Using CASL to Specify the Requirements and the Design: A Problem Specific Approach

Christine Choppy[1] and Gianna Reggio[2]

[1] LIPN, Institut Galilée - Université Paris XIII, France
[2] DISI, Università di Genova, Italy

Abstract. In [11] M. Jackson introduces the concept of *problem frame* to describe specific classes of problems, to help in the specification and design of systems, and also to provide a framework for reusability. He thus identifies some particular frames, such as the translation frame (e.g., a compiler), the information system frame, the control frame (or reactive system frame), ... Each frame is described along three viewpoints that are application domains, requirements, and design.
Our aim is to use CASL (or possibly a sublanguage or an extension of CASL if and when appropriate) to formally specify the requirements and the design of particular classes of problems ("problem frames"). This goal is related to methodology issues for CASL, that are here addressed in a more specific way, having in mind some particular problem frame, i.e., a class of systems.
It is hoped that this will provide both a help in using, in a really effective way, CASL for system specifications, a link with approaches that are currently used in the industry, and a framework for the reusability.
This approach is illustrated with some case studies, e.g., the information system frame is illustrated with the invoice system.

1 Introduction

It is now well established that formal specifications are required for the development of high quality computer systems. However, it is still difficult for a number of practitioners to write these specifications. In this paper we address the general issue of how to bridge the gap between a problem requirements and its specification. We think this issue has various facets. For instance, given a problem, how to guide the specification process? Often people do not know where to start, and then are stopped at various points. Another facet is, given a specification language, how to use it in an appropriate way? We address these facets here through the use of M. Jackson's problem frames (successfully used in industry), which we formalize by providing the corresponding specification skeletons in the CASL language ([12, 13]).

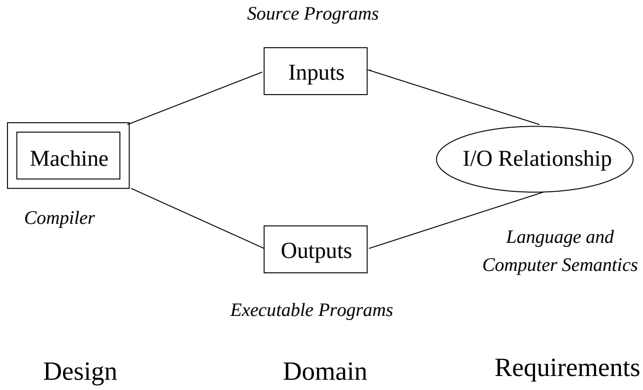
A Jackson problem frame [11] is a generalization of a class of problems, thus helping to sort out in which frame/category is the problem under study. The idea here is to provide a help to analyse software development problems and to choose an appropriate method for solving them (with the assumption that there is no "general" method). Then, for each problem frame, M. Jackson provides its expected components together with their characteristics and the way they are connected. The problem frame components always include a domain description, the requirements description, and possibly the design description. The domain description expresses "what already exists", it is an abstraction of a part of the world. The requirements description expresses "what there should be", that is what are the computer system expected concepts and facilities. The design description deals with "how" to achieve the system functions and behaviours. The problem frames identified are the translation (JSP), the information system, the reactive system (or control frame), the workpiece, and the connection frames. While these cover quite a number of applications, it may also be the case that some problems are "multiframe".

While Jackson problem frames may be used to start understanding and analysing the system to be developed, they have no formal underpinning. Our idea here is to still rely on them while providing the corresponding "specification frames". We thus provide a methodological approach to write problem specifications using the problem frames that importantly gives guidelines to *start* and to *do* the problem analysis and specification; combining these two "tools" yields a powerful approach to guide the problem understanding and the specification writing.

The issue of the choice of a formal specification language is non trivial [2]. We think that algebraic specification languages offer an adequate degree of abstraction for our needs, so we chose the latest and more general one, Casl, the Common Algebraic Specification Language ([12, 13]) developed within the Common Framework Initiative (CoFI), to express our proposed formal underpinning for problem frames. While Casl was developed to be "a central, reasonably expressive language for specifying conventional software", "the common framework will provide a family of languages" [12]. Thus, restrictions of Casl to simpler languages (for example, in connection with verification tools) will be provided as well as extensions oriented towards particular programming paradigms, e.g., reactive systems. While dealing with the translation frame (Sect. 2 and 3), Casl complies with our needs, but when moving to dynamic systems that may occur within the information system frame (Sect. 4 and 5), we propose an extension of Casl with temporal logic, named Casl-Ltl [15], based on the ideas of [3] and [7], that may be more appropriate (and will be shortly presented when used). Since the design of Casl was based on a critical selection of constructs found in existing algebraic frameworks, the reader familiar with these may feel quite "at home", while Casl offers for these convenient syntactic combinations. In this paper, we use some Casl constructs, that we introduce when they appear. While the Casl syntax and semantics are completed, some tools (e.g., parsers, libraries, ...) are being developed. In what follows we shall rely on the available library for basic data types [16].

This paper is organised as follows. In Sect. 2 and 3, we describe the translation problem frame, provide a a method to formalize it using Casl, and illustrate it on a short example that is the Unix grep utility. In Sect. 4 and 5, we work similarly on the information system problem frame and illustrate it with the invoice case study. The information system frame raises various issues, since we are dealing with bigger reactive systems. The size issue leads us to clearly identify sets of properties that need to be expressed and also to search for a legible way to present large specifications of this kind. For lack of room, we cannot report here the complete specifications of the considered case studies, they can be found in [6].

