Microprocessing and Microprogramming 38 (1993) 615-623
North-Holland

IDFG: An interactive applications specification model with phenomenological properties

A. Kameas[a] S. Papadimitriou[b] P. Pintelas[c] and G. Pavlides[d]

[a]Department of Computer Engineering & Informatics, University of Patras, Patras 26110, Greece

[b]Department of Computer Engineering & Informatics, University of Patras, Patras 26110, Greece

[c]Dept. of Mathematics, Sector of Computational Mathematics & Informatics, University of Patras, Patras 26110, Greece

[d]Department of Computer Engineering & Informatics, University of Patras, Patras 26110, Greece

In this paper, we present a model based on Data–Flow Graphs called Interactive Data–Flow Graph (IDFG) model. This model is used in the context of a CASE tool (SEPDS) for the specification of distributed interactive applications. Our effort was towards designing a model that would bridge the gap between designers and users perspective by providing the designers with a model that would permit them to "think in users terms". IDFG combines elements of control and data models together with characteristics of the object oriented paradigm, adopting an action–based description of user goals. It is this focus on user actions that gives IDFG a phenomenological nature. In addition, IDFG can be combined with the underlying data model of SEPDS in a straightforward manner, enabling the prototyping of distributed interactive applications. We show that IDFG meets many of the requirements that designers have from current interactive application specification systems, such as user–specified level of abstraction of interaction specification, support of mixed–initiative dialogues, modularity and reusability of interaction. We conclude the paper by presenting a simple example of the application of IDFG and by presenting the state of our research.

1. Introduction

The SEPDS (Software Environment for the Prototyping of Distributed Systems) system is an engineering framework for the prototyping of distributed systems. It provides an integrated set of tools that assist the designers through the prototyping, simulation and profiling phases of the construction of a distributed system [7]. Specification of interactive applications in SEPDS consists of two major phases: the specification of the "workings" of the application and the specification of its interactive features. The tools of the former subsystem were designed and implemented based on the Extended Data–Flow Graph (EDFG) concept, which is itself an extension of the Data-Flow Graph (DFG) concept [6]. In this paper, we will describe IDFG (Interactive Data–Flow Graph) model, which is used by the tools of the User Interface Generating Environment (UIGE), to construct the user interface of the target (produced) application. A description of the process of application specification can be found in [5].

The scope of the design was to bridge the gap that usually exists between the designers and users model of an interactive application. To this end, IDFG enables the designers "think in users terms", thus allowing for consideration of the users preferences and abilities [8] and supports the designers in specifying the interactive features of the application in a non–technical way, leading to a semi–automated user interface construction process based on the semantics of the application graph. In addition, integration and combination of such a model with EDFG, the data model that SEPDS uses for application specification, is straightforward [1].

In the specification of interactive applications, both these models are used: IDFG models the behavior of application objects, while EDFG models their functionality. In this way, designers do not have to use additional effort to learn a new specification language, and the prototyping subsystem of SEPDS is able to produce highly–interactive applications with few changes.

The model we present combines features from both state–based models (e.g. [4, 9]), which incorporate powerful control models, but do not handle data and user models well (such models are adequate for the fine–grained, low–level specification of user actions and action sequences, but are hard to use by non–computer experts), and user–oriented models (e.g. [3]) which focus on representing interaction with interconnected plans of various levels of detail and complexity, to meet the needs of all the categories of users between novice and technically expert. Description of the elements of the screen has been separated from the specification of actions that may be performed with these elements, leading to a more user–centered design. IDFG is *action–oriented*, assigning to user actions well–defined semantics based on the goal–based structuring of these actions, thus making the screen elements that represent these actions "invisible" (the ready–to–hand phenomenological property [12]).

A model (Propositional Production Systems - PPS) with similar characteristics was proposed in [10]. PPS systems however, are inadequate for our purpose for two reasons: they do not represent explicitly the user perspective and way of thinking (although they permit abstractions of the interaction to be formed), and they can not be integrated in SEPDS without adjustments.

