Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Formalization and composition of languages for the modeling of fire safety systems

H. Chanti [*, ***] **L. Thiry** [*] **M. Hassenforder** [*] **J-F. Brillhac** [**] **P. Fromy** [***]

[*] *MIPS EA 2332 - Université de Haute Alsace, 12 rue des Frères Lumière - 68093 Mulhouse (France); (e-mail: {houda.chanti, laurent.thiry, michel.hassenforder}@uha.fr).*
[**] *GRE EA2334 - Université de Haute Alsace - 3bis, rue Alfred Werner - 68093 Mulhouse (France); (e-mail: jean-francois.brillhac@uha.fr).*
[***] *CSTB, 84 avenue Jean Jaurès - Champs-sur-Marne - 77447 Marne-la-Vallée Cedex 2 (France); (e-mail: {houda.chanti, philippe.fromy}@cstb.fr).*

Abstract: Modeling complex systems, such as the ones found in the certification of fire protection systems, generally requires the intervention of many specialists, each one using its own formalisms, concepts and tools. To model such systems, many specific languages are required and to be integrated they should be formally described. In this proposal, we suggest to use functional programming concepts to formalize and integrate the languages involved in the field of fire safety systems. Formalization is done by specifying constructor functions and integration by the way of generic/higher-order functions.

Keywords: Domain Specific Languages, functional programming, fire safety.

1. INTRODUCTION

A complex system is generally studied by considering various points of view, each one based on specific models and languages. Thus, global modeling requires means (i.e. concepts and tools) to express in a unified way modeling languages and their relations. Moreover, the elements considered have to be precise (i.e. with mathematical foundations) to make possible proofs (e.g. no information is lost, safety properties are satisfied, etc.).

In the domain of computer science, languages used by specialists are called Domain Specific Languages (DSLs), see van Deursen et al. [2000] for a more precise definition of the concept and examples. These languages are: 1) based on concepts and features of a domain, and 2) used to describe and generate programs in a specific field. The use of DSLs offers the possibility for analysis, optimization and transformation of models, and has the advantage to enable the reuse of software artifacts, Biggerstaff [1998].

As a complex system, the evaluation of a fire safety system requires the use and the integration of many models and languages to describe the architecture, fire attitude, physical properties of the materials, behavior of security system, etc. The contribution of the present paper is in the precise definition of the needed languages, and the illustration of how functional programming concepts can facilitate their integration.

The current paper is divided into three parts: the first one describes the main components of fire safety systems and the fundamental concepts that make functional programming interesting. The second part gives a formal definition of the specific languages used in the process of evaluation of the safety level, and their integration. The last part concludes by summarizing the main points presented and by giving the perspectives considered.

2. FIRE PROTECTION AND FUNCTIONAL PROGRAMMING

2.1 Fire protection systems

The certification of a building against fire is based on the evaluation of the safety sub-systems installed. A fire safety system is composed of various components distributed in a building to collect information regarding to fire safety. To evaluate the fire safety level in a building, engineers try to reproduce the phenomena observed in a fire situation. Based on various configurations (figure 1), they use simulators to calculate physical quantities that can lead to destruction of materials or death of persons (what defines the undesired events). Certification of fire protection system involves:

- Many models to be integrated such as architectural model, undesired event, simulator, checker, etc.
- The use of a simulator to get system evolution $X(t) = [P_i(t), T_i(t)]$ by considering an initial state $X(0)$, pressures P_i of a location, temperature T_i, etc.
- The resulting behavior $X(t)$ is used to check safety properties (e.g. $P_i(t) > V$), and many behaviors have to be considered depending of the initial configuration $X(0)$, i.e. where a fire begins, or choices of fire protection sub-systems.

Copyright © 2014 IFAC

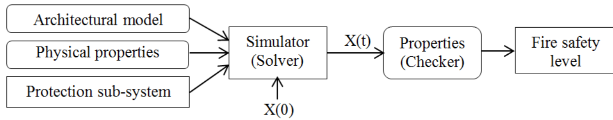

Fig. 1. Models in a fire protection system

Till now the evaluation of fire safety level depends on the knowledge of specialists and is done manually. There exists many available simulators based on various fire propagation models with for instance the ones described by Hu et al. [2007] or Parry et al. [2003].

