A Case-based Assessment of the FLUIDE Framework for Specifying Emergency Response User Interfaces

Erik G. Nilsson
SINTEF ICT
Oslo, Norway
Erik.G.Nilsson@sintef.no

Ketil Stølen
SINTEF ICT
Oslo, Norway
Ketil.Stolen@sintef.no

ABSTRACT
In this paper, we report the results from assessing the FLUIDE Framework for model-based specification of user interfaces supporting emergency responders. First, we outline the special challenges faced when developing such user interfaces, and the approach used in the FLUIDE Framework to meet these challenges. Then we introduce the framework, including its two specification languages. Thereafter, we present the case addressing the specification of user interfaces for three existing emergency response applications. Based on these specifications, we discuss how well we succeeded, concluding that we were able to describe the applications in a comprehensive and understandable way taking similarities and difference between the applications into account. The language constructs function as intended, having two languages has proven valuable, and the specifications scale quite well.

Author Keywords
User interface specification languages; Emergency response

ACM Classification Keywords
D 2.2 Design Tools and Techniques: User interfaces

INTRODUCTION
Emergency response operations are very varied, from simple everyday incidents to long-lasting serious catastrophes. More complex operations tend to have a fast changing nature, sometimes being almost unpredictable. Developing ICT solutions supporting such work is challenging. User interfaces (UIs) of ICT solutions need to adapt to and reflect these variations, and their design are therefore particularly challenging.

Support for UIs on equipment with different screen sizes is important to allow local leaders at the incident site employ the same applications in the same intuitive way on different kinds of equipment [15]. For field workers it is important to have non-intrusive ICT support, possibly offering non-visual modalities as an alternative to or in combination with visual presentation and interaction. To the extent emergency responders have ICT support today, needs for flexibility are often addressed by generic, data oriented ICT solutions forcing them to adapt to the solutions, and not the other way around.

There are on the other hand also many similarities and patterns between emergencies. These include types and occurrences of emergency response operations, as well as the actors involved in such operations. Furthermore, tasks and information needs have similar communalities across operation types, actual operations and agencies. In [16] we have argued that the similarities and pattern may be characterized by a limited number of categories of functionality.

The approach put forward in this paper is to provide components supporting these categories of functionality, combined with means for composing end-user solutions from these components. The components need to be flexible and tailor-friendly, i.e., they need to combine being ready-to-use with being highly configurable. Composition is primarily done at design time, while configuration (and certain types of composition) may also take place at run time. Developing UIs for this kind of solutions using traditional programming languages with connected libraries is very challenging to the extent it is at all possible. It is extremely resource demanding because all imaginable combinations of functionality, compositions and configurations must covered.

Model-based UI development approaches [11] are well suited to meet the needs for cross-platform and cross-modality support, but existing model-based UI development approaches are also seriously challenged by the requirements for flexibility. There is a need for building blocks that:

R1. Are at a sufficiently high level of abstraction to support development of UIs that work across platforms and modalities
R2. Provide compound structures of simple elements and containers/dialogs to support common specifications between platforms and modalities
R3. Have reflection mechanisms giving an awareness of model structures (including domain models) to support adaptation both at design and run time

EICS'16, June 21 - 24, 2016, Brussels, Belgium

ACM 978-1-4503-4322-0/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2933242.2933253

R4. Support development of UIs where the layout depends on the instances at run time, typically using icons, maps, graphical elements, as well as alternative modalities like speech
R5. Provide specific and explicit support for UI patterns and styles that are particularly useful in the emergency response domain

MARIA [20] meets R1 well, but fulfills R2 only partly. It offers building blocks that are abstractions of simple UI elements (elements for entering data, presenting data, activation functions, etc.) as well as abstractions of containers for structuring these (including top level dialogs), but not any composed ones. With the chosen building blocks the composition structure of the UIs – which is different when the platforms have large differences – is reflected in the specifications, also at the abstract level. It meets R3 quite well by offering adaptation between platforms through a mitigation mechanism (including a run time system), but this does not support other types of adaptation on single platforms. R4 is partly met by offering support for different modalities, and a possibility to show map-based UIs. The latter is though offered through composing external UI services and specifying such UIs does not seem to be directly supported. As MARIA is a general purpose language, R5 is not met.

Also UsiXML [8] meets R1 well, but fulfills R2 only partly for the same reasons as MARIA, although some compound components like tables are offered. It meets R3 to some extent. The connections to domain models are through transformation, but as these work both ways, a degree of traceability is achieved, and some models are available at run time to facilitate adaptation. The adaptation is though restricted to fitting a UI to similar devices with different form factors, not to platforms with large differences [12]. UsiXML is extensible, and there is an extensive family of related approaches providing various extensions [24]. Some of these extensions enhance the support for R3 [10], while other address R4 (including support for maps and 3D UIs), but there is no common, integrated support. Another extension supports development of a UI for a flight cockpit [7], an application area having similarities to emergency response. Except for this, R5 is not met.

CAP3 [23] meets R1 and R2 in similar ways as MARIA and UsiXML, although the building blocks are slightly more abstract. According to [23], CAP3 supports adaptation to context, but this seems to be achieved through service integration and not involving any reflection mechanisms, showing limited or no support for R3. Except for supporting maps and live content like movies, R4 is not met. Only visual modalities seem to be supported. R5 is not met.

ICOs [13] employs a different approach by focusing on the UI behaviour, abstracted using Petri nets. It does not support cross-platform specifications, and does therefore not meet R1, but it offers an abstract iWidget construct to embed the presentation part of UIs specified by other means. R2 is not met, neither. R3 is partly met by providing a run time system. Reflection is provided through such mechanisms in Java. Both seem to be used primarily to provide interactive development support. ICOs put emphasis on supporting development of post-WIMP UIs, so R4 is met. R5 is partly met, as ICOs have proven useful in closely related domains like command and control, air traffic control and cockpit systems.

The recent OMG standard IFML (Interaction Flow Modeling Language, available at www.ifml.org) meets R1 only partly, as the building blocks are on a lower abstraction level than MARIA, UsiXML and CAP3, and only visual web-based UIs are supported. It meets R2 to the same degree as MARIA, UsiXML and CAP3, but supports neither R3, R4 nor R5.

We have developed the FLUIDE Framework offering building blocks fulfilling R1-R5. How this is done is explained in the next section. In this paper we present the framework in an example-based manner, and report results from a case-based assessment of the framework.