Both EDFG and IDFG can be proved equivalent in expressiveness with Petri Net models that use time stamps. Although Petri Nets have been used before for representing interaction [2], IDFG has certain representational advantages: first, it is a more "compact" model, and thus, easier to use by the non expert designer. In addition, in IDFG we use semantic (and not technical) description of actor interaction (that is why links are typed and labeled). Furthermore, in IDFG, user actions and goals are explicitly represented; the same is true for system actions and screen condition. Finally, IDFG supports events caused both by the user and the system (modelled as user and system actions). This means that the system could as well initiate the execution of certain tasks, thus laying the modeling base for the specification of intelligent user interfaces.

In the next section, our contribution, the IDFG model, is described along with its most important properties. A discussion on the models properties follows with an example of an application of the model. The paper concludes with a presentation of our research directions.

2. The IDFG model

An IDFG is a bipartite graph consisting of a set L of *links* and a set A of *actors*. Actors are the computational components, while links are used for data transportation. There also exists a set E of directed arcs which connect actors to links and links to actors. The function each actor performs consists of two parts: the *behavioural part*, which is made up of rules, and the *functional part*, which contains code segments. For every rule of the behavioural part, there exist a set of left–hand–side conditions that must hold for it to fire, and a set of right–hand side conditions that result from the firing. These are represented by the PRE and the POST fields, respectively, of the actor. A restriction that applies is that *no two tokens of the same link may be consumed in the same actor firing.* The code segments of an actor (which are represented with field FUN) are EDFG actors; this is how IDFG can be integrated with the underlying model of SEPDS.

Links correspond to conditions; those that currently contain tokens describe the current situation. Links are typed (mainly to distinguish between the components of a situation). The types of links are contained in the set { *user actions, system actions, object conditions, goals, incommunication, outcommunication* }. To improve expressibility, more link types can be added to this set.

Two additional constraints are imposed on the system:

- *every actor must have a link of type user*

action or system action in its PRE, which means that state transitions take place only as a consequence of an event (user or system action)

- each user action must belong to a goal–leading sequence, that is why, *each actor has also a link of type goal in its PRE and POST* (a goal link in the POST is used to signal successful goal achievement)

Actors can be of two kinds: *action actors* and *context actors*. With each action offered by the application to its users (that is, with each command that is transferred by the user interface to the underlying application), an action actor is associated. The number of action actors is finite and equal to all the commands supported by the application. In order for such an actor to be ready–to–fire, all links in its PRE except user or system action links must already contain tokens. This means that the user interface must have reached the appropriate state (as represented by the condition links) and the actor must belong in one of the contexts the user is currently working with (as represented by the goal link) for the user action to be available (any missing links are interpreted as "don't care" conditions). The effects of its execution are modelled with the production of tokens in the actor's POST.

To model context of operation and to support the goal–based structuring of user actions, we use context actors. Their functionality is to correctly interpret user actions in order to appropriately decompose user goals into subgoals, so that eventually the correct action actor will fire. To infer the context of operation, these actors contain rules that fire depending on the user interface action that the user performs. Context actors may be formed by combining action actors or context actors; this process may be applied an adequate number of times so as to represent all user goals and subgoals.

Commands are modelled with actor firings, and sequences of commands with actor firing sequences (graphs). Any portion of an IDFG could be executed if tokens were created on the PRE of the first actor of the sequence. Thus, if we were to start the system from an arbitrary situation, we would have to provide a mechanism that would create tokens on the links of the actors that must be fired first.

To transparently support interaction across distributed contexts, we require that *there exists one separate IDFG for each autonomous application process*. In such a system, we have to model the effects that processes of this kind may have on one another. To this end, we use a special link type, the *communication* type. In effect, there exist *in-communication* and *outcommunication* link types to account for the direction of communication. The actors that contain rules that result in intercontextual communication are called *communication actors* [11]. On the other hand, links of type *system action* are used to model system–initiated communication among actors of the same IDFG.