Our institute the CSTB (the Scientific and Technical Center for Building), develops a specific simulator called CIFI (Curtat [2002],Bodart and Curtat [1987]) based on a two zones fire model, to get the behavior $X(t)$. An example of input configuration $X(0)$ for the simulator is given bellow (we will refer to this file by the name configuration):

```
- Simulation properties
&MISC JOBNAME='F3' TETAFIN=5400.0 DTETA=0.10
      ATOL=1.D-9 RTOL=1.D-7 NSAVE=600/

&REAC XRAD=0.30D0 COYIELD=0.004D0
      SOOTYIELD=0.015D0 NU_C=3.4D0
      NU_H=6.2D0 NU_O=2.5D0 /

- External temperature
&EXT  T=20.0 /

- Local description
&LOC  ID='LOC1' H=2.2 LONG=75. LARG=15. ALT=0.
      T16INI=21. E(1)=0.2 LAMBDA(1)=1.6
      RHO(1)=2300. CS(1)=1000. EMIP(1)=0.9
      ISOLANT(1)=.FALSE. E(2)=0.2
      LAMBDA(2)=1.6 RHO(2)=2300. CS(2)=1000.
      EMIP(2)=0.9 ISOLANT(2)=.FALSE./

- Openings description
&OUV  ID='OUV1' ALLEGE=0. LINTEAU=1.
      LARGEUR=3.6 FUITE=0.01 ALT=0.
      LOCIDS='LOC1','EXT1'  CTRLID='ETATOUV1' /

- Smoke evacuation
&OUV  ID='OUV2' ALLEGE=1.4 LINTEAU=2.2
      LARGEUR=4.5 FUITE=0.01 ALT=0.
      LOCIDS='LOC1','EXT3' CTRLID='ETATOUV2' /

- Fireplace properties
&SBO  ID='FOYER' LOCID='LOC1' ZF=0. TVAP=300.
      LVAP=1.8D6 AFMAX=12. ILOI=1 RAMP='LIN'
      TPLT=300. MPDOTMAX=0.6944 TDEC=300.
      TFIN=3300./

- Controllers
&CTRL ID='ETATOUV1' ON_INI=.TRUE. /
&CTRL ID='ETATOUV2' ON_INI=.TRUE. /

&END /
```

This configuration is a composition of many blocks:
The block &MISC describes the simulation properties as the duration of a simulation and the step time. These data are filled by a fire safety engineer.

The block &REAC is filled by a chemist, and consists on the parameters of the combustion model.

The block &LOC describes the geometry of one local in a building. This one is identified by an ID. Some parameters are filled by an architect, such as H, LONG, LARG and ALT, but some others are filled by a fire safety engineer as they represent the characteristics of construction materials and fire resistance (as E, LAMBDA, RHO, etc).

The &OUV block gives mainly architectural information, but also leakage (physics) when the opening is closed.

The &SBO block gives information about the location of the fireplace (LOCID), and also the parameters of the curve combustion based on the fuel material. It describes one kind of fireplaces (for example a table), if the fire safety engineer changes for another fireplace (a chair for example), he has to replace the whole block by new parameters.

Based on the knowledge of many specialists, many data are composed to make an input file (or a configuration) for the simulator.

The main problem is that an architect, a chemist or a fire safety engineer use their own languages, but they have to translate it into this configuration language to correspond to the structure of the input file (an example of an input configuration is given in subsection 2.1). So to allow the experts to use their preferred languages, a specific tool has to be proposed to merge these expert specific languages and to generate the initial configuration of the simulator.

Moreover as explained above, many initial configurations $X(0)$ have to be considered (e.g. $P_i(0) \in [P_{min}, P_{max}]$) what corresponds to another specific language. Finally, a language has to be defined for the checker and the properties to automatic the safety level detector.

2.2 Formalization of languages

Models represent abstractions of systems. They are expressed by using languages, i.e. set of terms generated by a set of production rules formalized by "a grammar", Chomsky [1959]. A grammar $G = (TS, NT, S, m)$ is defined by: 1) Terminal Symbols *TS*, e.g. *"+","("* or *"1"*, 2) Non-Terminal symbols *NT*, e.g. *Exp*, 3) A Start symbol $S \in NT$ and 4) A map *m:* $(TS \cup NT)^* \rightarrow NT$, where X^* denotes a sequence of X_s.

