% observers of the subsystem states
...
ss_r.id : $st \rightarrow ss_r$.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. 3.2.4, a transition property *cond* is transformed into $inf: S \xrightarrow{l} S' \Rightarrow cond'$, where *cond'* is obtained from *cond* by adding S as extra argument to each non-primed state observer, by adding S' as extra argument to each primed state observer, and by replacing each atom of the form "*eIn* **happen**" with $eIn \in l$, and each atom of the form "*lIn* **happen**" with $lIn \in inf$.

4.3 Constructive specifications

4.3.1 The specifications characteristics

The constructive specification method for structured system that we present is similar in many respects to the property-oriented one introduced in Sect. 4.2. We first determine the parts and the constituent features, and that can be done as before, the only difference is that in this case we do not define any state features, because the states are fully determined by the subsystems (they are just sets of subsystem states). Then, we do not give properties but instead we define precisely the additional information, the states, the labels and the transition of the generalized lts modelling the specified structured system.

The definition of the additional information, labels and of the states is quite trivial (the additional information are sets of local interactions, the labels are set of elementary interactions, and the states are sets of subsystem states together with their identities). For what concerns the transitions, similarly to the case of simple systems, we define them by means of conditional rules stating which groups of local transitions (of the subsystems) may be executed together resulting in a global (of the structured system) transition. The general

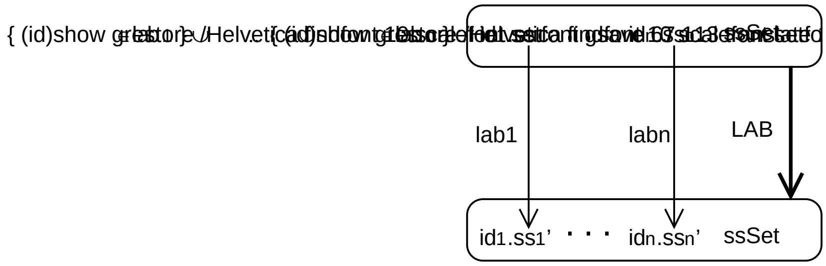


Fig. 33. Visual presentation of rules defining global transitions

form of such rules, visually depicted in Fig. 33, is

$$
\begin{array}{l}
(\#)\ \ \textbf{if}\ ss_1 \xrightarrow{lab_1} ss_1'\ \ \textbf{and}\ \ldots\ \textbf{and}\ ss_n \xrightarrow{lab_n} ss_n'\ \ \textbf{and} \\
\quad pos\text{-}cond(lab_1, \ldots, lab_n, LAB, ssSet)\ \ \textbf{then} \\
\quad \{(id_1.x) \mid x \in lab_1\} \cup \ldots \cup \{(id_n.x) \mid x \in lab_n\} : \\
\qquad \{id_1.ss_1, \ldots, id_n.ss_n\} \cup ssSet \xrightarrow{LAB} \{id_1.ss_1', \ldots, id_n.ss_n'\} \cup ssSet
\end{array}
$$

stating that if $ss_1 \xrightarrow{lab_1} ss_1'$, ..., $ss_n \xrightarrow{lab_n} ss_n'$ are local transitions (i.e., of subsystems), and their labels satisfy some condition, then we have a global transition (of the structured system), in whose source state we distinguish the subsystems subjected to a local transition (with source states ss_1, ..., ss_n) and those staying idle (with source states in $ssSet$), that is labelled by LAB, and that whose additional information is the set of the involved local interactions ($\{(id_1, x) \mid x \in lab_1\} \cup \ldots \cup \{(id_n, x) \mid x \in lab_n\}$). The condition may concern also the possible transitions of the subsystem staying idle ($ssSet$) and the label of the resulting global transition LAB.

To help the specifier write such complex rules our method proposes to proceed in the following way. Recalling that the labels of the systems consists of sets of elementary interaction, we first determine which groups of local interactions must be executed together (*synchronized sets*). Then, we assume that a set of local transitions can build a global transition if and only if their corresponding local interactions can all be combined into disjoint synchronized sets. Furthermore, we add to each synchronized set a global elementary interaction (of the structured system), expressing the interaction with the external environment resulting from the execution of the set of local interactions.

The transition shown in Fig. 19 may be, for example, motivated by the following synchronized sets (*null* stands for no global elementary interaction)
({ A.e, B.e}, *null*), ({ A.f }, X), ({ B.g,C.h }, Y)
Notice that A.f is not synchronized with other local interactions.

Thus the rule (#) can be specialized in the following way (we present it in four parts followed by the corresponding comments):

$$(1)\ \ \textbf{if}\ ss_1 \xrightarrow{lab_1} ss_1'\ \ \textbf{and}\ \ldots\ \textbf{and}\ ss_n \xrightarrow{lab_n} ss_n'\ \ \textbf{and}$$

(2) $\{(id_1.x) \mid x \in lab_1\} \cup \ldots \cup \{(id_n.x) \mid x \in lab_n\} =$
$linSet_1 \cup \ldots \cup linSet_k$ **and**
(3) $(linSet_1, EI_1)$, ..., $(linSet_k, EI_k)$ are synchronized sets **then**
(4) $linSet_1 \cup \ldots \cup linSet_k$:
$\{id_1.ss_1, \ldots, id_n.ss_n\} \cup ssSet \xrightarrow{\{EI_1,\ldots,EI_k\}} \{id_1.ss'_1, \ldots, id_n.ss'_n\} \cup ssSet$

If
(1) $ss_1 \xrightarrow{lab_1} ss'_1, \ldots, ss_n \xrightarrow{lab_n} ss'_n$ are local transitions (i.e., of subsystems)
(2) each local transition determines a set of local interactions
$\{(id_i.x) \mid x \in lab_i\}$ $i = 1, \ldots, n$
(2)&(3) local interactions may be combined to form synchronization sets $linSet_1$, ..., $linSet_k$
(3) accompanied by the global elementary interactions EI_1, ..., EI_k
then
(4) we have a global transition (of the structured system), in whose source state we distinguish the subsystems subject to a local transition (with source states ss_1, ..., ss_n) and those staying idle (with source states in $ssSet$), that is labelled by the global elementary interactions EI_1, ..., EI_k, and that whose additional information is the set of the involved local interactions $(linSet_1 \cup \ldots \cup linSet_k)$.

This rule states that any group of local transitions (of the subsystems) whose elementary interactions build a group of synchronized sets may result in a global transition (of the structured system) [free mode]. If we need to express that at most one of such set is selected at each step [inter mode], the above rules may be changed by setting $k = 1$. If instead, we need that only maximal groups of subsystem transitions are selected (i.e., the subsystems in $ssSet$ cannot perform some other synchronized set of local moves) [max mode], then we add in the premise the condition that $ssSet$ cannot move. Thus, to define the above rules, we only need to define which are the synchronized sets of local interactions and the mode in which the subsystem transitions are picked up (for the case of maximal parallelism we need also to define also an additional predicate *noMove*, again by conditional rules).

The set of synchronized local interactions may be in turn defined by conditional rules (synchronization rules) having the form
(+) **if** $pos\text{-}cond(\{sid_1.ei_1,\ldots,sid_k.ei_k\},EI)$ **then**
$(\{sid_1.ei_1,\ldots,sid_k.ei_k\},EI)$ is a synchronized set

The form of the constructive specifications of structured systems is summarized in Fig. 34.

The parts and the constituent features of a constructive specification of a structured system are visually presented as for the property-oriented specifications, see Fig. 22, but now in the name compartment we put an indication

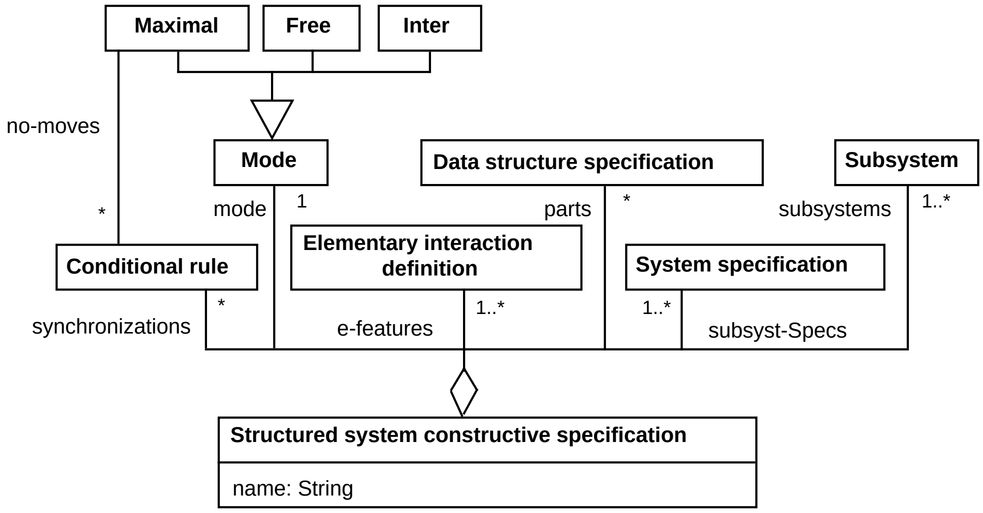


Fig. 34. Structured System Constructive Specification

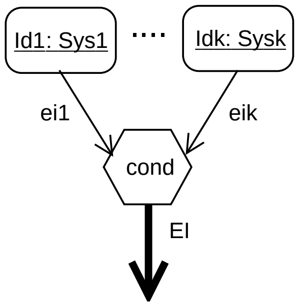
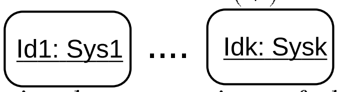


Fig. 35. Visual presentation of a synchronization rule

of the mode of the system (inter, max, or free). A rule defining the synchronized sets as in (+) above will be visually represented as in Fig. 35, where 

is a fragment of the configuration of the system. The visual presentations of all the synchronization rules may be then put together building an oriented graph by collecting all rounded boxes depicting the same subsystems. The rules defining the *noMove* predicate, if any, are visually depicted as a standard predicate, see Sect. 5.3.

4.3.2 Example: a Constructive Specification of a lift controller

As example we specify the design of another controller of the list system that differently from that of Sect. 3.3.2 consists of two processes, one taking care of the doors (detecting wrong doings and making sure that each order sent to them is correctly executed) and a main part taking care of interacting with the users, the motor and the cabin.

First we specify the two composing subsystems DoorHandler and Main, respectively in Fig. **??** and Fig. 36, and after the structured system D_Controller in

Fig. 38.

4.3.3 Casl-Ltl *View*

Here we present the Casl-Ltl [22] corresponding version of our constructive specifications of structured systems introduced before in Sect. 4.3, and whose form is abstractly/conceptually presented in Fig. 4.3.1.

Let *conSpec* be a constructive specification of structured systems having the form described in Fig. 34, and assume that

- *conSpec*.parts $= \{ds_1, \ldots, ds_j\}$ are the parts, and $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ the corresponding Casl-Ltl specifications;
- *conSpec*.subsyst-Specs $= \{ssp_1, \ldots, ssp_k\}$ are the subsystem specifications, and $\mathrm{SSP}_1, \ldots, \mathrm{SSP}_k$ are the corresponding Casl-Ltl specifications;
- *conSpec*.e-features $= \{ei_1, \ldots, ei_n\}$ are the elementary interactions;
- *conSpec*.s-features $= \{so_1, \ldots, so_m\}$ are the state observers;
- *conSpec*.subsystems $= \{ss_1, \ldots, ss_r\}$ are the subsystems.

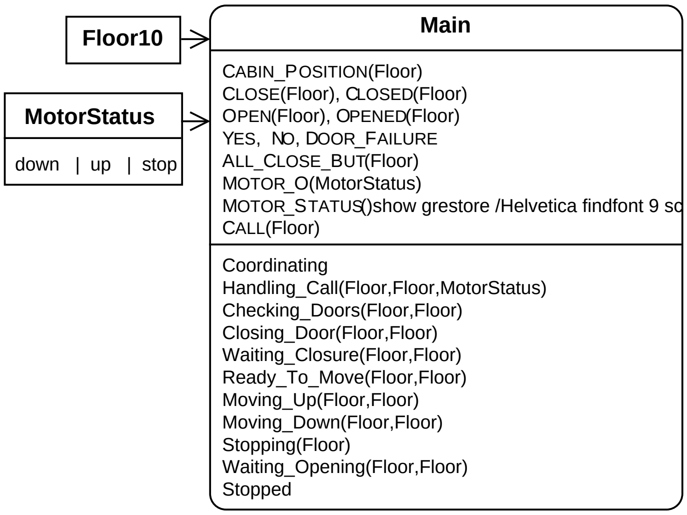
Below we give the Casl-Ltl specification corresponding to *conSpec*.