2 Translation Frame



The translation frame we consider here is a simple frame that is quite useful for a number of case studies [5] (it is close to the JSP frame where inputs and outputs are streams). The translation frame domain is given by the Inputs and the Outputs, the requirements are described by the input/output relationship, I/O Relationship, 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 Relationship is given by the language and computer semantics, and the Machine is the compiler. In the following, we shall provide the skeletons for the CASL formal specifications of Inputs, Outputs, the I/O Relationship, and the Machine, as well as conditions for correctness of the Machine as regards the I/O Relationship. This will be shortly illustrated on a case study (the Grep utility) in Sect. 3.

2.1 Domain and Requirements

To capture the requirements in this case means:
- to express the relevant properties of the application domain components, i.e., Inputs and Outputs;
- to express the I/O Relationship.
Let us note that this often yields to specify also some basic data that are required by the Inputs, the Outputs and/or by the I/O Relationship.

To use the CASL language to specify the above requirements means to give three CASL specifications of the following form:

spec INPUTS =
  ......

spec OUTPUTS =
  ......

spec IO_RELATIONSHIP =
  INPUTS **and** OUTPUTS **then**
  pred $IO\_rel : Input \times Output$
  axioms
  ......

where the IO_RELATIONSHIP specification extends (CASL keyword **then**) the union (**and**) of the INPUTS and OUTPUTS specifications by the *IO_rel* predicate. The axioms in CASL are first-order formulas built from equations and definedness assertions. We can here add some suggestions on the way the axioms of IO_RELATIONSHIP should be written. The *IO_rel* properties could be described along the different cases to be considered and expressed by some conditions on the *Input* and *Output* arguments. This approach has the advantage that the specifier is induced to consider all relevant cases (and not to forget some important ones). Therefore the axioms of IO_RELATIONSHIP have the form

either $IO\_rel(i, o) \Rightarrow cond(i, o)$

or $cond(i, o) \wedge def\ i \wedge def\ o \Rightarrow IO\_rel(i, o)$.

where i and o are terms of appropriate sorts and *cond* is a CASL formula.

2.2 Design

To design a solution of the problem in this case means to define a partial function (or a sequential program or an algorithm) *transl* that associates an element of Outputs with an element of Inputs. To use CASL to specify the above design means to give a CASL specification of the following form:

spec MACHINE =
  INPUTS **and** OUTPUTS **then**
free { %% this CASL "free" construction requires that no additional feature occurs
  op $transl : Input \to? Output$ %% the translation function
  axioms
  %% *transl* domain definition
  ......
  %% *transl* definition
  ......}

The axioms for *transl* should exhibit both (i) when is *transl* defined (*transl* domain definition), and (ii) what are *transl* results (*transl* definition).

Again here, we suggest a case analysis approach which yields for the *transl* domain definition axioms of the form

$cond(i) \Rightarrow def\ transl(i)$

and for the *transl* definition axioms of the form

$cond(i, o) \wedge def\ (transl(i)) \wedge def\ o \Rightarrow transl(i) = o$

where i and o are terms of the appropriate sorts, and *cond* is a positive conditional formula. Let us note that, in order to provide a more concise/readable presentation of the axioms, the $def\ (transl(i)) \wedge def\ o$ part may be left implicit.

2.3 Correctness

Here we add some notion of correctness which is not explicited in Jackson's presentation, and which we can deal with thanks to the formalization we provide. It may now be relevant to state under which conditions the MACHINE designed implements the IO_RELATIONSHIP. We propose below three conditions and introduce the following specification, requiring that the predicate *IO_rel* does not belong to the MACHINE signature.

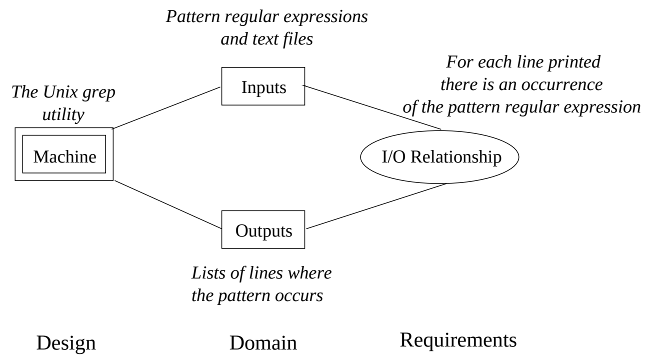
spec TRANSLATION = IO_RELATIONSHIP **and** MACHINE

MACHINE is correct w.r.t. IO_RELATIONSHIP iff

1. MACHINE is sufficiently complete and hierarchically consistent w.r.t. INPUTS and OUTPUTS.
2. TRANSLATION $\models \forall\ i : Input, o : Output, transl(i) = o \ \Rightarrow\ IO\_rel(i, o)$
3. TRANSLATION $\models \forall\ i : Input, o : Output, IO\_rel(i, o) \ \Rightarrow$ $\exists\ o' : Output \bullet transl(i) = o'$

Condition 1 requires that MACHINE does not introduce some new elements or properties in the specified descriptions of "what already exists" (the application domain), i.e., INPUTS and OUTPUTS. Condition 2 requires that, whenever *transl* is defined for a given i and yields o, then IO_RELATIONSHIP relates i and o, in other words, it ensures that the produced translation is correct. Finally, condition 3 expresses that whenever IO_RELATIONSHIP relates i with some o (recall *IO_rel* is just a relationship not a function), then *transl* applied to i must yield some o', in other words, it requires that the translation produces an output when appropriate given the requirements.

3 Case Study: The Grep Operation



In the previous section, the translation frame was presented with the typical compiler example. Here, we illustrate it with the grep utility that is provided by Unix and we sketch the corresponding specifications (see [6] for the full ones). The grep utility searches files for a pattern and prints all lines that contain that pattern. It uses limited regular expressions to match the patterns.