Grammars are generally expressed by using specific languages, as in particular the Backus-Naur Form (BNF) language, Chomsky [1959], or the Extended Backus-Naur Form (EBNF). As an illustration, the EBNF representation of the grammar for elementary arithmetic expressions *Exp* is:

```
<Exp>  ::= <Val>|<Exp> "+" <Exp>|"("<Exp>")"
<Val>  ::= (0 | 1 | 2 |...| 9)+
```

The symbol (::=) is called the definition symbol, it separates the right parts from the left parts of a rule. Rules having the same results are grouped together by using the choice symbol (|).

In the above example, an expression is either (|) a value *Val*, a sum between two expressions in an infixed notation *Exp + Exp*, or a sub-expression enclosed in parenthesis *(Exp)*. A value is a non empty sequence (+) of digits *0 | 1 | 2 |...| 9*.

In the definition below, *Exp*, and *Val* are non-terminal symbols, they are written between angle brackets <>; and +, (,) and digits are terminal symbols. The non-terminal *Exp* is considered as the start symbol.

The aim of a "parser" (i.e. the first component of a "compiler") is to check that a language (i.e. a sequence of terminal symbols) is conformed to the grammar by finding a sequence of production rules (or derivation) to get the language starting with the start symbol.

In modern compilers the internal data structure is converted to an "abstract syntactic tree" Aho et al. [2007] to store successive results. The tree structure can be visited to generate representations for other languages or to calculate particular values. To summarize, a grammar is then associated to a datatype (e.g. *Exp*) and the first step of a compilation is to transform (and parse) a sequence of characters to a value of this type, i.e.:

```
parseExp: String -> Exp
```

The next step is to interpret the result (e.g. evaluate an expression) or to generate code for another tool.

```
eval: Exp -> Integer
```

The function eval evaluates an expression and returns an integer.

2.3 Functional programming

The fundamental building blocks of Functional Programming are functions (not objects or procedures) that declare a relationship between two or more entities. Functional programming languages such as Haskell (Russell and Cohn [2012]) are mostly based on λ-*calculus* (Hudak [2000]). In particular, the models defined can use higher-order functions (i.e. functions that have other functions as a parameter or as a result) to stay compact and simple. A function f will correspond to a relation between two sets/types A and B, what is written in haskell:

```
f :: A -> B
```

As a remark, A or B can be functions sets (e.g. $A = C \to D$) and f is then called "an higher-order function". Now, the tree data structure *Exp* can be specified by a family of functions (or an algebraic specification, Ehrig and Mahr [1985]) usable to define a particular value, e.g. *"(1+2)+3"* corresponds to *plus(plus(val(1),val(2)),val(3))*.

```
val : Integer    -> Exp
plus: (Exp, Exp) -> Exp
```

The constructor functions can be grouped together in Haskell by the mean of a datatype definition as bellow:

```
data Exp = Val  Real
         | Plus (Exp, Exp)
```

Then functions (and its particular eval) can specify how each constructor is transformed:

```
eval : Exp          -> Integer
eval(val(v))        = v
eval(plus(e1,e2)) = eval(e1)+eval(e2)
```

From a theoretical point of view, "eval" is a particular case of a catamorphism (Meijer et al. [1991]), a concept of Category Theory used to express generic transformations, to calculate programs or to make proofs (Fokkinga [1992]).

Our group has studied for a while the benefits of the functional representations of languages and systems and the present paper gives another application of the concepts proposed, e.g. Thiry and Thirion [2008], Thiry and Thirion [2009], Thiry and Hassenforder [2013].

3. DSLS INTEGRATION FOR FIRE SAFETY

3.1 System descriptions

As explained, the description of fire safety systems is based on the architecture of the building, the technical and organizational safety measures and the fire simulator. The proposed fire safety detector proposed is composed of a generator (based on different models and simulation), a fire simulator and a checker as seen in figure 2.