ElInteraction and LocalInteraction have been defined in Sect. 4.2.4.

spec StructState =
    SSP_1 **and** ... **and** SSP_k **then**
free {
    generated type *structState* ::=
        ss_1.name : _(ss_1.type) | ... | ss_r.name : _(ss_r.type) |
        _ || _ : *structState* × *structState*
    axioms
        $ss1 \mid\mid ss2 = ss2 \mid\mid ss1$
        $ss1 \mid\mid (ss2 \mid\mid ss3) = (ss1 \mid\mid ss2) \mid\mid ss3$
        $ss \mid\mid ss = ss'$
} end

spec *Spec*.name =
    FiniteSet[ElInteraction] **and** FiniteSet[LocalInteraction] **and**
    StructState **and** DS_1 **and** ... **and** DS_j **then**
free {
    dsort *structState* **label** *FinSet*[*elInteraction*] **info** *FinSet*[*localInteraction*]
    preds *isSynchronizedSet* : *FinSet*[*localInteraction*] × *FinSet*[*elInteraction*]
        noMove : *structState*
    axioms
        formulae corresponding to the synchronization rules, see below
    %% a rule of the form below for all n, $1 \leq n \leq r$ and all k, $1 \leq k \leq n$
        $ss_1 \xrightarrow{lab_1} ss'_1 \;\wedge\; \ldots \;\wedge\; ss_n \xrightarrow{lab_n} ss'_n \;\wedge$

Parts and constituent features



Behaviour

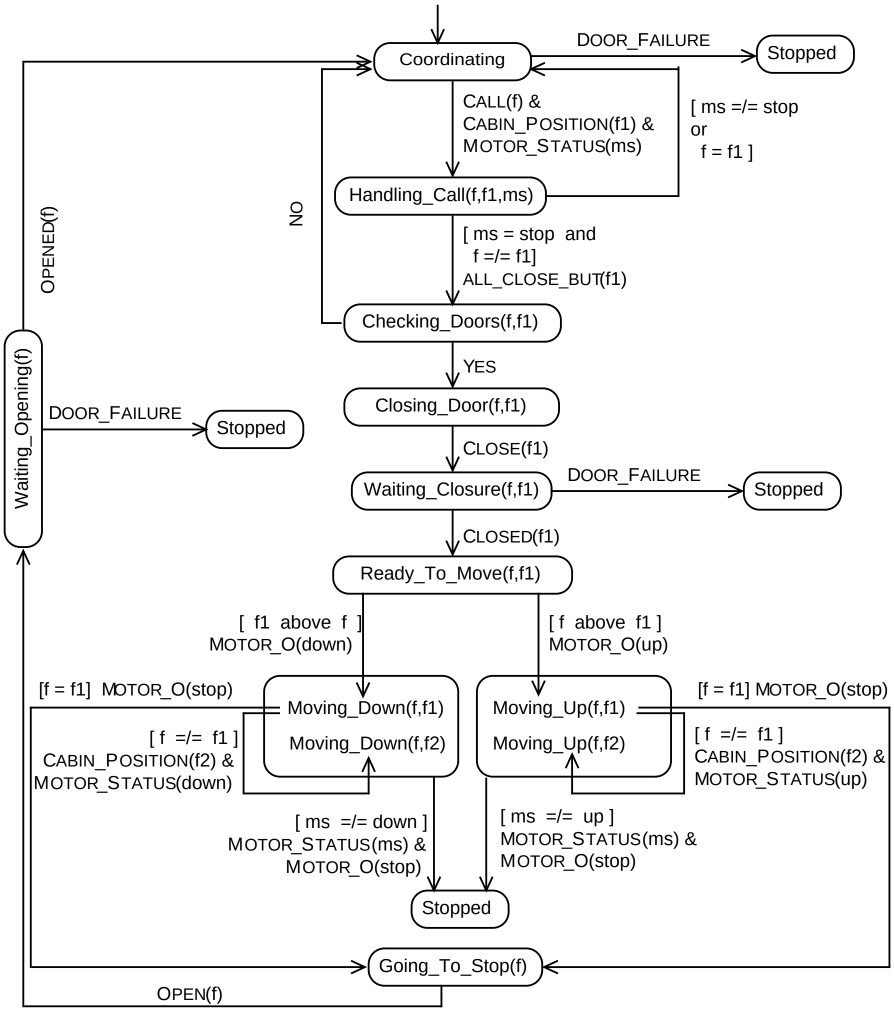
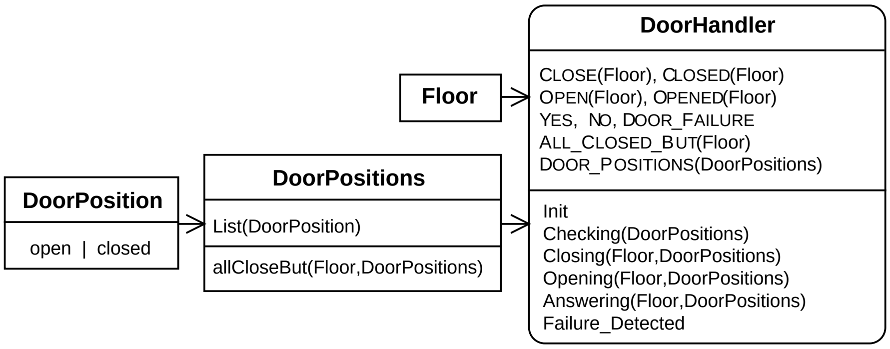


Fig. 36. Main: Specification

Parts and constituent features



Behaviour

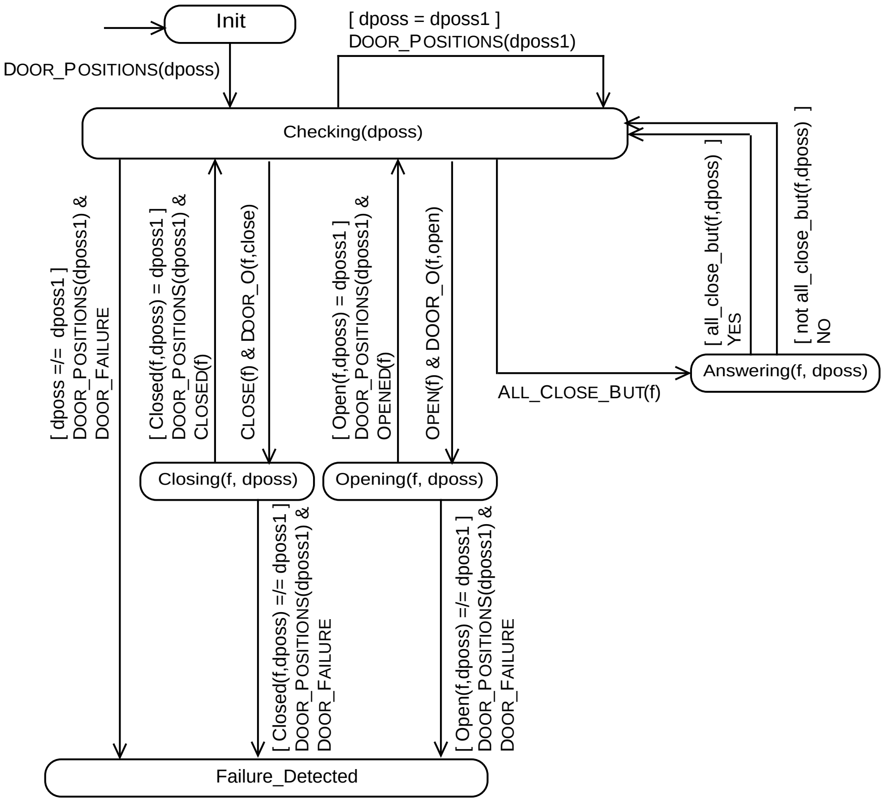


Fig. 37. DoorHandler: Specification

$\{(ss_1.\mathsf{Id}, ei) \mid ei \in lab_1\} \cup \ldots \cup \{(ss_n.\mathsf{Id}, ei) \mid ei \in lab_n\} =$
$linSet_1 \cup \ldots \cup linSet_k \ \wedge$
$isSynchronizedSet(linSet_1, EI_1) \ \wedge \ \ldots \ \wedge$
$isSynchronizedSet(linSet_k, EI_k) \ \Rightarrow$
$ss_1 \mid\mid \ldots \mid\mid ss_n \mid\mid ssSet \xrightarrow{\{EI_1,\ldots,EI_k\}} ss'_1 \mid\mid \ldots \mid\mid ss'_n \mid\mid ssSet$

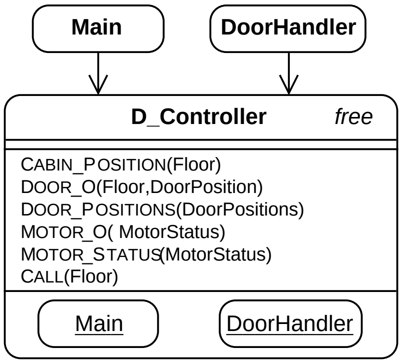
%% axioms corresponding to the conditional rules defining *noMove*,
%% if any, following Sect. 5.3.3

....

} end

A synchronization rule as (+) in Sect. 4.3 corresponds to the following CASL-LTL

Parts and constituent features



Synchronization rules

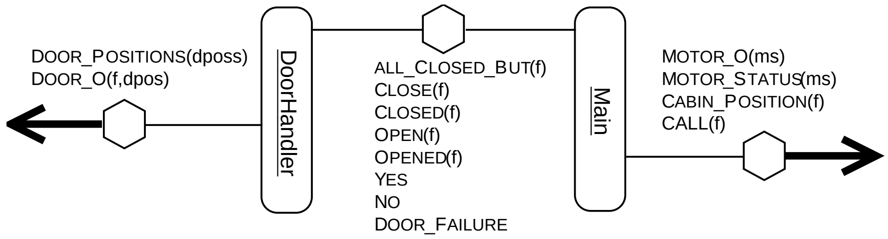


Fig. 38. D_Controller: Specification

formula

$$pos\text{-}cond(\{(Id1, ei1), \ldots, (Idk, eik)\}, EI) \;\Rightarrow$$
$$isSynchronizedSet(\{(Id1, ei1), \ldots, (Idk, eik)\}, EI)$$

The above specification correspond to the mode Free. If the mode is Inter, then the rules defining the transition are changed by setting $k = 1$. If it is instead Max you add in the premise the condition $noMove(ssSet)$.

5 Specification of Data Structures

5.1 Data Structure Items

A *data structure* consists of a set of values, some constructors for denoting them, and 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 (denoted by a "?" symbol). Constructors and operations may be constants (considered as 0-ary operations), and constants are always defined (or total).

We present two specification methods for data structures (property-oriented

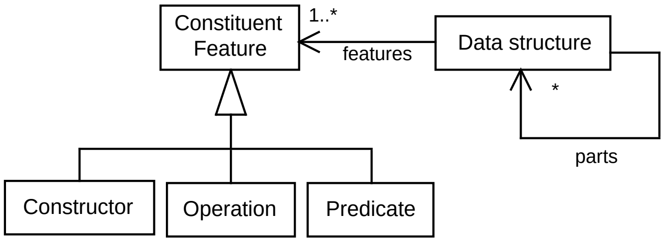


Fig. 39. Data Structure Item

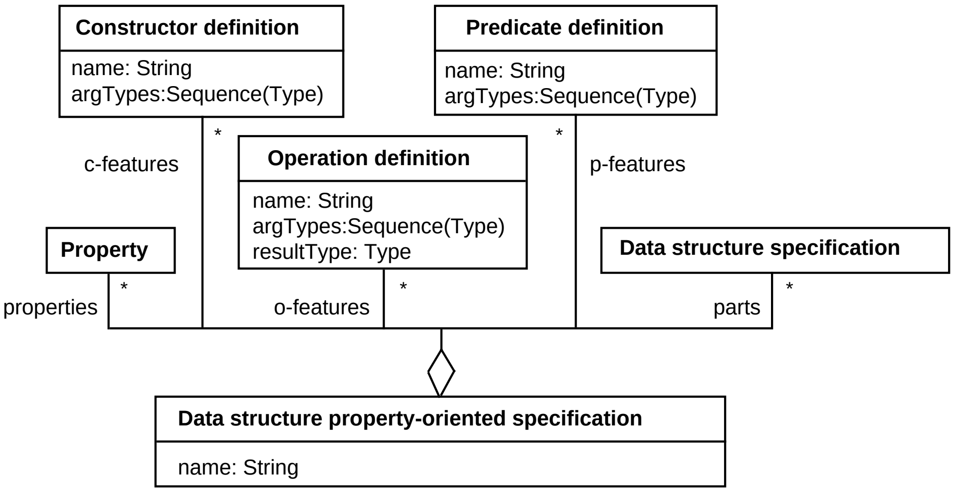


Fig. 40. Data Structure Property-Oriented Specification

and constructive) following the conceptual schema of Sect. 2.1. Both methods are based on the idea that 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. As shown in Fig. 39, data structures may be characterized by their constructors, operations and predicates, and so they will have three corresponding kinds of constituent features.