THE FLUIDE FRAMEWORK

The FLUIDE Framework contains: (a) a collection of ready-to-use and highly configurable components supporting flexible composition of end-user solutions for emergency responders, (b) composition and configuration approaches, (c) the FLUIDE Specification Languages, (d) a generic mechanism transforming specifications to components or applications, (e) the FLUIDE Method supporting the use of the framework. The current version of the FLUIDE Framework is a first prototype, and thus the maturity of the different parts vary. The most mature part, and the subject of this assessment, is the FLUIDE Specification Languages: FLUIDE-A is used for expressing abstract UI (AUI), while FLUIDE-D is used for expressing concrete designs, usually denoted concrete UI (CUI). FLUIDE-D provides specific support for the emergency response domain through a library of UI patterns that are particularly useful for this domain, and may be automatically transformed to a final UI (FUI).

The FLUIDE Specification Languages

The UI of an emergency response application must support the work performed by emergency responders. The four main language constructs in FLUIDE presented as rounded rectangles in Figure 1 support a natural breakdown of such work (rectangles). Emergency response work can be categorized with respect to responder types, responder roles and high level tasks, as well as combinations of these. The categories of functionality [16] support categories of work and the task structures these categories contain.

In FLUIDE-A, the *Category Manager* construct facilitates the specification of a whole application, or some part of it. A category of functionality supports certain work performed by emergency responders. Such work can be divided into tasks on different levels.

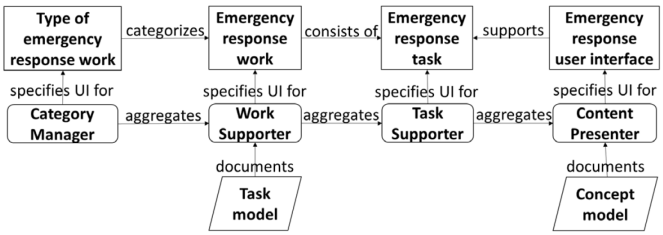

Figure 1. Overview of the main constructs in FLUIDE-A

These tasks may be categorized both in a hierarchical goals/means structure and through temporal constraints between sets of tasks. Such task structures are specified using the *Work Supporter* construct, which includes a task model to specify hierarchical and temporal structures. The concrete syntax of FLUIDE-A uses a neutral hierarchical task model syntax to express the hierarchical structure. The temporal structure is expressed using operators. A special kind of Work Supporter aggregates other Work Supporters recursively.

A UI supporting one task is required to manage information content relevant for solving the task. The information needs of individual tasks are specified using the *Task Supporter* construct. How the information content used in a Task Supporter is further broken down and structured in (part of) a UI is specified using the *Content Presenter* construct. The information to be presented by a Content Presenter is specified by a concept model where all entities are connected through relations. The concept model, together with the specification of an *anchor* (the root entity of the model), is sufficient for determining which information is to be presented in a FUI at run-time. FLUIDE-A employs a subset of the UML class model syntax (extended with the anchor) to express the models. Additional platform-independent visual properties are expressed using annotations. As the same information may be useful when solving different tasks, and because other tasks may only require a subset of the same information, Content Presenters may be specified hierarchically.

In the CAMELEON glossary[1], one of the definitions of *interactor* is: "A computational abstraction that allows the rendering and manipulation of entities (domain concepts and/or tasks) that require input and output resources". The four main constructs in FLUIDE-A may be understood as interactors in this sense. We use *interactor construct* as a common term for these constructs, and the term *interactor instance* to refer to an occurrence of an interactor construct in a specification.

FLUIDE-D is used for specifying designs for FLUIDE-A specifications, and contains variants of the four main constructs in FLUIDE-A, using the same names with the suffix *design*. The interactor design constructs in FLUIDE-D are used to specify which parts of the domain and task models that are to be included in a FUI. FLUIDE-D's core is the library of UI patterns, operationalized in the *view* constructs, including *content views* for presenting the instances corresponding to the concept models. Views are used to specify how some part a FLUIDE-A specification is to be presented on a given UI platform using certain modalities and UI styles. The view constructs in FLUIDE-D make it possible to specify designs for a given FLUIDE-A specification for different target platforms through adding minimal amounts of platform specific specifications.

In the FLUIDE Specification Languages, R1 is met by having building blocks both for the abstract and concrete UIs that are at a higher abstraction level than corresponding building blocks in the approaches discussed above. R2 is met by providing compound building block as constructs in the languages. The difference compared to other approaches is most evident for the *content views* used as part of the FLUIDE-D specifications. Such views specify how the instances corresponding to a concept model fragment is presented. They provide means for specifying quite advanced designs in a very compact way through exploiting pairs of UI patterns and model patterns. These views support UI patterns that are particularly useful in the emergency response domain, and in this way FLUIDE meets R5. We use the term *UI pattern* to denote a user interface design pattern, i.e., a pattern focusing on reoccurring visual and structural aspects as well as generic behaviour of user interfaces. Compound building blocks are in their nature more specialized than simple ones. To counter for this, the views provide versatility through being based on model patterns. We use the term *model pattern* to denote patterns of the same type as Gamma et al. [6], i.e., expressed in terms of a concept model. This means that the views may be used to specify advanced UIs managing a wide variety of information as long as the information to be presented has a structure that matches the model patterns used in the view. E.g., the *Map Icons View* provides means for specifying an icon-based presentation of any type of information in a map UI as long as the model follows a given structure (including providing locations) – working just as well for presenting incident objects, resources, victims, important locations or risks. Thus, such views combine being specialized and powerful with regards to emergency response need with being versatile with regards to the actual information they present. The compound view types in FLUIDE-D also support specification of UIs where the layout is depending on the instances at run time, thus meeting R4. R3 is supported in FLUIDE through enabling reflection mechanisms by embedding domain models as part of the specifications. This includes the concept models used in the content views just discussed, as well as task models. This also provides traceability, which reduces the challenges connected to roundtrip engineering which is inherent in model-based systems development.

EMERGENCY RESPONSE CASE STUDY

In order to assess the FLUIDE Framework we retrospectively specified the UI of three existing emergency applications

[1] http://giove.isti.cnr.it/projects/cameleon/glossary.html

(without any connections to FLUIDE), all three of which were developed as part of the research project BRIDGE (www.bridgeproject.eu). The three applications are: MASTER, eTriage and Resource Manager. We denote this the target UI. The advantages of specifying already existing applications are realism and that we do not need to do UI design from scratch. Using three applications in the case provides variation with regards to users, tasks, managed information, platform and style. The disadvantage on the other hand is that we mainly assess the suitability of the FLUIDE Specification Languages. The full description of the three applications and the corresponding FLUIDE specifications are available in [17]. In the following, we present excerpts from these specifications to provide more background on the FLUIDE languages and their usage.