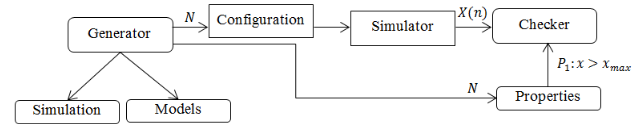

Fig. 2. Components used to evaluate the fire safety levels

Evaluation of the safety level in a building, needs to study several configurations and this is the goal of the generator. It automatically build many configurations and randomly selecting values into intervals which constitute the input files of the fire simulator.

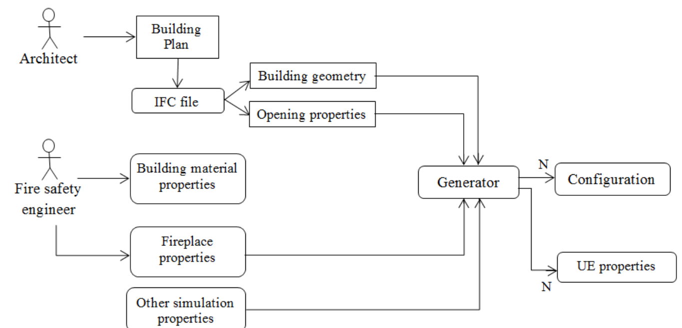

Fig. 3. Detailed process of the generation of configurations

Each configuration is made by composition of architectural model, fireplace and undesired events. A configuration describes the architecture of the building using an architectural model based on a digital map with information concerning the geometry of the building and dimension and position of the different openings. These data are expressed in a specific language used by the architects.

Characteristics of the materials (e.g. information about the stability of materials against fire) and other parameters which describe a fireplace are also necessary to make a configuration as seen in figure 3. A library is proposed in order to save these kind of information. These data are collected and merged in a special format, and then sent to the generator to make many configurations. A configuration is then sent to the fire simulator to calculate some physical quantities ($X(t)$).

The simulation results are controlled to check if some conditions are achieved or not (figure 4). These values

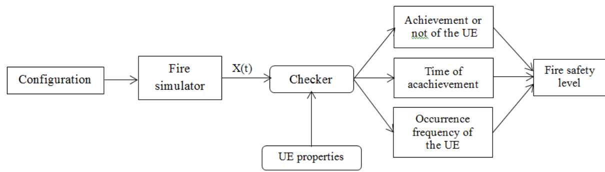

Fig. 4. Process of the evaluation of fire safety level

characterize the undesired events (UE) that be expressed by using another specific language. The generator has to describe the undesired events properties too, as they depend on the configuration. More precisely, the evaluation of the safety level in a building is based on the frequency occurrence of the UE. The checker is in charge of this task.

3.2 The architectural model

Each configuration describes the architecture of the building using an architectural model. Based on a digital map, information concerning the geometry of the building, dimension and position of the different openings are extracted on an IFC format, the specific language of architects, Spearpoint [2006].

The architecture of the building is described by a geometry model exported as an IFC (Industry Foundation Classes) file, Spearpoint [2006] and Martens and Herbert [2004]. Industry Foundation Classes (IFC) is a standard which allows building simulation software to automatically acquire building geometry and other building data from project models. An example of an IFC file is:

```
#9512= IFCBUILDINGSTOREY('3y21AUC9X4yAqzLGUny16E',
                         #16,'Story',$,$,
                         #9509,
                         $,$,.ELEMENT.,6200.);
#9509= IFCLOCALPLACEMENT(#115,#9506);
#9506= IFCAXIS2PLACEMENT3D(#9502,#9498,#9494);
#9502= IFCCARTESIANPOINT((0.,0.,6200.));
#9498= IFCDIRECTION((0.,0.,1.));
#9494= IFCDIRECTION((1.,0.,0.));
```

For a more abstract point of view, this file is fundamentally a map $I \rightarrow C(X_1, X_2, ...X_n)$ where I is an index, C a component (e.g. *IFCDIRECTION*), and X_i can be a value V, an index I or a component C.
By taking back, the concepts presented in the previous section, such a language can be summarized by a grammar whose an extract is given below.

```
<IFC>     ::= <ENTRY>*
<ENTRY>   ::= #<INTEGER>= <ID>(<PARAMS>)
<INTEGER> ::= (<DIGIT>)*
<PARAMS>  ::= epsilon | <PARAM> <CONT>
<PARAM>   ::= #<INTEGER> | <ID>(<PARAMS>) |
              '<TEXT>' | ...
<CONT>    ::= espilon | , <PARAM> <CONT>
<ID>      ::= (<LETTER> | <DIGIT>)*
```

An IFC has many entries, each one began with a special character # followed by an integer and then an equal symbol, an identifier *ID* and a set of parameters between brackets *(PARAMS)*. An identifier *ID* is a sequence of a character and an integer *#INTEGER*. A parameter *PARAM* can be an identifier, an identifier followed by other parameter or just a text.