Let us recall that in the property-oriented case, the listed constituent features are necessary (but not restrictive), while in the constructive case they are fully described (and thus restrictive).

5.2 Property-oriented specifications

The property-oriented specification method for data structures we propose is a specialization of GPSm introduced in Sect. 2.2. After having identified the parts and constituent features (Fig. 40 with the visual presentation in Fig. 41), the properties are expressed using the cell filling approach.

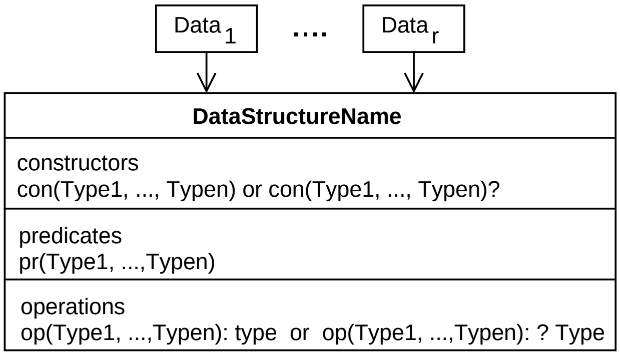


Fig. 41. Visual presentation of a Data Structure: parts and constituent features

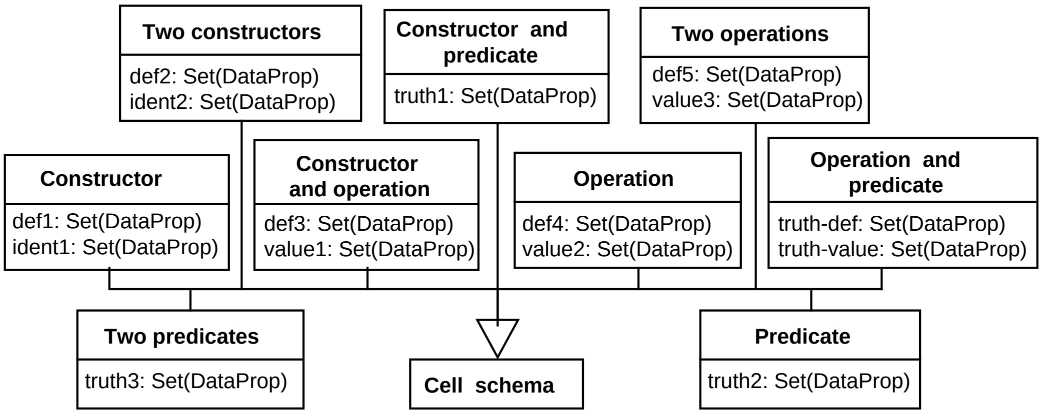


Fig. 42. Data Structure: Cell Schemata

5.2.1 Finding properties (cell fillings)

We present the schemata for all relevant types of cells for the case of data structures, summarized in Fig. 42, in Fig. 43, ..., 51. The constituents of data structures are of three kinds, constructors, predicates and operations, and so we have to consider nine kinds of cells. 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 as $t = t'$ in the case t is defined implicitly requires also that t' must be defined. In order to avoid the undefined case, the premises of the many properties used in the cell fillings require the definedness of all the elements involved in the property, thus their form is

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

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

whenever everything is defined *cond*

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

cond

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

ident1 The values represented by *con* are/are not identified with those represented by other constructors.

whenever everything is defined *cond*

where *cond* includes atoms of the form $con(arg) = \ldots$

Fig. 43. Constructor (*con*)

def2 Conditions on the relationships between the definedness of con_1 with that of con_2 (required only for partial constructors):

cond

where *cond* includes atoms of the form **def**($con_1(arg_1)$) and of the form **def**($con_2(arg_2)$)

ident2 The values represented by con_1 are/are not identified with values represented by con_2:

whenever everything is defined *cond*

where *cond* includes atoms of the form $con_1(arg_1) = con_2(arg_2)$

Fig. 44. Two constructors (con_1:con_2)

def3 Conditions on the definedness of the application of *op* to values represented by *con* (required only for partial operations):

cond

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

value1 Conditions on the values returned by the application of *op* to values represented by *con*:

whenever everything is defined *cond*

where *cond* includes terms of the form $op(con(arg))$

Fig. 45. Constructor and operation (*con*,*op*)

truth1 Conditions on the truth of *pr* over the values represented by *con*:

whenever everything is defined *cond*

where *cond* includes atoms of the form $pr(con(arg))$

Fig. 46. Constructor and predicate (*con*,*pr*)

---

def4 Conditions on the definedness of op (required only for partial operations):

$cond$

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

value2 Conditions on the values returned by op:

whenever everything is defined $cond$

where $cond$ includes terms of the form $op(arg)$

Fig. 47. Operation (op)

---

def5 Conditions on the relationships between the definedness of op_1 with that of op_2 (required only for partial operations):

$cond$

where $cond$ includes atoms of the form $\mathbf{def}(op_1(arg_1))$ and of the form $\mathbf{def}(op_2(arg_2))$

value3 Conditions on the values returned by op_1 with that returned by op_2:

whenever everything is defined $cond$

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

Fig. 48. Two operations (op_1,op_2)

---

truth-def Conditions on the relationships between the truth of pr and the definedness of op (required only for partial operations):

whenever everything is defined $cond$

where $cond$ includes atoms of the form $pr(arg_1)$ and of the form $\mathbf{def}(op(arg_2))$

truth-value Conditions on the relationships between the truth of pr and the values returned by op:

whenever everything is defined $cond$

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

Fig. 49. Operation and predicate (op,pr)

---

truth2 Conditions on the truth of pr:

whenever everything is defined $cond$

where $cond$ includes atoms of the form $pr(arg)$

Fig. 50. Predicate (pr)

truth3 Conditions on the relationships between the truth of pr_1 and that of pr_2:

whenever everything is defined *cond*

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

Fig. 51. Two predicates (pr_1,pr_2)

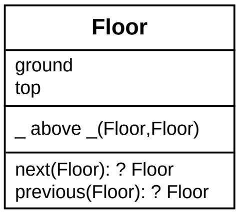


Fig. 52. Floor: parts and constituent features

5.2.2 *Example: a Property-Oriented Specification of* Floor

We specify the Floor data structure used in the Lift examples. Fig. 52 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. The properties given below were worked out using our cell filling approach, then redundant properties were removed and the result was reorganized.

- There exists a ground and a top floor, and they are different.
  $ground \neq top$
- *above* is total order over the floors with *top* as maximum and *ground* as minimum.
  if $f \neq ground$ **then** f *above* *ground*
  if $f \neq top$ **then** *top* *above* f
  $f_1 = f_2$ **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 $next(f)$.
  def$(next(ground))$
  not def$(next(top))$
  def$(next(f))$ **iff** *top* *above* f
  whenever everything is defined
  $next(f)$ *above* f **and**
  not exists $f_1 \bullet (next(f)$ *above* f_1 **and** f_1 *above* $f)$
  whenever everything is defined $next(previous(f)) = previous(next(f)) = f$
- Properties on *previous* are similar to those of *next*, and are given in Appendix C.

5.2.3 CASL *View*

Here we present the CASL [13] corresponding version of our property-oriented specification of data structures introduced in Sect. 5.2 where Fig. 40 displays its conceptual form, and Fig.41 its visual presentation.

Let *Spec* be a property-oriented specifications of data structures having the form described in Fig. 40, 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 "?").

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 $\rightarrow$ op_1.resType
    ...
    op_m.name : op_m.argTypes $\rightarrow$ 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.

5.3 Constructive specifications

5.3.1 The specifications characteristics

The constructive specification method for data structure is similar in many respects to the property-oriented one introduced in Sect. 5.2. First, we determine the used data structures (the parts), the constructors, the predicates and the operations (the constituent features). Then, we state when the constructors and operations are defined, which constructors represent the same values,

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

which are the values returned by the operations and when the predicates hold. That is done by means of groups of conditional rules defined in the following.

Recall (cf. Sect. 1.2, the spirit of the CASL free construct) that our constructive specifications by conditional rules follow the basic principle that "something is true in the specified item if and only if it can be deduced by the rules". Thus a partial constructor/operation is undefined except if there is a rule explicitly stating that it is defined. In the same way, a predicate does not hold except if there is a rule explicitly stating that it holds.[14] Let us also remind that constructors, operations and predicates are strict.

def-con For each constructor *con*, a set of conditional rules expressing when *con* is defined:

if $\textit{pos-cond}(\textit{pat})$ (**and** $_{t \text{ is a term appearing in } \textit{pat}}$ $\mathbf{def}(t)$) **then** $\mathbf{def}(\textit{con}(\textit{pat}))$

where *pos-cond* is a conjunction of positive atoms where the operations of the data structure do not appear[15], and *pat* are expresssions built only by constructors and variables

The above restrictions on the form of the rules should help avoid to implicitly introduce the definedness of other combinations of constructors and should help the specifiers have a clear idea of when the constructor is defined.

con-ident For each constructor *con*, a set of conditional rules expressing in which cases it represents values that may be represented also by using other constructors:

whenever everything is defined
if $\textit{pos-cond}(\textit{pat}_1,\textit{pat}_2)$ **then** $\textit{con}(\textit{pat}_1) = \textit{pat}_2$

where *pos-cond* is a conjunction of positive atoms where the operations of the data structure do not appear[16], and $\textit{pat}_1$, $\textit{pat}_2$ are expresssions built only by constructors and variables.

Note that all constructors represent different values except if there is a rule explicitly stating that they are the same. If this part is empty all constructors denote different values.

def-op For each operation *op*, a set of conditional rules expressing when *op* is defined:

if $\textit{pos-cond}(\textit{pat})$ (**and** $_{t \text{ is a term appearing in} \textit{pat}}$ $\mathbf{def}(\mathrm{t})$) **then**
$\mathbf{def}(\textit{op}(\textit{pat}))$

where *pos-cond* is a conjunction of positive atoms, and *pat* are expresssions built only by constructors and variables.

op-val For each operation *op*, conditional rules stating which are the values

[14] In a 2-valued logics, we do not address the issue of undefinedness of a predicate since it means it does not hold.
[15] To our knowledge, this "constructors only" constraint needs to be relaxed in some exceptional cases, as our example in Sect. 5.3.2.
[16] Ibid.

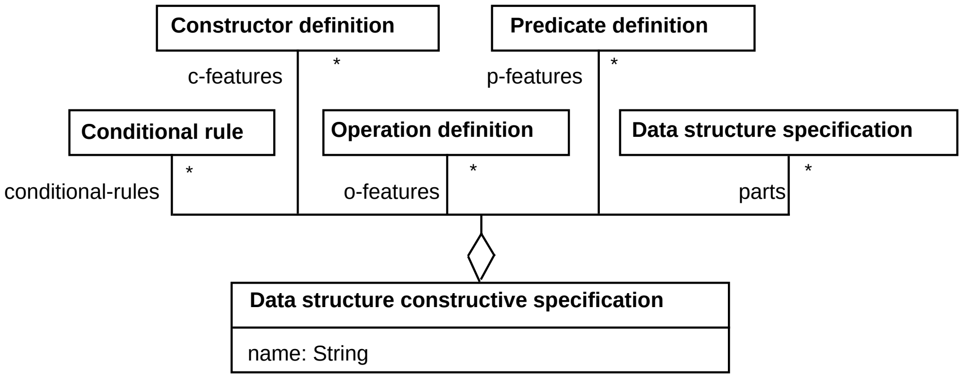


Fig. 53. Data Structure Constructive Specification

returned by *op*:

whenever everything is defined
if *pos-cond*(*pat*,*expr*) **then** *op*(*pat*) $=$ *expr*
where *pos-cond* is a conjunction of positive atoms, and *pat* are expresssions built only by constructors and variables.

pr-truth For each predicate *pr*, a set of conditional rules stating when *pr* holds:

whenever everything is defined if *pos-cond*(*pat*) **then** *pr*(*pat*)
where *pos-cond* is a conjunction of positive atoms, and *pat* are expresssions built only by constructors and variables.