3.1 Domain and Requirements

In order to provide a specification of the domain, we need to specify the inputs, which are regular expressions and files, the outputs, which are lists of lines, and also the basic data that are required, which are characters, strings and lists.

Basic Data To specify the basic data we use some specifications provided in [16], e.g., CHAR and STRING. For example the specification for strings of [16] is an instantiation of the generic specification LIST[ELEM] together with some symbol mapping ($\mapsto$) for sort names:

spec STRING = LIST[CHAR]
  with sorts $List[Char] \mapsto String$

Inputs We sketch below the specifications of the Inputs which are regular expressions and files.

spec GREP_INPUTS = REGULAR_EXPRESSION **and** FILE

The CASL construct **free type** allows one to provide for the *Reg_Expr* type constants (*empty* and Λ), operations ($\_\_+\_\_$ and $\_\_*$), and also to state that any character may yield a *Reg_Expr*.

spec REGULAR_EXPRESSION = CHAR **then**
free type Reg_Expr ::=
  $empty \mid \Lambda \mid \_\_+\_\_ : (Reg\_Expr\ Reg\_Expr) \mid \_\_* : (Reg\_Expr) \mid$ **sort** $Char$;

spec FILE = STRING **then**
free type $File$ ::= $empty \mid \_\_\ \_\_ : (Char\ ; File)$;
  ops $first\_line : File \to ?\ String$;
    $drop\_line : File \to ?\ File$;
%% with the corresponding axioms

Outputs is a list of lines, that is a list of strings.

spec GREP_OUTPUTS = LIST[STRING]
  with sorts $List[String] \mapsto Grep\_Output$

I/O Relationship The I/O Relationship between the Inputs and the Outputs is sketched in the following specification, where the *grep_IO_rel* predicate properties are expressed by means of the predicates *is_gen* (stating when a string matches a regular expression) and *appears_in* (stating when a string is a substring of another one).

spec GREP_IO_REL = GREP_INPUTS **and** GREP_OUTPUTS
then preds $grep\_IO\_rel : Reg\_Expr \times File \times Grep\_Output$;
    $\_\_\ is\_gen\ \_\_ : String \times Reg\_Expr$;
    $\_\_\ appears\_in\ \_\_ : String \times String$;
  vars $reg : Reg\_Expr$; $ol, ol' : Grep\_Output$; $f : File$;

axioms
$grep\_IO\_rel(reg, empty, ol) \ \Leftrightarrow \ ol = nil;$
$\neg\ f = empty \ \Rightarrow$
$(grep\_IO\_rel(reg, f, ol) \ \Leftrightarrow$
$((\exists\ s \bullet\ is\_gen\ (reg, s) \ \wedge\ s\ appears\_in\ first\_line(f) \ \wedge$
$grep\_IO\_rel(reg, drop\_line(f), ol') \ \wedge\ ol = first\_line(f) :: ol')$
$\vee\ (\neg\ \exists\ s \bullet\ is\_gen\ (reg, s) \ \wedge\ s\ appears\_in\ first\_line(f) \ \wedge$
$grep\_IO\_rel(reg, drop\_line(f), ol)));$
%% axioms defining $appears\_in$ and is_gen

3.2 Design

The Machine yields an *Grep_Output* given a *Reg_Expr* and a *File*.

spec Grep_Machine = Grep_Inputs **and** Grep_Outputs
then op $grep\_transl : Reg\_Expr \times File \rightarrow Grep\_Output;$
pred $match : Reg\_Expr \times String;$
vars $reg : Reg\_Expr;\ f : File;$
axioms
$grep\_transl(reg, empty) = empty;$
$\neg\ (f = empty) \ \wedge\ match(reg, first\_line(f)) \ \Rightarrow$
$grep\_transl(reg, f) = first\_line(f) \ ::\ grep\_transl(reg, drop\_line(f));$
$\neg\ (f = empty) \ \wedge\ \neg\ match(reg, first\_line(f)) \ \Rightarrow$
$grep\_transl(reg, f) = grep\_transl(reg, drop\_line(f));$
%% axioms defining $match$

3.3 Correctness

To express correctness we need to introduce the following specification, requiring that the predicate *grep_IO_rel* does not belong to the Grep_Machine signature.

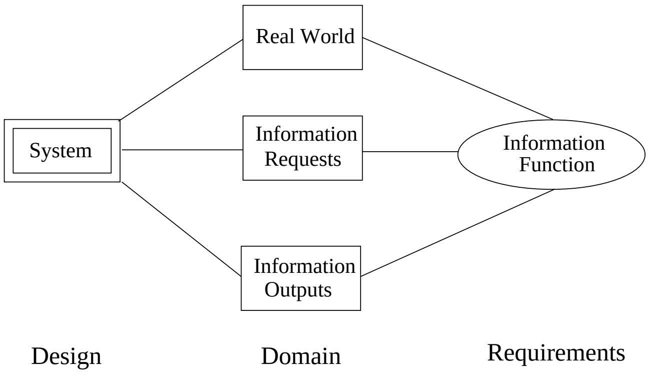
spec Grep_Translation = Grep_IO_Rel **and** Grep_Machine

Grep_Machine is correct w.r.t. Grep_IO_Rel iff

1. Grep_Machine is sufficiently complete and hierarchically consistent w.r.t. Grep_Inputs and Grep_Outputs.
2. Grep_Translation $\models \forall\ f : File, reg : Reg\_Expr, ol : Grep\_Output,$ $grep\_transl(reg, f) = ol \ \Rightarrow\ grep\_IO\_rel(reg, f, ol)$
3. Grep_Translation $\models \forall\ f : File, reg : Reg\_Expr, ol : Grep\_Output,$ $grep\_IO\_rel(reg, f, ol) \ \Rightarrow\ \exists\ ol' : Grep\_Output \bullet grep\_transl(reg, f) = ol'$

4 Information System Frame

The information system frame domain description is given by the Real World, the Information Requests and the Information Outputs, the requirements are described by the Information Function, and the design is the System. To quote [11], "In its simplest form, an information system provides information, in response to requests, about some relevant real-world domain of interest." The Real World 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" [11]). Here we consider information system frames with a dynamic domain, so "The Real World is *dynamic* and also *active*." [11].