By taking back the concepts introduced in part 2.2, the language is formalized by a grammar and the next step is to define a datatype to store information (and the parse function associated). Here an IFC can be abstracted by a list *[ ]* composed of an integer (the index), a string (the component type) and values (the parameters):

```
data Ifc   = [(Int, String, Value)]
```

A $Value$ is either an integer (for references), a string (for values) or a pair composed of a string and a list of values (for sub components):

```
data Value = Int | String | (String,[Value])
```

Finally a parse function has been defined by encoding the grammar with dedicated tool, i.e.

```
parseIfc :: String -> Ifc
```

Utility functions has been defined to extracts particular information component for instance, i.e.

```
get:  (Ifc, Int)   -> (String,[Value])
```

3.3 Materials and fireplaces

Some additional properties must be defined to constitute the configuration. Information about construction materials and characteristics of potential fireplaces are defined in a library. The library has many entries divided into categories: the first one describes the construction materials which informs about the tenability of the walls, it corresponds to the entry &*MAT*. The second one lists and describes the combustion curves of potential fireplaces. It corresponds to the &*SBO* entry. Each entry is defined by a unique identifier (ID). An example of a library is:

```
&MAT ID='CONCRETE_1' E=0.20 LAMBDA=1.2
     RHO=2100. CS=900. EMIP=0.9
     ISOLANT=.FALSE. /
&MAT ID='WOOD' E=0.20 LAMBDA=1.6
     RHO=2300. CS=1000. EMIP=0.9
     ISOLANT=.FALSE. /

&SBO ID='TABLE' ZF=0.8 TVAP=300.
     LVAP=1.8D6 AFMAX=5. ILOI=5
     RAMP='LIN' TPLT=600. MPDOTMAX=0.045
     TDEC=1200. TFIN=1800. /
&SBO ID='CHAIR' ZF=0. TVAP=300. LVAP=1.8D6
     AFMAX=5. ILOI=2 RAMP='LIN' TPLT=600.
     MPDOTMAX=0.1 TDEC=1800. TFIN=2700. /
```

In the example above two kinds of materials are defined: CONCRETE_1 and WOOD, each one with its own properties. Two kind of fireplaces are defined too: a fireplace for a table and another for a chair. Each one corresponds to a list of parameters describing the curve combustion of the fireplace. The grammar used is based on the configuration grammar and is not specific to the fire safety engineer, but can be changed freely without disturbing the software. Its structure can be summarized by the following grammar:

```
<LIB>   ::= <ENTRY>*
<ENTRY> ::= <CAT>
<CAT>   ::= <MAT>|<SBO>
<MAT>   ::= &MAT <ID><PARAM>
<SBO>   ::= &SBO <ID><PARAM>
<ID>    ::= ID= ' <LETTER>*[<DIGIT>*] '
<PARAM> ::= <LETTER>*= <VAL>
<VAL>   ::= <DIGIT>*[.<DIGIT>*]|' <LETTER>* '
```

The library is composed of many entries $ENTRY^*$ divided into the description of materials MAT for the description of the fireplaces SBO. Each entry began with the special character & and the keyword MAT or SBO. Each entry has an identifier ID followed by an equal symbol (=) and a list of parameters $PARAM$. Each one is defined by a sequence of letters $LETTER^*$, followed by an equal sign (=) and a value VAL.

By taking back the concepts introduced in part 2.2, the language is formalized by a grammar and the next step is to define a datatype to store information (and the parse function associated).

A datatype *Lib* can be seen as a list:

```
data Lib = [(Cat, String,Params)]
```

Where a category *Cat* corresponds to:

```
data Cat = MAT String | SBO String
```

Finally a parse function has been defined by encoding the grammar with dedicated tool, i.e.

```
parseLib :: String -> Lib
```

Utility functions has been defined to extracts parameters of a particular entry i.e.

```
get:  (Lib, Cat, String) -> [Params]
```

The function get returns a list of parameters corresponding to the specific entry defined by a library, a category and an identifier (string).

