Rule-Enhanced Task Models for Increased Expressiveness and Compactness

Werner Gaulke
University of Duisburg-Essen
Duisburg, Germany
werner.gaulke@uni-due.de

Jürgen Ziegler
University of Duisburg-Essen
Duisburg, Germany
juergen.ziegler@uni-due.de

ABSTRACT
User centered design and development of interactive systems utilizes theoretically well-grounded, yet practical ways to capture user's goals and intentions. Task models are an established approach to break down a central objective into a set of hierarchical organized tasks. While task models achieve to provide a good overview of the overall system, they often lack detail necessary to (semi-) automatically generate user interfaces. Based on requirements derived from a comprehensive overview of existing task model extensions, improvements and development methods, an approach that integrates logical rules with task models is introduced. By means of practical examples it is shown, that the integration of rules enables advanced execution flows as well as leaner task models thus improving their practical value.

Author Keywords
Model-Driven Development; User-Interfaces; Task-Modeling; Rules

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous;

INTRODUCTION
A structured representation of user goals and tasks is a key activity in the process of user-centered engineering of interactive systems. Task models can serve both as tools for analyzing the requirements for an interactive application and as input to (semi-) automated user interface (UI) generation techniques. For the latter, formal models are needed that contain detailed and precise definitions of all aspects relevant to the structural and temporal properties of an UI. The well-known CAMELEON reference framework [5], for example, relies on task models as the first step to capture requirements that are subsequently transformed into abstract as well as concrete interface representations.

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

http://dx.doi.org/10.1145/2933242.2933243

A range of modeling approaches can be applied to represent user tasks, offering different perspectives on a task and/or different levels of detail and granularity [15]. A high-level distinction can be made between process models which represent the control flow between tasks of approximately similar level of abstraction, and hierarchical task models which focus on the breakdown of tasks into a hierarchy of subtasks. Hierarchical models have gained considerable popularity in the HCI field due to their ability to incrementally transform high-level cognitive user goals into low-level actions at the interface (for an overview and comparison, see e.g. [18]). They are, therefore, quite suitable for user-centered design processes which begin by analyzing user goals and requirements independently from their technical implementation. By including temporal operators, hierarchical task models can represent task flow at different levels, thus providing specifications that are needed to transform a model into an abstract or concrete UI. Concur Task Trees (CTT), first defined in [25], are an example of this approach and can be considered a de facto standard in current task modeling.

Process models such as Business Process Model and Notation (BPMN) [21], while often not considered as proper task models, have their strengths in being able to show complete task flows involving parallelism, and, in particular, conditional branching which is typically less easy to represent in hierarchical models. Both approaches, hierarchical as well as process models, rely on graphical representations which are illustrative and relatively easy to understand, providing a good overview of the task structure shown. A major shortcoming of both, however, lies in the problem that tasks involving alternative execution flows due to the complex decision processes involved may considerably inflate the model with additional subtasks or flows, or cannot be represented at all. As an example, consider a typical UI where certain inputs are activated or deactivated, either based on the user's previous interaction or the state of some data item in the domain model. Actually, modern UIs frequently make use of the context-dependent activation or deactivation of controls for increasing usability by guiding users through their interaction task. Such processes, however, are typically cumbersome to model, both in flat process models as well as in hierarchical models. In the area of business process modeling, methods based on Business Rules have been proposed to capture the manifold conditions and case distinctions that are commonly found in such scenarios. A prominent technique for this purpose is the Object Constraint Language

(OCL), used in the UML context [22]. A recent example of the approach can be found in the Decision Model and Notation (DMN) [23], an extension of BPMN which captures conditional choices using formal methods like decision tables. With this technique the authors aim at avoiding overly complex process flows and at facilitating the reusability of process definitions. While hierarchical task models such as CTT have been extended to provide means for specifying pre- and postconditions of tasks, the approach is rather limited and is, for example, not suited to enable general rule-based modifications of task control flow.

In general, current task modeling techniques do not sufficiently leverage the potential of rule-based techniques for increasing *expressiveness*, for *simplifying* the models and for facilitating the *reusability* of definitions for recurring situations across different task models. Furthermore, a systematic and powerful linking with domain models is needed to allow rules to check conditions based on data and to manipulate the state of domain object. This aspect is also not sufficiently covered by existing methods.

In this paper, we propose an approach that extends hierarchical task modelling based on the CTT notation by a generic rule mechanism that can check a wide range of interaction and data-related conditions as well as enable and disable tasks in a flexible manner. Furthermore, an event mechanism is introduced by which rules can react on events and raise them, thus enabling communication across distant parts of a task tree. The task model is closely linked with a domain model which is represented in a semantic format as an ontology to support cross-application use and the definition of generally valid rules. We introduce the concept of rule-based tasks that model entire subtask structures with complex execution flows in a single task entity, thus simplifying the overall task model and supporting reusable task patterns.

In the following, we first provide a comprehensive overview of existing extensions of CTT-based task models and review the respective development environment, concluding with requirements for modeling tasks in a way that enables the generation of complex dialogs. Next, we introduce our rule-enhanced task modeling technique which integrates CTT task trees with the rule language Reaction RuleML. We also provide an overview of our domain modeling approach. Finally, applications are demonstrated by means of practical examples.

Related Work

Extensive research has been conducted in the area of task modeling for extending the basic techniques such as CTT for better manageability and increased uptake. This research can be divided into three main directions: First, concepts have been proposed that aim to improve expressiveness of the language, reducing limitations and enabling a broader range use cases. Second, reusability of task model concepts has been addressed in order to reduce modeling effort and to achieve more consistent task models. Third, user friendly development environments and visualization tools were designed to foster the practical adoption of the method. In the following, we give a brief overview of these developments and highlight limitations that still exist.

Task models – extending expressiveness

Although the original CTT model covers the basic temporal structures of task execution well, certain situations such as error handling cannot be solved with the original language features. Several extensions have been proposed to solve limitations by adding *new language concepts* or *task types*.

The authors of [31] propose new operators to enable advanced error handling in task models. If, for example, a task cannot be completed due to an error, the extension allows to define error handlers that in turn define alternative tasks that will be invoked in such cases. The definition of errors and error handlers allows to deal with models where the successful completion of a task cannot be guaranteed. In addition to error handling capabilities, a basic concept to enable communication between instantiated task models is proposed. Specific actions in a task model can be connected to another task model thus influencing its tasks.