4.1 Domain and Requirements

To capture the requirements in the case of the Simple Information System means:

- to find out the relevant properties of the Real World;
- to determine the Information Requests and the Information Outputs;
- to determine the Information Function.

To use CASL-LTL [15] to specify the above requirements means to give four specifications corresponding to the four parts respectively, as follows.

We consider the case where the Real World is a dynamic system, thus is specified using CASL-LTL by logically specifying an lts (a *labelled transition system*) that models it. A *labelled transition system* (shortly *lts*) is a triple $(S, L, \rightarrow)$, where S and L are two sets, and $\rightarrow \subseteq S \times L \times S$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$. Using the **dsort** construction introduced in CASL-LTL is a way to declare the lts triple $(S, L, \rightarrow)$ at once and to provide the use of temporal logic combinators in the axioms (as defined in CASL-LTL).

Given an lts we can associate with each $s_0 \in S$ the tree (*transition tree*) whose root is s_0, where the order of the branches is not considered, two identically decorated subtrees with the same root are considered as a unique subtree, and if it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from n to n'.

We model a dynamic system D with a transition tree determined by an lts $(S, L, \rightarrow)$ and an initial state $s_0 \in S$; the nodes in the tree represent the intermediate (interesting) situations of the life of S, and the arcs of the tree the possibilities of S of passing from one situation to another. It is important to note here that an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: D in the situation s has the *capability* of passing into the situation s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus l contains information on the conditions on the environment for the capability to become effective, and on the transformation of such environment induced by the execution of the transition.

We assume that the labels of the lts modelling the Real World are finite sets of events, where an *event* is a fact/condition/something happening during the system life that is relevant to the considered problem. So we start by determining which are the events and by classifying them with a finite number of kinds, then we specify them with a simple CASL specification of a datatype, where any kind of event is expressed by a generator.

spec EVENT =
 ... **then**
 free type *Event* ::= ...

At this stage it is not advisable to precisely specify the states of the lts modelling the Real World; however, we need to know something about them, and we can express that by some CASL operations, called *state observers*, taking the state as an argument and returning the observed value. Finally we express the properties on the behaviour of the Real World along the following schema:

- **Incompatible events:** Express when sets of events are incompatible, i.e., when they cannot happen simultaneously.
- **Relationships between state observers and events** For each state observer *obs* express (i) its initial value, (ii) its value after $E(\ldots)$ happened, for each kind of event E modifying it.
- **Event specific properties** For each kind of event E express the
  - **preconditions** properties on the system state and on the past (behaviour of the system) required for $E(\ldots)$ to take place
  - **postconditions** properties on the state and on the future (behaviour of the system) that should be fulfilled after $E(\ldots)$ took place
  - **liveness** properties on the state under which $E(\ldots)$ will surely happen and when it will happen.

Then the specification of the Real World has the following form (the specification FINITESET has been taken from [16]).

spec REAL_WORLD =
 FINITESET[EVENT **fit** $Elem \mapsto Event$] ... **then**
 dsort *State* %% **dsort** is a CASL-LTL construction
 free type *Label_State* ::= **sort** *FinSet*[*Event*];
 pred *initial* : *State* %% determines the initial states of the system
 %% State observers
 op $obs : State \times \ldots \to \ldots$
 ......
 axioms
 %% Incompatible events
 ...
 %% Relationships between state observers and events

...
%% Event specific properties
...

The postconditions and the liveness properties cannot be expressed using only the first-order logic available for axioms in CASL, thus CASL-LTL extends it with combinators from the temporal logic ([7]) that will be introduced when they will be used in the case study.

The Information Requests and the Information Outputs are two datatypes that are specified using CASL by simply giving their generators.

spec INFORMATION_REQUESTS =
    ... **then**
    free type *Info_Request* ::= ...

spec INFORMATION_OUTPUTS =
    ... **then**
    free type *Info_Output* ::= ...

We assume that the Information Function takes as arguments, not only the information request, but also the history of the system (a sequence of states and labels), because it contains all the pieces of information needed to give an answer.

The Information Function is specified using CASL-LTL by defining it within a specification of the following form.

spec INFORMATION_FUNCTION =
  INFORMATION_REQUESTS **and** INFORMATION_OUTPUTS **and** REAL_WORLD **then**
free {
  %% histories are partial system lifecycles
  type $History ::= init(State) \mid \_\_\ \_\_\ \_\_(History; Label\_State; State)?$;
  op $last : History \to State$;
  vars $st : State;\ h : History;\ l : Label\_State$;
  • $def\ (h\ l\ st) \iff last(h) \xrightarrow{l} st$ %% __ __ __ is partial
  • $last(init(st)) = st$
  • $def\ (h\ l\ st) \Rightarrow last(h\ l\ st) = st$
  op $inf\_fun : History \times Info\_Request \to Info\_Output$;
  axioms
  %% properties of *inf_fun* ...

where the properties of *inf_fun* are expressed by axioms having the form
$def\ (h) \wedge def\ (i\_req) \wedge def\ (i\_out) \wedge cond(h, i\_req, i\_out) \Rightarrow$
$inf\_fun(h, i\_req) = i\_out$
and *cond* is a conjunction of positive atoms.