The MASTER Application

The MASTER application consists of a large map display showing information overlays (icons and other visual representations) with relevant information for the emergency response at hand. All information overlays on the map belong to one of five categories. The application also includes a *ribbon* showing the information elements in the overlays grouped by these categories, which are further divided into sub categories. On the top level, the ribbon contains a set of buttons for accessing the information in each category, as well as a ticker showing summary information – as illustrated in Figure 2. The ribbons for each of the categories are similar to each other, but present different types of information. We only show the ribbon for the *victims* category (Figure 3). It contains some overview information at the left and icons for each of the sub categories in a horizontal scrollable view at the right. Each victim is represented by an icon in the ribbon and on the map. The plus icons represent functionality for adding elements of the given sub category. The information in the victims category may also be presented in a tabular form, as shown in Figure 4. The *Victims Presenter* (Figure 5) specifies both the right hand part of the ribbon for the victims category and the tabular presentation of victim information in FLUIDE-A, using the Basic Content Presenter construct. The outer border with a name is used for all FLUIDE-A interactor constructs. Decorations on the border determine the construct and whether it is basic or aggregated.

Figure 2. Ribbon showing the ticker and the top level buttons

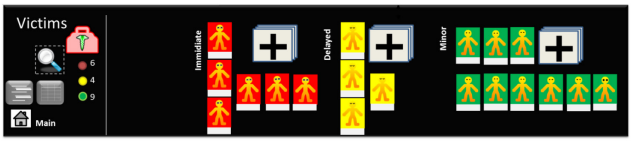

Figure 3. Ribbon content for the victims category

The UI annotations specified in Figure 5 express which icon that should be used to visualize instances of the entity *Victim*, rules for displaying icons, labels, as well as visualization of entities or attributes. The left side of the ribbon for the victims category is specified by a Basic Content Presenter called *Victim Summary Presenter*, which is not included in this paper. This presenter, together with the *Victims Presenter* (Figure 5), contain the needed information for specifying the ribbon category in Figure 3. The specification of this in FLUIDE-A is shown in Figure 6 using an Aggregated Content Presenter. Aggregated presenters include only the border parts of their members, and the members' names are shown inside the presenter instead of in the heading.

Figure 7 shows a FLUIDE-D Basic Content Presenter Design specifying the right hand part of the ribbon for the victims category (Figure 3). The outer border of a FLUIDE-D specification resembles the FLUIDE-A border, but it also specifies UI style(s) and modalities/platform(s) the design is targeted at.

Patients

	Status	Description	BP	Temp.	Name	...
1		Unconscious	139/90	38	Unknown	...
8		Seriously bleeding	80/40	35	Unknown	...
3		Deliriously	200/140	40	Per Olsen	...
5		Broken leg	...	...	Jon Tronstad	...
2		...	...	...	...	...
11		Scratches	...	...	...	...
15		Dead	...	...	...	...

Figure 4. Tabular presentation of victim information

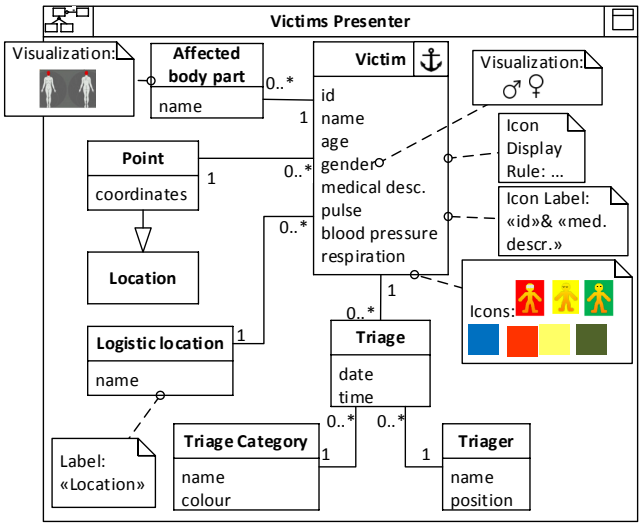

Figure 5. FLUIDE-A spec. of all the Victims sub categories

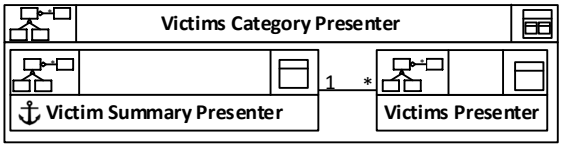

Figure 6. FLUIDE-A spec. of the Victims Category (Figure 3)

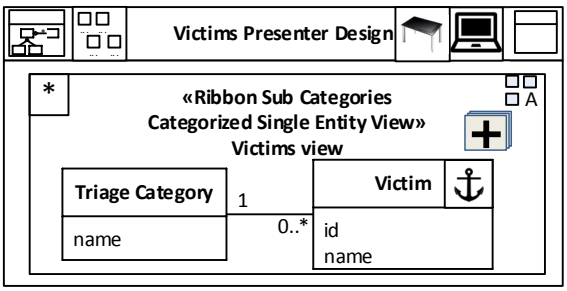

Figure 7. FLUIDE-D spec. of all the Victims sub categories

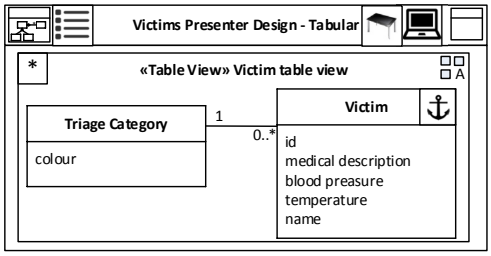

Figure 8. FLUIDE-D spec. of the Tabular UI in Figure 4

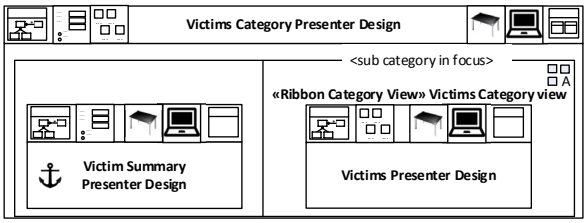

Figure 9. FLUIDE-D specification of the Victims Category

Presenter designs contain one or more views. There are four main types of views, i.e., layout manager view, decorational view, content view and content integration view. The view shown in Figure 7 is a content view, i.e., a view that presents instances of one or more entities. Content and content integration views use UML stereotype notation to denote the view type before its name. A *1* or * on the top left of a content view denotes whether the view presents one or a number of instances of the anchor entity. The available content and content integration views make up the FLUIDE library of emergency response UI patterns.

All content views impose restrictions on the model fragment they may present, expressed in FLUIDE-D as a model pattern fitting the UI pattern supported by the content view. The model pattern for the domain specific content view used in Figure 7 must contain the entity to be presented *(Victim)*, and a categorizing entity *(Triage Category)*. It presents a number of instances of the presented entity as a set of grids or tables of icons. The number of grids is determined by the number of instances of the categorizing entity. These instances also determine the labels for each sub type. The plus icon on the right hand side of the view is specified using a property setting for the view. The icons to use in the grid are obtained from the icon annotation on the *Victim* entity in the corresponding FLUIDE-A specification.

Figure 8 shows a Basic Content Presenter Design specifying the Tabular presentation of victim information (Figure 4). It contains one generic content view. Its model pattern must contain one entity (possibly with subtypes) that determines the rows in the table *(Victim)*. It may also include related entities, as long as the cardinality on the side of the related entity is one.

Figure 9 shows an Aggregated Content Presenter Design specifying the Ribbon content for the victims category (Figure 3). It only includes the border parts of the member presenter designs. The anchor symbol indicates which of the member presenter designs the aggregated one inherits the anchor from. The presenter design in Figure 9 contains a content integration view. Content integration views integrate related content from different presenter designs. Content integration views require that their member presenter designs use specific content views. The Ribbon Category View used in Figure 9 has two slots. The part to the left of the vertical line may only aggregate exactly one presenter design containing a Ribbon Category Overview View. The right hand slot may aggregate different view types, including the one used in Figure 7.

To specify the coupling of the 5 ribbon categories (of which one is specified in Figure 6 and Figure 9) in FLUIDE-A, we use a Task Supporter with 5 children, shown in Figure 10. We do not show the corresponding Task Supporter Design. To specify the coupling of the ribbon buttons (specified in the Task Supporter *Use ribbon buttons* which is not shown) and the ribbon categories (Figure 10), the Basic Work Supporter shown in Figure 11 is used. Each task in a Work Supporter may have a connected Task Supporter. In the Work Supporter shown in Figure 11, there are three tasks, of which two have Task Supporters. We do not show the corresponding design for the Work Supporter.

To specify the entire ribbon (buttons, ticker and the individual ribbon categories), the Aggregated Work Supporter shown in Figure 12 is used. An Aggregated Work Supporter must add exactly one task (possibly with a Task Supporter) on the level above the member supporters. The supporter in Figure 12 couples the ribbon contents (Figure 11) with the Work Supporter for the ribbon ticker. We do not show the corresponding design.

The map part of MASTER in FLUIDE-A is specified using a number of Basic and Aggregated Content Presenters, a Task Supporter, and a Basic Work Supporter.

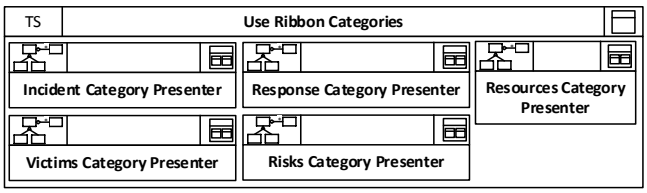

Figure 10. FLUIDE-A specification of a Task Supporter connecting the five ribbon categories

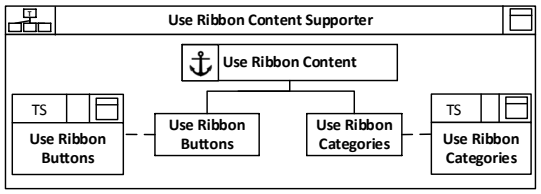

Figure 11. FLUIDE-A spec. of a Basic Work Supp. coupling the ribbon categories with the top level set of ribbon buttons

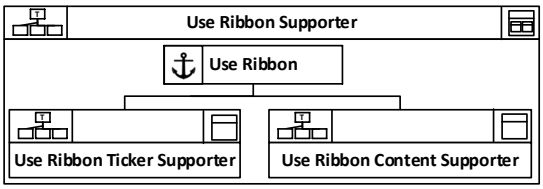

Figure 12. FLUIDE-A specification of an Aggregated Work Supporter for the Use Ribbon task

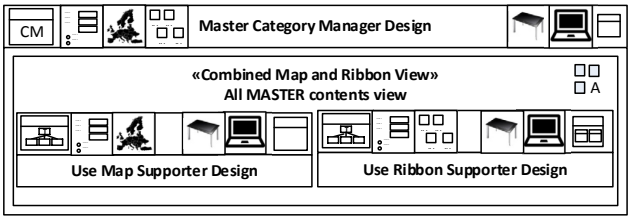

Figure 13. FLUIDE-D spec. of the MASTER UI

To specify the coupling of the ribbon and the map, a Category Manager is used. A Category Manager may aggregate both Content Presenters and Work Supporters. The Master category manager (not shown) aggregates a Basic and an Aggregated Work Supporter. The corresponding design is shown in Figure 13. It uses a domain specific content integration view which puts together one design containing a Map View with another design containing a Ribbon View, together making up a complete UI with the map and the ribbon working together. All designs having design children exploit the sum of styles and modality/platforms of their children (and their children recursively).

The Resource Manager Application

The Resource Manager is a smartphone application supporting personnel in the field by managing locations as well as receiving and responding to task allocations. Figure 14 shows two of the UIs it consists of. We do not show any of the Content Presenters or Task Supporters specifying the UIs of the Resource Manager. Instead we show the Work Supporter in Figure 15. It uses a more complex task model than the Work Supporter shown for the MASTER application (Figure 11). There are four levels in the task model in the Work Supporter, and four of the nine tasks have connected Task Supporters. The task model contains some operators (using the same symbols and precedence rules as CTT [19]): ">>" indicates sequence in task performance, while "[]" indicates a choice between tasks. No operator indicates that the tasks may be performed in an arbitrary sequence, including in parallel. The corresponding design is shown in Figure 16.

This Work Supporter Design utilizes a decorational view specifying that the child designs are a set of loosely connected windows (or full screen dialogs on a mobile device). The close icon indicates that windows are used, while the stacking look is used to indicate that the view contains a number of windows. The child Task Supporter Designs correspond to the children of the Work Supporter shown in Figure 15. Note also that the design uses touch-based mobile device modality/platform.

Figure 14. Two example Resource Manager UIs

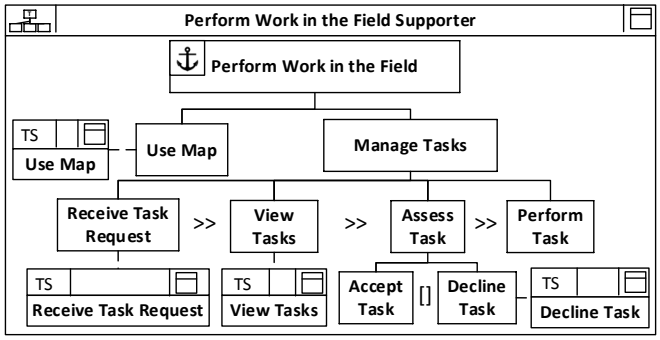

Figure 15. FLUIDE-A specification of the Basic Work Supporter *Perform Work in the Field Supporter*

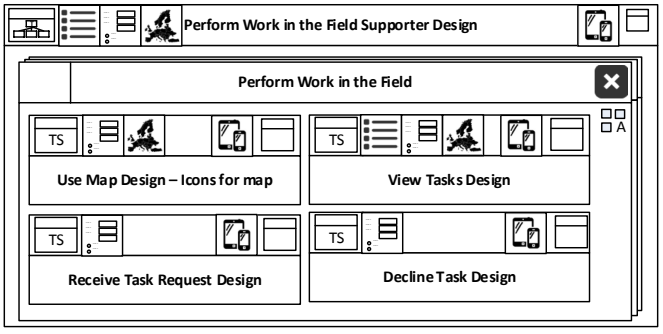

Figure 16. FLUIDE-D specification of a Basic Work Supporter Design for the Basic Work Supporter in Figure 15

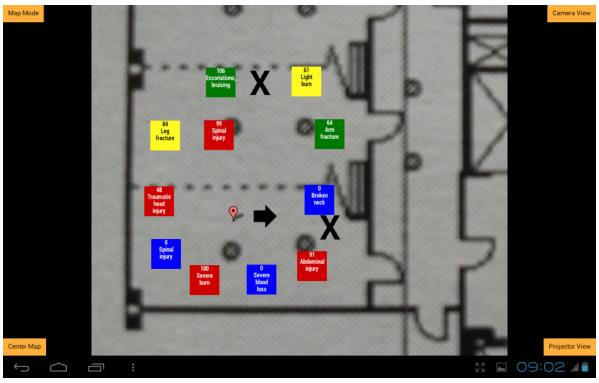

Figure 17. Map based UI giving an overview of triaged victims

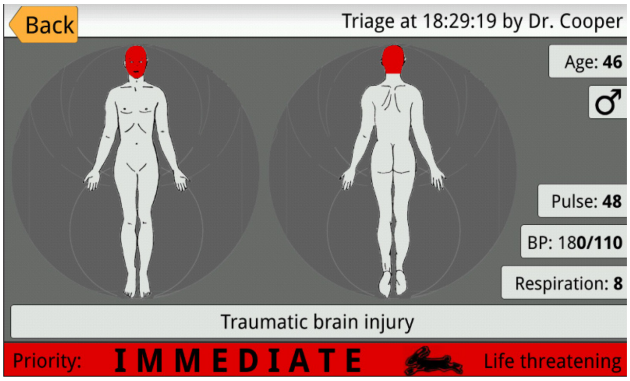

Figure 18. UI for managing details about a triaged victim

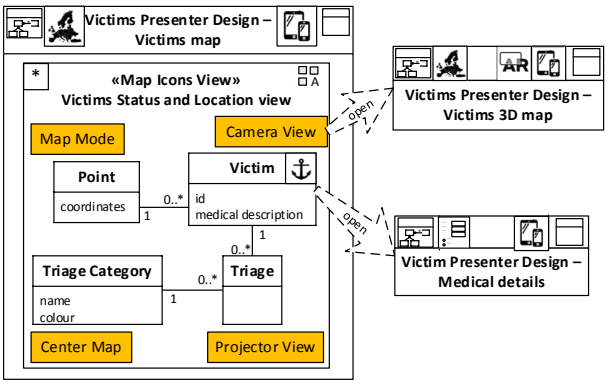

Figure 19. FLUIDE-D specification of the UI in Figure 17

The eTriage Application

The eTriage application supports personnel in the field triaging victims. The UI giving an overview of victim already triaged is shown in Figure 17. The UI employed when triaging a single victim or looking at the details for a victim already triaged is shown in Figure 18. The content managed by these UIs are covered by the FLUIDE-A Basic Content Presenter *Victims Presenter* shown in Figure 5.

The FLUIDE-D specification of Figure 17 is shown in Figure 19. It uses a map-based UI style, shown by the icon on the left side in the header. It contains one Map Icons View, which is a domain specific content view type. The model pattern must contain one entity (possibly with subtypes) that has a relation to a location entity (providing a point). It may also include related entities, as long as the cardinality on the side of the related entity which is presented is one. It presents a number of instances of the presented entities as icons on a map. If the presenter design using this type of view is member of a Map View (a domain specific content integration view), either directly or one or more times among its parents, the icons are shown on the map provided by the Map View highest up in the hierarchy. If a Map Icons View does not have a parent providing a Map View, it will provide its own map. Three of the buttons inside the views are property values of the view, the fourth (i.e., the button labeled "Camera View") is a button providing navigation. The presenter design shown in Figure 19 also specifies the intended dialog navigation to take place when an icon in the map is tapped. It is shown as a dashed-lined arrow with a growing size. The type of dialog navigation (in this case *open*) is shown as text on the arrow. The small end indicates which element of the UI that triggers the dialog navigation. The point of the arrow identifies the target. The target of one of the dialog navigation specifications is a representation of the presenter design shown in Figure 20.

A FLUIDE-D specification of Figure 18 is shown in Figure 20. It puts together five content views using a number of layout manager views (the designs used in MASTER leave most of the layout to the content and content integration views).

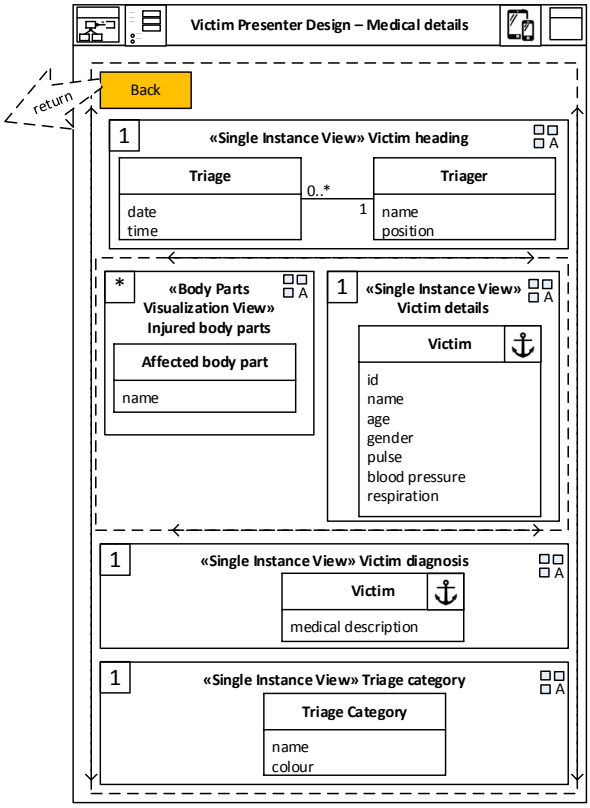

Figure 20. FLUIDE-D specification of UI in Figure 18

Layout manager views are not given names, and are shown using dashed lines (to indicate that they are usually not visible). The arrows on the dashed line specify whether the children are organized horizontally or vertically. Four of the content views used are Single Instance Views, a generic content view type for presenting one instance at a time of a single entity. The presenter design also uses the domain specific Body Parts Visualization View, which presents multiple instances together graphically. The presenter design also contains a button for navigating back to the dialog from which the *Medical details* was opened using the "return" type of dialog navigation.

DISCUSSION

As a result of the case study we have obtained three specifications – one for each application. In the following, we discuss the suitability of the FLUIDE languages by a careful inspection of these specifications as available in [17]. The discussion is structured according to six research questions.

When conducting the assessment, we have put emphasis on dealing with the challenges related to the fact that the assessment was performed by the researchers that develop the FLUIDE Framework. Firstly, these challenges were addressed by formulating research questions that were possible to address solely by assessing the specifications and the corresponding UIs. Secondly, the discussions addressing the research questions are to a large extent based on quantitative data obtained by careful analyses of the specifications and the UIs they specify. We claim that both these measures have contributed to an objective assessment with comparable conclusions to a possible outcome of an assessment performed by other researchers.

To what extent were we able to successfully express the three applications?

We were able to fully describe all three applications. In doing this, we met no major obstacles. We claim that the specifications of the three applications contain sufficient information for the target UIs to be schematically deducible from them. To support this claim, we will use two examples to show how all the different parts of a target UI are reflected in the corresponding specification: In Figure 21 we show where the different parts of the UI in Figure 3 are specified. In Figure 22 we show the correspondence between the elements in the UI in Figure 18 and elements in its FLUIDE-D specification (Figure 20).

To what extent are the specifications comprehensible to third parties?

The fact that the model fragment are expressed using known modelling languages gives systems developers (and end users) knowing UML class models and task models a head start. Also, as can be seen in the presentation of the case, the powerful constructs in the languages – particularly the view constructs – make the specifications fairly simple.

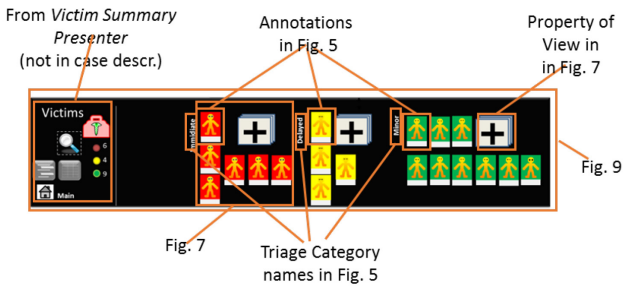

Figure 21. Where the parts of the UI in Figure 3 are specified

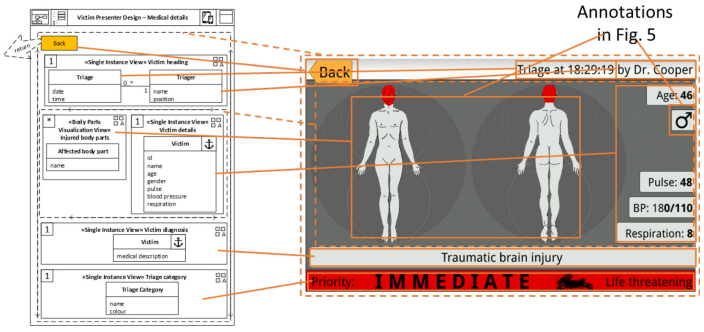

Figure 22. Connections between the UI in Figure 18 and its specification

The naming of the different content and content integration views are made to highlight which kind of UI pattern they support and how they are used. To add to this, the decorations on the FLUIDE-D specifications give comprehensive and understandable information about style, modality and platform in a compact manner. These observations are supported by an experiment using another case [18].

Are there common patterns/differences between the three specifications?

The three applications all support emergency responders, and they manage overlapping information. They have many commonalities, but they also vary regarding number of UIs, their complexity, style, platform, modality, and whether they are data or task oriented. The number of UIs is reflected in which interactor construct that are used. MASTER and Resource Manager use all constructs, while eTriage having few UIs only use the Content Presenter and Category Manager (Design) constructs. As Resource Manager and eTriage only manage one category of information each, while MASTER manages 5, there is only one Content Presenter instance in the FLUIDE-A specifications of Resource Manager and eTriage respectively, while there are 20 such instances in the FLUIDE-A specification of MASTER.

MASTER is data oriented and puts few restrictions on the sequence in which the different parts are to be used. Resource Manager is task oriented with a natural sequences in which the different UIs are used to solve specific tasks. This difference is reflected in the task models in the Work

Supporters in their specifications. The task models in the Work Supporters in MASTER (like the one in Figure 11) are quite simple, while the task model in the Work Supporter in Resource Manager (Figure 15) is more complex, containing operators to specify the expected sequence the UIs are to be used.

Which constructs functioned well; which constructs functioned less so?

All the interactor constructs in FLUIDE-A and all interactor design constructs in FLUIDE-D are used in the case. In the case description, examples are given for all the interactor (design) constructs, either the FLUIDE-A, the FLUIDE-D, or both variants. Based on this, we may conclude that the interactor (design) constructs match the needs of the case. All sub constructs (construct that may only be used as part of an interactor instance) except one match the needs of the case. The annotation sub construct is not used in any of the Work Supporters. This may be an indication that annotations are not very useful for this construct.

All the four view types in FLUIDE-D are used in the case, and the distribution between them reflects characteristics of the case. Content views may be further divided into domain specific and generic ones. The domain specific ones are more used than the generic ones, although one of the generic ones (Single Instance View) is among the individual view constructs used most often. These findings indicate that having a library combining domain specific and generic view types is useful in an emergency response case.

Are both languages needed or should they be integrated into one?

The main rationale for having the traditional split AUI and CUI [3] in the FLUIDE languages is to support development across platforms, styles, modalities and even applications, by having the common parts of the specifications expressed in FLUIDE-A and the more specific parts expressed in FLUIDE-D, among other to avoid redundant specifications. This split works well for the Content Presenters (which represents 57% of the interactor instances in the specification). All these instances are used as basis for at least two designs, and there are on average 3.1 designs for each instance of Basic Content Presenter. The most versatile (among them the *Victims Presenter* in Figure 5) are used as a basis for as much as six designs. Approximately a third of the Basic Content Presenters are used as the basis for presenter designs in two different applications, showing that our aim of using FLUIDE-A specifications across applications is achievable. Furthermore, there is one Basic Content Presenter (the *Victims Presenter* in Figure 5) that is used as basis for presenter designs exploiting all four styles used in the case (ribbons, tabular, maps and forms).

All the instances of the other interactor constructs are used by exactly one interactor design instance. The main explanation for this finding is that the tasks and task models tend to be more specialized than the concept models, indicating that the Task Supporters and Work Supporters in the FLUIDE-A specifications, although intended to be abstract, will naturally be specific for one (part of) a UI on a given platform. This is in line with the findings of e.g., Clerckx et al [4]. This shows that the split between FLUIDE-A and FLUIDE-D is important and successful for Content Presenters, but not important for the other constructs. As the Content Presenters represent more than half the interactor instances in the case, keeping the split for this construct seems advisable. An option would be to merge the languages for the other constructs, but this would violate the symmetry between the languages. Thus, our conclusion is that although the case has shown that the split is not important for all the constructs, we find the benefits from keeping the split higher than the disadvantages of having the split only for some constructs.

To what extent do the specifications scale?

The main means for specifying UIs of different size and complexity in FLUIDE is the aggregation mechanisms in the languages. They enable splitting the specification of large and/or complex UIs into smaller, manageable pieces through reusing lower level interactor (design) instances in higher level ones. Such mechanisms are available both in FLUIDE-A and FLUIDE-D. Through these mechanisms, it is possible to keep each interactor instance on a reasonable size. The specifications show that all the main aggregation mechanisms are used, and that each time an aggregation mechanism is used, a limited number of occurrences are aggregated (but usually more than one). This makes the specifications simple and manageable, and is a natural consequence of having four levels of interactor constructs, of which two may be used recursively. The most complex interactor (design) instances are included in the case description (Figure 5 for FLUIDE-A and Figure 20 for FLUIDE-D). Both these are of reasonable size.

RELATED WORK

There are quite a few languages and approaches supporting model-based UI development [11]. As shown in the introduction, neither some of the most influential of these [8, 13, 20] nor OMG's IFML standard meet all the requirement we have identified. In particular, none of the assessed approaches meet the requirement of having compound building blocks. UsiXML, MARIA, CAP3 [23] and RBUIS [2] build on or relates to the CAMELEON Framework [3], which depicts a close connection between AUIs and concept and task models. Despite this, neither of these embed such models, including their structure, as part of specifications. In UsiXML, the language supports specification of such models, but the connection to the AUI is through graph transformations. In MARIA, elements from concept models are referred in the AUI, as are operators from task models. CAP3 have explicit relations to task models in the AUIs, but use this mainly to specify behaviour, not for aggregation as we do. RBUIS keeps the connection to task models, even at run time where these

models are used as the basis for simplification of layout and features to roles, operationalized through adaptation of the CUIs. The AUIs are also used in this process, which enables fine-grained adaptations that are difficult to include in specifications. Our adaptations to roles are more coarse-grained, and based on the task models at design time. Adaptation at run time is partly focused on keeping the context of use in UIs on different devices, and partly on adaptations to changes in the external context. In realizing the latter, the approach used in RBUIS will be considered. We use the concept *interactor* in a similar way as it is used in the CAMELEON Framework, MARIA, CAP3 and Trætteberg's work [22].

The concrete syntax we use in FLUIDE-D is inspired by the way Canonical Abstract Prototypes [5] embed abstract interactors in an abstract layout, but it also differs significantly through our use of views and model element instead of containers and simple UI elements. Our view constructs rely on combining and coupling UI patterns and model patterns. Using UI patterns in UI development approaches is not uncommon. Ahmed and Ashraf [1] use patterns extensively, but focus on task and UI patterns. Lin and Landay [9] also use UI patterns in their cross-device development tool, but they rely on correspondence between CUI elements on different platforms rather than abstractions. In MyUI, Peissner et al [21] make extensive use of patterns combined with state charts for their AUI. In addition to UI patterns, they use patterns for categorizing devices, user groups, UI elements, as well as adaptation to these. They do not apply model patterns. Vanderdonckt and Simarro [25] support patterns both for domain and UI models, but do not combine them. Using model patterns as part of model-based UI development is not very common. Trætteberg [22] uses such patterns as part of his languages, but does not apply it to a view mechanism in the CUIs. The transformation mechanism presented in [14] also uses model patterns.

CONCLUSIONS AND FUTURE RESEARCH

We have presented the FLUIDE Framework supporting development of flexible UIs for emergency response applications through a component-based approach. The framework contains the FLUIDE Specification Languages with a unique combination of feature. The languages provide compound building blocks, which for AUIs enables common specifications across platforms and modalities with large differences, and for CUIs enables compact specifications of advanced UIs, including UIs where the layout is depending on instances at run time. The compound building blocks are made versatile by supporting model patterns, and provides a library of UI patterns that are particularly useful in the emergency response domain. The specifications embed domain model (concept and task models), including their structure, enabling reflection to support composition and adaptation, as well as traceability to support roundtrip engineering.

We have assessed the FLUIDE specification languages by specifying three existing emergency response applications. The experience from using the FLUIDE languages for specifying the case indicates that they are well suited for specifying UIs in applications supporting emergency responders working at the incident site. More precisely, we were able to fully describe all the three applications without meeting any major obstacles. Moreover, we have argued that the specifications contain sufficient information for the target UIs to be schematically deducible. We have tried to highlight that the specifications are comprehensible to third parties because they use known modelling style, have powerful constructs making the specifications fairly simple, yet carrying comprehensive information about the UI being specified.

The commonalities and the variations between the three applications are well reflected in the three specifications, both with respect to which constructs that are used, and level of details in the specifications. The specifications contain occurrences of all the interactor and interactor design constructs. All these, as well as most of their sub constructs, including the view types, were used as expected and intended. The case indicates that annotations on Work Supporters are probably not needed. We experienced that having a library combining domain specific and generic view types is useful when specifying emergency response UIs.

The specification includes a set of very versatile Content Presenters, working across both styles, modalities and applications. Even though the other construct proved less versatile, this shows that the split between FLUIDE-A and FLUIDE-D works, as the Content Presenters represent more than half the interactor instances in the case. The specifications scale very well because the reuse mechanisms are extensively used in the case, both on construct and instance level, contributing to keeping the specifications simple. The case also shows that the complexity of the target UIs is well reflected in the complexity of the FLUIDE specifications.

Among our planned future research is to complement the framework, including tool support and adaptation mechanisms.

ACKNOWLEDGMENTS

The work on which this paper is based is supported by the EMERGENCY project (187799/S10), funded by the Norwegian Research Council and the following project partners: Locus AS, The Directorate for Civil Protection and Emergency Planning, Geodata AS, Norwegian Red Cross, and Oslo Police District.

REFERENCES

1. S. Ahmed and G. Ashraf. 2007. Model-based user interface engineering with design patterns. *Elsevier Journal of Systems and Software 80(8),* 1408-1422.

2. P. A. Akiki, M. Keynes and A. K. Bandara. 2013. RBUIS: Simplifying Enterprise Application User Interfaces through Engineering Role-Based Adaptive Behavior. *Proceedings of EICS'13. ACM*
3. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. 2003. A Unifying Reference Framework for Multi-Target User Interfaces. *Oxford Journals Interacting with Computers 15 (3),* 289–308.
4. T. Clerckx, K. Luyten,,and K. Coninx. 2004. Generating context-sensitive multiple device interfaces from design. *Proc. of CADUI'04. Springer.*
5. L. L. Constantine. 2003. Canonical Abstract Prototypes for abstract visual and interaction. *Proc. of DSV-IS'03. Springer.*
6. E. Gamma et al. 1995. Design Patterns – Elements of Reusable Object-Oriented Software. *Addison-Wesley.*
7. J. Gonzalez-Calleros, J. Vanderdonckt, A. Lüdtke and J-P. Osterloh 2010. Towards Model-Based AHMI Development. *Proc. of HCI-Aero'10. ACM.*
8. Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, V. López-Jaquero. 2004. USIXML: a language supporting multi-path development of user interfaces. *Proc. of EHCI-DSVIS'04. Springer.*
9. J. Lin, J.A. Landay. 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. *Proc. of CHI'08. ACM.*
10. V. López-Jaquero, J. Vanderdonckt, F. Montero and P. González. 2007. Towards an Extended Model of User Interface Adaptation: The ISATINE Framework. *Proc. of EIS'07, LNCS 4940, Springer.*
11. G. Meixner, F. Paternò and J. Vanderdonckt 2011. Past, Present, and Future of Model-Based User Interface Development. *De Gruyter i-com 10(3)*, 2-11.
12. V. G. Motti and J. Vanderdonckt 2013. A Unified Model for Context-aware Adaptation of User Interfaces. *Revista Română de Interacţiune Om-Calculator 6(3),* 211-248.
13. D. Navarre, P. Palanque, J-F. Ladry, and E. Barboni. 2009. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact. 16(4).*
14. E.G. Nilsson, J. Floch, S. Hallsteinsen, and E. Stav. 2006. Model-based User Interface Adaptation. *Elsevier Computers & Graphics, 30(5),* 692-701.
15. E.G. Nilsson and K. Stølen. 2010. Ad Hoc Networks and Mobile Devices in Emergency Response – a Perfect Match? *Proc. Second International Conference on Ad Hoc Networks*. *Springer.*
16. E.G. Nilsson and K. Stølen. 2011. Generic functionality in user interfaces for emergency response. *Proc. OZCHI'11. ACM.*
17. E.G. Nilsson and K. Stølen. 2016. Assessment of the FLUIDE Specification Languages Using an Emergency Response Case. *SINTEF Report A26920 (ISBN 9788214059014).*
18. E.G. Nilsson and K. Stølen. 2016. The FLUIDE Framework for Specifying Emergency Response User Interfaces Employed to a Search and Rescue Case. *Proceedings of ISCRAM 2016.*
19. F. Paternò. 1999. Model-based Design and Evaluation of Interactive Applications, *Springer.*
20. F. Paternò, C. Santoro, and L.D. Spano. 2009. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. on Computer-Human Interaction 16(4).*
21. M. Peissner, D. Häbe, D. Janssen and T. Sellner. 2012. MyUI: generating accessible user interfaces from multimodal design patterns. *Proc. of EICS'12. ACM.*
22. H. Trætteberg. 2002. Model-based User Interface Design. *PhD thesis, NTNU.*
23. J. Van den Bergh, K. Luyten and K. Coninx. 2011. CAP3: context-sensitive abstract user interface specification. *Proc. of EICS'11. ACM.*
24. J. Vanderdonckt 2008. Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures. *Proc. of ROCHI'08. Matrix Rom.*
25. J. Vanderdonckt and F. M. Simarro 2013. Generative Pattern-Based Design of User Interfaces. *Proc. of PEICS'10. ACM.*