2.1. Actor composition

When context actors are constructed, the construction process must be defined, together with the link types and the execution semantics of the resulting actor. As far as goals and user and system actions are concerned, *lower level goals are derived from user or system actions and goals of the next higher level*. To achieve such a transformation when a context actor is formed, we use Primitive Graphs (PGs) to specify the type of the context actor.

Actor composition is a model property that enables the designer define subgraphs that would correspond to user goals, with the context of user actions encapsulated in their structure. These subgraphs have many valuable properties, like integral design and execution, incremental goal representation, reusability of interaction portions, and may be used for the automated production of user interfaces which are specified by the designer in a goal–structured way.

2.2. Types of context actors

PGs are special actors with well–defined executional semantics, which represent basic operations that come up often enough to make us represent them in a distinct way. The PGs that we will use are: And PG (APG), Or PG (OPG), Not PG (NPG), Sequence Start (SEQS), Sequence Continue (SEQC), Sequence End (SEQE), Enable

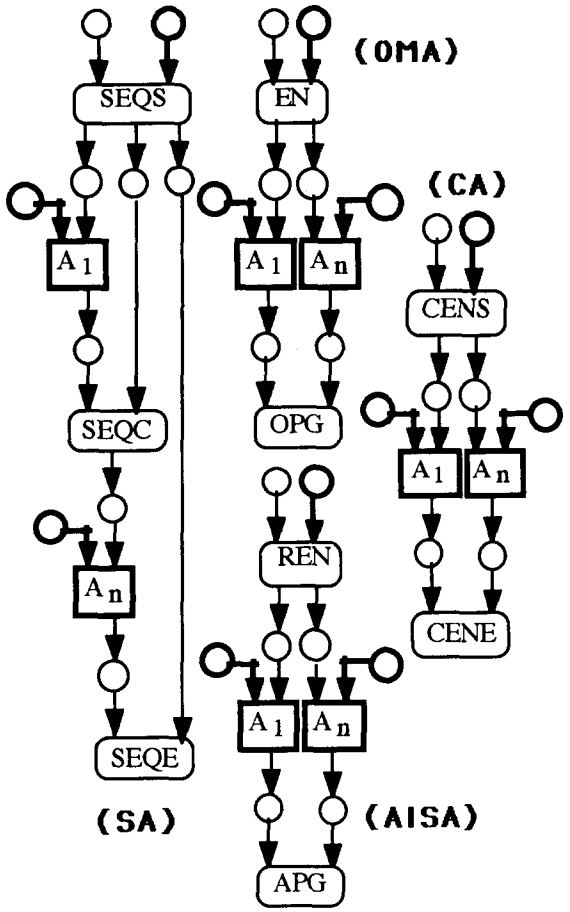

Figure 1. The four types of context actors used in IDFG model

(EN), Repeatedly Enable (REN), Concurrently Enable Start (CENS) and Concurrently Enable End (CENE) (their functionality can be deduced from the example at section 3.4), while some of them are analytically described in [11]. PGs may be used to model link combinations and actor interrelation and are needed because complex actors model situations that are difficult to represent analytically.

Context actors (depicted in figure 1) are of the following types (in all figures containing IDFGs, we have used thick circles to represent user action links, plain circles to represent other links, rectangles to represent actors, double-line rectangles to represent EDFG actors, filled rectangles to represent communication actors and round-edge rectangles to represent PGs):

Sequence (SA): this actor decomposes a higher level goal and action to a lower level set of goals that must be achieved sequentially. After the sequence is initiated (as signalled by PG SEQS), each actor in the sequence is ready to fire. An actor fires after the appropriate event takes place and the previous actor has successfully terminated execution (PG SEQC). The construct is exited after the last actor in the sequence has terminated execution successfully (PG SEQE).