3.4 The Undesired Event (UE)

An undesired event (UE) is defined by some safety properties. It is specified as the last event in a chain of events with devastating impact on one or more individuals, ecosystems or materials. It is strongly correlated to the thermal conditions, the amount of toxic gases and smokes, and the pressure in a local. It is considered as an extrem condition.

An UE can be the reaching of a certain temperature in the upper layer (T_U) of a local (gas temperature), a certain height of the smoke or the achievement of a critical pressure. Based on the frequency occurrence of the undesired events (UE), the safety level is evaluated. Some examples of Undesired Events are:

```
2 TU > 550
1 TU > 450 && ZD < 0.55
1 teta < 650 && TU > 450 && ZD < 0.55
```

Concerning the grammar, an event is defined by symbols S (e.g. TU, ZD), some values V (e.g. 0.55, 450), a room N in a building and the following grammar:

```
<E>::= <N> <U>
<U>::= <S> < <V> | <S> > <V> | <U> && <U>
```

We then define the abstract syntactic tree:

```
data UE = Inf (Int, String, Float)
        | Sup (Int, String, Float)
        | And (Int, UE, UE)
```

Next, the result of the simulator is used to define a "context" function to get the value of a particular symbol.

```
type Time    = Integer
type Context = (Time, String) -> Float
```

To evaluate an event in a context, the function eval is defined:

```
eval :: (UE, Context, Time)
eval(Inf(i,s,v),c,t)   = c(t,s) < v
eval(Sup(i,s,v),c,t)   = c(t,s) > v
eval(And(i,e1,e2),c,t) = eval(e1,c,t) & eval(e2,c,t)
```

Finally a $parseUE$ function is defined and used with the eval function to get the checker component: $checker = eval \circ parseUE$

3.5 Global integration

The simulation language has now to refer the previously defined languages and to specify a range for some quantities (e.g. [18.0..38.0/2.0], ['BOX','TABLE','CHAIR']). An extract of this file is given by:

```
&MISC NAME='conf' PLAN='room.ifc' LIB='library.data'
MAX_CONFIGURATION=8 TETAFIN=3600. DTETA=1. ATOL=1.D-9
RTOL=1.D-7  NSAVE=60/

&REAC REAC='REAC_SIMPLE' /

&EXT T= [ 18.0 .. 38.0 / 2.0 ] DP(1)=0./

&LOC ID='LOC1' GEO='LOC1' MAT='CONCRETE_1' T16INI=20.1 /
- A door
&OUV ID='OUV1' GEO='OUV1' ON_INI=.FALSE. /

- Windows
&OUV ID='OUV2' GEO='OUV2' ON_INI=.TRUE. /
&OUV ID='OUV3' GEO='OUV3' ON_INI=.FALSE. /

&SBO ID='FOYER' LOCID='LOC1' SBO= ['BOX','TABLE','CHAIR']/

&ENS ID='GOODS' LOCID='LOC1' EXPR='TMZH > 600' /
&ENS ID='HUMAN' LOCID='LOC1' EXPR='TMZH > 180 && ZD < 1.8'/

&END /
```

To specify a range of values, the configuration grammar is augmented to deal with an interval notation into square brackets with initial, final and the step values (see the &EXT block) or just a comma separated interval values (see the &SBO block) enclosed by a square brackets.

A function read the content of the external files simulation, archi and lib. Based on specific parsers, the simulation file refers the archi and lib files. The simulation file is sent to the generator which get randomly a value in intervals to make a configuration. These steps are repeated many times to produce several configurations (input to the fire simulator). The following code reads the external files, then transforms their contents in specific types (Ifc, Lib, ...), and generates an initial state (ctx[0]). By applying this initial state on the architectural and fire models, it calculates a new state and save it in result file. This step is repeated many times ("N" times).

```
simu      = parseSimu(read("simulation.txt"))
archi     = parseIfc(read("archi.txt"))
lib       = parseLib(read("lib.txt"))
model     = parse(read("model.txt"))
for i  = 1 to get(ctx, "MAX_CONFIGURATION")
    config  = generate-random(seq(simu, archi, lib))
    ctx[0]  = exec(config,[])
    for n  = 1 to get(config,"N")
        ctx[n] = exec(seq(model, config), ctx[n-1])
    write(config ,"config-"+i+".txt")
```

```
write(ctx    ,"result-"+i+".txt")
```

The context "ctx" contains successive values of the quantities calculated by the model for example the temperature in upper layer. The evolution state of the system is saved in "result-" files. The "config" contains the different configurations generated by the generator and saved in the "config-" files.