The form of the constructive specifications of data structure is summarized in Fig. 53 (details of the constructor, predicate and operation definitions are as in Fig. 40).

The visual presentation of the parts and the constituent features of a data structure constructive specification is the same as for the property-oriented ones, see Fig. 41.

The conditional rules belonging to a ***def-con*** slot are visually presented as follows. First we group together all rules whose consequences have the same form; then each group, say, e.g., **if** $cond_1$ **then def**(*con*(*pat*)), ..., **if** $cond_K$ **then def**(*con*(*pat*)), will be visually presented by **def**(*con(pat)*) **if** $\left[\begin{array}{l} cond1 \\ \cdots\cdots \\ condK \end{array}\right.$.

Similarly, we represent the rules belonging to a slot ***def-op***; whereas each group of rules belonging to a slot ***pr-truth*** having the same consequence will be presented by *pr(pat)* **holds if** $\left[\begin{array}{l} cond1 \\ \cdots\cdots \\ condK \end{array}\right.$.

For what concerns the rules belonging to a ***op-val*** slot, we first group together all rules whose consequences (they are equations) have the same left side; then

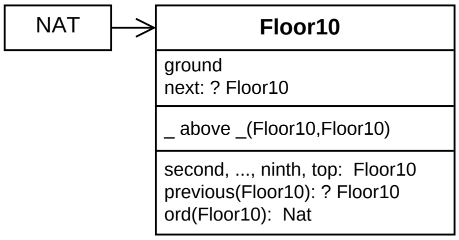


Fig. 54. Floor10: parts and constituent features

each group, say, e.g.,
if $cond_1$ **then** $op(pat) = expr_1, \ldots,$ **if** $cond_K$ **then** $op(pat) = expr_K$,

will be then visually presented by $op(pat) = \begin{array}{|l} exp1 \textbf{ if } cond1 \\ \hline \cdots\cdots \\ \hline expK \textbf{ if } condK \end{array}$.

Similarly, we represent the conditional rules in a ***con-ident*** slot.

The visual ordering of the rules should help detect lacking or overlapping cases, which may lead, e.g., to define two different values of an operation for the same argument. As usual, we then drop repeated formulae, after having checked the absence of contradictions, and slightly rearrange the others to improve readability.

5.3.2 Example: a Constructive Specification of Floor10

Here we consider the Floor example Floor10 (used in Sect. 3.3.2 and Sect. 4.3.2) with a maximum of ten floors. Fig. 54 shows the constructors, *ground* and *next*, the predicate *above*, and the operations *second*, ..., *ninth*, *top*, *previous* and *ord*. The conditional rules defining the various constituents of Floor10 are given below, since there are quite simple, we do not give a visual presentation.

- Definedness of constructors (only *next* is partial) $\mathbf{def}(next(f))$ **if** $ord(f) < 10$ [17]
- Operations *second*, ..., *ninth* are just shortcuts.
  $second = next(ground)$
  ...
  $top = next(ninth)$
- Definition of the partial operation *previous*
  $\mathbf{def}(previous(f))$ **if** $ord(f) > 1$
  $previous(next(f)) = f$ **if** $\mathbf{def}(next(f))$
- Definition of the total operation *ord*
  $\mathbf{def}(ord(f))$
  $ord(ground) = 1$

[17] Here, the "constructors only" constraint (cf. footnote 15) is relaxed for legibility sake, and we allow the use of *ord*.

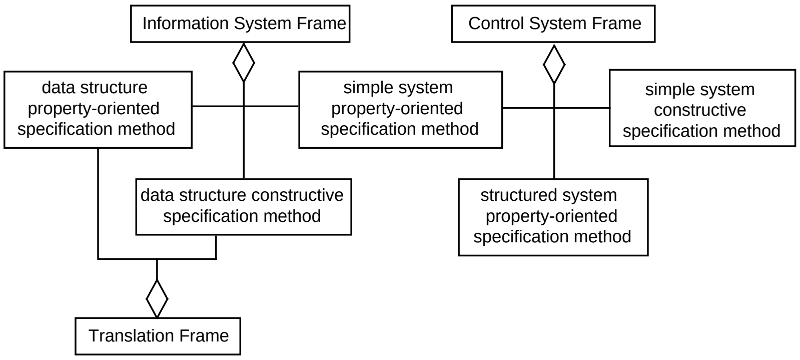


Fig. 55. How the basic FG specification methods are used for problem frames

$\quad ord(next(f)) \;= ord(f) + 1$ **if def**$(next(f))$

– Definition of the predicate $above next(f)$ $above$ f

$\quad$ **if** f_1 $above$ f_2 **and** f_2 $above$ f_3 **then** f_1 $above$ f_3

It is interesting to compare this with the Floor example given in Sect. 5.2.2 (and Fig. 52). Note that the constructors are not the same to start with, and we do not need to express the same properties.

5.3.3 Casl *View*

The Casl corresponding version of our constructive specifications of data structure is quite similar to that for the property-oriented case; the only difference is that in this case the resulting algebraic specification has an initial semantics (this results from using the **free** construct) instead of a loose one.

spec *conSpec*.name =
$\quad$ DS$_1$ **and** ... **and** DS$_j$ **then**
free {
$\quad$ **type** *conSpec*.name ::= con_1.name(con_1.argTypes)? | ... | con_n.name(con_n.argTypes)
$\quad$ **ops** op_1.name : op_1.argTypes →? op_1.resType
$\quad\quad$ ...
$\quad\quad$ op_m.name : op_m.argTypes → op_m.resType
$\quad$ **preds** $\quad pr_1$.name : pr_1.argTypes
$\quad\quad$ ...
$\quad\quad\quad$ pr_p.name : pr_p.argTypes
$\quad$ **axioms**
$\quad\quad$ *formulae corresponding to the conditional rules*
} **end**

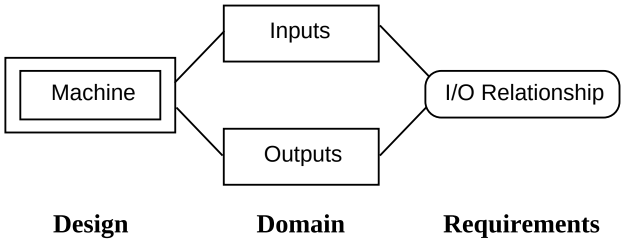


Fig. 56. Translation Frame Schema

6 Applying our FG Specification Methods to Classes of Systems ("Problem Frames")

In Fig. 55 we show by means of a simple UML class diagram how the basic FG specification methods introduced before can be used to support the specification of the most relevant problem frames of M. Jackson (see [16, 17]).

6.1 Translation Frame

The translation frame domain, see Fig. 56, is given by the Inputs and the Outputs, the requirements are described by the input/output relationship, I/O Relation, and the design is the Machine. An example of a translation frame problem is a compiler, where the Inputs are the source programs, the Outputs are the executable programs, the I/O Relation is given by the language and computer semantics, and the Machine is the compiler.

To develop a software system matching the frame "Translation" using our method means to specify the various components of the frame as presented in Fig. 56. Technically, the domain is a pair of data structures, Inputs and Outputs, which should be specified by constructive specifications following Sect. 5.3; the I/O Relation may be seen as a data structure having Inputs and Outputs as parts and a predicate representing I/O Relation, and can be specified by a property-oriented specification following Sect. 5.2. Finally the design is seen as a partial function (or a sequential program or an algorithm) *Tran* that associates an element of Outputs with an element of Inputs, and so it is again a data structure having Inputs and Outputs as subparts and an operation *Tran*, that be specified by a constructive specification following Sect. 5.3.

6.2 Information System Frame

To quote [16], "In its simplest form, an information system provides information, in response to requests, about some relevant real-world domain of interest."

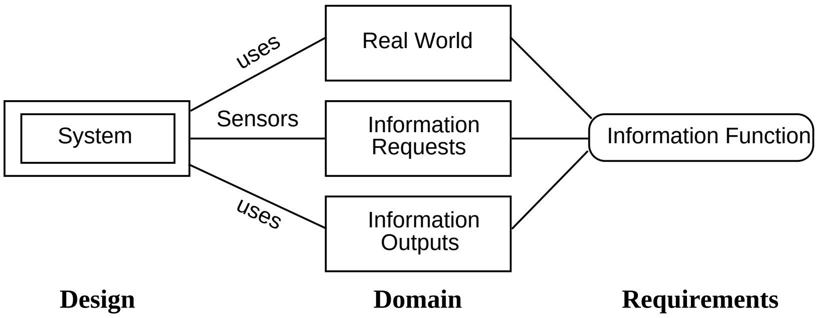


Fig. 57. Information System Frame Schema

The information system frame domain is given by the RealWorld, the InformationRequests and the InformationOutputs, the requirements are described by the InformationFunction, and the design is the System. The RealWorld may be a *static* domain (e.g., if the system provides information on Shakespeare's plays), or a *dynamic* domain (e.g., "the activities of a currently operating business" [16]). Here we consider information system frames with a dynamic domain, so "The RealWorld is *dynamic* and also *active*." [16].

To develop a software system matching the frame "Information System" using our method means to specify the various components of the frame as presented in Fig. 57.

The RealWorld is a dynamic system, thus it is specified following Sect. 3.2 but with some peculiarity corresponding to its particular nature. All its elementary interactions correspond to signal, throughout some sensors, something happening inside it that is relevant for the information system. The InformationRequests and the InformationOutputs are two data structures that are specified by simply giving their constructive specifications, following Sect. 5.3.

Clearly, InformationFunction to produce the correct information output, given an information request, should take into account the real world, more precisely its current state and also its past history as known by the values communicated by the sensors. Technically, these information may be abstractly represented by a sequence of sets of elementary interactions (recall that each elementary interaction corresponds to some value communicated by a sensor). So we can define it by a standard data structure, HISTORY, whose definition is given once. Thus to specify InformationFunction means to give a constructive specification of a particular data structure following Sect. 5.3; such data structure must have as subparts InformationRequests, InformationOutputs and HISTORY, and an operation InformationFunction : $History \times InformationRequets \rightarrow InformationOutputs$.

To design an "Information System" means to design the System, a dynamic system interacting with the RealWorld (by receiving the values sensed by the sensors), and with the users (by receiving the information requests and sending

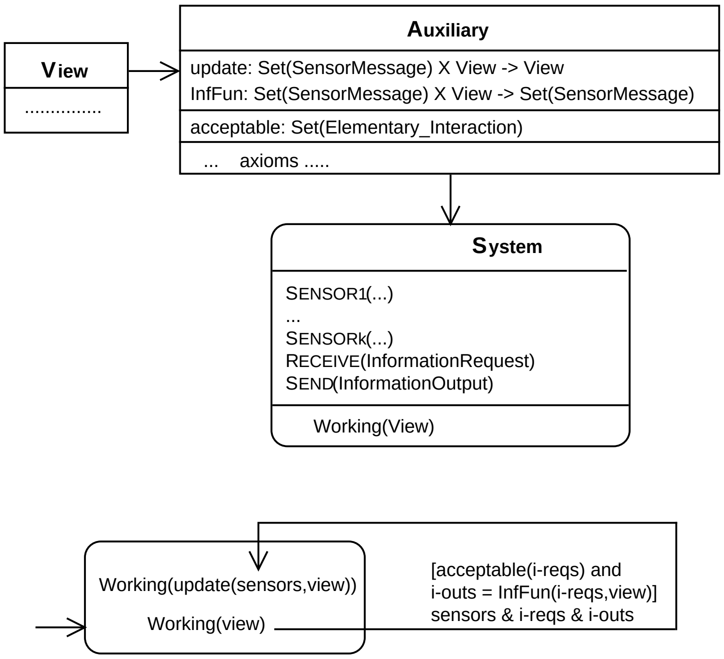


Fig. 58. System Specification: elementary constituents, parts and behaviour

back the information outputs). We assume that the System:

- keeps a view of the actual situation and of the past history of the RealWorld,
- updates it depending on the info received by the sensors,
- decides which information requests from the users to accept in each instant,
- answers to such requests with the appropriate information outputs using its view of RealWorld.