One-out-of-many (OMA): this actor decomposes a higher level goal and action to a lower level set of goals, one of which will eventually be achieved (this situation is equivalent to providing the user with different decision paths). All actors are ready to fire (as signalled by PG EN); the one that will eventually fire is determined by the next event. The construct is exited when one actor terminates execution successfully (PG OPG).

All-independent-of-sequence (AISA): this actor decomposes a higher level goal and action to a lower level set of goals, all of which must eventually be achieved, but the sequence of achievement is not important (this situation is equivalent to providing the user with alternative decision paths). All actors are ready to fire; the firing sequence is determined by the next event (PG REN). The construct is exited when all actors have terminated execution successfully (PG APG).

Concurrent (CA): this actor decomposes a higher level goal and action to a lower level set of goals, one or all of which will eventually be achieved, but the sequence of achievement is not important (the construct is initiated with PG CENS). The difference here is that subgoals may be achieved concurrently, enabling the user to work with several contexts at the same time. The construct is exited when all IDFGs terminate execution successfully (PG CENE).

3. Application of the model

In this section we discuss some general properties of IDFG model and present an example of its

application.

3.1. Abstraction and inheritance

Inheritance, as contrasted with composition which is a model property, is an implementation property. In IDFG, inheritance is incorporated by abstracting away the functionality of actors, much like in the object–oriented paradigm. To incorporate inheritance, we use field INHERITS, which may be used to represent whether an actor inherits the functionality of one or more classes of actors (multiple inheritance), or is a primitive one.

Abstraction can be used to improve reusability when the specified user interface will be implemented by the system. Reusability is achieved by using *slots* in the PRE and POST of the actors. Slots are also typed, and can accept links of the same type only. In this way, designers will not have to start from scratch each time they produce a new application. Conversely, subgraphs that represent user goals at any level can be stored in a library and reused or adapted to the needs of the new application.

3.2. Prototyping of applications

Prototyping of applications is a task of SEPDS, and therefore its description is not entirely within the context of this work; we will however, give an outline of the process, while more can be found in [7].

SEPDS supports the top–down refinement of system prototypes; this process consists of two steps: object (actor) refinement and functional refinement. In addition SEPDS supports the partitioning of the prototype in order to find a more appropriate size of objects and exploit the parallelism inherent in the application. The application specification models supported by SEPDS offer the appropriate constructs (for example, actor composition, inheritance, actor abstraction, PGs etc) to support these processes. The prototyping subsystem of SEPDS consists of the DFG Modeler and the Template Manager. The former provides the tools for building and maintaining a DFG, while the latter contains tools for maintaining a software base and for building an executable prototype. Note the key role of PGs in prototyping: the designer must eventually program the functional part of actors (or use an actor from the actor base); PGs however, are code segments automatically provided by the system.

For example, in IDFG, we assume that there exists an *external event–handler*, (EEH) which gets user input and sends it to the other IDFGs. EEH should not wait for response to the token it communicates; instead, *it must communicate a token each time a user action is recorded and identified.*

3.3. The phenomenological nature of IDFG model

The interaction specification model must help the designers transfer effectively the model of the application in their minds to the end–users of the target application. That is why it must enable the designers think in users terms. When using a system, users have in mind a *goal* they want to achieve, and try to do that through a sequence of *operations* that the system supports. If the system is interactive, users adapt their next operation to the effect that the last (or a sequence ending with the last) produced. One cannot help noticing a *recursion* in this description: to achieve the overall effect, users must achieve intermediate effects.

Definition 1. Action: every operation users perform, which may affect their goal–pursuing strategy; usually, each action is reflected in the user interface. The set of actions includes all the operations supported by the user interface and the application, as well as those actions that may be taken by the system itself. Subsequences of actions lead to the achievement of subgoals, whether users intended to pursue them or not.

The emphasis must be placed on the availabilty of actions: users in every moment think of *available actions* that will lead to their goal. At any moment, a set of actors (the actor–ready list) contain tokens in all their PRE links except the action link. These actors represent the actions that are available to the user (or the actions that the system may take).