In many cases the above four specifications share some common parts, by using the CASL constructs for the declaration of named specifications, such parts can be specified apart and reused when needed. These specifications are collected together and presented before the others under the title of basic data.

4.2 Design

To design an "Information System" means to design the System, a dynamic system interacting with the Real World (by detecting the happening events), and with the users (by receiving the information requests and sending back the information outputs).

We assume that the System:

- keeps a view of the actual situation of the Real World,
- updates it depending on the detected events,
- 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 the situation of the Real World.

We assume also that the System can immediately detect in a correct way any event happening in the Real World and that the information requests are handled immediately (more precisely the time needed to detect the events and to handle the requests is not relevant).

The design of the System will be specified using CASL-LTL by logically specifying an lts that models it. The labels of this lts are triples consisting of the events detected in the Real World, the received requests and the sent out information output.

spec SYSTEM =
  SITUATION **and** FINITESET[EVENT **fit** *Elem* $\mapsto$ *Event*] **and**
  FINITESET[INFORMATION_REQUESTS **fit** *Elem* $\mapsto$ *Info_Request*] **and**
  FINITESET[INFORMATION_OUTPUTS **fit** *Elem* $\mapsto$ *Info_Output*] **then**
free {
  dtype
    System ::= **sort** *Situation*;
    Label_System ::= _ _ _(*FinSet*[*Event*];
      FinSet[*Info_Request*]; *FinSet*[*Info_Output*]);
  ops $update : Situation \times FinSet[Event] \to Situation$;
    $inf\_fun : Situation \times Info\_Request \to Info\_Output$;
  pred $acceptable : FinSet[Info\_Request]$;
  axioms
    $i\_reqs = \{i\_req_1\} \cup \ldots \cup \{i\_req_n\} \ \wedge \ acceptable(i\_reqs) \wedge$
    $i\_outs = \{inf\_fun(sit, i\_req_1)\} \cup \ldots \cup \{inf\_fun(sit, i\_req_n)\} \ \Rightarrow$
      $sit \xrightarrow{evs \ i\_reqs \ i\_outs} update(sit, evs)$;
  %% axioms defining *update*, *acceptable* and *inf_fun*
    ......
}

where SITUATION specifies 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 Real World.

Thus to specify the design of the System it is sufficient to give:

- the specification SITUATION;
- the axioms defining the operation *update* describing how the System updates its view of the Real World when it detects some events;
- the axioms defining the predicate *acceptable* describing which sets of requests 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.

4.3 Correctness

We introduce the following specification:

spec INFORMATION_SYSTEM =
  INFORMATION_FUNCTION **and** SYSTEM **then**
  pred $Imp : History \times Situation$
  axioms
  ......

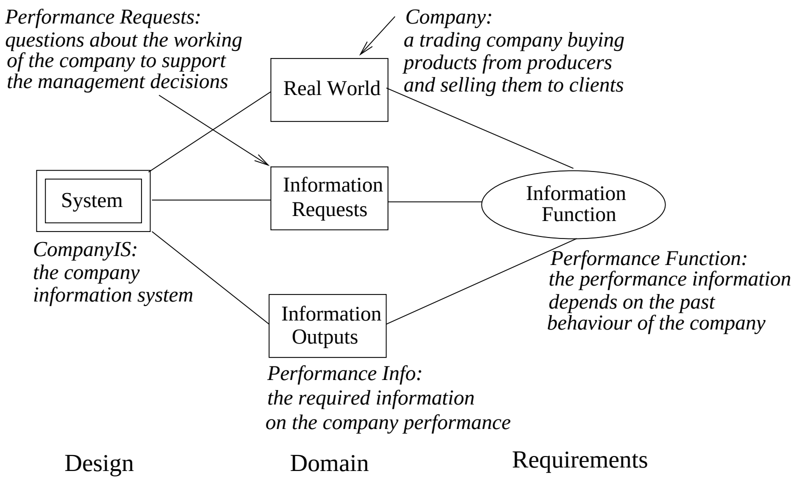
SYSTEM is correct w.r.t. INFORMATION_FUNCTION iff

1. SYSTEM is sufficiently complete and hierarchically consistent w.r.t. EVENT, INFORMATION_REQUESTS and INFORMATION_OUTPUTS.
2. INFORMATION_SYSTEM $\models \forall\ st : Situation, i\_req : Info\_Request,$ $i\_out : Info\_Output, inf\_fun(st, i\_req) = i\_out \Rightarrow$ $\exists\ h : History \bullet Imp(h, st) \wedge inf\_fun(h, i\_req) = i\_out$

Notice that the proof has to be done in the realm of the first-order logic, and not require to consider the temporal extenxion of CASL-LTL, in this frame the temporal combinators are used only to express the properties of the Real World, i.e., of the application domain.

5 Case Study: The Invoice System

5.1 The Invoice System



This case study, the invoice system, is inspired from one proposed in [1]. The problem under study is an information system for a company selling products to clients to support the management decisions. The clients send orders to the company, where an order contains one and only one reference to an ordered product in a given quantity. The status of an order will be changed from “pending” to “invoiced” if the ordered quantity is less or equal to the quantity of the corresponding referenced product in stock. New orders may arrive, and there may be some arrivals of products in some quantity in the stock. We also consider that this company may decide to discontinue some products that have not been sold for some given time (e.g., six months). An order may be refused when the product is no more traded, or when the quantity ordered is not available in the stock and a decision was taken to discontinue the product; this refusal should take place within one month after the order was received. We take the hypotheses that the size of the company's warehouse is unlimited and that the traded products are not perishable.

The picture above shows how the invoice system matches the IS frame.

Due to lack of space, the complete specifications of the requirements and the the design part will not be given here but they are available in [6].