Based on the definition of undesired events, the results of simulation are sent to the checker to verify if the undesired events have been reached or not. The results of checking and the fire safety level correlated to the occurrence frequency of the UE are saved in the "result" file.

4. CONCLUSION

In order to integrate the different models and (specific) languages required in the field of fire safety on a unique platform, this paper proposed a particular use of functional programming. It allows in a formal way and the definition of higher level function the transformation, integration and composition of several models that do not necessarily use the same language.

In order to formalize and integrate the models of a fire safety system, the first step is to define the languages used by a specific model and to describe it concepts by a grammar. Then define the abstract datatype (or abstract tree) to store the information. Based on the tree definition, specific parsers and some higher order functions must be defined in rder to convert information from a datatype to another, to compose many heterogeneous models and extract required information.

The perspective of this work is to integrate the human behavior in the evaluation system. How to describe and integrate it and which languages and models are necessary?

REFERENCES

A.V. Aho, M.S. Lam, S. Ravi, and J.D. Ullman. *Compilers: principles, techniques, & tools.* Pearson/Addison Wesley, Boston, 2nd ed edition, 2007. ISBN 0321486811.

T.J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5(1):169–226, January 1998. ISSN 1022-7091, 1573-7489.

X Bodart and M Curtat. CIFI computer code: Air and smoke movement during a fire in a building with ventilation ducts networks equipment. In *CIFI Computer Code: Air and Smoke Movement During a Fire in a Building With Ventilation Ducts Networks Equipment*, volume 104, 1987.

N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1959. ISSN 0019-9958.

M. Curtat. *Traité de physique du bâtiment: Tome 3, Physique du feu pour l'ingénieur.* Traité de physique du bâtiment. CSTB, 2002. ISBN 9782868913050.

H. Ehrig and B. Mahr. *Fundamentals of algebraic specification.* Number 6 in EATCS monographs on theoretical computer science. Springer-Verlag, Berlin New York, 1985. ISBN 0-387-13718-1.

M.M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, sep 1992.

L.H. Hu, N.K. Fong, L.Z. Yang, W.K. Chow, Y.Z. Li, and R. Huo. Modeling fire-induced smoke spread and carbon monoxide transportation in a long channel: Fire dynamics simulator comparisons with measured data. *Journal of Hazardous Materials*, 140(12):293–298, February 2007. ISSN 0304-3894.

P. Hudak. *The Haskell school of expression: learning functional programming through multimedia.* Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-64408-9.

B. Martens and .P. Herbert. *ArchiCAD.* Springer, January 2004. ISBN 9783211407554.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. page 124144. Springer-Verlag, 1991.

R. Parry, C. A. Wade, and M. Spearpoint. Implementing a glass fracture module in the BRANZFIRE zone model. *Journal of Fire Protection Engineering*, 13(3):157–183, January 2003. ISSN 1042-3915, 1532-172X.

J. Russell and R. Cohn. *Haskell.* Book on Demand, May 2012. ISBN 9785510997262.

M. J. Spearpoint. Fire engineering properties in the IFC building product model and mapping to BRANZFIRE. 2006.

L. Thiry and M. Hassenforder. Micro languages for systems. *Transaction on Control and Mechanical Systems*, 1(8), January 2013. ISSN 2345-234X.

L. Thiry and B. Thirion. Functional metamodels for the development of control software. *International Federation of Automatic Control, IFAC*, 8, 2008.

L. Thiry and B. Thirion. Functional metamodels for systems and software. *Journal of Systems and Software*, 82(7):1125–1136, July 2009. ISSN 0164-1212.

A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):2636, June 2000. ISSN 0362-1340.