Traditional DFG models interpret the notion of state as the distribution of tokens on the DFG links. Our model extends this notion:

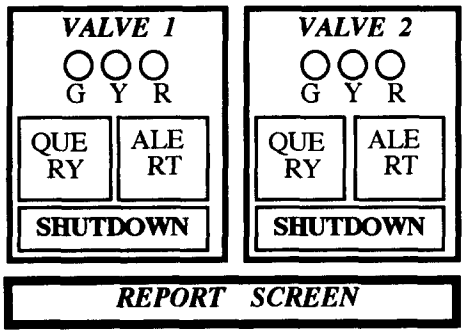

Figure 2. The user interface of the example application

Definition 2. State is the set of actors in the actor–ready list, or equivalently, the set of user or system actions that the actors in the actor–ready list represent. Since these actions correspond to goals in a lower–level, we may equivalently say that state is represented with the set of goals that may be achieved as a consequence of user of system actions permitted by the actors in the actor–ready list.

Definition 3. State transitions occur as a consequence of an actor firing which causes the output of tokens in the actors OFS links, modifying the actor–ready list. Note that although an actor firing may affect links of object condition type, these are invisible, as far as the user is concerned.

Since, however, an actor firing depends on its PRE, it is clear that among a set of otherwise identical actors that are ready to fire (the actor–ready list), the one that fires is determined by the link of type *user action*. That is why we claim that our model has phenomenological properties [12]. Note in addition, that the firing actor may be determined also by a link of type *system action*. This enables us to model systems where state transitions may occur independently of user actions, such as knowledge–based systems (where the system itself may initiate a phase of interaction) and real–time systems (where the system may have to take default actions on the absence of human response within a pre-specified time limit). In order to correctly recognize the context of the next event, each independent IDFG has its own locus of control, which is used to determine the next actor that will fire. Furthermore, this property can be used for the resolution of firing conflicts: all actors that can eventually fire, do so, and the consequences of firing appear in the graph in the form of a new marking.

To construct an application using IDFG, a designer must think of the goals and subgoals that a user may achieve using the application, the actions that need to be made available to the user, and the action sequences that lead to these goals. The objects affected by users actions are directly dependent on the context of operation, and thus are not so important (these are mainly used to represent screen effects of user actions, and not as carriers of semantic information). Designers are therefore forced to think in a more "goal–oriented" and "goal–efficient" way, which we believe, comes closer to a users way of thinking when using an application.

3.4. An example

To demonstrate the capabilities of IDFG, we will use it to describe a simple interactive application, where a user checks two screens that show the condition of two valves and acts accordingly (figure 2). Each screen contains three lights: green light means that valve is OK, yellow light means that valve is not functioning in its full capability and red light means that valve is malfunctioning. The user may react to valve condition by first querying a database about valve condition readings and then either sounding an alarm, or shuting down the valve. Although the two valves reside in remote sites, the user can handle them concurrently. To keep the application simple, we assume that the system imposes no real–time requirements on the user.

Figure 3.a shows that the user may work with any of two screens (context actor of type CA is used for actors V1 and V2; note that at this level of design, the location of actors is not important). To represent the functionality of each screen, each actor is analyzed (figure 3.b). It seems that (context actor of type SA) first a light is turned on and subsequently the user must press check

button (communication actors LGT and CHK, each of which must contain an EDFG actor) and then act (context actor ACT of type OMA) by pressing either alert or shutdown button (SEPDS actors ALR and SHD in figure 3.c, respectively).

Figure 4 contains the description of some of the actors of figure 3.

4. Conclusions

We have described IDFG, a DFG–based model that can be used for the specification of interaction features of distributed applications. This model explicitly represents events (user and system actions), event effects (object conditions) and user plans (goals), enabling the designers to describe applications from the users perspective. Although IDFG combines state representation with rule–based description of execution, emphasis is placed on availability of actions to the user, giving this model a phenomenological nature. IDFG can be proved equivalent to Petri Net models, and in addition, it is more expressive and easy to use by the non–technical designer.