5.2 Domain and Requirements

As explained in Sect. 4, to specify the requirements in this case means to provide four specifications corresponding to the four parts of the frame, which are reported in the following subsections. Some specification modules are quite large, thus, for readability sake, we provide some "friendly" abbreviated presentation of them. The domain and requirements specifications share some common data structures, and by using the CASL construct for the declaration of named specifications, we have specified them apart and collected together under the title of basic data.

Basic Data Some obvious basic data are the codes for products, orders, and clients, and the quantities. We need also a notion of time encoded into a date (day/month/year). The components of an order are the date when it is received, the product ordered (referenced by its code), the quantity ordered, the client who issued the order (referenced by its code), and an order code. Moreover, to specify the invoice system, we need also to use the elaboration status of an order and the trading status of a product.

spec CODE =
  sorts *Product_Code*, *Order_Code*, *Client_Code*
  %% codes identifying the products, the orders and the clients

spec QUANTITY =
  sort *Quantity* %% the quantities of the considered products
  ops $0 :\to Quantity$;
    $\_\_+\_\_ : Quantity \times Quantity \to Quantity$, **comm**, **assoc**, **unit** 0;
    $\_\_-\_\_ : Quantity \times Quantity \to ?\ Quantity$, **unit** 0 ;
  pred $\_\_\leq\_\_ : Quantity \times Quantity$;
    ...

spec DATE =
  NAT **then**
free {

type $Date$ ::= $\_\_/\_\_/\_\_(Nat; Nat; Nat)$;   %% dates as day/month/year
pred $\_\_ \leq \_\_ : Date \times Date$;
op $initial\_date :\to Date$;
...

spec ORDER =
CODE **and** QUANTITY **and** DATE **then**
free type $Order$ ::= $mk\_order(product : Product\_Code; quantity : Quantity;$
$date : Date; code : Order\_Code; client : Client\_Code)$

spec STATUS =
free types
$Product\_Status$ ::= $traded \mid not\_traded$;
$Order\_Status$ ::= $pending \mid invoiced \mid non\_existing \mid refused$;
%% elaboration statuses of products and trading statuses of orders

Real World To specify the Real World component of the application domain, we have to express the relevant properties of its behaviour, following the schema introduced in Sect. 4.1; thus, we first determine the "events" and the "state observers", and then we look for the "incompatible events", the "relationships between state observers and events", and for the "event specific properties". We provide below the abbreviated presentation and a sketch of the corresponding CASL-LTL specification (see [6] for the full specification).

Events We present the events by listing the generators (written using capital letters) with their arguments and a short comment.

- $RECEIVE\_ORD(Order)$ to receive an order
- $SEND\_INVOICE(Order)$ to send the invoice for an order
- $REFUSE(Order)$ to refuse an order
- $RECEIVE\_PROD(Product\_Code; Quantity)$ to receive some quantity of a product
- $DISCONTINUE(Product\_Code)$ to discontinue a product
- $CHANGE(Date)$ to change the date

State Observers We simply present the state observers by listing them with the types of their arguments and result, dropping the standard argument of the dynamic sort $State$. We use the notation convention that sort identifiers start with capital letters, whereas operation and predicate identifiers are written using only lower case letters.

- $product\_status(Product\_Code) : Product\_Status$ trading status of a product
- $order\_status(Order\_Code) : Order\_Status$ elaboration status of an order
- $available\_quantity(Product\_Code) : Quantity$ available quantity of a product in the stock
- $date : Date$ actual date

With the corresponding formal specification (see [6] for the full specification):

spec STATE_OBSERVERS =
  ORDER **and** STATUS **then**
  sort *State*
  ops $product\_status : State \times Product\_Code \to Product\_Status$
  %% trading status of a product
...

Incompatible Events We simply present the incompatible events by listing the incompatible pairs.

- All events referring to two orders with the same code are pairwise incompatible.
  - $RECEIVE\_ORD(o), SEND\_INVOICE(o')$ **s.t.** $code(o) = code(o')$
  - $RECEIVE\_ORD(o), REFUSE(o')$ **s.t.** $code(o) = code(o')$
  - $SEND\_INVOICE(o), REFUSE(o')$ **s.t.** $code(o) = code(o')$
  - $RECEIVE\_ORD(o), RECEIVE\_ORD(o')$ **s.t.** $code(o) = code(o') \wedge \neg (o = o')$
  - $SEND\_INVOICE(o), SEND\_INVOICE(o')$ **s.t.** $code(o) = code(o') \wedge \neg (o = o')$
  - $REFUSE(o), REFUSE(o')$ **s.t.** $code(o) = code(o') \wedge \neg (o = o')$
- All events referring to the same product are pairwise incompatible.
  - $RECEIVE\_PROD(p, q), SEND\_INVOICE(o)$ **s.t.** $product(o) = p$
  - $RECEIVE\_PROD(p, q), DISCONTINUE(p')$ **s.t.** $p = p'$
  - $SEND\_INVOICE(o), DISCONTINUE(p)$ **s.t.** $product(o) = p$
  - $RECEIVE\_PROD(p, q), RECEIVE\_PROD(p, q')$ **s.t.** $\neg q = q'$
- All change date events are pairwise incompatible.
  - $CHANGE(d), CHANGE(d')$ **s.t.** $\neg d = d'$

In the corresponding CASL specification (see [6]) each pair corresponds to an axiom, e.g., the first two axioms below.

$$st \xrightarrow{l} st' \wedge RECEIVE\_ORD(o) \in l \wedge SEND\_INVOICE(o') \in l \Rightarrow \neg (code(o) = code(o'))$$

$$st \xrightarrow{l} st' \wedge RECEIVE\_ORD(o) \in l \wedge REFUSE(o') \in l \Rightarrow \neg (code(o) = code(o'))$$

Relationships between State Observers and Events We simply present the relationships between state observers and events by listing for each state observer its initial value, which events modify it and how. This last part is given by stating which is the observer value after the happening of the various events. Notice that such value could be expressed by using also the observations on the state before the happening the event. Thus "**after** $RECEIVE\_PROD(p, q)$ **is** $available\_quantity(p) + q$" below means that the new value is the previous one incremented by q.