We assume also that the System can immediately receive in a correct way any message sent by the sensors of the RealWorld and that the information requests are handled immediately.

The design of the System will be specified by a constructive specification of a simple system following Sect. 3.3. The elementary interactions are those corresponding to receive some info on some sensors, to receive information requests and the send out information outputs. The unique state constructor is *Working* parameterized by a real world view. This specification is visually reported in Fig. 58, where VIEW is a data structure describing in an appropriate way (i.e., apt to permit to answer to all information requests) the System's views of the possible situations of the RealWorld.

Thus to specify the System it is sufficient to give:

- the specification VIEW;
- the axioms defining the operation *update* describing how the System updates its view of the RealWorld when it detects some events;
- the axioms defining the predicate *acceptable* describing which sets of re-

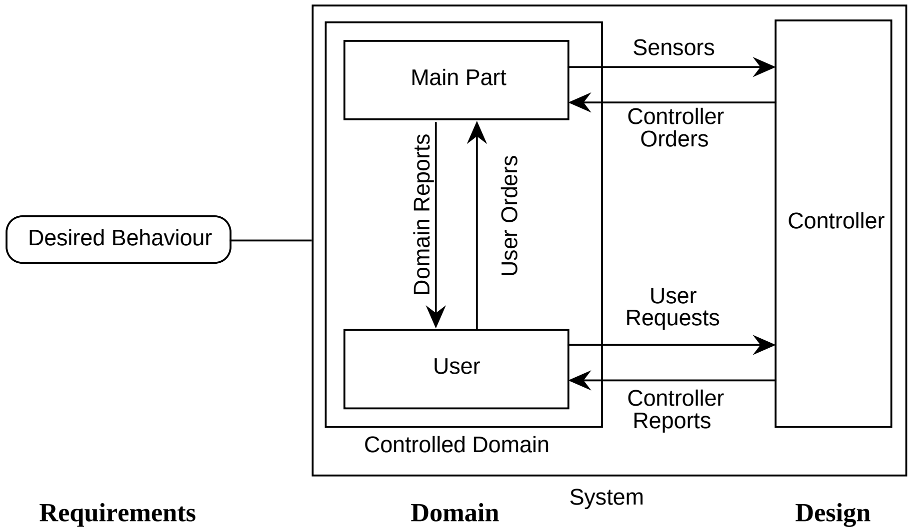


Fig. 59. Control System Frame Schema

quests may be accepted simultaneously by the System;
- the axioms defining the operation *Inf_Fun* describing what is the result of each information request depending on the System's view of the the Real-World situation.

That means to give a constructive specification of a data structure, AUXILIARY in Fig. 58, having as parts InformationOutputs, InformationRequests, VIEW and the predicate *acceptable* and the operations *update* and *Inf_Fun*.

6.3 Control System Frame

We report the general schema of the control system frame following Jackson's book [16] in Fig. 59. The controlled domain may include the users of the thing controlled by the software that we are going to develop, thus we call the "controlled thing" the *main part* (shortly indicated as MainPart in the following). Thus the first step is to see precisely how the considered problem matches the "control system frame".

To use our FG specification methods to support the whole development of a software system matching the frame "Control System with User" means to specify System and Desired Behaviour in Fig. 59; clearly some parts of this specification correspond to the requirements, some other to the domain, and the remaining to the design.

Technically, System is a structured system and Desired Behaviour is a set of additional properties over it. We specify System following Sect. 4.2, clearly taking into account its peculiarity, and splitting its properties in those desired (Desired Behaviour) and in those corresponding to its intrinsic nature (the others). Now, we list the peculiar features of System.

System has three subsystems, whose types are respectively User, MainPart and Controller.

System is closed (i.e., no interactions with the external world, and thus the label of its transitions will be always the empty set of elementary interactions), for this reason we drop the elementary interaction part in its specification.

The only state observers are the selectors of the states of the MainPart and of the User. We do not consider the observer corresponding to the state of Controller, because we will never express properties about it, but we will just fully specify when specifying the Controller.

The elementary interactions of the three subsystems are classified into six categories listed below, and the interactions of any particular category belong to the two subsystems to whom they are connected in Fig. 59; e.g., Sensors are interactions of both the MainPart and of the Controller.

Sensors or better the elementary interactions corresponding to sending/receiving the data collected by the sensors present on the MainPart (precisely MainPart sends and Controller receives).

Controller Orders or better the elementary interactions corresponding to sending/receiving orders to modify MainPart (precisely Controller sends and MainPart receives).

User Requests or better the elementary interactions corresponding to sending/receiving the requests of the User to for Controller about the wanted behaviour of the MainPart (precisely User sends and Controller receives).

Controller Reports or better the elementary interactions corresponding to sending/receiving the reports on what is going on in the MainPart sent (precisely Controller sends and User receives).

User Orders or better the elementary interactions corresponding to sending/receiving orders from the User to modify MainPart (precisely User sends and MainPart receives).

Domain Reports or better the elementary interactions corresponding to sending/receiving the reports on what is going on in the MainPart (precisely User sends and MainPart receives).

The three subsystems behave in a truly parallel way and cooperate only by performing simultaneously the shared interactions of the above six categories. Thus the ***synchr1*** and the ***local-global1*** properties of System are standard and do not depend on the particular case.

The used data structures, the elementary interactions of the subsystems and the architecture diagram of System are collected in a unique visual presentation, sketched in Fig. 60, where D1, ..., Dr are the used data structures.

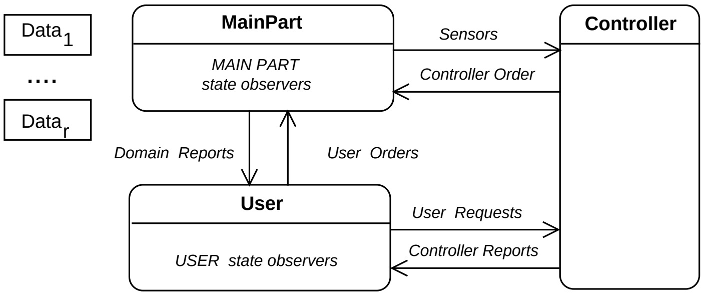


Fig. 60. Visual presentation of the structure of a control frame

The specification of the MainPart is obtained by specializing the method for property-oriented[18] specifications of simple systems of Sect. 3.2.

Some parts of the specification of MainPart have been already determined (e.g., the elementary interactions, already determined together those of the other subsystems), many are standards (e.g., the incompatibility between elementary interactions whose kind is different, as a user order and a controller order), and other have a particular form (e.g., the properties about a Controller order, where only the ***post-cond1*** slot is not empty).

Similarly, also the specification of the User is obtained by specializing the method for simple system of Sect. 3. Let us note that, while concentrating on properties that are relevant to the system, often this specification will be very short (especially in the case of a human user). However, it may reflect some rules of the directions for use. Indeed, the system to be developed may be guaranteed under the condition that the directions for use are followed by the user.

The Desired Behaviour, that is the final aim of the system that we have to develop, i.e., the system requirements, consists technically of a set of particular properties on System of Fig. 59.

- ***post-cond1*** properties for all local moves User: UR with UR user request
- ***pre-cond1*** and ***vitality1*** properties for all local moves User: CR with CR controller report
- properties on the state observers corresponding to the states of the MainPart and of User
- properties on the relationships between the state observer corresponding to the states of the MainPart and the one to the states of User

The Controller (i.e., the design part of the frame) should be specified by a constructive specification following the method proposed in Sect. 3.3 if it is a simple system or the one proposed in Sect. 4.3 otherwise; but with the

[18] In the case that we fully know how MainPart, for example, when it is a mechanical/ electromechanical appliance, we may instead give a constructive specification.

particularity that

7 Requirement Specification of an Internet Based Lottery Application

7.1 The problem

We have to develop an application ALL (ALgebraic Lottery) to handle algebraic lotteries. The lotteries are said "algebraic" because the tickets are numbered by integer numbers, the winners are determined by means of an *order* over such numbers, and a client buys a ticket by selecting its number. Whenever a client buys a ticket, he gets the right to another free ticket, which will be given at some future time, fully depending on the lottery manager decision. The number of a free ticket is generated by the set of the numbers of the already assigned tickets following some *law*. Thus a lottery is characterized by an *order* over the integers determining the winners and a *law* for generating the numbers of the free tickets. To guarantee the clients of the fairness of the lottery, the order and the law, expressed rigorously with algebraic techniques, are registered by a lawyer before the start of any lottery.

The application will be Internet based, thus the tickets will be bought and paid on-line using credit cards with the help of an external service handling them. Possible clients must register with the lottery system to play; and clients access the system in a session-like way. An external service takes care of the authentication of the clients. The winners are informed by email messages, and email is used also to inform the registered clients of the start and of the end of the various lotteries.

FG *Requirement Specification*

FG *Requirement Specification: The technique*

We use our FG method for property oriented specifications of structured systems (see Sect. 4.2) to specify the requirements for ALL. Indeed, ALL together with the external entities interacting with it, named in the following *context entities* (e.g., the manager and the email system) is a structured system, which we name SYSTEM.

ALL and all the context entities will be modelled by simple systems, because there is no need to investigate the internal structure of the context entities and of ALL. Indeed, for the former, only how they interact with ALL is relevant, and for what concerns ALL, to give it a structure in terms of cooperating subsystems means to fix already in the requirement phase an architecture for it, mixing the early design phases with the requirement one.

SYSTEM is a closed system, i.e., it cannot interact with its external world, and thus it has no elementary interactions. Moreover, all possible cooperations among its subsystems are binary ones between ALL and one context entity, and do not result in an elementary interaction of SYSTEM.

ALL is the most relevant of the subsystems of SYSTEM, and its specification is more complex of those of the other ones. Its properties are exactly the requirements for the application that we have to develop.

Notice that using our specification method we express all the requirements for the application to be developed (ALL), but also what we either assume or know about the context entities, thus making clear which is the context of the application.

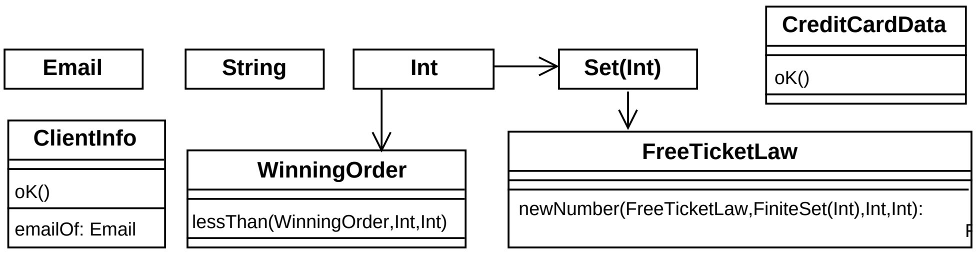
Because the specification of SYSTEM is quite complex, to make it more readable

- we collect all the specifications of data structures appearing in the specification of SYSTEM (parts of it and of its subsystems) together in a unique diagram Data View, and assume that they are implicitly added as parts whenever needed;
- The diagram showing the parts and constituents features of SYSTEM and the one showing the cooperations among its subsystems are collected together in a Context View, indeed such diagrams show the entities appearing in the context of the application and how it interacts with them. The specifications of such context entities are given separately.
- we define auxiliary/additional state observers;
- we use in properties a derived if-then-else logical combinator:
  if *cond* **then** $cond_1$ **else** $cond_2$ stands for
  (**if** *cond* **then** $cond_1$) **and** (**if not** *cond* **then** $cond_2$);
- when all the elementary interactions of a system are mutually pairwise incompatible, we just write *all incompatible* in the elementary interaction compartment.
- Only elementary interactions with the same name and the same arguments may synchronize; and thus it is sufficient to decorate any cooperation icon by a list of elementary interaction names. To further improve the readability, an elementary interaction owned by both two subsystems, say S1 and S2, which will be synchronized will be named either S1_XX or S2_XX depending on which of the two subsystem is the responsible of the resulting cooperation.

FG Requirement Specification: The Artifact

Here we report only the Data View, the Context View and the specification of one of the context enties (the authorization service) and of the group of requirements about the registration of a client, the remaining one may be found in the Appendix D.

Data View



lessThan(*wo*,_,_) is a total order over the integer, for any winning order *wo*.

$lessThan(wo,n,n)$

if $lessThan(wo,n_1,n_2)$ **and** $n_1 \neq n_2$ **then** $lessThan(wo,n_2,n_1)$

if $lessThan(wo,n_1,n_2)$ **and** $lessThan(wo,n_2,n_3)$ **then not** $lessThan(wo,n_1,n_3)$

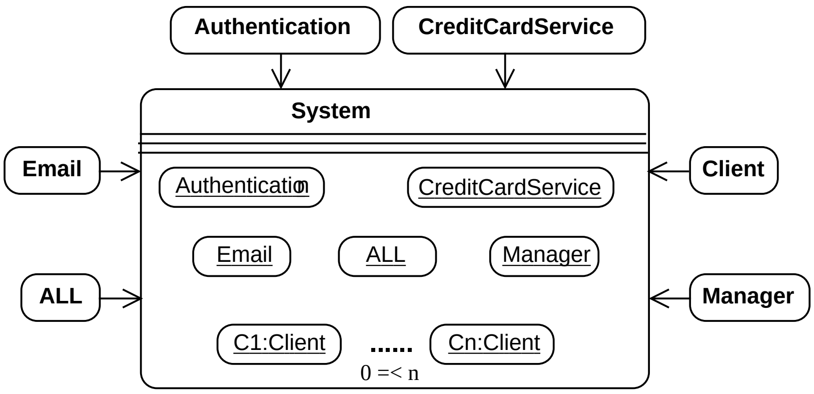
For each *ftl*, *newNumber*(*ftl*,*ns*,*d*,*m*) returns a set containing *m* integer numbers, between -*d* and *d*, and not belonging to *ns*.

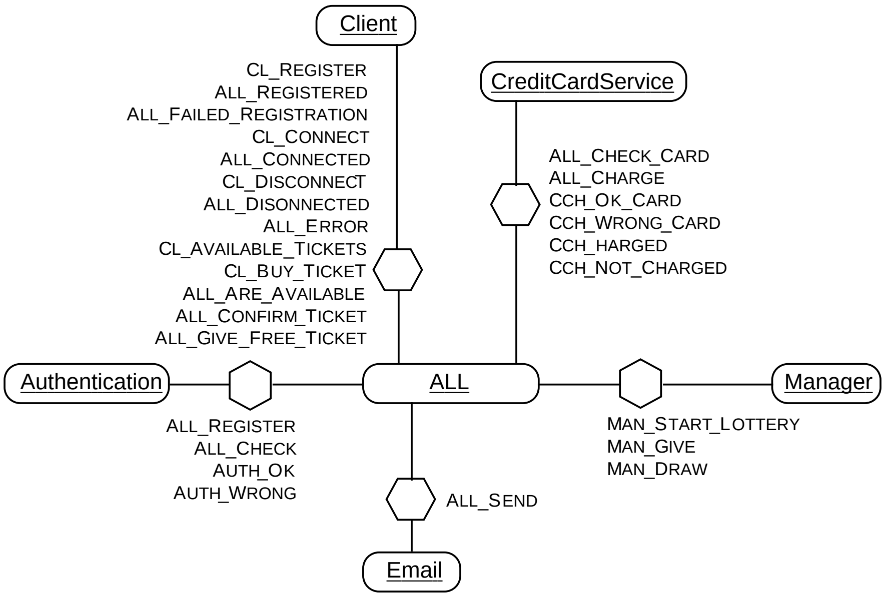
if $newNumber(ftl,ns,d,m) = ns'$ **then**

$size(ns') = m$ **and** $ns \cap ns = \{\}$ **and**

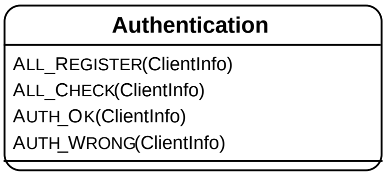
for all x $\bullet$ **if** $x \in ns'$ **then** $x \leq d$ **and** $-d \leq x$

Context View





Auhtentication Service Specification



The authentication service cannot send out simultaneously
a positive and a negative answer to a request to check a client.
AUTH_OK(ci) **incompatible with** AUTH_WRONG(ci)

The authentication service is always ready to accept a request to register a person.
at least in one case next ALL_REGISTER(ci) **happen**

The authentication service never confirms a registration of a client.

The authentication service is always ready to accept a request to check a person.
at least in one case next ALL_CHECK(ci) **happen**

The authentication service after receiving a request to check a client will inform
that either is ok or not.
if ALL_CHECK(ci) **happen then in any case eventually**
AUTH_OK(ci) **happen or** AUTH_WRONG(ci) **happen**

The Requirements for the Application ALL

AL_L
MAN_START_LOTTERY(WinningOrder,FreeTicketLaw,Int) MAN_GIVE(Int) MAN_DRAW ALL_SEND(Email,String) ALL_CHECK_CARD(CreditCardData) ALL_CHARGE(CreditCardData,Int) CCH_OK_CARD(CreditCardData) CCH_WRONG_CARD(CreditCardData) CCH_HARGED(CreditCardData,Int) CCH_NOT_CHARGED(CreditCardData,Int) CL_REGISTER(ClientInfo,CreditCardData) ALL_REGISTERED(ClientInfo,CreditCardData) ALL_FAILED_REGISTRATION(ClientInfo,CreditCardData) CL_CONNECT(ClientInfo) ALL_CONNECTED(ClientInfo) CL_DISCONNECT(ClientInfo) ALL_DISONNECTED(ClientInfo) ALL_ERROR(ClientInfo) CL_AVAILABLE_TICKETS(ClientInfo) CL_BUY_TICKET(ClientInfo,Int) ALL_ARE_AVAILABLE(ClientInfo,FiniteSet(Int)) ALL_CONFIRM_TICKET(ClientInfo,Int) ALL_GIVE_FREE_TICKET(ClientInfo) ALL_REGISTER(ClientInfo) ALL_CHECK(ClientInfo) AUTH_OK(ClientInfo) AUTH_WRONG(ClientInfo)
registered(ClientInfo) connected(ClientInfo) dimension: Int winningOrder: WinningOrder freeTicketLaw: FreeTicketlaw owner(Int): ? ClientInfo creditCard(ClientInfo): ?CreditCardData lotteryRunning freeTickets(ClientInfo): ?Int

Auxiliary/derived state observers

assignedTickets: FINITESET(INT)

returns the set of the numbers of the tickets already assigned to some client (i.e., bought or given away by the manager).

$n \in$ *assignedTickets* **iff** **def**(*owner*(n))

availableTickets: FINITESET(INT)

returns the set of the numbers of the still available tickets.

$n \in$ *availableTickets* **iff** $n \leq$ *dimension* **and** $n \geq$ -*dimension* **and** $n \notin$ *assignedTickets*

winners: FINITESET(*Int*)

returns the set of the numbers of the winning tickets, which are the first K numbers w.r.t. the current winning order, where K is the dimension of the lottery module 5000.

size(*winners*) = *dimension mod* 5000 **and**

for all n_1, n_2 •

if $n_1 \in$ *winners* **and** *lessThan*(*winningOrder*, n_1, n_2) **then** $n_2 \in$ *winners*

A client registers with ALL

ALL cannot handle two different registration requests simultaneously.
CL_REGISTER(*ci*,*ccd*) **incompatible with** CL_REGISTER(*ci'*,*ccd'*)

When ALL receives a registration requests from a client,
if the presented personal information or credit card data are wrong, or
the client is already registered, or
there is another registered client with the same email,
then
ALL informs the client that his registration has failed
otherwise,
ALL checks the credit card with the credit card service.
If the answer from the credit card service is positive, then
ALL registers the client with the authentication service,
and informs him that has been registered;
otherwise ALL informs him that his registration has failed.

if CL_REGISTER(*ci*,*ccd*) **happen then**
if (**not** *ok*(*ci*) **or not** *ok*(*ccd*) **or** *registered*(*ci*) **or**
exists *ci'* **s.t.** *registered*(*ci'*) **and** *emailOf*(*ci*) = *emailOf*(*ci'*)) **then**
in any case next ALL_FAILED_REGISTRATION(*ci*,*ccd*) **happen**
else
in any case next ALL_CHECK_CARD(*ccd*) **happen and**
(**eventually** CCH_OK_CARD(*ccd*) **happen and**
next ALL_REGISTER(*ci*) **happen and**
next ALL_REGISTERED(*ci*,*ccd*) **happen**
or
eventually CCH_WRONG_CARD(*ccd*) **happen and**
next ALL_FAILED_REGISTRATION(*ci*,*ccd*) **happen**)

ALL is always ready to accept registration requests by the clients.
at least in one case next CL_REGISTER(*ci*,*ccd*) **happen**

If the credit card of a cleint is recorded, then the client is registered, and
the data about such card are correct.
if def(*creditCard*(*ci*)) **then** *registered*(*ci*) **and** *ok*(*creditCard*(*ci*))

The credit card of a client can never be changed.
if *creditCard*(*ci*) $\neq$ *creditCard'*(*ci*) **then** *False*

ALL receives a message about the validity of a card from the credit card service,
only when it has asked for it.
if (CCH_OK_CARD(*ccd*) **happen or** CCH_WRONG_CARD(*ccd*) **happen**) **then**
in any case sometimes ALL_CHECK_CARD(*ccd*) **happened**

There are no properties concerning ALL_REGISTER; this means that ALL assumes that the authentication service always accept its requests, and that it never sends back a confirmation of the registration.

If ALL asks the authentication service to check a client, then it must be able
to receive its answer (positive or negative).
if ALL_CHECK(*ci*) **happen then in any case eventually**
AUTH_OK(*ci*) **happen or** AUTH_WRONG(*ci*) **happen**

ALL receives a message about the identity of a client from the authentication service, only when it has asked for such info.
if (AUTH_OK(*ci*) **happen or** AUTH_WRONG(*ci*) **happen**) **then**
  in any case sometimes ALL_CHECK(*ci*) **happened**

When a client is informed that he has been registered,
  then he made a registration request, and
  now he is registered together with his credit card data, and
  he has no right to receive any free ticket.
if ALL_REGISTERED(*ci*,*ccd*) **happen then**
  in any case sometimes CL_REGISTER(*ci*,*ccd*) **happened and**
  registered'(*ci*) **and** *creditCard'*(*ci*) $=$ *ccd* **and** *freeTickets*(*ci*) $= 0$

If ALL asks the credit card service to check a card, then
  it will be able to receive its answer (positive or negative).
if ALL_CHECK_CARD(*ccd*) **happen then in any case eventually**
  CCH_OK_CARD(*ccd*) **happen or** CCH_WRONG_CARD(*ccd*) **happen**

When a client is informed that his registration request is failed,
then he made a registration request.
if ALL_FAILED_REGISTRATION(*ci*,*ccd*) **happen then**
  in any case sometimes CL_REGISTER(*ci*,*ccd*) **happened**

For any registered client is available a credit card data and
the number of free tickets to which it has the right to receive.
if *registered*(*ci*) **then def**(*creditCard*(*ci*)) **and def**(*freeTickets*(*ci*))

Only ALL, as a consequence of a request of the client, may register a client.
if not *registered*(*ci*) **and** *registered'*(*ci*) **then** ALL_REGISTERED(*ci*,*ccd*) **happen**

Once a client is registered he cannot be cancelled.
if *registered*(*ci*) **and not** *registered'*(*ci*) **then** *False*

8 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 *formally grounded* we mean methods

- which have all the good properties of those commonly used (friendly notation based on simple intuitive visual metaphors, easy to understand and to learn, relevant for real applications, providing precise and helpful guidelines, ...),
- but where any used 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.[19] We mean that the underlying concepts are reflected in the method and used as such (although they are distilled to the potential user through methodological guidelines and nice visual notations).

As a formal basis for grounding our methods we have chosen the algebraic specification language CASL [9] and its extension for behavioural/dynamic specifications CASL-LTL [22]. 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" [7]. 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" [7]. Finally, its goal is a family of related extension and restriction languages.

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 have proposed some methods for developing the basic specification/modelling blocks for 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 address two kinds of specifications, the more abstract property-oriented ones, and the more concrete constructive ones.

To present our specification method 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 [16, 17] as a kind of workbench (see [11] for a first attempt and Sect. 6 for a summary). 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

[19] For example, a classical imperative programming language that has a formal semantics based on rewriting rules is not formally grounded on rewriting in our terms; indeed, users develop programs thinking in terms of assignments and not of rewrites.

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 see Sect. 7. 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 [20]. Interesting enough, the development of the formally-grounded specification helped finding some small errors in the UML based one and highlighted some peculiar aspects of the specified requirements that were less obvious in the UML one. On the other side, the FG specification could express all the relevant properties of the considered application, in a different style, not scenarios based and not object-oriented, but in a property oriented fashion. Another difference between the two approaches is that in the FG one any data used in the application is naturally picked up and explicitly specified (in FG, email addresses, Sect. 7, are a data structure, whereas in the UML one [5, 6] they are just strings) and so less type mismatches should be possible.[20] We have used the same case study and the two approaches in student projects, and the results were that the levels of the UML were quite different, including some very bad ones, whereas the FG specifications were quite homogeneous as regards the quality level, and quite similar. We think this is probably due to the fact that our method provides precise detailed guidelines.

Another point we would like to recall is how this work on FG methods may also result in the development of new specification/modelling techniques, perhaps based on new paradigms, to be integrated in practical common methods.[21] For example, our method is not object-oriented but *active-entities oriented* (or also agent-oriented), and offers a new kind of visual diagrams to present the behaviour of active entities (e.g., processes or agents), the behaviour diagrams (see Sect. 3.3). These new diagrams can be the basis for an extension of the UML statecharts (see [21]) overcoming their limitation to model only reactive behaviour. Also the new cell-filling technique for finding the properties, used here to find the axioms for the underlying logical-algebraic specification, may be at the basis of rigorous techniques for generating precise UML models, or for their inspection, to help to check whether all aspects of the modelled system have been considered.

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

[20] We would like to recall that an undetected type mismatch was one of the many concatenated facts causing the disaster of the Ariane 5.
[21] along the topics of http://www.dsi.unive.it/ mont2002/

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 [12]); see also [7] for further considerations on our view of the relationships between formal and practical used methods.

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 [18], for more references look at [25]).

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 [8]).

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.

We plan make further experiments concerning the use of the FG methods presented in this paper. In particular, we would like to try to develop specializations of them w.r.t. the application domain or the solved problem; for example, showing how they can be used for new problem frames, such as business automation, web applications and distributed mobile systems.

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 [10].

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 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.

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*. ACM Press, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio02a.pdf`.

[6] E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications. Technical Report DISI–TR–03–05, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03c.pdf`.

[7] E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Proc. of The 10th Anniversary Colloquium of the United Nations University International Institute for Software Technology (UNU/IIST): Formal Methods at the Crossroads from Panacea to Foundational Support. Lisbon - Portugal, March 18-21, 2002.* , Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003. To appear. The pdf version of this paper is available through anonymous ftp at ftp.disi.unige.it, in /person/ReggioG/AstesianoEtAll02a.pdf.

[8] V. Del 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'03*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003. To appear.

[9] Michel Bidoit and Peter Mosses. *CASL User Manual*, 2002. http://www.brics.dk/Projects/CoFI/.

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

[11] 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.

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

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

[14] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.

[15] W. Grieskamp, M. Heisel, and H. Dörr. Specifying Safety-Critical Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science, pages 88–106. Springer Verlag, Berlin, 1998.

[16] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[17] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[18] 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.

[19] P.D. 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.

[20] Rational. Rational Unified Process© for System Engineering SE 1.0. 2001.

[21] G. Reggio and E. Astesiano. An Extension of UML for Modelling the non Purely-Reactive Behaviour of Active Objects. Technical Report DISI–TR–00–28, DISI, Università di Genova, Italy, 2000. Available at ftp://ftp.disi.unige.it/person/ReggioG/ReggioAstesiano00b.pdf.

[22] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. Available at ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps.

[23] 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.

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

[25] G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa (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`.

[26] G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[27] 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/` .

[28] The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Technical report, 2001. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[29] UML Revision Task Force. *OMG UML Specification 1.3*, 1999. Available at `http://www.rational.com/media/uml/post.pdf`.

A Simple system, cell schemata

Two property cell schemata for simple systems are presented in Sect. 3.2.2 and Fig. 11. While the schemata for the "Elementary interaction" and "State observer" cells are described in Sect. 3.2.2, the others are detailed below.

In what follows *arg* stands for generic expressions of the correct types, possibly with free variables, and *cond*(*xxx*) for a generic condition where the free variables of *xxx* may appear.

---

incompatibility2 (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)

---

value2 (state property)

The results of the observation made by so_1 and so_2on 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)

---

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 only non-primed state observers may appear in $cond(arg)$ and *so* must appear in $cond(arg)$

post-cond2 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 non-primed and primed state observers may appear in $cond(arg)$ and *so*' must appear in $cond(arg)$

vitality2

If the result of the observation made by *so* on a state satisfies some condition, then any sequence of transitions starting from it will eventually contain a transition whose label contains *ei*. Note that vitality properties may have also the form "at least in a case" (instead of "in any case") or "next" (instead of "eventually").

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

Elementary interaction and state observer (ei:so)

B Example: Fragment of a Property-Oriented Specification of a Lift plant

The lift example is introduced in Sect. 1.3, and its property-oriented specification is presented in Sect. 3.2.3 where only properties on the orders are given. Here, we present the other properties on the sensors, on the cabin, and on the users entering/leaving the cabin.

On the sensors

- A sensor cannot signal two different values simultaneously.
  MOTOR_STATUS(ms_1) **incompatible with** MOTOR_STATUS(ms_2)
  if $ms_1 \neq ms_2$
  CABIN_POSITION(f_1) **incompatible with** CABIN_POSITION(f_2)
  if $f_1 \neq f_2$
- A sensor always signals the correct data.
  if MOTOR_STATUS(ms) **happen then** $motor\_status = ms$
  if CABIN_POSITION(f) **happen then** $cabin\_position = f$
- A sensor cannot break down, thus it may always be able to signal the correct value.
  at least in one case next MOTOR_STATUS($motor\_status$) **happen**
  at least in one case next CABIN_POSITION($cabin\_position$) **happen**

On the cabin

- If the motor is moving up (down), then the cabin position will change.
  if $motor\_status = up$ **then** $cabin\_position' = next(cabin\_position)$
  if $motor\_status = down$ **then** $cabin\_position' = previous(cabin\_position)$
- The cabin position changes only if the motor is working in the corresponding versus and the change corresponds to one floor.
  if $cabin\_position \neq cabin\_position'$ **then**
  ($cabin\_position' = next(cabin\_position)$ **and** $motor\_status = up$) **or**
  ($cabin\_position' = previous(cabin\_position)$ **and** $motor\_status = down$)

On the users entering/leaving the cabin

- At most one elementary interaction of kind TRANSIT may happen each time.
  TRANSIT(i) **incompatible with** TRANSIT(i') **if** i $\neq$ i'
- The users may enter/leave the cabin only if they are not too numerous the cabin is at a floor with the door open and the motor is stopped.

if $\text{T}\textsc{ransit}$(i) **happen then**
$\quad$ *users_inside*+ i ≤ 15 **and** *users_inside*+ i ≥ 0 **and**
motor_status = *stop* **and** *door_position*(*cabin_position*) = *open*

- The number of people inside the cabin changes only if and only if someone enter/leaves it.
  if $\text{T}\textsc{ransit}$(i) **happen then** *users_inside*′ = *users_inside*+i
  if *users_inside*′ = *users_inside*+i **and** i $\neq 0$ **then** $\text{T}\textsc{ransit}$(i) **happen**
- At most 15 people may be inside the cabin simultaneously.
  users_inside ≤ 15
- If the door at a floor of the cabin is open and the motor is stopped, then any appropriate number of people may enter/leave the cabin.
  if *motor_status* = *stop* **and** *door_position*(*cabin_position*) = *open* **and**
  $\quad$ *users_inside*+ i ≤ 15 **and** *users_inside*+ i ≥ 0 **then**
  $\quad$ **at least in one case next** $\text{T}\textsc{ransit}$(i) **happen**

C Example: Fragment of a Property-Oriented Specification of Floor

This example is given in Sect. 5.2.2 where the properties of *previous* given below were skipped.

- *previous* returns the floor immediately below a given one, if it exists. There is no floor between *previous*(f) and f.
  $\quad$ **def**(*previous*(*top*))
  $\quad$ **not def**(*previous*(*ground*))
  $\quad$ **def**(*previous*(f)) **iff** f *above ground*
  $\quad$ **whenever everything is defined**
  $\quad\quad$ f *above previous*(f) **and**
  $\quad\quad$ **not exists** $f_1 \bullet$ (f *above* f_1) **and** f_1 *above previous*(f))

D Fragment of the Requirement Specification of an Internet Based Lottery Application

Context Entity Specifications

Manager
MAN_START_LOTTERY(WinningOrder,FreeTicketLaw,Int) MAN_GIVE(Int) MAN_DRAW $\quad$ *all incompatible*

When the manager starts a new lottery/draws the winners/gives away free tickets, he cannot do any other things.

Email
ALL_SEND(Email,String)

The email service is always able to receive any request to send an email message.
at least in one case next ALL_SEND(*em*,*s*) **happen**

CreditCardService	
ALL_CHECK_CARD(CreditCardData)	
ALL_CHARGE(CreditCardData,Int)	
CCH_OK_CARD(CreditCardData)	
CCH_WRONG_CARD(CreditCardData)	
CCH_CHARGED(CreditCardData,Int)	
CCH_NOT_CHARGED(CreditCardData,Int)	*all incompatible*

The credit card service when receive a request or send out an answer cannot do any other things.

The credit card service after receiving a request of controlling a card, will answer with either an ok or wrong card message.
if ALL_CHECK_CARD(*ccd*) **happen then in any case eventually**
CCH_OK_CARD(*ccd*) **happen or** CCH_WRONG_CARD(*ccd*) **happen**

The credit card service is always ready to accept a request to check a card.
at least in one case ALL_CHECK_CARDALL_CHECK(*ccd*) **happen**

The credit card service after receiving a request to charge some money on a card will inform that the same has been either charged or not.
if ALL_CHARGE(*ccd*,*m*) **happen then in any case eventually**
CCH_CHARGED **happen or** CCH_NOT_CHARGED **happen**

The credit card service is always ready to accept a request to charge some amount on a card.
at least in one case ALL_CHECK_CARDALL_CHARGE(*ccd*,*m*) **happen**

Client	
CL_REGISTER(ClientInfo,CreditCardData)	
ALL_REGISTERED(ClientInfo,CreditCardData)	
ALL_FAILED_REGISTRATION(ClientInfo,CreditCardData)	
CL_CONNECT(ClientInfo)	
ALL_CONNECTED(ClientInfo)	
CL_DISCONNECT(ClientInfo)	
ALL_DISONNECTED(ClientInfo)	
ALL_ERROR(ClientInfo)	
CL_AVAILABLE_TICKETS(ClientInfo)	
CL_BUY_TICKET(ClientInfo,Int)	
ALL_ARE_AVAILABLE(ClientInfo,FiniteSet(Set))	
ALL_CONFIRM_TICKET(ClientInfo,Int)	
ALL_GIVE_FREE_TICKET(ClientInfo)	*all incompatible*

All the possible activities of a client are mutually incompatible.

The client after any request to ALL is ready to accept its answer, whatever it may be.
if CL_REGISTER(*ci*,*ccd*) **happen then in any case eventually**
ALL_REGISTERED(*ci*,*ccd*) **happen or**
ALL_FAILED_REGISTRATION(*ci*,*ccd*) **happen**
if CL_CONNECT(*ci*) **happen then in any case eventually**

ALL_CONNECTED(ci) **happen or** ALL_ERROR(ci) **happen**

if CL_AVAILABLE_TICKETS(ci) **happen then in any case eventually**
exists is **s.t.** ALL_ARE_AVAILABLE(ci,is) **happen**

if CL_BUY_TICKET(ci,n) **happen then in any case eventually**
ALL_CONFIRM_TICKET(ci,n) **happen or** ALL_ERROR(ci) **happen**

The Requirements for the Application ALL

The manager starts a new lottery

When ALL receives a request from the manager of starting a new lottery,
it cannot handle anything else.
$\textsc{Man\_Start\_Lottery}(wo,ftl,d)$ **incompatible with** ei

When ALL receives a request of starting a new lottery,
the proposed dimension is greater than 1 and a multiple of 5000, and no lottery is running.
Then, a lottery will be running, characterized by the proposed parameters
(winning order, free ticket law and dimension), no client owns a ticket, and
all registered clients will be informed by an email message of the fact.
if $\textsc{Man\_Start\_Lottery}(wo,ftl,d)$ **happen then**
  $d \geq 1$ **and** d *mod* $5000 = 0$ **and**
  not *lotteryRunning* **and** *lotteryRunning'* **and**
  dimension' $= d$ **and** *winningOrder'* $= wo$ **and** *freeTicketLaw'* $= ftl$ **and**
  for all $n\bullet$ **not def**$(owner'(n))$ **and**
  for all $ci\bullet$ **if** $ci \in$ *registered* **then**
    in any case next $\textsc{All\_Send}(emailOf(ci),$"Started New Lottery"$)$

When no lottery is running ALL must be able to accept a request from the manager,
with appropriate parameters, of starting a new one.
if not *lotteryRunning* **and** $d \geq 1$ **and** d *mod* $5000 = 0$ **then**
  at least in one case next $\textsc{Man\_Start\_Lottery}(wo,ftl,d)$ **happen**

The dimension of the lottery must be a multiple of 5000 and greater than 1.
dimension mod $5000 = 0$ **and** *dimension* ≥ 1

The dimension, the winning order and the free tickets law of the lottery
changes only when a new lottery is started.
if *dimension* $\neq$ *dimension'* **then**
  exists *wo*, *ftl* **s.t.** $\textsc{Man\_Start\_Lottery}(wo,ftl,dimension')$ **happen**
if *winningOrder* $\neq$ *winningOrder'* **then**
  exists *d*, *ftl* **s.t.** $\textsc{Man\_Start\_Lottery}(winningOrder',ftl,d)$ **happen**
if *freeTicketLaw* $\neq$ *freeTicketLaw'* **then**
  exists *d*, *wo* **s.t.** $\textsc{Man\_Start\_Lottery}(wo,freeTicketLaw',d)$ **happen**

A lottery becomes running only after the manager started it.
if not *lotteryRunning* **and** *lotteryRunning'* **then**
  exists *d*, *wo*, *ftl* **s.t.** $\textsc{Man\_Start\_Lottery}(wo,ftl,d)$ **happen**

The manager gives away some free tickets

When ALL receives a request from the manager of giving away some free tickets,
it cannot handle anything else.
MAN_GIVE(*nbil*) **incompatible with** *ei*

When ALL receives a request to give away *nbil* free tickets,
a lottery is running, there are still available at least *nbil* tickets, and
at least half of the tickets have been already assigned.
Then, ALL selects *nbil* clients having the right to receive a free ticket, and
each of them will get a ticket, whose number is determined by the current free ticket law.

if MAN_GIVE(*nbil*) **happen then**
  lotteryRunning **and** $size(availableTickets) \geq nbil$ **and**
  $size(assignedTickets) \geq dimension$ **and**
  exists *luckies* **s.t.** $size(luckies) = nbil$ **and**
    for all *ci* • **if** $ci \in luckies$ **then**
    $freeTickets(ci) > 0$ **and**
    exists unique $n \in freeTicketLaw(dimension, assignedTickets, nbil)$ **s.t.**
      in any case next ALL_GIVE_FREE_TICKET(*ci*,*n*) **happen**

When a lottery is running, there are still available at least *nbil*, and
at least half of the tickets have been already assigned,
ALL may receive a request to give away *nbil* free tickets.

if *lotteryRunning* **and** $size(assignedTickets) \geq dimension$ **and**
  $size(availableTickets) \geq nbil$ **then**
    at least in one case next MAN_GIVE(*nbil*) **happen**

When a client receives a free ticket,
the number of free tickets which he has the right to get decrease by 1, and
he is the owner of such ticket.

if ALL_GIVE_FREE_TICKET(*ci*,*n*) **happen then**
  $freeTickets'(ci) = freeTickets(ci)$ -1 **and**
  $owner(n) = ci$

If the number of free tickets o which a client has the right to get decrease by 1, then
he received a free ticket.

if $freeTickets'(ci) = freeTickets(ci)$ -1 **then**
  exists *n* **s.t.** ALL_GIVE_FREE_TICKET(*ci*,*n*) **happen**

The manager draws the winners

When ALL receives a request from the manager of drawing the winners,
it cannot handle anything else.
MAN_DRAW **incompatible with** *ei*

When ALL receives from the manager a requests to draw the winners, then
a lottery is running and there are no more available tickets.
Then, ALL informs by an email message the owners of the winning tickets
(as many as the dimension of the lottery module 5000,
whose numbers are the first w.r.t. the current winning order).
Finally, the lottery is terminated, and
ALL informs all the registered clients, always by an email message, of the end of the lottery.
if MAN_DRAW **happen then**
  lotteryRunning **and** *availableTickets* $= \{\}$ **and**
  (**for all** *n* • **if** *n*∈ *winners* **then**
    in any case next ALL_SEND(*emailOf*(*owner*(*n*)),"Won Prize") **happen**)
  and not *lotteryRunning'* **and**
  (**for all** *ci* • **if** *ci*∈ *registered* **then**
    in any case eventually ALL_SEND(*emailOf*(*ci*),"Lottery terminated") **happen**)

When a lottery is running and there are no more available tickets,
ALL must be able to receive from the manager a requests to draw the winners.
if *lotteryRunning* **and** *availableTickets* $= \{\}$ **then**
  at least in one case next MAN_DRAW **happen**

A lottery ends only when the winners are drawn by the manager.
if *lotteryRunning* **and not** *lotteryRunning'* **then** MAN_DRAW **happen**

A client connects to ALL

When ALL receives a connection request by a client,
if the client is registered and not already connected, then
ALL checks his identity with the authentication service. Thus,
if the answer of the authentication service is positive, then
ALL informs the client that he is connected,
otherwise
ALL informs the client that his request is an error;
otherwise
ALL informs the client that his request is an error.

if CL_CONNECT(*ci*) **happen then**
if *registered*(*ci*) **and not** *connected*(*ci*) **then in any case**
next ALL_CHECK(*ci*) **happen and eventually**
AUTH_OK(*ci*) **happen and next** ALL_CONNECTED(*ci*) **happen**
or
AUTH_WRONG(*ci*) **happen and next** ALL_ERROR(*ci*) **happen**
else
in any case ALL_ERROR(*ci*) **happen**

ALL is always ready to accept connection requests by the clients.
at least in one case next CL_CONNECT(*ci*) **happen**

When a client is informed that he is connected, then
he made a connection request and now his connection is registered.
if ALL_CONNECTED(*ci*) **happen then**
in any case sometimes CL_CONNECT(*ci*) **happened and** *connected'*(*ci*)

When a client is informed that his request was an error, then
he made either a connection or a disconnection request or
a request to show the available tickets or to buy a ticket.
if ALL_ERROR(*ci*) **happen then in any case sometimes**
CL_CONNECT(*ci*) **happened or** CL_DISCONNECT(*ci*) **happened or**
CL_AVAILABLE_TICKETS(*ci*) **happened or**
CL_BUY_TICKET(*ci*,*n*) **happened**

A client disconnects from ALL

When ALL receives a disconnection requests from a client,
    if the client is connected, then
        ALL informs him that he has been disconnected
    otherwise
        ALL informs him that his request was an error.

if CL_DISCONNECT(ci) **happen then**
    if *connected*(ci) **then**
        in any case next ALL_DISCONNECTED(ci) **happen**
    else
        in any case next ALL_ERROR(ci) **happen**

ALL is always ready to accept disconnection requests by the clients.
at least in one case next CL_DISCONNECT(ci) **happen**

When a client is informed that his request was an error, then he made either disconnection request or some other request. Notice again that the fact that the same error message is used by ALL to answer to two different requests is made explicitly clear by our FG method, while is neglected by use-case driven approaches, the error message appears in two different use cases.

A connected client must be also registered.
if *connected*(ci) **then** *registered*(ci)

Only ALL, as a consequence of a request of the client, may connect a client.
if not *connected*(ci) **and** *connected'*(ci) **then** ALL_CONNECTED(ci) **happen**

Only ALL, as a consequence of a request of the client, may disconnect a client.
if *connected*(ci) **and not** *connected'*(ci) **then** ALL_DISCONNECTED(ci) **happen**

When a client is informed that he has been disconnected, then
    he has made a disconnection request and now is disconnected.
if ALL_DISCONNECTED(ci) **happen then**
    in any case sometimes CL_DISCONNECT(ci) **happened and**
    not *connected'*(ci)

Notice, how this property precisely states that ALL cannot realize any mechanism of automatic disconnection (e.g., after some time). Similar requirements are usually neglected in classical use case base approaches.

A client checks which are the tickets still available

When ALL receives a request of showing the available tickets,
  a lottery is running and
  if the client is connected, then
    ALL will inform him of which are the available tickets
  otherwise
    ALL will inform him that his request was an error.
if $\text{CL\_AVAILABLE\_TICKETS}(ci)$ **happen then**
  lotteryRunning **and**
  if $connected(ci)$ **then**
    in any case next $\text{ALL\_ARE\_AVAILABLE}(ci, availableTickets)$ **happen**
  else
    in any case next $\text{ALL\_ERROR}(ci)$ **happen**

When a lottery is running, then
  ALL may accept requests to show the available tickets.
if *lotteryRunning* **then**
  at least in one case next $\text{CL\_AVAILABLE\_TICKETS}(ci)$ **happen**

When a client is informed that his request was an error, then he made either a request to show the available tickets or some other request.

If ALL says to a client that *tckts* are the available tickets, then
  a lottery is running, *tckts* are the available tickets, and
  the client has requested such information
if $\text{ALL\_ARE\_AVAILABLE}(ci, tkcs)$ **happen then**
  lotteryRunning **and** $tkcs = availableTickets$ **and**
  in any case sometimes $\text{CL\_AVAILABLE\_TICKETS}(ci)$ **happened**

A client buys a ticket

When a client tries to buy a ticket with number n,
a lottery is running and
if the client is connected and n is still available, then
ALL asks the credit card service to charge the sum of 1000 and
if the credit card service confirms the chargement, then
ALL confirms to the client that he has got the ticket n
otherwise
ALL informs the client that his request was an error;
otherwise
ALL informs the client that his request was an error.

if $\textsc{Cl\_Buy\_Ticket}(ci,n)$ **happen then**
lotteryRunning **and**
if $connected(ci)$ **and** $n \in availableTickets$ **then**
in any case next $\textsc{All\_Charge}(creditCard(ci), 1000)$ **happen and**
eventually
$\textsc{Cch\_Charged}(creditCard(ci), 1000)$ **happen and**
next $\textsc{All\_Confirm\_Ticket}(ci,n)$ **happen**
or
$\textsc{Cch\_Not\_Charged}(creditCard(ci), 1000)$ **happen and**
next $\textsc{All\_Error}(ci)$ **happen**
else
in any case next $\textsc{All\_Error}(ci)$ **happen**

When a lottery is running, ALL may accept requests to buy tickets.
if *lotteryRunning* **then**
at least in one case next $\textsc{Cl\_Buy\_Ticket}(ci,n)$ **happen**

When a client is informed that his request was an error, then he made either a request to buy a ticket or some other request.

When ALL confirms to a client that him bought a ticket n,
the client has asked for n, n was available, and after
the client is the owner of n, and has gained the right to another free ticket.
if $\textsc{All\_Confirm\_Ticket}(ci,n)$ **happen then**
in any case next $\textsc{Cl\_Buy\_Ticket}(ci,n)$ **happen and**
$n \in availableTickets$ **and**
$owner'(n) = ci$ **and** $freeTickets'(ci) = freeTickets(ci) + 1$

Only registered clients may own tickets.
if def$(owner(n))$ **then** $registered(owner(n))$

The number of free tickets to which a client has the right to get increases by 1,
then he bought a ticket.
if $freeTickets'(ci) = freeTickets(ci) + 1$ **then**
exists n **s.t.** $\textsc{All\_Confirm\_Ticket}(ci,m)$ **happen**

If ALL asks the credit card service to charge a sum on a card, then
it will be able to receive a message confirming or negating the operation.
if ALL_CHARGE(*ccd*,*m*) **happen then in any case eventually**
CCH_CHARGED(*ccd*,*m*) **happen or** CCH_NOT_CHARGED(*ccd*,*m*) **happen**

If ALL receives a message about the result of a request to charge a card
from the credit card service, then
it asked such chargement.
if (CCH_CHARGED(*ccd*,*m*) **happen or** CCH_NOT_CHARGED(*ccd*,*m*) **happen**) **then**
in any case sometimes ALL_CHARGE(*ccd*,*m*) **happened**