Furthermore, by integrating data, control and user models, IDFG supports mixed–initiative interaction, inheritance, modularity and reusability of interaction parts, and provides a user–specific level of abstraction, enabling the designer to "fly over the forest and select the appropriate tree".

This model serves our needs because it can be combined in a straightforward manner with EDFG, the DFG–based model that SEPDS supports. By tying description of interaction to application development, IDFG points directly to an object oriented implementation of the application by SEPDS prototyping subsystem.

Our next step is twofold: on one hand, we are studying the processing overhead that UIGE has on the overall SEPDS performance, and that of a user interface constructed with UIGE on the target application performance and on the other, we are building a knowledge base of design guidelines that will be integrated with UIGE, so that these can be used by the prototyping subsystem of SEPDS in association with the description of the application. We expect the rule–based nature of IDFG to make this integration much easier.

REFERENCES

1. J.D. Foley, D.J.M.J. de Baar and K.E. Mullet, *Coupling application design and user interface design.* Proceedings of the CHI92 Conference: Striking a balance, May 3-7, 1992, Monterey, USA, pp 259-266.
2. R. Bastide and P. Palanque, *Petri Net Objects for the design, validation and prototyping of user–driven interfaces.* Proceedings of INTERACT 90, The IFIP TC 13 Third International Conference on Human–Computer Interaction, August 27-31, 1990, Cambridge, U.K., pp 625-631.
3. J. Bonar and B. Liffick, *Communicating with high–level plans.* In Intelligent User Interfaces (J. Sullivan and S. Tyler eds), ACM Press, 1991, pp 129-157.
4. A.J. Dix and C. Runciman, *Abstract models of interactive systems.* In Proceedings of the British Computer Society Conference on People and Computers: Designing the Interface (P. Johnson and S. Cook eds), Cambridge University Press, 1985, pp 13-22.
5. A. Kameas, S. Papadimitriou and G. Pavlides, *Coupling interaction specification with functionality description.* Proceedings of the 1993 East–West International Conference on HCI, August 3-6, 1993, Moscow, Russia.
6. K. Kavi, B. Buckles and V. Bhat, *A formal definition of data flow graph models.* IEEE Trans. on Computers, C-35(11), 1986.
7. A. Levy, J. van Katwijk, G. Pavlides and F. Tolsma, *SEPDS: A support environment for prototyping distributed systems.* Proceedings of the 1st International Conference on System Integration, April 1990, New Jersey, USA.
8. A. Marcus and A. van Dam, *User Interface developments for the nineties.* IEEE Computer, 24(9), 1991, pp 52.
9. D.R. Olsen, *Push–down automata for user interface management.* ACM Trans. on Graphics, 3(3), 1984.
10. D.R. Olsen, *Propositional Production Systems for dialog description.* Proceedings of the CHI90 Conference: Empowering People, April 1-5, 1990, Seattle, USA, pp 57-63.
11. S. Papadimitriou, A. Kameas, P. Fitsilis and

G. Pavlides, *A new compression technique for tools that use data–flow graphs to model distributed real–time applications.* Proceedings of the 5th International Conference on Software Engineering and its Applications, December 7-11 1992, Toulouse, France, pp 235-244.

12. T. Winograd and F. Flores, *Understanding Computers and Cognition: A new foundation for design.* Ablex publishing, 1988, p 207.