- $product\_status(p)$
  initially is *traded*
  after $DISCONTINUE(p)$ **is** *not_traded*
  not changed by other events
- $available\_quantity(p)$
  initially is 0

after $SEND\_INVOICE(o)$ **s.t.** $product(o) = p$
  is $available\_quantity(p) - quantity(o)$
after $RECEIVE\_PROD(p, q)$ **is** $available\_quantity(p) + q$
not changed by other events

– $order\_status(oc)$
  initially is $non\_existing$
  after $RECEIVE\_ORD(o)$ **s.t.** $code(o) = oc$ **is** $pending$
  after $SEND\_INVOICE(o)$ **s.t.** $code(o) = oc$ **is** $invoiced$
  after $REFUSE(o)$ **s.t.** $code(o) = oc$ **is** $refused$
  not changed by other events
– $date$
  initially is $initial\_date$
  after $CHANGE(d)$ **is** d
  not changed by other events

Below, as an example, we report the complete axioms expressing the relationships between the state observer *product_status* and the events.

$$initial(st) \;\Rightarrow\; product\_status(st, p) = traded$$
$$st \xrightarrow{l} st' \;\wedge\; DISCONTINUE(p) \in l \;\Rightarrow\; product\_status(st', p) = not\_traded$$
$$st \xrightarrow{l} st' \;\wedge\; DISCONTINUE(p) \notin l \;\Rightarrow\; product\_status(st', p) = product\_status(st, p)$$

Event Specific Properties We present the event specific properties by listing for each event the properties on the system state necessary to its happening (preconditions), the properties on the system state necessary after it took place (postconditions), and under which condition this event will surely happen. The system state before and after the happening of the event are denoted by st and st' respectively. It is recommended to provide as well for each event a comment summarizing its properties in a natural way. We give below the presentation of $RECEIVE\_ORD$.

$RECEIVE\_ORD(o)$
Comment: If the order o is received, then the product referred in o was traded, no order with the same code of o existed, the date of o was the actual date, and in any case eventually o will be either refused or invoiced.

before
  $product\_status(st, product(o)) = traded,$
  $order\_status(st, code(o)) = non\_existing$ **and** $date(o) = date(st)$
after
  $order\_status(st', code(o)) = pending$ **and**
  $in\_any\_case(st', eventually\ state\_cond(x \bullet$
  $order\_status(x, code(o)) = refused \vee order\_status(x, code(o)) = invoiced))$

"$in\_any\_case(st', eventually\ state\_cond(x \bullet \ldots))$" is a formula of CASL-LTL built by using the temporal combinators. "$in\_any\_case(s, \pi)$" can be read "for every path σ starting in the state denoted by s, π holds on σ", where a path is a sequence of transitions having the form either (1) or (2) below:

(1) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots$ (infinite path)

(2) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots\ s_n$ $n \geq 0$

where for all i ($i \geq 0$), $s_i \xrightarrow{l_i} s_{i+1}$ and there does not exist l, s' such that $s_n \xrightarrow{l} s'$.

"*eventually state_cond*$(x \bullet F)$" holds on σ if there exists $0 \leq i$ s.t. F holds when x is evaluated by s_i.

Now we can give the formal specification of Real World for the invoice case, i.e., the company.

spec INVOICE_REAL_WORLD =
  ... **then**

  %% *RECEIVE_ORD*(o)

$$
\begin{array}{l}
st \xrightarrow{l} st' \ \wedge\ RECEIVE\_ORD(o) \in l \ \Rightarrow \\
product\_status(st, product(o)) = traded \ \wedge \\
order\_status(st, code(o)) = non\_existing \ \wedge \\
date(o) = date(st) \ \wedge\ order\_status(st', code(o)) = pending \ \wedge \\
in\_any\_case(st', eventually\ state\_cond(x \bullet \\
\qquad order\_status(x, code(o)) = refused \ \vee \\
\qquad\quad order\_status(x, code(o)) = invoiced))
\end{array}
$$

  ......

Information Requests We present the information requests by listing their generators with the types of their arguments; similarly for the information outputs.

- *available_quantity_of*?(*Product_Code*)
  what is the available quantity of a product in the stock?
- *quantity_of Product_Code sold_in Date − Date*?
  what is the quantity of a product sold in the period between two dates?
- *last_time_did Client_Code ordered*?
  what is the last time a client made some order?

Information Outputs

- *the_available_quantity_of Product_Code is Quantity*
- *error* : *prod_not_traded* the product appearing in the request is not traded
- *error* : *wrong_dates* the dates appearing in the request are wrong
- *the_quantity_of Product_Code sold_in Date − Date is Quantity*
- *Client_Code ordered_last_time_at Date*

Information Function Recall that the *inf_fun*, in this case named *invoice_inf_fun*, takes as arguments an information request and a history (a partial system lifecycle), defined as a sequence of transitions, i.e., precisely a sequence of states and labels. We simply present *invoice_inf_fun* by showing its results on all possible arguments case by case; each case is presented by starting with the keyword **on**, followed by the list of the arguments.