An advanced approach to support task models for collaborative scenarios is elaborated in [35]. The Collaborative Task Modeling Language (CTML) introduced there adds preconditions to control execution by enabling or disabling tasks based on defined requirements. In addition, events are used to define dependencies between tasks that have no direct connection, as well as to allow communication between multiple model instances. In [8] the authors use messages to enable communication between distant tasks as well as task conditions to restrict access based on user roles and rights. A detailed discussion of the influence of preconditions in task models is presented in [16]. It is shown that preconditions can both be used in task models to refine interaction flows and to clarify ambiguous parts. On the other hand, preconditions are limited to the activation or deactivation of the tasks they are attached to, they cannot perform changes in other parts of the task tree and are usually limited to single predicates. If task flow can be dependent on declarative conditions as well as on trigger events sent by other tasks, a simple precondition mechanism is not powerful enough to fully specify task activation.

The addition of input and output ports to task elements is elaborated in [13], aiming to support semi-automatic generation of user interfaces. Ports are used to explicitly define the data elements necessary to start task execution as well as to create output that can be consumed by following elements. Explicit integration of objects and domain knowledge is also addressed by the authors of the HAMSTERS approach [18]. Domain knowledge is, however, usually considered as application-specific. Shareable domain models, e. g. expressed as ontologies in Semantic Web formats, are not addressed in existing task modeling techniques. Yet this aspect is gaining more relevance, for example, in e-commerce applications where product ontologies may be exchanged between manufacturers, online-shops, or search engines.

All introduced concepts share the common goal to enable the construction of more complete and sound task models. On the downside, all additions lead to more complex models that require more effort to create and maintain, hence additional techniques to counteract this disadvantage are necessary.

Task models – patterns and reusability

In contrast to features that extend expressiveness by primarily adding new language concepts, the elements discussed in the following aim at improving the manageability of task models. This can be achieved by several measures. First, *redundancies* should be avoided to prevent the need to recreate identical tasks. Second, a better overview can be achieved if the *modularization* of task models is supported, enabling to divide large models into multiple, clearly representable parts. Third, concepts for *abstracting* fine grained task structures should be available to reduce the overall number of elements and to achieve more compact models.

To avoid multiple definitions of identical tasks the concept of references is introduced in [31]. Instead of defining identical elements multiple times a reference can be used to point to an already existing task thus reducing the overall modeling effort. Changes to the original task are applied automatically to all references. For example, an *Input Name* task could be referenced whenever the data is necessary. In addition, the authors propose the decomposition of monolithic models into smaller submodels that can be edited independently. Providing reusability for single elements as well as entire model fragments is deliberated in [19]. Whole parts of the model are extracted as fragments. Variables are added as placeholders to enable parameterization of the fragments in other contexts. By comparing regular models with models that utilize references and fragments, the authors demonstrate that the latter models can reduce the number of operators needed by about 20%. The extraction and use of complete task patterns can ease model development significantly [7]. Based on the Pattern Language Markup Language (PLML) [30] demonstrates a concept to collect and apply patterns for task models. Implementation and practical usage of patterns for the USer Interface eXtended Markup Language (UsiXML) is shown in [34]. Eventually, a combination of all introduced approaches is demonstrated by [6]. Task references (labelled sub-models), task fragments (sub-routines) as well as a collection of task patterns led to a reduction of about 41-46% in terms of necessary elements and operators, resulting in smaller and easier to handle models.

While modularization and reusability of tasks and the use of common task patterns leads to smaller task models there is still a challenge as to how the patterns are specified effectively and how they can capture complex interactive behavior in a compact form.

Task models – development

To foster the adoption of modeling methodologies usable tools are necessary which are tailored towards particular needs of the prospected user group.

The CTT Environment (CTTE) [20] provides a workbench to create CTT models. The hierarchic structure of task models is represented as a tree. Nodes are used to depict composite tasks whereas leaves define concrete, potentially executeable tasks that should not be further divided. Associations between task elements are typed with temporal operators to specify possible execution orders, flows as well as task dependencies. While providing a usable platform to create simple as well as large task models, especially latter ones are hard to create and maintain due to the necessary screen size needed to visualize these models. Ways to improve usability and facility of CTTE are introduced in [28]. Extensions range from simple changes, like abbreviated labels or visually merged associations, to complex features like a fish eye view that scales elements depending on the current focus. Further extensions influenced by modern web development techniques are shown in [1]. Visualization and control elements adapt in accordance to available screen size.

In addition to the introduced extensions of [28] described above, a text based representation is proposed that can be used as an alternative to the tree visualization. Tasks are displayed as text elements in a list where indenting is used to reflect the hierarchy and icons to depict temporal operators. In a similar fashion, the tool TaskArchitect [32] complements the common tree visualization with an additional tree-table. Every task is represented as a row, whereas columns are used to display and configure properties like difficulty or role.

An editor (AMBOSS) to create CTT models, extended specifically for security critical systems, is described in [8]. In addition to basic CTT elements, modeling of domain objects, roles, preconditions and messages are supported. Roles can be used in preconditions, tasks can be connected with domain concepts and messages are utilized to enable communication between tasks without a direct connection. The tree visualization contains icons to depict concepts whereas additional information is available in a detailed view. Although icons provide sufficient information, they can be challenging to learn and memorize for new users.

Alternatives to tree visualizations of CTT are proposed in [4] and [14]. Former approach uses activity diagrams from the Unified Modeling Language (UML) specification, whereas in the latter case tasks are expressed using the Business Process Modeling Language (BPMN) notion. Both cases have in common, that an already existing and well defined process modeling language is used in conjunction with a mapping that defines the transition from process model to task model and vice versa. Due to the different paradigms, transitions are subject to loss of information as not every expression can be mapped to a distinct counterpart. To solve this particular problem, the approach described in [33] restructures the interface generation process to be solely based on BPMN thus avoiding a transition to a hierarchical task model completely. The usage of an existing process modeling language in all three variants is seen by the authors as an advantage regarding the availability of already existing tools and trained users.

However, in practice and due to the different focus of process languages, they can lead to even larger models or ambiguities not translatable to a hierarchical task model.

REQUIREMENTS FOR ADVANCED TASK MODELING

Concluding from the analysis of related work concerning the three aspects expressiveness, reusability and development the following requirements should be considered for state of the art task modeling.

Language constructs beyond basic structural and temporal concepts:

- *Events:* As described in related work, the integration of events enables communication between tasks that are not directly connected. In addition, external services are enabled that both consume or produce such events.
- *Conditions:* Conditions for different phases of task execution (e.g. pre-, post- and contextual) to enable flexible conditional task flows. When defining conditions, domain model elements to be checked must be accessible in a flexible manner (e. g. based on query techniques).
- *Domain model integration:* Methods to read and manipulate (application independent) domain model information should be provided.

Patterns and reusability*:*

- *References*: References provide a simple, yet practical way to avoid multiple definitions of identical tasks within a model.
- *Variables and patterns:* Utilizing variables, single tasks or whole fragments can be reused. This facilitates the collection of patterns to ease development of common scenarios whilst enabling to focus on crucial parts.

Model development:

- *Modularization*: The option to split a single model into multiple fragments improve maintainability and feasibility. It should also be possible to modularize and adapt models for different phases or user roles.
- *Flexible tree layout and alternative view*: Task trees should both adapt dynamically to available screen real estate as well as avoid visual clutter wherever possible. In addition, an alternative view, based on tables or lists, can further increase usability.

Current extensions either try to extent CTT in terms of expressiveness or try to simplify development by modularization and reuse. However, especially modeling challenges that require a high level of detail or depend on conditional workflows can lead to large, thus difficult to maintain CTT models. The subsequently suggested rule based task aims to both improve expressiveness as well as enable a leaner representations of expressed behaviors.

RULE-ENHANCED TASK MODELS: MOTIVATION AND REQUIREMENTS

One of the strengths of modern graphical UIs is that they can guide the user through interactions with complex dependencies on prior inputs or the current state of the application. This may be done, for example, by activating or deactivating interaction objects, by expanding additional parts of a screen form, or by changing the values that can be selected in a widget. Such fine-grained and complex dependencies are typically hard, if not impossible, to express with standard CTT-like techniques. Especially when complex case distinctions need to be considered, simple concepts like task preconditions are insufficient to model the resulting processes economically and precisely. Consider, for example, two dropdown lists in a form from which the user can select in order-independent fashion, but where the values selectable in one list depend on the selection in the other. The only solution in standard CTT would be to create one subtask for each combination permitted which would inflate the model considerably without providing additional insight in the overall task flow. The general point is here that many interaction flows depend on the underlying logic of the application. Aspects of what is usually called ‘business logic’ should therefore be expressible in the task model, if they influence the user’s interaction with the system.

For these reasons, we propose to extend tasks with a rule concept that is powerful enough to cover different types of business logic as well as event-based dependencies between tasks. In respect to the forward modeling approach intended by the CAMELEON framework, as demonstrated in [27], rules are not designed to anticipate information of the abstract or concrete interface, rather they provide means to leverage their derivation. In detail, rule-enhanced task descriptions can serve different purposes. First of all, rules can be used to *conditionally influence the execution* of single tasks as well as whole task flows. In addition, it should be possible that rules create ‘virtual’ subtasks on the fly without defining them in the hierarchical model first. Moreover, rules can be used to *improve soundness and utility* of task models by embedding decision models within the task model, covering all use case specific information in a single model. By using rules to cover the definition of fine grained workflows instead of modelling each step explicitly, the overall size of models can be reduced, thus *increasing transparency* and *facilitating the development of models*. For example, a single rule could be used to define order, datatypes and dependencies of user inputs and replace several single tasks that would otherwise be necessary. Finally, rule-based tasks might be *reused* in different applications or task models reducing redundancy and effort in building the models.

In order to realize described goals, three main requirements have to be met. First, a *well-defined domain model* is needed to allow the formulation of conditions that check application state and to manipulate state in a rule’s action part. Second, a *powerful rule language* is necessary, providing a suitable

level of expressiveness to enable both the definition of logical expressions as well as the utilization of domain and task-model knowledge. It should also be able to use events for triggering actions. Third, the semantics of *rule based tasks* have to be defined in order to specify its use in task models.

Running Example

The following example outlines a typical process of a customer in an e-commerce (web-)frontend. Although tasks like product browsing or search are rather simple, particularly the customization of products, ranging from single selections to multiple interconnected choices, can lead to negligible modeling challenges.

Goal of the task model is to encompass a shopping scenario for customizable fashion products. A domain model is used that contains conceptual knowledge about garments as well as instance data of sold products. First, the customer starts by picking a category (product class) from a set of available categories. Next, he has to choose the desired product from a list of available products. To ease search, filters are provided. After selecting a product, a detail view enables further product configuration. The properties color, size and logo can be configured. Configuration is restricted by the following constraints: A logo is only available if the selected size is 'L (5)' or larger. If the color 'black' is selected, no logo selection is possible.

Figure 1 depicts a basic task model for the given scenario. *Buy Customized Product* is modeled as the parent task, specialized in three subtasks. Initially, a product class like t-shirts or sweaters has to be selected (*Select Product Class*). Upon selection the transition *enabling with info exchange* is used to activate the *Choose From Product List* task whilst passing the selected class as parameter. This task is further differentiated into the subtasks *Filter Products* and *Select Product*. While former is used to provide filter mechanisms, latter one is used to select the chosen product. Finally, after a product is selected, the product instance is passed, enabling the task *Configure Product*. Although the model already provides a sufficient overview additional details are required if it shall be used for (semi-)automatic interface generation.

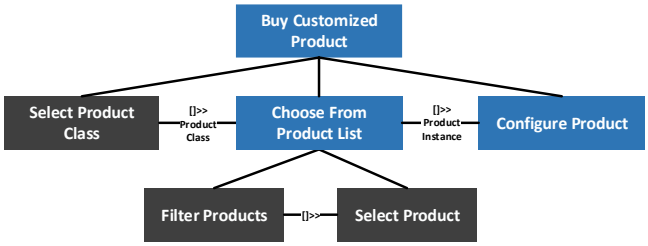

Figure 1. Comprehensive task model for the described scenario. First, a product class is selected to enable product selection. Second, a product is chosen to activate product configuration. Notation derived from CTT Environment [20]**.**

According to the described scenario, *Configure Product* is restricted by certain conditions. Using basic CTT, several subtasks could be added to reflect configuration of individual product properties. In this case *Select Color, Select Size and Select Logo* are added as individual tasks (figure 2). In addition, choice options could be added as additional subtasks, thus providing more detail. However, described restrictions like the option to enable or disable logo selection based on certain color or sizes would result in a not reasonable effort in comparison to the rather simple complexity. Moreover, advanced constraints could not be expressed with basic CTT concepts thus making an additional concept, like the proposed rule type, necessary.

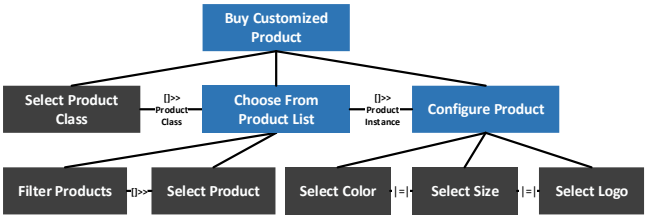

Figure 2. Task model extended with configuration options for color, size and logo. Addition of all possible combinations would lead to an unfeasible, hard to maintain model.

Domain model

Task models and domain models represent two complementary perspectives on an interactive application and must be connected if the models are to be transformed into an abstract or concrete UI, in particular if automated generation is intended [18]. Domain model data serve as input and output of tasks but also for testing conditions that activate tasks. For generating systems, domain models need to contain not only conceptual entities but also the instance data to be processed. There are several ways to realize domain models that can be differentiated regarding the underlying model paradigm and the level of expressiveness. For instance, relational models can be used to define entities and their relationships. Although already providing sufficient information for a wide range of use cases, advanced concepts like inheritance or custom datatypes with fixed value ranges are missing, thus making more expressive formats necessary.

Semantic formats like the resource description framework (RDF) [17] or the ontology web language (OWL) [9], rely on a graph-based data model that offer a high level of flexibility and expressiveness. They provide capabilities to create platform independent domain models which is increasingly a requirement if a model is to be shared among different stakeholders or applications, as is the case, for instance for product models in e-commerce. Semantic models provide constructs for, amongst other things, custom data types, structured relationships and inheritance. In addition, rules to define generalized domain level constraints are supported either by adding restrictions for relationships or datatypes or by using an extension like the semantic web rule language (SWRL) [11], or Reaction RuleML which we introduce in the next section. However, to avoid possible duplication of domain knowledge, rules embedded in task models should be clearly separated from rules in the domain model. Rules in the domain model should be universally valid, whereas

rules ins task models should only be used to add use case specific knowledge relevant for interactive behavior. For instance, if the domain model contains a user concept and every user has to supply a password with at least eight characters, this should be added as a constraint within the domain model. In contrast, a restriction to allow registration only for certain email addresses is most likely application specific and therefore a candidate for a task model rule.

To make use of domain model concepts in task model elements as well as rules, mechanisms to establish a connection are necessary. A connection can be achieved either by creating a static link or with a query mechanism that provides loose coupling and a higher flexibility. Advantages of latter approach are discussed in [12], where database queries are used to connect elements of discourse models with domain model elements. In regard to semantic domain models, specialized query languages like SPARQL [10] are available to create a connection.

To conclude, a well-defined domain model is necessary to enable rules that make use of defined concept by utilizing a suitable query language. Rules itself should be used to express application specific requirements that extend available domain model knowledge.

Rule language

A rule language used to define constrains for task models has to meet several requirements. First of all, a generic rule structure, that is capable to support the definition of different specific rule types, is necessary. In addition, an expressive rule language is required to build logical equations in accordance to the introduced generic structure.

The RuleML project aims to provide an overarching specification of web rules by standardizing commonly used rule concepts [2]. To enable cross platform use, rules can be serialized using the Rule Interchange Format (RIF) [29]. RuleML pursues an extensible approach and is divided into multiple modules. Within this context, the module Reaction RuleML [24] serves as a suitable archetype to derive the concept of rules for task models due to its particular focus on rule based event processing. As shown in figure 3, a rule is separated into two essential parts: Initially a *premise* is defined, stating facts that either evaluate to true or false. A *consequence* is provided to define changes or actions to be executed if given premise evaluates as true. In Reaction RuleML both basic parts are further differentiated to enable a wide range of use cases. On the one hand, premises are divided into events (*on*) and conditions (*if*). Events are used to specify triggers that invoke evaluation when they occur within the model whereas optional conditions can be used to provide further restrictions that have to be evaluated before subsequent processing. On the other hand, consequences are divided into conclusions that can derive knowledge (*then*) or invoke actions (*do*).

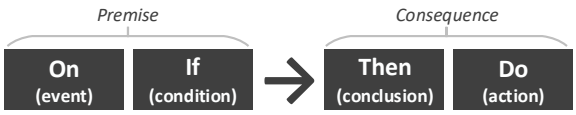

Figure 3. Basic building blocks of Reaction RuleML. Premises are build using events or conditions whereas consequences lead to conclusions and actions.

Building blocks of Reaction RuleML can be combined to create different rule types, each geared towards a specific use case. Table 1 lists valid variations and explains each rule type briefly. Beyond introduced elements, the specification provides additional constructs like scopes or logical alternatives (else) to enable definition of advanced rules. In the scope of this paper presented basic elements are sufficient for further discussion and demonstration.

Combination	Name / Purpose
[if] → [then]	Derivation rule: Tests existing knowledge to gather new insights.
[if] → [do]	Production rule: Tests existing knowledge and executes actions if conditions evaluated as true.
[on] → [do]	Trigger rule: Is invoked on occurrence of a certain element and does execute an action.
[on][if] → [do]	Event Condition Action (ECA) rule: Invoked on occurrence of a certain event with an additional condition to determine execution of an action.

Table 1. Typical rule types made from of Reaction RuleML building blocks.

To finally make use of described rule types, it is on the one hand necessary to provide ways to use domain and task model knowledge as well as state information in premises. On the other hand, means to express consequences are necessary (see figure 4). Such operations are enabled by *n-ary predicates*. For instance, E*quals(x,y)* is a commonly used predicate to compare two variables for equality, resulting either in true or false. Thus, to make use of available information, predicates specifically designed for task model and domain models need to be defined.

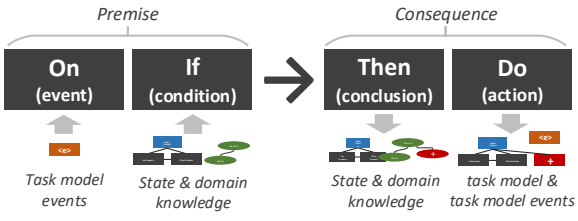

Figure 4. Generic structure of rules for task models. Premises are checked against domain model or the current state, whereas consequences are applied to the task model.

Basic predicates

Comparisons, set operations or logical expressions are essential building blocks to create rules. Therefore, predicates to enable these operations have to be provided. Typical concepts like *Equals(x,y), Greater(x,y) or Less(x,y)* are used to enable comparisons and support common datatypes. Testing, whether an element belongs to a set or not, is provided by set operations like *Contains (x,y)*. Finally, logical expressions to connect (*and, or)* or negate (*not)* atomic parts are essential to compose advanced rules. Given examples are non-exhaustive and used to give a general impression about basic predicates.

Event predicates

ECA or trigger rules use events as enabler to invoke their execution. Therefore, the following predicates are used to create listeners for automatically or manually raised events within the task model. Automatically raised events occur during execution of the task model. Every action or change, like the activation of a task caused by user interaction, is made available as an event and can be used as a trigger. On the contrary, manual events are raised explicitly by action predicates used in rule consequences. To conclude, the event predicates defined below provide flexible means to react on activities within the task model. Concepts like pre- and postconditions or error handling are covered by this approach and benefit from the higher expressiveness of rule terms.

Predicate	Description
OnTaskStart(tx) OnTaskFinish(tx) OnTaskInterrupt(tx) OnTaskResume(tx)	Listener for state changes of specified task tx. If tx is omitted, the trigger is bound to the task it was defined in.
BeforeTaskStart(tx) BeforeTaskFinish(tx)	Listener for state changes specifically to recreate pre- and postconditions.
OnInput(dx)	Listens for inputs for element dx from the domain model made within a task.
OnEvent(ex)	Generic event listener that registers for a specified event ex
OnError(ex)	Listener for errors raised during the execution of tasks.

Table 2. List of predicates to register for events occurring within task models. Events are either raised explicitly by action predicates or automatically by task model state changes.

Condition predicates

Conditions can make use of available information contained both in the task model as well as in domain model. In the former case, state information about the task model is of particular interest. As demonstrated in [3], task models can be transformed to state charts that enable to query state information. Therefore, predicates are necessary to get hold of such state information. In the latter case, a predicate to test or get hold of additional domain model concepts is needed.

Predicate	Description
TaskStarted(tx) TaskActive(tx) TaskFinished(tx) TaskCancelled(tx)	To determine the current task state. Tests, if task tx (identified by unique name or id) is currently in respective state (started, active, finished, cancelled).
QueryConcept(qx)	Executes a query against the domain model and makes the result available.

Table 4. Predicates to create conditions. Used to get information from the task model and domain model.

Conclusion predicates

Derivation of new knowledge is the purpose of conclusion predicates. In this particular case, new domain or task model knowledge could be deduced. Imagine a task model that contains many concurrent and optional tasks that reflect extensive activities found, for example, in the area of enterprise resource planning. State information about finished or unfinished tasks could be used to derive a user role which in turn could trigger certain rights or features. Although providing a powerful mechanism, conclusion predicates and knowledge derivation are out of scope for this paper.

Action predicates

Enabling reactive behavior is the purpose of action predicates. In detail, either existing information is altered or new information is added. For example, in the former case an existing task could be activated or deactivated whereas in the latter case a new task, initially not existent within the model, could be added. Tasks that are created by such an action are called *virtual tasks.* They are particularly useful if, for example, a lot of simple tasks for data entry or manipulation need to be created. By creating virtual tasks procedurally, fine grained workflows can be realized without the need to define all concrete tasks and possible branches manually thus reducing visual clutter.

In addition to manipulation and creation of task elements, action predicates are provided to raise events or errors. Both can be utilized by using the previously introduced *OnEvent* or *OnError* predicates. Possible event types should be collected centrally in order to ease the definition of rules using respective events as triggers.

Predicate	Function
EnableTask(tx) DisableTask(tx)	Enables/Disables an *existing task* tx and makes it available/unavailable for execution. Does not transition to that task, just enables it for subsequent processing. Most likely to be used for sub tasks within the current hierarchy.

UpdateTask(tx, px)	Updates property px of task tx. Used to update available task properties like min/max executions.
CreateInput(dx,tx) RemoveInput(dx,tx)	Creates a *new* or removes an existing *virtual task* with the purpose to enable input of new information. Parameter dx specifies a plain datatype or references a concept of the domain model. Parameter tx is used to define the task where the new virtual task is added. If omitted, the current task is used.
DisableInput(dx,tx)	A nondestructive alternative to remove input. The Input task is disabled but could still be visible in the interface in order to give a hint of possible options.
AddValue(id,v) UpdateValue(id,v) DeleteValue(id,v)	Used to add/update/delete a specified value v in the current execution state, identified by an id. Acts like a key-value store and is suitable for rather simple items like a role name.
AddInstance(tx,v) UpdateInstance(tx,v) DeleteInstance(tx,v)	Adds/update/deletes a value v of the given type tx from the referenced domain model. Used to update values that may be picked in a selection.
RaiseEvent(tx) RaiseError(tx)	Creates an event / error event with a type tx as a parameter. Types should be defined centrally to enable trigger rules for such events.

Table 3. Action predicates to update task state information and domain concepts, add/remove virtual tasks or raise events and errors that can be utilized as triggers by other rules.

To conclude, introduced predicates provide a collection of functionalities to be used to both define rule conditions and consequences. Practical examples were used to derive and refine proposed predicated. Proposed predicates are the result of made experiences of their application and discussion in real world scenarios. If, however, additional requirements arise due to the specifics of a given scenario, extensions can be added easily. Present predicates are designed to extend expressiveness and foster simplification of common use-cases whilst integrating seamlessly with existing task model concepts. Finally, the next section conceptualizes a frame to put rules and predicates to use.

Rule based task

To finally make use of the introduced rule concept it is necessary to extend the existing CTT meta model to provide means that enable to add rule definitions. Basic CTT differentiates two main concepts: on the one hand, abstract (also composite) tasks are composed out of an arbitrary number of subtasks whereas on the other hand concrete (also atomic) tasks mark single executeable units. Specified in a formal meta model (e.g. [26]), abstract and concrete tasks usually derive common properties from a task superclass. Furthermore, concrete tasks are specialized into user-, interaction-, and system tasks. Rule definitions should be possible for both types in order to indicate their area of influence on the overall model. For example, rules for abstract tasks could be used to process events raised by associated children, whereas rules for concrete tasks are more likely to define conditions or actions that are relevant during its execution.

In order to enable the definition for both abstract and concrete tasks, the task superclass is extended with two additional associations as shown in figure 5. Firstly, the class *DomainObjectReference* is used to reference domain model concepts that are going to be used in rule definitions. Secondly, rules are added within the *RuleDefinition* class. This separation of concerns was chosen to avoid a mixture of query and rule terms aiming at a better readability. However, an implementation may decide to merge both if the used query language can seamlessly be embedded into rule definitions without making them too complex.

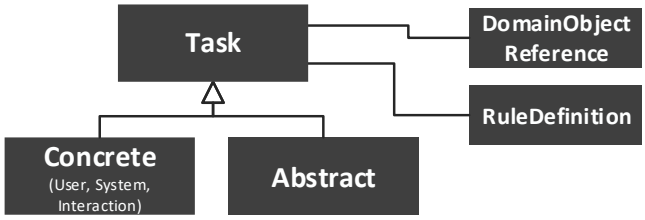

Figure 5. The rule based task element serves as a frame to integrate rules. It inherits all common task properties and contains defined rules.

The application of the extended meta model is done in two phases. Existing domain objects that are going to be used in rule definitions have to be collected initially. Technically this is done by specifying queries that select concepts from the domain model, making them available as variables. If, as previously advocated, semantic domain models are used, a semantic query language like SPARQL would be utilized. In case no domain object is used, this part may be omitted. Finally, rule terms are added that abide to the specified rule scheme and make use of the previously introduced predicates as well as allocated variables resulting from executed queries. To summarize, the following information has to be supplied:

1. **Domain Objects**: Reference / query to domain objects to be used in rule definitions. Query results are available as variables in rule definitions.
2. **Rules**: Definition of rules by using available variables as well predicates according to the previously define rule scheme.

Application of rule based task element

Figure 6 demonstrates the application of the rule based task for the initially introduced running example. Even though

original CTT was able to provide a basic solution, constraints between product properties could not be expressed properly. Proposed rules and predicates will be used to model demanded constraints.

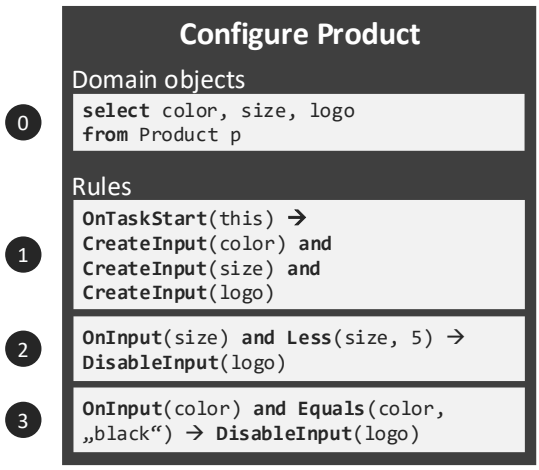

Figure 6. Rule element to configure products. ❶ domain concepts to be used in rule definitions are selected. ❶ enables input for all selected concepts. ❷ checks the entered sized and disables logo input if the size is less than 5. ❸ checks the selected color and disables logo input if the color was black.

Initially, before any rules can be defined, the domain properties *size, color* and *logo* are selected in order to make them available as variables (❶). For the matter of simplicity, selection of these properties is done using a rudimentary query-pseudocode. Selected variables are used subsequently within the definition of rule terms. As mentioned above, it is necessary that respective properties are specified in a well-defined data structure and results are either an object type or a primitive like a string. Only if this is the case, basic predicates can be used to define tests like comparisons.

The first rule (❶) is defined as a trigger rule ([on]→[do]) with the *OnTaskStart* predicate used to bind to the activation of the current rule by using the *this* keyword as a parameter. In the action part of this initial rule the *CreateInput* predicate is used to create virtual tasks to request input for color, size and logo. The next two rules are used to define constraints between user selections and available options. In the first instance (❷), a rule to disable input for logo is created if the selected size is less than medium (5). *OnInput* is used to trigger execution of this rule that occurs whenever data is entered. The above described condition, regarding the selected size, is expressed using the *Less* predicate. If the term evaluates true, input for *logo* is disabled by the *DisableInput* predicate. In the second instance (❸), a rule is defined to disable logo input if the color is black. Similar to previous rule, an event listener for the respective datatype is added as well as a condition (*Equals)* to test the value of color. In case the condition evaluates true, *DisableInput* is used as well to deactivate input for logo.

Integration and execution

The created rule based task can now be integrated in the task model. The abstract task *Configure Product* with three concrete subtasks from figure 2 is replaced with the new rule based task element. As depicted in figure 8a), on initial execution rule ❶ is activated, creating three virtual tasks that enable input for color, size and logo. Selecting a color would trigger rule ❸. And, as shown in 8b), if the chosen color equals black, the selection for logo is disabled.

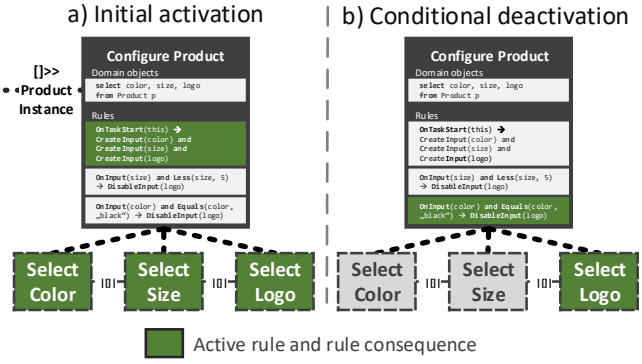

Figure 7. The rule element replaces manual created subtasks. a) On initial activation three virtual subtasks (select color, select size, select logo) are generated. b) If a color is selected the last rule is triggered. If the color was "black" the conclusion is applied and the logo selection would be disabled.

The given example provides no means to reactivate the selection once they are deactivated although this could easily be solved by adding an alternative (else) to the clause. In general, the modeler is responsible to avoid conflicts or deadlocks though this also concerns other CTT extensions and could be solved by model-checking techniques [31].

To conclude, within this section a rule based task was motivated, specified and demonstrated with an initial basic example, yet to fully embrace possible applications and benefits, additional examples are given in the next section.

USE CASE SCENARIOS

Following examples briefly show the use of rule based tasks in diverging scenarios to demonstrate their benefits.

Compact task models

Rule based tasks can be used to create compact task models by replacing concrete tasks with virtual tasks created by the CreateInput predicate. Especially tree visualizations, used commonly by task model editors, benefit in terms of clarity.

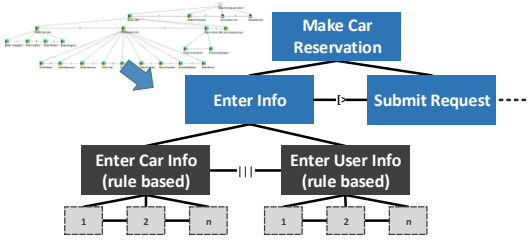

Figure 8. The example of [26] **is significantly reduced in size by using virtual task, thus allowing to focus on critical parts.**

Use-case dependent constraints

Domain models are typically aimed towards reuse and use-case overarching application. However, in order to comply with given requirements, it is often necessary to adapt minor details. Rules can be used to add such details without changing the domain model.

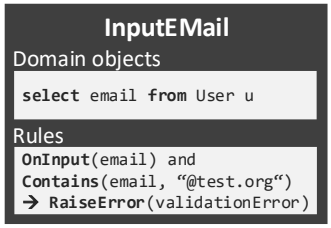

Figure 9. Simple use case specific rule that disallows email addresses from the test.org domain.

Constraint based behavior

Rule based tasks enable to express complex constraint based behaviors to enable use cases that cannot be expressed using original task model concepts.

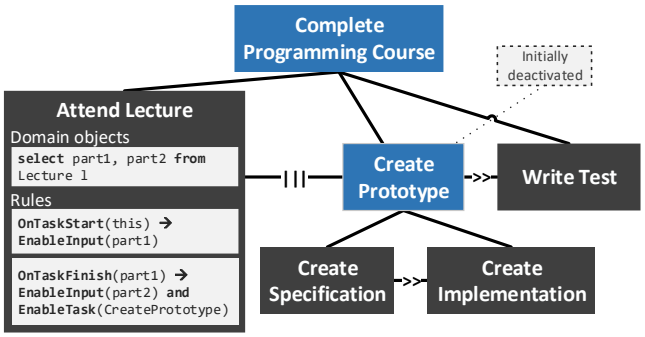

Figure 10. Task model for a programming course. The lecture is divided in two parts. Only if the first part is finished, the Create Prototype task should be activated.

Events to collect information

The integrated event mechanism allows rules to both raise as well as consume events. This can be used to create a central task that is notified by other tasks whenever they are completed successfully in order to count and conditionally activate a task or subtree.

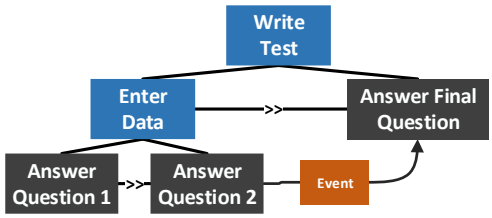

Figure 11. Events are raised if a question was solved successfully. Only if a certain number of correct answers is counted, the final question is activated.

Reuse by rule parametrization

As demonstrated in [19], placeholders can be utilized to enable reuse and parametrization of single tasks and whole subtrees. Applied to rules, parametrization can be used to prepare rules for reuse across different task models. For example, as indicated in figure 12, a generic *Process Payment* task tree is defined which, due to its parameter, can be inserted whenever payment needs to be handled.

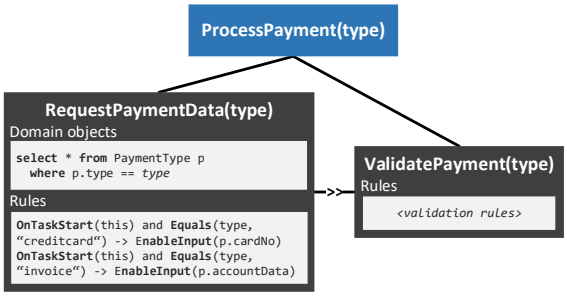

Figure 12. Simplified example of a reusable task tree for payment processing. The payment type is added as a placeholder. Embedded into another context, the placeholder would be replaced with the actual payment method.

CONCLUSIONS AND FUTURE WORK

We have proposed a method for extending hierarchical CTT-based task models with a powerful rule language that provides more capabilities than existing pre-/post-condition definitions. In particular, event-based control flow can be seamlessly integrated with conditional task activation, and the temporal operators provided by CTT. We introduce task-based rules that encapsulate complex interactive behavior in their rule descriptions thus removing the need to model each possible subtask (structure) explicitly. The creation of virtual tasks through rules can greatly reduce the complexity of the visual model, especially if complex dependencies control the activation of interaction tasks. The rule language makes it possible to embed those parts of the business logic in the task model that determines task flow. By using graph-based domain models in semantic format, application-independent models can be coupled with different task models. Generally valid rule-based knowledge can be represented at this level, allowing to separate knowledge depending on its application-specificity. The loose coupling via query-based domain model access allows for flexible querying and manipulation of domain entities and for abstracting tasks by parameterizing them with domain concepts.

By providing means to write more compact task models, the technique also supports the need to provide modelers with a good overview even of complex models. Developers can use the graphical tree model to observe the overall, essential task flow while hiding detailed flows in the rule section of tasks. At the same time, the language seems powerful enough to precisely define all control flow aspects that are needed to automatically generate UIs. Currently, we work on implementing an editor for the method. To address usability concerns, rule editing will be enabled in an interactive, graphical way. Future work will address testing the method in different use cases and on using it in automated UI generation processes.

REFERENCES

1. Davide Anzalone, Marco Manca, Fabio Paternò, and Carmen Santoro. 2015. Responsive task modelling. *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '15*, ACM Press, 126–131.
2. Tara Athan, Harold Boley, and Adrian Paschke. 2015. RuleML 1.02: Deliberation, Reaction and Consumer Families. *Joint Proceedings of the 9th International Rule Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, RuleML2015 Industry Track Doctoral Consortium*.
3. J Brüning, M Kunert, and B Lantow. 2012. Modeling and executing ConcurTaskTrees using a UML and SOIL-based metamodel. *12th Workshop on OCL and Textual Modelling, OCL 2012 - Being Part of the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012*: 43–48.
4. Jens Brüning, Anke Dittmar, Peter Forbrig, and Daniel Reichart. 2008. Getting SW engineers on board: Task modelling with activity diagrams. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4940 LNCS: 175–192.
5. Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. 2003. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers* 15, 3: 289–308.
6. Peter Forbrig, Célia Martinie, Philippe Palanque, Marco Winckler, and Racim Fahssi. 2014. Rapid Task-Models Development Using Sub-models , Sub-routines and Generic Components. *Human-Centered Software Engineering: 5th IFIP WG 13.2 International Conference, HCSE 2014, Paderborn, Germany, September 16-18, 2014. Proceedings*: 144–163.
7. A. Gaffar, Daniel Sinnig, A. Seffah, and Peter Forbrig. 2004. Modeling patterns for task models. *Proceedings of the 3rd annual conference on Task models and diagrams* 2, 1: 99–104.
8. Matthias Giese, Tomasz Mistrzyk, Andreas Pfau, Gerd Szwillus, and Michael Von Detten. 2008. AMBOSS : A Task Modeling Approach for Safety Critical Systems. *Engineering Interactive Systems 2008* 5247.
9. Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. 2008. OWL 2: The next step for OWL. *Web Semantics* 6, 4: 309–322.
10. Steven Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. Retrieved from http://www.w3.org/TR/sparql11-query/
11. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. 2004. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Retrieved from http://www.w3.org/Submission/SWRL/
12. Filip Kis and Cristian Bogdan. 2015. Generating Interactive Prototypes from Query Annotated Discourse Models. *i-com* 14, 3.
13. Tobias Klug and Jussi Kangasharju. 2005. Executable task models. *Proceedings of the 4th international workshop on Task models and diagrams - TAMODIA '05*, ACM Press, 119.
14. Jens Kolb, Manfred Reichert, and Barbara Weber. 2012. Using concurrent task trees for stakeholder-centered modeling and visualization of business processes. *Communications in Computer and Information Science* 284 CCIS: 237–251.
15. Quentin Limbourg and Jean Vanderdonckt. 2003. Comparing task models for user interface design. *The handbook of task analysis for human-computer interaction*: 135–154.
16. Marco Manca, Fabio Paternò, Carmen Santoro, Lucio Davide Spano, and Via Moruzzi. 2014. Considering Task Pre-Conditions in Model-based User Interface Design and Generation. *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems*, ACM, 149–154.
17. Frank Manola, Eric Miller, and Brian McBride. 2014. RDF 1.1 Primer. Retrieved from http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/
18. Célia Martinie, Philippe Palanque, Martina Ragosta, and Racim Fahssi. 2013. Extending procedural task models by systematic explicit integration of objects, knowledge and information. *Proceedings of the 31st European Conference on Cognitive Ergonomics - ECCE '13*, ACM Press, 1.
19. Célia Martinie, Philippe Palanque, and Marco Winckler. 2011. Structuring and composition mechanisms to address scalability issues in task models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6948 LNCS: 589–609.
20. Giulio Mori, Fabio Paterno, and Carmen Santoro. 2002. CTTE: support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering* 28, 8: 797–813.
21. Object Management Group. 2011. *Business Process Model and Notation (BPMN) Version 2.0*. Retrieved from http://www.omg.org/spec/BPMN/2.0/
22. Object Management Group. 2014. *Object Constraint Language v2.4*. Retrieved from http://www.omg.org/spec/OCL/2.4/

23. Object Management Group. 2014. *Decision Model and Notation (DMN)*. Retrieved from http://www.omg.org/spec/DMN/1.0/
24. Adrian Paschke, Harold Boley, and Zhili Zhao. 2012. Standardized Semantic Reaction Rules. *Rules on the Web: Research and Applications*: 100–119.
25. Fabio Paternò, C. Mancini, and S. Meniconi. 1997. ConcurTaskTrees: A diagrammatic notation for specifying task models. *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction*: 362–369.
26. Fabio Paternò, Carmen Santoro, Dave Raggett, and Spano Lucio Davide. 2014. MBUI - Task Models. Retrieved from http://www.w3.org/TR/task-models/
27. Fabio Paterno, Carmen Santoro, and Lucio Davide Spano. 2011. Engineering the authoring of usable service front ends. *Journal of Systems and Software* 84, 10: 1806–1822.
28. Fabio Paternò and Enrico Zini. 2004. Applying information visualization techniques to visual representations of task models. *Proceedings of the 3rd annual conference on Task models and diagrams*: 105–111.
29. Axel Polleres, Harold Boley, and Michael Kifer. 2013. RIF Datatypes and Built-Ins 1.0 (Second Edition). Retrieved from http://www.w3.org/TR/2013/REC-rif-dtb-20130205/
30. Frank Radeke and Peter Forbrig. 2010. Patterns in Task-Based Modeling of User Interfaces. In *Task Models and Diagrams for User Interface Design*, David England, Philippe Palanque, Jean Vanderdonckt and Peter J. Wild (eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–197.
31. Daniel Sinnig, Maik Wurdel, Peter Forbrig, Patrice Chalin, and Ferhat Khendek. 2007. Practical Extensions for Task Models. In *Task Models and Diagrams for User Interface Design*. Springer Berlin Heidelberg, Berlin, Heidelberg, 42–55.
32. Jon Stuart and Richard Penn. 2004. TaskArchitect. *Proceedings of the 3rd annual conference on Task models and diagrams - TAMODIA '04*, ACM Press, 145.
33. H Trætteberg. 2008. UI design without a task modeling Language - Using BPMN and diamodl for task modeling and dialog design. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5247 LNCS: 110–117.
34. Jean Vanderdonckt and Francisco Montero Simarro. 2010. Generative pattern-based design of user interfaces. *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems - PEICS '10*: 12–19.
35. Maik Wurdel, Daniel Sinnig, and Peter Forbrig. 2008. CTML: Domain and Task Modeling for Collaborative Environments. *Journal of Universal Computer Science* 14, 19: 3188–3201.