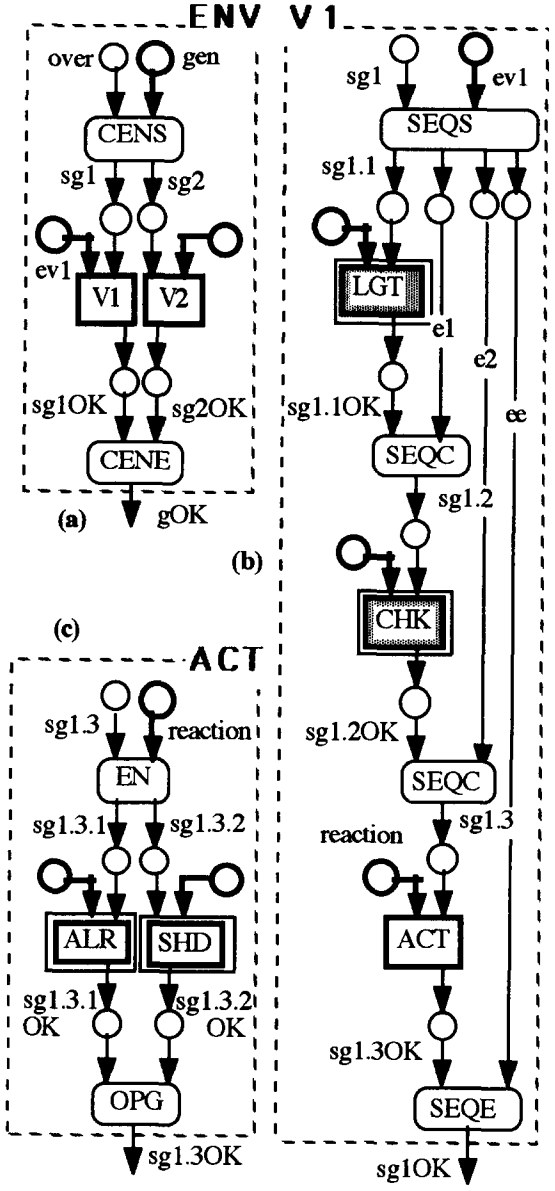

Figure 3. The IDFGs that represent the application of the example

Definition of actor ENV

goal: over(control valves)
user actions: gen(inside valve 1, inside valve 2)
system actions: gOK
behavioural part
rules: PG CENS
control valves, inside valve 1 -> sg1
control valves, inside valve 2 -> sg2
PG CENE
sg1OK -> gOK
sg2OK -> gOK

Definition of actor ACT

goal: sg1.3
user actions: reaction(press ALR button, press SHD button)
system actions: sg1.3OK
behavioural part
rules: PG EN
sg1.3, press ALR button -> sg1.3.1
sg1.3, press SHD button -> sg1.3.2
OPG
sg1.3.1OK -> sg1.3OK
sg1.3.2OK -> sg1.3OK

Definition of actor BUTTON

goal: < * * * >
user actions: < * * * > (press, release)
object conditions: normal, selected
incommunication: < * * * >
system actions: < * * * > OK
outcommunication: < * * * >
behavioural part
rules:
<goal>, press, normal -> selected
<goal>, release, selected -> normal, <outcommunication>
<goal>, <incommunication> -> <system actions>OK

Definition of actor LGT

goal: sg1.1
system actions: enLGT
incommunication: LGTstatus, LGTlit
system actions: sg1.1OK
outcommunication: setLGT
behavioural part
rules:
sg1.1, enLGT, LGTstatus -> setLGT
sg1.1, enLGT, LGTlit -> sg1.1OK
functional part
Actor LGT is a communication actor that receives the state of the light from the application DFG and sets the light appropriately. Consequently, the functional part is the mechanism that sets the light

Definition of actors ALR and SHD

Actors ALR and SHD (and a sub-actor of actor CHK) function similarly, as buttons. Thus, they may inherit their functionality from another actor of type BUTTON. In such an actor, slots are represented with < *** >. For example, for actor ALR, INHERITS=button, with goal=sg1.3.1 (sound alarm), user actions=alarm, incommunication from the application DFG that sounds the alarm, system actions=sg1.3.1OK and outcommunication to the DFG that sounds the alarm. Rules describe its functionality. Specifically, when the button is released, outcommunication is sent to the alarm sounding DFG. When incommunication is received from the DFG, the subgoal that corresponds to the button is achieved

Figure 4. Description of the IDFGs of the example application