on $available\_quantity\_of?(p),\ h$
  if $product\_status(last(h), p) = traded$ **returns**
    $the\_available\_quantity\_of\ p\ is\ available\_quantity(last(h), p)$
  if $product\_status(last(h), p) = not\_traded$ **returns** $error : prod\_not\_traded$
on $quantity\_of\ p\ sold\_in\ d_1 - d_2?,\ h$
  if $\neg\,(d_1 \leq d_2$ **and** $d_2 \leq date(last(h))$ **returns** $error : wrong\_dates$
  if $d_1 \leq d_2$ **and** $d_2 \leq date(last(h))$ **returns**
    $the\_quantity\_of\ p\ sold\_in\ d_1 - d_2\ is\ sold\_aux(p, d_1, d_2, h)$
on $last\_time\_did\ cc\ ordered?,\ init(st)$ **returns** $initial\_date$
on $last\_time\_did\ cc\ ordered?,\ h\ l\ st$
  if $RECEIVE\_ORD(o) \in l$ **and** $client(o) = cc$ **returns**
    $cc\ ordered\_last\_time\_at\ date(st)$
  if $\neg\,(\exists\ o : Order \bullet SEND\_INVOICE(o) \in l$ **and** $client(o) = cc)$ **returns**
    $invoice\_inf\_fun(last\_time\_did\ cc\ ordered?, h)$

The auxiliary operation $sold\_aux$ returns the quantity of a product sold in a certain time interval, for its complete definition see [6].

As an example, we show below the complete CASL axioms corresponding to the definition of $invoice\_inf\_fun$ for the first case.

$product\_status(last(h), p) = traded \ \Rightarrow$
  $invoice\_inf\_fun(available\_quantity\_of?(p), h) =$
  $the\_available\_quantity\_of\ p\ is\ available\_quantity(last(h), p)$
$product\_status(last(h), p) = not\_traded \ \Rightarrow$
  $invoice\_inf\_fun(available\_quantity\_of?(p), h) = error : prod\_not\_traded$

6 Conclusions and Future Work

While it is clear that methods are needed to help developing formal specifications, as extensively advocated in [4], this remains a difficult issue. This problem is addressed in [9,10] that define the concept of *agenda* used to provide a list of specification and validation tasks to be achieved, and apply it to develop specifications with Statecharts and Z. [14] also uses agendas addressing "mixed" systems (with both a static and a dynamic part), and provides some means to generate parts of the specification. [5] is the first work we know of that provides a formal characterisation of M. Jackson problem frames. Along this approach, we provide here a formalization of the translation frame and of the information system frame using the CASL language together with worked out case studies. Being in a formal framework lead us to add to the issues addressed by problem frames, the issue of correctness.

Following the approach proposed in this paper to use formal specifications in the development process of real case studies becomes an "engineering" kind of work. Indeed, for each frame we propose an operative method based on "normal" software engineering (shortly SE) concepts (inputs, outputs, events, ...) and not on mathematical/formal ones (existence of initial models, completeness, ...). Moreover, working with large case studies lead us to provide some legible presentations of the various parts of the specifications removing/"abstracting from" some conventional mathematical notations/overhead (while the corresponding complete specifications may be easily recovered from these) as for example, in Sect. 5.2.

We have based our work on some well established SE techniques and concepts (as the clear distinction supported by Jackson among the domain, the requirements and the design) that, for what we know, are not usually very well considered in the formal method community ([5] beeing an exception). Previous algebraic specifications of the case studies considered in this paper made by the authors themselves, without considering the SE aspects, were quite unprecise and perhaps also wrong. In the grep case everything was considered as "requirements" and then realized in the design phase, and so we had implemented also the regular expressions and the files. Instead, for the invoice case the old specifications were confused as regards what should be the responsibilities of system that we have to build (e.g., the information system was responsible to guarantee that an order eventually will be either invoiced or refused instead of simply taking note of when an order is invoiced).

Let us note that, while the selection of the correct frame and the specification of the requirements and the design are essential, specifying the domain part is necessary to produce sensible requirements, and may also be needed to discuss with the clients, or to check about possible misunderstandings with the domain experts (most of the worst errors in developing software systems are due to wrong ideas about the domain).

Another relevant aspect of our work is clearly "reuse": but here we reuse what can be called, by using a current SE terminology, "some best practices", not just some specifications. The ways to handle particular problem frames that we propose encompass the practice on the use of algebraic specifications of the authors; and so our work may be considered in the same line of the use of "patterns" ([8]) for the production of object oriented software. The most relevant difference between [8] and the work presented here is the scale: we consider as a reusable unit a way to solve a class of problem, the patterns of [8] consider, instead, something of much smaller (pieces of the design).

Acknowledgements We would like to thank the anonymous referees for their careful reading and helpful comments.

References

1. M. Allemand, C. Attiogbe, and H. Habrias, editors. *Proc. of Int. Workshop "Comparing Specification Techniques: What Questions Are Prompted by Ones Particular Method of Specification". March 1998, Nantes (France).* IRIN - Universite de Nantes, 1998.
2. E. Astesiano, B. Krieg-Bruckner, 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. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
4. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236, 2000.
5. D. Bjørner, S. Kousoube, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In M.G. Hinchey and Liu ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997.* IEEE CS Press, 1997.
6. C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach – Complete Version. Technical Report DISI-TR-99-33, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.
7. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2), 1997.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. 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 LNCS. Springer Verlag, Berlin, 1998.
10. M. Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Proceedings Systems Implementation 2000.* Chapman & Hall, 1998.
11. M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.
12. 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 LNCS, Berlin, 1997. Springer Verlag.
13. The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
14. P. Poizat, C.Choppy, and J.-C. Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, number 1709 in LNCS. Springer Verlag, Berlin, 1999.
15. 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. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.
16. M. Roggenbach and T. Mossakovski. Basic Data Types in CASL. CoFI Note L-12. Technical report, 1999. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .