# Agent-based simulation with iTasks for navy patrol vessels

*Author:*
Wessel van Staal

*Supervisors:*
Prof. Dr. Ir. M.J. Plasmeijer
Dr. J. van Diggelen



*Thesis number:*
671

Master's Thesis
Computing Science
Radboud University Nijmegen
August, 2013

# Contents

**1 Introduction** **3**

**2 CWA and iTasks** **6**
2.1 Cognitive work analysis . . . . . 6
2.2 iTasks . . . . . 9
2.2.1 Task concepts . . . . . 9
2.2.2 Semantics of iTasks . . . . . 11
2.2.3 GiN: A graphical iTasks notation . . . . . 12
2.2.4 iTasks in practice . . . . . 13
2.3 iTasks and CWA in software development . . . . . 14

**3 Agent-based simulation** **15**
3.1 BDI agents . . . . . 16
3.2 Hierarchical task networks . . . . . 17
3.3 2APL . . . . . 18
3.4 TÆMS . . . . . 19
3.5 Brahms . . . . . 20
3.5.1 KAoS integration . . . . . 22

**4 An iTasks agent framework** **23**
4.1 Introduction . . . . . 23
4.2 Architecture . . . . . 24
4.3 Dualism of tasks and agents . . . . . 25
4.4 Extensions to iTasks . . . . . 26
4.4.1 Extended service representation . . . . . 26
4.4.2 Annotating tasks with tags . . . . . 27
4.4.3 Changes for agent framework . . . . . 28
4.5 Semantics of agents . . . . . 29
4.5.1 Events and states . . . . . 29
4.5.2 Tagging tasks . . . . . 30
4.5.3 Running agents . . . . . 30
4.5.4 Agent primitives . . . . . 31
4.6 Towards automatically deriving agents . . . . . 33
4.7 An example . . . . . 35

**5 Case study: damage control with iTasks** **38**
5.1 Problem definition . . . . . 38
5.2 Building the iTasks prototype . . . . . 41
5.2.1 Handling fire alerts . . . . . 42
5.2.2 Starting blanket searches . . . . . 43
5.2.3 Searching compartments . . . . . 43
5.2.4 Executing attack plans . . . . . 44
5.3 Using the iTasks agent framework . . . . . 45
5.3.1 Introduction . . . . . 45

5.3.2 Task costs . . . . . 45
5.3.3 Agent behaviour . . . . . 46
5.3.4 Implementation . . . . . 48
5.4 Performing simulations . . . . . 51
5.5 Discussion . . . . . 53

**6 Conclusions** **54**
6.1 Future work . . . . . 56

# 1 Introduction

Designing automation for complex domains is a time consuming and error prone process. Such a process usually consists of a domain analysis and a requirements analysis, followed by the implementation and testing of the software system. Multiple iterations of these phases are required to obtain the product as desired. In the domain of our interest, navy vessels, the complexity of the design is increased because of reduced (human) resources on vessels due to budget cuts. Furthermore, budget cuts reduce the amount of time that is available to build and test software.

Domain experts think in terms of high level concepts that do not always translate well to software. Using their domain knowledge, they know precisely (explicitly or implicitly) which tasks humans need to perform under what circumstances to achieve certain goals. These high level concepts need to be translated into very precise program statements. The amount of time required to implement a software system is proportional to the level of abstraction and expressiveness that the programming language and tools offer. With existing software development methods, the gap between concepts used by programmers and domain experts is often too large, increasing the amount of time required to build usable prototypes. We would like to validate functionality as quickly as possible, to identify issues early on and minimize the effort required for reparation.

Once requirements have been defined and a prototype is developed, experiments need to be conducted in order to verify claims about design decisions. Such experiments can be dangerous in more complicated domains such as the navy, where mistakes can have severe consequences. Moreover, it is costly to conduct and organize such experiments when human end users are involved. Another method that can be used to estimate the impact of design decisions is simulation. Human end users are simulated with computer programs, such that experiments can be conducted more efficiently in less time. This is used extensively in the space domain using the Brahms simulation tool [36] [4].

To perform simulations, it must be possible to model the users of the software system. Software components that are used to simulate human behaviour are referred to as *agents*. Each agent is able to observe, reason and act upon its environment. In existing agent-based systems, the tasks that can be performed are included in the definition of the agents. A disadvantage of this approach is that it imposes constraints on the amount of reuse of tasks. Functionality that is common to agents cannot be abstracted and isolated in building blocks to reduce the amount of time required to build future agents. Furthermore, it is harder to model team-based cooperation when all tasks are decentralized. Agents in existing tools often lack a coordination artefact to coordinate task distribution, making it more difficult to efficiently divide tasks among agents.

Another important feature that is underdeveloped in existing agent-based simulation methods is the ability to perform human-in-the-loop [37] or crew-in-

the-loop [10] simulations. In order to support actual humans participating in simulations, user interfaces must be defined along with all the machinery to handle interactions. With existing methods, this functionality is not automatically derived from agent specifications or task models. Therefore, it is not trivial to interchange humans with software agents. Human-in-the-loop simulations are especially relevant in training scenarios. Furthermore, actual visualizations of task models used with simulations help discussions between domain experts and developers to improve requirements.

TOP provides the concept of a task to build fully functional software by specifying relations between primitive tasks. In this work, we propose to use the iTasks framework, which is an implementation of TOP. Developers only need to concern themselves with issues that are related to the requirements of the software system. All of the other concerns are handled by the iTasks framework, including the generation of a user interface, handling data persistence, handling user input, etcetera. iTasks is built with Clean, a state of the art functional language supporting a high degree of abstraction. The concept of a task creates an opportunity to bridge a gap between analysts and developers, providing a shared idea that is understandable for both expertises. Analysts can use task analysis methods to gain insight about what tasks humans perform on a high level, which can be used to assist the iTasks programmer in constructing the task model in software. If developers can share concepts with domain experts to communicate in an efficient manner, the time to build a software system can be greatly reduced. Further, if software developers do not need to concern themselves about matters that are not important to the functional requirements of the system, they can use their time in the most efficient manner possible.

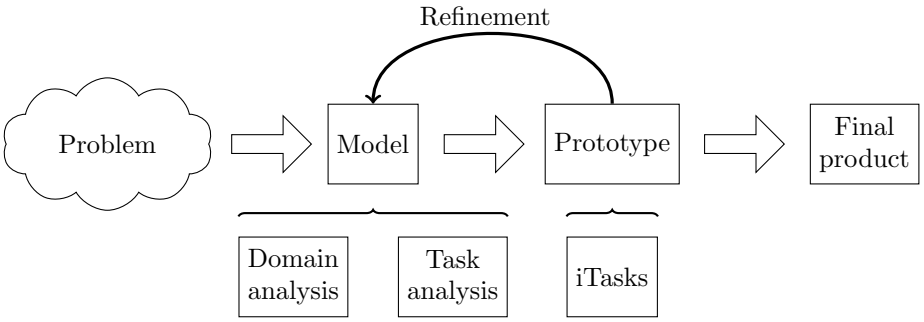


Figure 1: Overview of research domain

In figure 1, we show an abstracted view of a development process involving iTasks as prototyping tool. Given some real-world problem, we construct a contextual model by performing a domain analysis. Based on such an analysis, we can derive requirements and claims that must be satisfied by the solution of the problem. Requirements are implemented in prototypes iteratively, possibly refining requirements and claims during the development process. Our aim is to

use iTasks as a tool to perform rapid prototyping and agent-based simulation, providing us with a tool to build a formal and executable task model that forces domain experts to think in concrete tasks. We focus on agent-based simulation with iTasks to enable automated testing, allowing us to improve requirement refinement and claim validation. We use domain analysis tools as context for requirements: in this work, we also study the use of Cognitive Work Analysis to derive requirements for iTasks prototypes.

We propose an agent framework that is defined in terms of iTasks to overcome issues with other agent-based tools. Instead of designing tasks specifically for an agent in a decentralized manner, it can be more appropriate to use a centralized task model defining the constraints and order of tasks to be performed by (a team of) agents. A combination of approaches is also possible, enabling the option of moving tasks between the centralized task model and individual agents. Because we define software agents in iTasks with a functional language, we have the opportunity to create abstract building blocks that promote reuse in agents.

We ask the following questions:

How can we use iTasks as a tool to assist in designing automation for complex domains?

- How can we perform simulations with agents in iTasks?
- How do agents in iTasks relate to agents in other frameworks?
- How can iTasks be used to rapidly build prototypes in a non-trivial domain, based on products from task analysis methods and input from domain experts? In particular, how can cognitive work analysis assist in designing requirements for iTasks models?

This research is conducted in cooperation with the Netherlands Organisation for Applied Scientific Research (TNO) as part of a project to reduce the amount of personnel on navy patrol vessels. Within this project, TNO intends to use iTasks as a tool to experiment with task models and to provide a mechanism for personnel training. Computer simulation is a commonly used tool at TNO to test various solutions. With that in mind, the goal of our research is to enable such functionality for iTasks. We compared several simulation tools used at TNO with the framework we propose in this thesis. Furthermore, we developed the prototype in our case study in cooperation with domain experts at TNO.

The aim of this research is to gain insight about how iTasks can be used to improve the development process. A part of that aim is to extend iTasks where necessary in order to move the framework forward towards a product that can be used in a variety of domains. In this work, we discuss how we extend iTasks to support agent-based simulations. We explain how these agents relate to existing agent-based systems. Furthermore, we discuss how we design and build a proto-

type based on input from domain experts and products from analysis methods. Also, we show how we can perform simulations with this prototype.

# 2 CWA and iTasks

In this chapter, we discuss both Cognitive Work Analysis (CWA) and iTasks. We also explain how decision ladders in CWA relate to iTasks.

## 2.1 Cognitive work analysis

Analysts use various methods in order to understand and optimize tasks that humans execute in complex environments. Examples of these methods include cognitive work analysis (CWA) [31] [16] and hierarchical task analysis [38]. In this study, we mainly focussed on CWA, since this was the method used to analyse the domain of the prototype as described later in this work.

Cognitive work analysis is a method to gain understanding about complex domains. Using the method, analysts can answer high level questions such as why a system exists and what tasks are performed by whom. Rather than very explicitly describing workflows of tasks that are performed, CWA focusses on describing the constraints and goals of tasks. An advantage of this approach is that it provides a way to handle situations that were not anticipated during analysis. This makes the method especially useful for the navy domain where unexpected events occur on a daily basis. However, it is not clear how usable the analysis results are in a software development process, when details are important. We claim that CWA products do not contain enough detail to be able to translate into iTasks implementations with little effort. However, we did use some of CWA as high level (informal) specifications. We will now describe the phases that are included in CWA.

**Work domain analysis** During domain analysis, the elements of the domain and their relations are defined using an abstraction hierarchy. In this hierarchy, it becomes clear what the higher level goal of the system is and of what parts it is composed. At the lower level of the hierarchy, all of the important physical objects are described that support the system on the operational level. On the top of the hierarchy, the high level goals are described. This hierarchy has been used to construct user interface designs [40], but is primarily used to gain a high level insight about the domain.

**Control task analysis** The goal of control task analysis is to investigate the activities (tasks) that are performed in the domain. The main products of this phase are *decision ladders* [31] that describe a sequence of information processing steps that occur during a task. A decision ladder describes what has to be achieved and the constraints that must be enforced, but

not in terms of precise steps as one would expect in a workflow. It leaves room for human interpretation, making it more suitable to handle unanticipated events. Decision ladders also support short cuts (*shunts*) in order to achieve a goal in less time. This enables the option of modelling paths for experienced and less experienced actors. Decision ladders were used as the basis for the prototype as described later in this work.

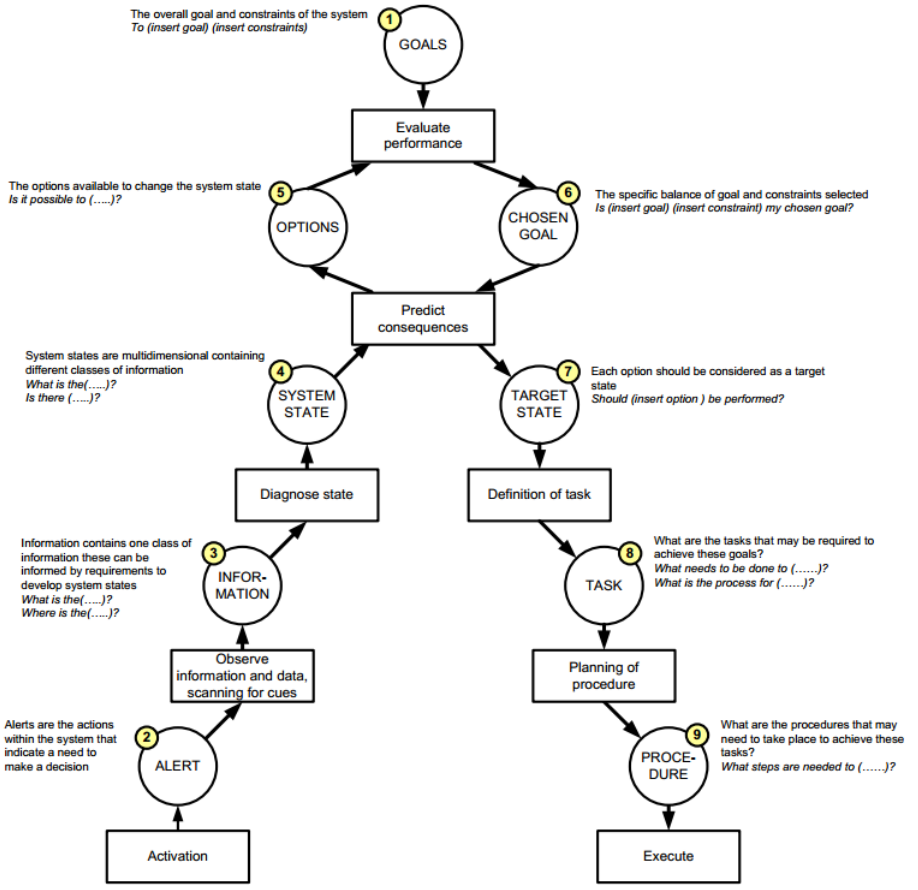


Figure 2: The decision ladder template [15]

Decision ladders are instances of a generic structure; the overall process remains the same for each ladder. We use 2 as an example of a decision ladder. The left hand side of the ladder in figure 2 corresponds to activation of the decision making process and building the *internal picture* of the context. An activation may be caused by a triggered sensor or a human reporting a calamity during a blanket search on a vessel. In iTasks, we can model triggers that fire when an environment changes; if a predicate is satisfied, we can allocate a task and assign it to some user to aid in the

decision making process. Part of this task might be to inform the user about the environment such that the user can build an internal picture. The decision ladder does not explicitly define whether tasks are performed by an information system or by human users; this yields an opportunity to experiment with the level of automation. Sheridan & Verplank [34] describe the level of autonomy (LOA) of an information system. A high LOA corresponds to a system where most tasks are performed without any human interaction. iTasks is especially suitable to experiment with the LOA: a rule-based task can easily be converted into a task that requires interaction with a human. Furthermore, with iTasks, one can use an adaptive strategy, where a task is only performed by a human when the system is unable to do so.

The top level of the ladder describes the knowledge-based steps in the decision ladder. In iTasks task models, this would correspond to tasks that only serve a supporting and informing role; almost no tasks can be automated (easily) in this case. An example of this might be the prioritizing of calamities by the commanding officer on a marine vessel. While an information system can play a supporting role, it is up to the commander to make these important decisions.

The right hand side corresponds to the planning and execution part in a decision making process. With iTasks, this would correspond to tasks that present the user with options and known procedures to resolve some issue. In command and control, it is important to be aware of the available resources (i.e. the capabilities) that can be allocated to handle a crisis situation. iTasks has built-in functionality to delegate tasks to users in a flexible manner. Users can monitor the progress of delegated tasks, optionally reassigning them when needed.

Each part of the decision ladder corresponds to a set of requirements and task types for iTasks. In our case study, we further elaborate on ways to interpret one instance of a decision ladder.

**Strategy analysis** The goal of a strategy analysis is to describe recurring paths of decision ladders in more detail. This phase identifies the most common ways in which a task is performed, allowing for optimizations. However, the strategy analysis will never yield an exhaustive list of all possible ways to perform a task. The focus lies on the most common paths.

**Social organization & cooperation analysis** In this phase, the goal is to analyse which actors must perform what tasks. Products from earlier phases are annotated with actor indicators to describe which actors can be involved.

**Worker competencies analysis** Tasks require skills and experience. A worker competency analysis yields a matrix indicating what task requires which competencies from workers.

## 2.2 iTasks

iTasks [14] [26] [27] is a software library that can be used to specify high level work tasks performed by users. An iTasks application is a task composed of subtasks, where each individual subtask can also be a composition of tasks. When viewed as a hierarchy, basic primitive tasks such as displaying information on the screen or filling out a form exist on the bottom. At the top of the hierarchy are the tasks that correspond to higher level functionality of the application.

Tasks are composed with various types of composition. These compositions correspond to the flow of an application, which can expressed by workflows. In fact, an extension exists for iTasks that allows users to construct tasks visually by drag and drop operations. The basic forms are sequential and parallel composition, where tasks are executed sequentially and in parallel respectively. From these basic compositions, more complicated variants can be derived.

While other workflow systems are often fixed and static, iTasks provides the possibility of creating highly dynamic task models. Tasks produce values that can be used to dynamically decide what task to perform successively. Tasks can be assigned dynamically to users based on complicated decision procedures. With iTasks, it is possible to model an application that can suggest multiple alternatives to perform a task, based on the current context.

One of the advantages of iTasks is that a multi-user application is completely derived from the tasks that are specified. This application is web-based, only requiring devices that are capable of browsing websites. During development, one does not need to be concerned about implementation details of the user interface. This makes iTasks especially suitable for rapid prototyping, which is one of the main motivations of this research. It is possible to extend and adapt the user interface, but this is not required to build a fully functional application.

iTasks provides functionality to define and manage users. Tasks can be delegated to different users and can share information, allowing users to monitor task progression and share workload. Since all of these features are implemented as tasks, iTasks is especially useful to construct a formal model of tasks performed by teams. This model produces a fully functional application and forces domain experts to think in more exact terms.

### 2.2.1 Task concepts

To build task specifications, we have several constructs available. Among the primitive tasks are *interaction tasks* that require input from users to complete. We show `viewInformation`, `enterInformation` and `updateInformation` as examples:

```
viewInformation     :: d [ViewOption m] m       → Task m | descr d & iTask m

enterInformation    :: d [EnterOption m]        → Task m | descr d & iTask m

updateInformation   :: d [UpdateOption m m] m   → Task m | descr d & iTask m

:: Stability :== Bool

:: TaskValue a = NoValue
               | Value a Stability
```

Each of these functions produce the abstract notion of a `Task a`, which is the fundamental type in the iTasks system. A `Task a` is a description of all the computation required to construct the user interface, handle input from users and construct task values. Tasks produces values (`TaskValue a`) that are either non-existing (`NoValue`), stable or unstable. Stable values no longer change as opposed to unstable values. This is a powerful concept in iTasks: it is possible to act upon incomplete information.

With `viewInformation`, some description $d$ describes the task that must be performed by the user. `ViewOption` is used to produce an optional *view* for the user; information that is defined in terms of $m$. `enterInformation` defines a task that produces a $m$ out of 'nothing': the user is responsible for doing so. The iTasks system generates a user interface using type information at runtime. The `enterInformation` task produces `NoValue` until the user has entered all required information to produce $m$. The framework has similar functions for entering (multi) choice values and viewing/updating shared information.

Primitive tasks can be composed with *combinators* to produce more complex tasks. In iTasks, there are two core combinators: `step` (sequential composition) and `parallel` (parallel composition). There is also value transformation, but we omit this part. The other combinators are derived from these core combinators.

```
:: TaskStep a b
   =       OnValue             ((TaskValue a)  → Maybe (Task b))
   |       OnAction    Action  ((TaskValue a)  → Maybe (Task b))
   | ∃ e:  OnException         (e              → Task b)          & iTask e
   |       OnAllExceptions     (String         → Task b)

step :: (Task a) [TaskStep a b] → Task b | iTask a & iTask b

:: ParallelTaskType
   = Embedded
   | Detached ManagementMeta

:: ParallelTask a  :== (SharedTaskList a) → Task a

parallel :: d [(ParallelTaskType,ParallelTask a)] → Task [(TaskTime,TaskValue a)]
```

```
16     | descr d & iTask a
```

With `step`, `Task a` is evaluated to produce values. A list of possible steps to be performed based on these values is used to execute a new task in sequence, producing a new value $b$. This is slightly similar to the observer pattern known in object oriented programming, where one can 'hook into' values that are produced from some source. A task in iTasks can be seen as a data source that is continuously producing new values until it stabilizes. A `TaskStep` is some step requiring action from a user (a button) or a rule-based step in which some predicate is used to decide the next task to be performed. When no `TaskStep` matches on a value produced by `Task a`, `step` produces `NoValue` and continues to evaluate `Task a`.

The `parallel` combinator takes a list of tasks to be performed in parallel. The combinator collects the results from these tasks continuously and combines them with a time-stamp. Tasks supplied to the parallel combinator can be performed in the current *session*, allocating them to the current user. It is also possible to assign tasks to other users, using the `Detached` constructor in `ParallelTaskType`. The derived `@:` combinator makes use of this functionality. An important feature of the parallel combinator is the ability to monitor values produced by tasks by providing a `SharedTaskList`. Each task is able to monitor its *siblings*, enabling more advanced collaboration scenarios. Each of the following combinators can be defined in terms of the parallel and step combinator:

```
1 (≫*) infixl 1 :: (Task a) [TaskStep a b]  → Task b   | iTask a & iTask b
2 (>>=) infixl 1 :: (Task a) (a → Task b)   → Task b   | iTask a & iTask b
3
4 (-||-) infixr 3 :: (Task a) (Task a)    → Task a       | iTask a
5 (-&&-) infixr 4 :: (Task a) (Task b)    → Task (a,b)   | iTask a & iTask b
6 (@:)   infix 3  :: worker   (Task a)    → Task a       | iTask a & toUserConstraint worker
7 allTasks        :: [Task a]             → Task [a]     | iTask a
8 anyTask         :: [Task a]             → Task a       | iTask a
```

Both ≫* and `>>=` can be defined in terms of `step`: ≫* is the infix representation of `step` and `>>=` is the monadic style bind operator. `Task b` is produced once `Task a` produces a stable value. `-||-` is the or-combinator, performing two tasks in parallel from which one must complete. Similarly, `-&&-` is the and-combinator, performing two tasks at the same time that must be completed.

### 2.2.2 Semantics of iTasks

Semantics of software systems known in academic literature are often described using a syntax grammar and derivation rules. These semantics often contain mistakes, are difficult to read and they need to be verified manually. The semantics of iTasks are defined differently; the framework is expressed as a term rewriting system implemented in Clean∗ [28]. This way, we can use the Clean

compiler to check for trivial errors (e.g. type errors or typing mistakes) and actually execute them for testing.

A task in iTasks can be seen as a function that modifies some state based on an event, producing both a value and a *reduct* describing the continuation of the task. Evaluating a task comes down to providing events and a state, rewriting the resulting reduct to continue evaluation until the task value stabilizes or some error occurs. We briefly discuss the details of the task evaluation. Further details are specified in [28].

```
1  :: Task a :== Event → *State → *(Reduct a, Reponses, *State)
2
3  evaluateTask :: Task a → *World → *(Maybe a, *World) | iTask a
4  evaluateTask ta world
5  # st = {taskNo = 0, timeStamp = 0, mem = [] , world = world}
6  # (ma,st) = rewrite ta st
7  = (ma, st.world)
8
9  rewrite :: Task a → *State → *(Maybe a, *State) | iTask a
10 rewrite ta st=:{world}
11 # (ev,world) = getNextEvent world
12 # (t, world) = getCurrentTime world
13 # st = {st & timeStamp = t, world = world}
14 # (Reduct res nta, rsp, st) = ta ev st
15    =   case res of
16        ValRes _ (Val a Stable) = (Just a, st)
17        ExcRes _ = (Nothing, st)
18        _ = rewrite nta
19 = {st & world = informClients rsp st.world}
```

The function `rewrite` obtains an event through some function `getNextEvent` and applies the task function. When the task value is stable, the recursion ends and the value is returned. Similarly, if an exception occurs, a `Nothing` is returned. Otherwise, the reduct obtained from the task is used to continue evaluation recursively. Each time the task is evaluated, the clients are informed by some function `informClients`. The details of both `getNextEvent` and `informClients` are not relevant to the semantics. Later in this work, we use the semantics proposed in [28] as foundation for the semantics of our agent framework.

### 2.2.3 GiN: A graphical iTasks notation

Graphical notations are common to many workflow systems. A graphical representation often clarifies the flow and structure of a program, increasing the readability for non-technical users. This can help tremendously when discussing implemented functionality with domain experts. Such a notation exists for iTasks in the form of GiN [11]. GiN provides graphical elements for some of the combinators that we discussed in section 2.2.1, but it is slightly outdated: the step (≫*) combinator and shares in iTasks are not yet supported. We use

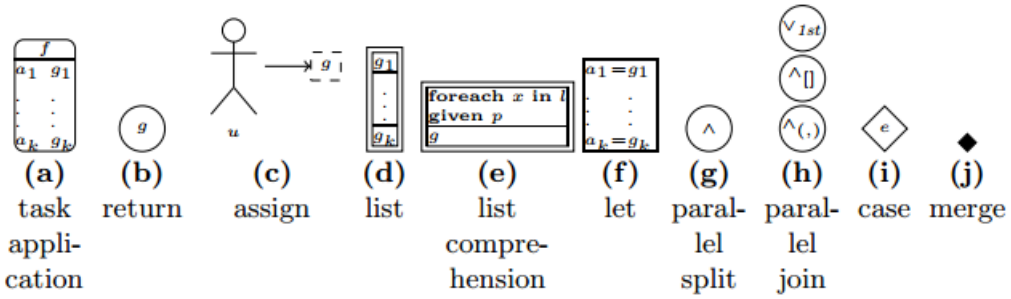


Figure 3: Elements of GiN [11]

ad-hoc solutions in this work to express these concepts visually. GiN can be translated directly to an iTasks task specification, enabling the option of executing the graphically designed workflows. In this work, we use some elements of GiN to clarify some of the code examples. However, we do use abstracted views that omit information to keep the diagrams clear and simple.

The following elements are outlined in figure 3. Function $f$ applied to arguments $g_1, \cdots, g_k$ corresponds to **a)**. Lifting a value $g$ into the task domain with `return` corresponds to **b)**. Assigning a task $g$ to user $u$ is expressed with **c)**. The elements **g)** and **h)** are used for parallel composition: $\vee_{1st}$ maps to `anyTask` and $\wedge_{[]}$ maps to `allTasks`, while $\wedge_{(,)}$ maps to `-&&-`. The elements in figure 3 are connected with arrows that correspond to the bind operations. $\xrightarrow{p}$ corresponds to `>>= λp.` and $\rightarrow$ corresponds to `>>|`.

### 2.2.4 iTasks in practice

The iTasks system was used to build a prototype for the Netherlands Coast Guard, to handle incident response scenarios. The Netherlands Coast Guard handles many incidents daily, each requiring a different workflow that dynamically changes as more details about the incident arise. This work by Lijnse et al [20] [21] showed that the iTask workflow definition language (WDL) is powerful enough to capture highly dynamic tasks that occur during an incident. This work demonstrates the flexibility of iTasks: task models need not be fixed nor static as with other workflow systems, a task model can be modelled such that the system suggests tasks or different alternatives to approach the same problem, depending on the situation. This leaves more initiative to the human user, which is sometimes required with knowledge-based tasks. A knowledge-based task is difficult to automate; it requires human experience and judgement.

The work done for the cost guard also showed that higher order tasks provide a powerful concept in iTasks. Functionality that is useful in multiple instances

(e.g. providing suggestions for alternative tasks) can be generalized by such constructs. Since tasks can be treated as data, they can be stored in data structures along with meta information, further improving the reusability of functionality.

## 2.3 iTasks and CWA in software development

In this research, both CWA and iTasks are positioned in a development method called Situated Cognitive Engineering (SCE) [24]. We illustrate this in figure 4. We distinguish three components: foundation, design specification and testing. In the foundation, we explore the domain of the problem. Part of this exploration is to define the roles, constraints and goals in the domain. Based upon this foundation, we formulate requirements that are optionally based on design patterns. In software engineering, a design pattern relates to recurring patterns that have proven to be working in practice. It has a similar meaning in SCE, where design patterns describe proven ideas for requirements. We also specify requirements for training (i.e. defining the capabilities required to perform tasks) and scenarios usable for (human-in-the-loop) tests.

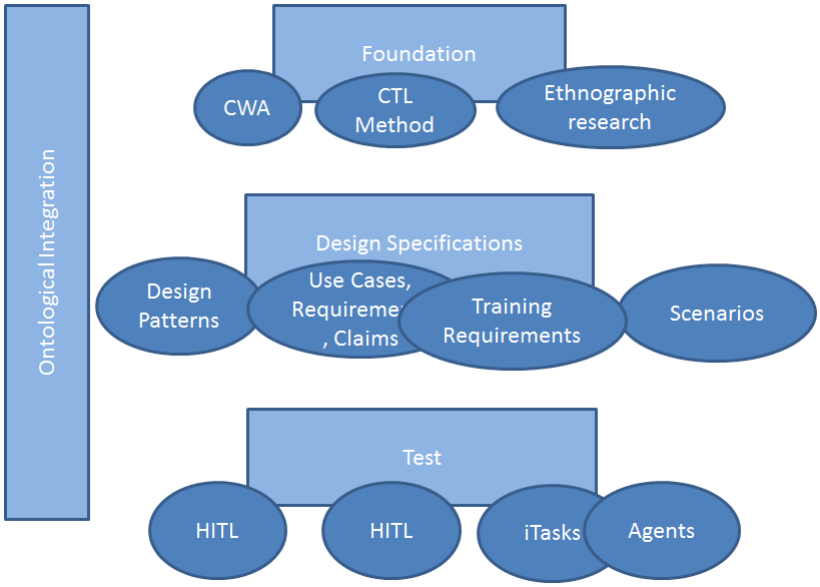


Figure 4: CWA and iTasks positioned in SCE

Cognitive Work Analysis can be used in the foundation component: part of a CWA analysis is to define an abstraction hierarchy and decision ladders. These high level concepts aid in domain exploration. For example, an abstraction hierarchy defines relations between the goals of a system and the physical entities

and systems available to achieve these goals. Decision ladders can also serve as foundation for high level requirements for iTasks.

iTasks is positioned as a rapid prototyping tool to test implementation of requirements. One of the main reasons to use iTasks is the minimal amount of effort required to build fully functional prototypes. Furthermore, one forces domain experts to think about the desired behaviour of a software system in more detail. By using agent-based simulation, tests can be performed by a combination of both human and software agents (human-in-the-loop). Claims that were defined in the design specification can be verified by results of these tests.

# 3 Agent-based simulation

Simulation is used for a wide variety of purposes: some examples include gaining understanding about human behaviour in complex scenarios (e.g. emergency evacuations), to train personnel in dealing with crisis situations and to compute effects of changes during the design of a product. Simulations can be performed in various ways. A real-life exercise on a patrol vessel where several different scenarios are trained is an example of such a simulation. Another example is the use of computational simulation, where some approximation of an environment and scenario is defined in a computer model to obtain results through computation. This has the advantage that simulations can be performed with relatively few resources as opposed to real-life exercises.

Commonly known methods of computer simulation include discrete time event simulation [7] and stochastic modelling [17]. With discrete time event simulation, a time line of events is processed. An event resembles a system state change at a particular time. Entities that have state are effected by these events. An example of stochastic modelling is the use of discrete time markov chains (DTMCs), where one can model a system by a set of states and transitions. Each transition is associated with a probability. Using DTMCs, one can compute the probability of reaching a particular state within $n$ steps.

Another method of computer simulation involves the use of *agents* that interact with an environment and each other. There is no real consensus on what it takes to call something an agent. According to Macal & North [22], an agent is an autonomous, self-contained entity attempting to achieve goals with respect to its behaviours. Furthermore, an agent is flexible and has the ability to adapt its behaviour by learning from experience. An agent can represent a human, but it is also possible to model entities such as vehicles.

Creating models for agent-based simulation is often called *bottom-up* modelling [32], as behaviour is defined at the individual level from which the global behaviour emerges. A simulation may involve a large number of agents, all communicating with each other and their environment.

Macal states that an agent-based model consists of (*i*) a set of agents with their behaviours and attributes; (*ii*) a set of inter-agent relationships and methods of interaction; (*iii*) the environment that agents interact with.

According to Bonabeau [1], agent-based simulation (ABS) offers the following advantages when compared to other modelling techniques:

- ABS captures emergent phenomena;
- ABS provides a natural description of a system;
- ABS is flexible.

Bonabeau argues that emergent phenomena are hard to predict and can be counter-intuitive, making them difficult to create and observe with other modelling methods. Bonabeau also argues that some systems are natural to define in terms of agents: an example of a problem that is suitable for ABS is traffic modelling [23], where each (driver of a) vehicle is represented by an agent.

Sheridan & Verplank [34] describe the level of autonomy regarding human-machine interactions. As the level of autonomy (LOA) increases, the machine requires less interaction to complete its tasks. Simulation can be used to experiment with various LOA and explore the effects in different situations. It has been shown that ABS is particularly useful to experiment with situated tasks that are driven by communication [3] [13].

## 3.1 BDI agents

There are many different variants of agents and levels of intelligence and autonomy. BDI agents are defined in terms of beliefs, desires and intentions [30]. An agent has a *belief state* representing beliefs held about the environment at time $t$. All agents observe their environment through sensors and act upon their environment with actuators. By interpreting sensor information, an agent can update its belief state if it desires. This can be modelled with the following function [18]:

$$remember : S \times P \to S$$

where $S$ represents the belief state and $P$ represents percepts that are produced by sensors. A belief state is different from the actual environment: an agent can have beliefs about the environment that are false. Beliefs can be quantitative: an agent can believe that some proposition about the environment is true to some degree. Belief states are used as a temporary memory to accumulate facts during the agent's lifetime and aid in making decisions. When combined with sensor information, agents can decide what actions to perform. This can be modelled with the following function:

$$do : S \times P \rightarrow C$$

where $C$ represents *commands* that can be given by agents to change their environment or interact with other agents.

BDI agents also have the concept of *desires* and *intentions*. Desires are goals the agent wishes to achieve at some time in the future. Intentions are the steps the agent intends to take to achieve a goal. By observing the environment and interacting with other agents, an agent can decide to change its intent or desire.

## 3.2 Hierarchical task networks

In planning and artificial intelligence, a commonly known concept is a *hierarchical task network* [33] [19]. Such a network is based on a hierarchical decomposition of a problem domain into tasks. A network consists of primitive tasks and non-primitive tasks that can be decomposed into other tasks. This network can be seen as a plan to achieve a certain goal and can be obtained by applying a hierarchical task analysis [38]. The concept of a HTN is used by TÆMS to define software agents. Interestingly, a method is described to map task hierarchies to concepts of BDI agents [9].

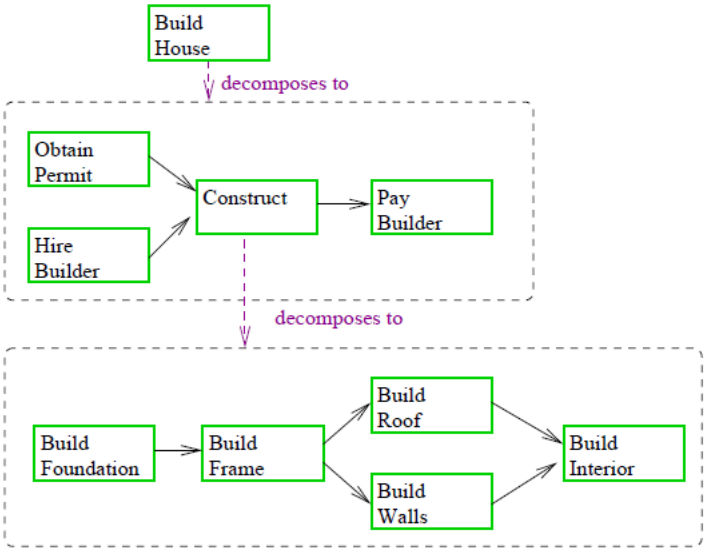


Figure 5: Example of a hierarchical task network

In [9], a *task hierarchy language* is defined. A task is defined as $T(Type, [(T_1, C_1), \cdots, (T_n, C_n)])$ where $Type$ denotes the type of task $T$, $[T_1, \cdots, T_n]$ denote the subtasks of $T$

| Task hierarchy | BDI agent |
|---|---|
| State | Beliefs |
| Main task | Goal |
| Subtask | Goal or plan |
| Primitive task | Action (atomic plan) |

Table 1: Mapping from task hierarchies [9] to BDI agents

and $C_1, \cdots, C_n$ denote the conditions under which subtasks can be adopted. *Type* specifies what composition is used: sequential, all, one or prim. Sequential composition specifies that all subtasks must be executed one after the other. All or one composition specifies that all subtasks must be executed or just one respectively. Prim denotes that the task is primitive and is not composed of any subtasks. The paper then shows that such a hierarchy can be translated to an agent in 2APL.

Conceptually, there are many similarities between HTNs and task models we have for iTasks. For example, primitive actions correspond to the primitives we have in iTasks. Further, non-primitive tasks use some form of composition composing other tasks. We have a similar notion in iTasks with task composition. However, task models for iTasks are defined formally in Clean. Task hierarchy networks are less expressive when compared to task models in iTasks in terms of constructs for abstraction. For example, we can define higher order tasks in iTasks. Additionally, we can identify common patterns in task models and create abstractions to maximize the amount of reuse.

## 3.3 2APL

2APL [5] is a programming language to implement BDI-based agents. It provides higher level constructs to implement concepts commonly used in agent-based programming such as plans, goals and beliefs. The programming language consists of a combination of declarative and imperative (Java) constructs, using the declarative subset to formulate beliefs and goals while the imperative subset is used to formulate *plans* and interfaces to external environments. A goal is a desired world state (i.e. a predicate over the belief state and environment), while a plan is a sequence of actions to achieve a goal. An important concept of 2APL is the notion of *planning goal* (PG) rules of the form $goal \leftarrow belief \mid plan$. To achieve *goal* when an agent believes $belief$, it must execute *plan*. A plan consists of actions such as updating the belief state, communicating with other agents or interacting with the environment. One of the more interesting features of 2APL are *repair rules*. If some plan fails, a repair rule can be used to alter the plan such that the agent can recover from the failure. This adds flexibility to how agents need to deal with errors.

2APL is a general purpose agent language: agents can interact with any envi-

ronment and are not tailored towards a specific technology. A large difference with this work is that we define agents that are specifically designed to operate on iTasks task specifications. The tight coupling between task specifications and agents performing tasks is expressed by the duality of the two concepts. Whereas 2APL is a completely separate framework specifically for the specification of agents, we utilize the same model for both task specifications (programs) and agent definitions.

With 2APL, each agent has a high level of autonomy in the sense that there are no constraints on what tasks can be performed at a particular time. This is different from agents we define in this work; we have a central iTasks model that defines the constraints and order of tasks to be performed by agents. This means that the level of autonomy is constrained. An advantage of this approach is that we have a central point of coordination. In team-based scenarios, we can define all tasks that must be performed by a team along with coordination in one model. With 2APL, tasks are coupled with agents and coordination must be done by inter-agent communication.

There is some similarity between error handling of 2APL agents and iTasks agents: 2APL uses *plan repair rules* as mentioned earlier. This is a way to alter plans that have failed. iTasks agents are also capable of altering their tasks when failures occur by using the iTasks exception model. Using the step combinator, one can define an exception handler to produce a new task (or plan) to be executed. If the exception is not handled, it is propagated onwards in the task tree. This makes it possible to define general error handlers for iTasks agents.

## 3.4 TÆMS

TÆMS (Task Analysis, Environmental Modelling and Simulation) [6] [12] is a modelling language for describing task models for agents. The goals that agents should achieve are defined at the top of such a tree, composed of tasks that the agent should perform. Tasks are in turn composed of other tasks and *methods*, which cannot be decomposed further. A method corresponds to primitive actions that an agent can perform (e.g. manipulate the environment). Each method is associated with a *quality* which is a indicator of the quality of the outcome. Methods are also associated with a cost and duration distribution. Tasks are then composed by defining Quality Accumulation Functions (QAFs) describing the semantics of executing and accumulating the quality of outcomes of subtasks/methods. With TÆMS, each agent is assigned a task model. It is also possible to define multi-agent systems, allowing inter-agent relationships between task models. For example, an agent might depend on another agent to perform a certain task.

The task models that are defined with TÆMS correspond to hierarchical task networks as described earlier, with additional features such as a notion of quality

and duration. TÆMS shows that it is possible to define agents with task models, which is similar to what we present in this work. One of the differences between the method presented in this work and TÆMS is that TÆMS is specifically designed to model tasks for simulation and is limited in terms of extensibility. In contrast, our method is embedded in the Clean language offering greater flexibility. We provide a foundation for agents, which can be extended at will with concepts such as quality and duration. One would have to alter the core of TÆMS to add features to agents.

Another difference with iTasks is that TÆMS is not meant to support interactions with actual humans. iTasks supports human-in-the-loop simulations, where humans can take over roles of software agents. This makes iTasks more suitable for training scenarios. Task models in iTasks were originally designed to be executed by actual humans, by using advanced technology to automatically construct user interfaces from task specifications.

## 3.5 Brahms

Brahms [36] [4] is an agent-based simulation tool that combines comprehensive models involving human-machine interaction, collaboration, cognitive modelling, traditional business process modelling and more. Brahms is used to model *work practice*, groups of people using resources and collaboration to perform work. The Brahms developers use the following definition for work practice:

> The collective performance of contextually situated activities of a group of people who coordinate, cooperate and collaborate while performing these activities synchronously or asynchronously, making use of knowledge previously gained through experiences in performing similar activities.

Brahms uses a form of BDI style agents, where each agent executes activities based on local beliefs. Brahms is capable of simulating complex interactions between agents and their environment, such as agent interactions with physical devices and social interactions. Results of simulations can be used to derive work flow diagrams that indicate how goals were achieved by agents. These workflows can then be used to adjust the agent model or alter requirements. The Brahms architecture is structured around the following constructs:

Brahms distinguishes two types of knowledge: declarative and procedural. Declarative knowledge is captured in terms of beliefs of agents. A belief is a predicate in first-order logic, an assertion that is perceived to be true by an agent. Procedural knowledge is captured in workframes and thoughtframes, expressed with rule-like logic. Thoughtframes update the belief set of an agent, while workframes perform actual activities in an environment. Both types of frames are guarded by preconditions: if a precondition of a rule matches against the belief set of an agent, it is put on a queue to be processed.

```
GROUPS are composed of
  AGENTS having
    BELIEFS and doing
  ACTIVITIES executed by
    WORKFRAMES defined by
        PRECONDITIONS, matching agents beliefs
        PRIMITIVE ACTIVITIES
        COMPOSITE ACTIVITIES, decomposing the activity
        DETECTABLES, including INTERUPTS, IMPASSES
        CONSEQUENCES, creating new beliefs and/or facts
    DELIBERATION implemented with
      THOUGHTFRAMES defined by
      PRECONDITIONS, matching agents beliefs
      CONSEQUENCES, creating new beliefs
```

Figure 6: Brahms taxonomy

An interesting similarity between Brahms and iTasks agents is the use of explicit *detectables* and *sensing actions*. With Brahms, agents can explicitly observe the environment and update their belief sets accordingly. This explicit nature is also used in iTasks agents, where information about the environment (a task model) must be observed explicitly with `readInformation` or derived combinators. iTasks agents can react upon these changes to formulate new beliefs and start new tasks.

Since Brahms enjoyed over a decade of extensive research and development, its features are far more extensive than what we propose in this work. However, there are some conceptual differences similar to the other tools we have discussed. A large difference is that we model tasks that need to be performed by teams in one central task model. This task model is agnostic with respect to simulation of agents; it does not describe any properties related to individuals (e.g. cognitive attributes, capabilities). From this central task model, the iTasks platform derives a fully functional application which can be used immediately. This yields an executable, formal specification of tasks that need to be performed in a domain.

It is interesting to reason about the amount of autonomy of agents in both Brahms and iTasks. Franklin et al [8] defines an *autonomous agent* as follows:

> An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

For both iTasks agents and Brahms agents, the environment seems to be the 'main' limitation to autonomy. iTasks agents are required to follow tasks according to a task model, but are free to decide *how* and *when* a task should be performed. They are also capable of social interactions as with Brahms agents. With Brahms agents, they are limited to what actions can be performed in an environment. Arguably, a Brahms environment is far more extensive than an iTasks task model. For example, a Brahms environment is situated in space and time while an iTasks task model is not. For iTasks agents, there is no built-in notion of movement or geography.

### 3.5.1 KAoS integration

KAoS [39] [2] is a set of policy and domain services that can be used to define *policies*. According to the KAoS developers, a policy is a statement enabling or constrained some action that can be performed by an actor in a particular situation. In the context of software agents, KAoS can be used to constrain the capabilities of agents to make sure agents behave according to the intended purpose. In addition, it can be used to model team-based policies as well as individual policies.

KAoS can be used to define policies for agents, while remaining agnostic about the agent technology that is used. This way, one can reuse the same policies for multiple agent-based frameworks and other distributed systems. Efforts were made to bridge KAoS with Brahms [35] [37], such that Brahms agents can make use of KAoS policies. In order to preserve autonomy of agents, agents are not required to follow KAoS policies, but agents have the option to do so. Note that KAoS does not rely on any implementation details of agents; the KAoS services can be supported by any agent framework.

*Domain services* are used to support the abstract notion of a domain: they can be used to structure agents into groups, ranging from simple teams to complex organizational structures. Such a domain can be used to generalize policies for agents (i.e. domain specific policies). *Policy services* can be used to specify and enforce policies within domains. Part of the service is conflict resolution, resolving contradicting policy issues. An *actor* uses a policy service to learn the policies applicable to the current situation. Often times, such an actor is an agent in some agent framework. An *enforces* intercepts actors to enforce policies.

Brahms agents resemble agents in iTasks more strongly with the Brahms/KAoS bridge: KAoS can be seen as a central coordination model, a service that dictates what actions are allowed in the current context. Using this bridge, it is possible to define a central model for Brahms agents, such that individual agent tasks can be defined (or constrained) in a centralized, more team-based manner. This is similar to the central task model we have in iTasks. The two methods will also have similar challenges. For example, should a particular behaviour or task

be defined in the agent model or in the central model?

iTasks agents could also benefit from a KAoS integration. The KAoS services can be used to assist in agent reasoning. For example, KAoS can be used to reason about the actions an agent can perform in the future, based on the policies that are currently enforced. Furthermore, one of the interesting parts of KAoS is the modular architecture and the modules that it currently supports. One of these modules is *spatial reasoning*, in which policies are also situated in space (and time). Since we do not yet support tasks models situated in space, we could potentially use KAoS as alternative if we can build a KAoS-iTasks agent bridge.

# 4 An iTasks agent framework

## 4.1 Introduction

We build software with iTasks by specifying task models that capture work performed by humans. iTasks reduces the amount of time required to produce prototypes by automatically deriving tedious development work from task specifications. This allows us to test requirements specifications early with user experiments. However, it is time consuming to arrange the people necessary to perform these experiments, especially in complex domains such as navy vessels. Furthermore, in training scenarios, it should be possible to train individuals without the need of complete human teams. In order to solve both of these issues, we need a way to simulate human behaviour for iTasks.

An iTasks user is working on an instance (i.e. a personal view) of the task model, similar to instantiation of classes to objects in object oriented languages. Each user is associated with an instance and performs work by providing the iTasks server with input (e.g. keystrokes and mouse clicks). Each time input is received, the iTasks server modifies the instance and responds by presenting the user with a visualization of the new instance. The user then has the opportunity to observe the changes and can decide what input to supply next. If we would like to model human users or systems with software in the form of software agents, we need a mechanism that supports interpreting task representations. Furthermore, the mechanism needs to support producing input to tasks.

Agents defined for iTasks can interact in various ways. One way is to model inter-agent communication in the task model. For example, it is possible to model chat-like functionality as tasks that are performed in parallel with other tasks that agents might have. The communication method is then defined in the central task model. Another way is to let agents communicate using primitives that are defined in the agent framework. Messaging and similar forms are commonly used in existing agent frameworks. In our work, the agent framework

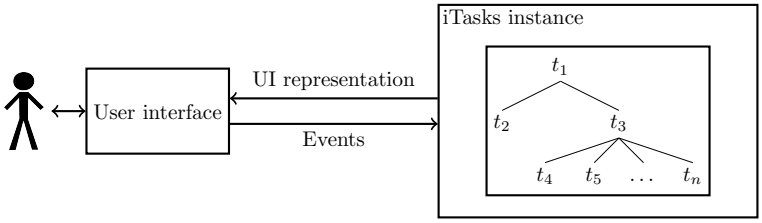


Figure 7: User-iTasks interaction

is implemented as extension to the iTasks platform. Inter-agent messaging can be implemented by using the shares provided by iTasks.

Agent frameworks often support modelling of cognitive and social attributes. The agent framework we propose in this work does not include these features, as it is merely the foundation of defining agents with iTasks. However, we do show that concepts such as costs for performing tasks can be defined with little effort in our case study. This shows the extensibility of our approach: since we define agents with iTasks in a general purpose language, we can add functionality without having to modify a complex code base.

## 4.2 Architecture

Users perform tasks by supplying *events* to an iTasks instance. An event is produced by keystrokes, mouse clicks and any other form of input. The iTasks instance then processes the event and produces a user interface representation of the tasks that are being performed. A user then interprets the new information displayed on the screen, making a decision to produce new events to continue working on tasks.

We want to be able to model *task performers* that perform tasks defined with iTasks. We refer to these as *agents*, where each agent is a separate entity capable of observing, reasoning and performing actions to complete tasks. With iTasks, tasks are defined in a single task model that describes the constraints and order of tasks to be performed. This means that agents are only capable of performing tasks that are assigned to them as defined by the central task model. The level of atonomy of each agent is thus dependent on how strict the task model is defined.

The architecture as shown in figure 11 applies to agents as well. A small difference is that agents do not require the complex UI representation of tasks, but rather a minimal version that includes task tags to be able to distinguish tasks.

An important observation is that **we consider observing, reasoning and performing actions to complete tasks as tasks**. This has the implication

that we can define agents in iTasks, exploiting the full potential of the iTasks framework. Agents can use shared data to communicate with other agents or use interaction tasks to influence their behaviour. Each agent can be considered as a single iTasks instance with a task model, enabling the option of simulating agents with agents.

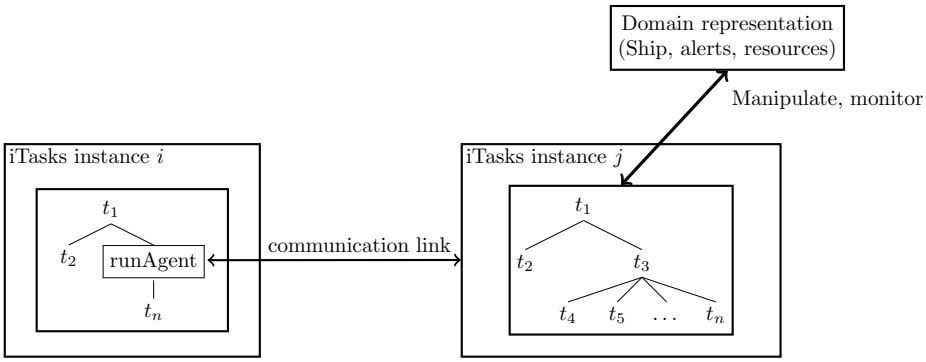


Figure 8: Overview of architecture

We introduce a new iTasks primitive `runAgent` that runs a simulation task representing an agent. The `runAgent` task is connected to some source providing a task model. In figure 8, an agent task is running in a separate iTasks instance $i$ and is connected to iTasks instance $j$. The agent task is performing tasks that are defined in $j$. Note that an agent task and source can run in the same iTasks instance ($i = j$). Also note that $i$ itself is an iTasks instance, allowing us to define some instance $k$ containing agents to work on tasks in $i$.

## 4.3 Dualism of tasks and agents

A dualism exists between how we define tasks in iTasks and how we model agents. The idea is that a task consumes *events* produced by agents and produces a value in addition to some intermediate representation of itself. We take the dual of this function to model agents: given some representation of tasks at hand and their values, produce events to perform work. Conceptually, we have the following:

$$Tasks \equiv Event \times State \rightarrow Value \times TaskRep \times State$$
$$Agents \equiv Value \times TaskRep \times State \rightarrow Event \times State$$

By extending the notion of *Event* and *State*, we can model *Agents* as a subset of *Tasks*. Because of the dualism between the two domains, operations on *Tasks* are also useful for *Agents*. For example, the semantics of the `parallel` combinator for *Tasks* can be used for *Agents* to simulate performing two tasks

in parallel. We define new primitives for *Agents* that are the dual of primitives for *Tasks*. For instance, we have `enterInformation` for *Tasks* to interactively obtain input from users (agents). This task consumes events to update and produce a value representing the information. We define the dual of this function, `writeInformation`, to produce events in order to update the value of a `enterInformation` task. `writeInformation` completes when the corresponding `enterInformation` task is available and the events are produced.

Now that this dualism is acknowledged, we could exploit some of its properties in various ways. We discuss some of these ideas informally. For example, agent definitions could be (partially) derived from task specifications. Furthermore, we could define some notion of agent soundness: if tasks (activities) in agent $a$ are defined satisfying duality with the tasks in model $b$, we could say that agent $a$ is sound with respect to task model $b$. Similarly, if tasks in agent $a$ cover all tasks defined in model $b$, we could say that agent $a$ is complete with respect to task model $b$. These properties could assist in agent development. For instance, a soundness check could be performed by static analysis during development of agents. However, it remains to be seen whether these properties can be verified automatically. Verifying these properties is presumably undecidable or extremely complicated due to the fact that iTasks is nested in a general purpose language, allowing all sorts of highly dynamic task models. Future research could indicate what subset of iTasks programs are verifiable.

## 4.4 Extensions to iTasks

In order to support agent-based simulations with iTasks, the iTasks platform is extended in various ways. These extensions are not just useful for the agent framework presented in this work; it also allows us to create connectors to third party frameworks such as Brahms.

### 4.4.1 Extended service representation

We define a new type `TaskServiceRep` as task representation for *service-based* access to iTasks. With service-based access, we mean access by (agent) software with the intent to perform tasks in iTasks. A variant of this type exists in the original iTasks implementation, but is only partially implemented and does not produce all the information required for services. For example, we require that a tag is included for services to differentiate between tasks. `TaskServiceRep` replaces the original `TaskPart` type in iTasks. The type represents all running interaction tasks in an iTasks session. Note that an iTasks session represents one user that is performing tasks, so each user has a different service representation as well as a different user interface. Continuation actions that are grouped by the iTasks platform are also included in the structure. Each interaction task produces this structure as result to serve as an alternative to the (more complex) GUI representation.

```
//Task representation for web service format
:: TaskRepKind = ActionRep       (String, Bool)
               | EditorRep       JSONNode

//Task representation for web service format
:: TaskServiceRep = TaskServiceRep TaskId (Maybe JSONNode) TaskRepKind

:: ServiceRep :== [TaskServiceRep]
```

The `JSONNode` type in the `EditorRep` construct represents any type that is produced by an interaction task. In the iTasks implementation, interaction tasks provide the option to use *views* to alter the information that is shown to users in the UI. The service representation that we implemented will always contain this *view* of information, since we do not want to discriminate between users that perceive information through the UI and extensions that interpret information from the service representation. However, we also would like that the information in the service representation is free from any UI related values. In order to implement this, we had to change the implementation of interaction tasks slightly. We show the implementation of `viewSharedInformation` as an example:

```
viewSharedInformation :: !d ![ViewOption r] !(ReadWriteShared r w) → Task r | descr d & iTask r
viewSharedInformation d [ViewWith tof] shared
    = interactSharedInformation d (toReadOnly shared) (λr → Display (tof r))
      <<# tof //new : change service representation

//alter the editor value in the service representation for editor tasks
(<<#) infixl 1 :: (Task a) (a→b) → Task a | iTask a & JSONEncode{|*|} b
```

The value that is shown to the user in `viewSharedInformation` is a `Display`, which is UI related information. It informs the iTasks platform to generate a read-only user interface element. We strip the `Display` constructor in the service representation with the `<<#` combinator.

### 4.4.2 Annotating tasks with tags

The *tag* part of `TaskServiceRep` is another extension to iTasks: tasks can be *tagged* in order to identify them in applications that consume the iTasks service. Note that this extension is not just useful for agents, but also for other applications that must interface with iTasks. We introduce two new operators to the iTasks platform:

```
class Tag a
    | JSONEncode{|*|}
    , JSONDecode{|*|} a

//Tag tasks with identifiers
(<<:) infixl 2 :: (Task a) t → Task a | iTask a & Tag t
(>>:) infixl 2 :: t (Task a) → Task a | iTask a & Tag t
```

Tags can be of any type, with the restriction that the type can be encoded and decoded to the JSON format. JSON is a generic representation commonly used in web development. JSON encoding and decoding is supported by almost all common development platforms. Tagging is especially useful when task models are highly dynamic (e.g. tasks are allocated dynamically at runtime, executed in parallel, etc.). Using tags, agents can easily identify what tasks are available to them.

### 4.4.3 Changes for agent framework

We now introduce changes to the iTasks platform specifically to support our agent framework. As explained before, an agent can be defined as the dual of a task. However, by extending `IWorld` and `Event`, we can express agents as tasks:

```
:: Event    = EditEvent     !TaskId !String !JSONNode
            | ActionEvent   !TaskId !String
            | FocusEvent    !TaskId
            | AgentEvent    !ServiceRep //Task representations as input
            | RefreshEvent

:: *IWorld =
    {
    ... //many other fields
    , eventTarget :: Maybe (Event *IWorld → *(ServiceRep, *IWorld))
    }
```

The idea is that the duality is encapsulated in the existing types. The input for agent tasks (task representations) is encoded into the `Event` type. When agent tasks produce events, they do so implicitly by using the the `eventTarget` function in `IWorld`. We now introduce a new core combinator for iTasks, `runAgent`:

```
:: TaskRepShare :== ReadOnlyShared ServiceRep
:: EventConsumer :== Event *IWorld → *(MaybeErrorString ServiceRep, *IWorld)

runAgent :: TaskRepShare EventConsumer (Task a) → Task a | iTask a
```

`runAgent` encapsulates a task that represents an agent. It takes a share that contains a task representation to be performed by the agent, a function to consume events and a task that represents the agent. It essentially implements the observer pattern by monitoring the share and running the agent when a new task representation is available. The share can represent any data source, normally being an iTasks instance.

We now introduce functions to connect agents to an iTasks instance. The function `webServiceTaskProducer` takes an URL, a port, an interval and a `SessionId` to periodically poll an external iTasks instance for available tasks. The `webServiceEventConsumer`

function takes similar arguments and sends events to an iTasks instance. We illustrate the usage of these functions with a simple example in section 4.7.

```
:: URL :== String
:: Port :== Int

webServiceTaskProducer  :: URL Port Int SessionId → TaskRepShare
webServiceEventConsumer :: URL Port SessionId Event → Task Void

getSessionId :: URL Port → Task SessionId
```

Recently, the iTasks framework introduced push-based events. A push-based event model is far more efficient than the polling mechanism that we use currently. The idea of push-based events in iTasks is that the iTasks server provides a notification mechanism when new information is available for clients. Instead of continuously checking whether new information is available (polling), the server notifies the client when new data is available. We do not use this functionality in the current version of the agent framework, but there is no reason why it could not be implemented in the future.

## 4.5 Semantics of agents

In this section, we semantically define the concepts of agents in iTasks. We use and extend the semantics defined in [28] with Clean∗. We give the semantics of the `runAgent` function along with `matchAgentEvent` and `doEvent`. Furthermore, we show how tasks are tagged. As in [28], we used framed boxes to indicate semantics and unframed boxes to indicate the actual API. In section 4.7, we give a short example of agents performing some tasks.

### 4.5.1 Events and states

To express our semantics, we extend `Event` and `EditorResponse`. We add a new event `AgentEvent` that includes the task representations to be processed by agents. In addition, we include a tag field in the existing `EditorResponse` type. We only allow editor tasks to be tagged in our semantics, keeping most of the existing semantics in tact. We also add a field `invalidateAgent` to `State`; we explain the purpose of this field in section 4.5.4.

```
1 :: Event =  RefreshEvent
2             | EditEvent TaskNo Dynamic
3             | ActionEvent TaskNo Action
4             | AgentEvent Responses //new: task responses as input
5
6 :: EditorResponse =
```

```
    { description   :: String
    , editValue     :: EditValue
    , editing       :: EditMode
    , tag           :: Maybe String //new: tags for editors
    }

:: *State =
    { taskNo            :: TaskNo
    , timeStamp         :: TimeStamp
    , mem               :: [Dynamic]
    , world             :: *World
    , invalidateAgent   :: Bool //new: flag whether agent task tree is invalid
    }
```

### 4.5.2 Tagging tasks

We introduce two operators to tag tasks:

```
(<<:) infixl 2 :: (Task a) String → Task a | iTask a
(>>:) infixl 2 :: String (Task a) → Task a | iTask a
```

We use the function `tagEditor` to alter the response produced by a task. If the response is an `EditorResponse`, we update the *tag* field.

```
tagEditor :: String → (Task a) → Task a | iTask a
tagEditor tag tsk = newTask eval
where
    eval tn t ev st
        #(Reduct res nTsk, rspTsk, st) = tsk ev st
        #rspTsk = case rspTsk of
                    [(t, EditorResponse e)]
                          = [(t, EditorResponse {e& tag = Just tag})]
                    rsp   = rsp
        = (Reduct res nTsk, rspTsk, st)

(<<:) infixl 2 :: (Task a) → String → Task a | iTask a
(<<:) t tag = tagEditor tag t

(>>:) infixl 2 :: String → (Task a) → Task a | iTask a
(>>:) tag t = tagEditor tag t
```

### 4.5.3 Running agents

We will now define the semantics of the `runAgent` function that is responsible for running an agent. `runAgent` takes an argument representing an agent ($a$) and an

argument representing the task to be executed by the agent ($b$). When an event is available, it is passed to $b$. We also pass the event to $a$, since $a$ can contain interaction tasks. After that, we collect the task representation from $b$ and pass it as an `AgentEvent` to $a$, so that $a$ is able to produce events. When $a$ produces events, they are added to the event queue of iTasks, effectively creating a loop of $a$ producing events to perform $b$. We define this seemingly complex mechanism as follows:

```
1  runAgent :: (Task a) → (Task b) → Task b | iTask a & iTask b
2  runAgent agent tsk = newTask eval
3  where
4      eval tn t ev st=:{invalidateAgent=ia}
5          #(Reduct res nTsk, rspTsk, st)  = tsk ev st
6          | isExcRes res                  = (Reduct res (runAgent agent nTsk), [], st)
7          #(Reduct _ nAgt, _, st)         = agent ev st
8          #(Reduct _ nAgt, rspAgt, st)    = nAgt (AgentEvent rspTsk)
9                                                {st&invalidateAgent=False}
10         = (Reduct res (runAgent nAgt nTsk), rspTsk ++ rspAgt,
11                 {st&invalidateAgent=ia})
```

For simplicity, we assume that exceptions are handled within agents and do not need to 'bubble up' in the task tree. We update `invalidateAgent` in order to indicate that we have a fresh task representation for the agent; the reasoning behind this choice is explained in chapter 4.5.4.

Note that the type of `runAgent` in our semantics is different from the type in the actual implementation discussed earlier. The only difference is the source of the task representations and the output of the events: in the actual implementation, we abstract from the source of task representations and the output of events. In our semantics, we simplify this by explicitly specifying the task to be executed by the agent. The events produced by agents are directly placed in the iTasks queue instead of using an abstraction. Other than a difference in the way input/output is coordinated, the implementation corresponds to the semantics as defined above.

### 4.5.4 Agent primitives

We will now define the semantics of the primitives `takeAction`, `writeInformation` and `readInformation`.

```
1  takeAction          :: String   → Task Void
2  readInformation     :: String   → Task a    | iTask a
3  writeInformation    :: String a → Task Void | iTask a
```

We first introduce a few helper functions to define these primitives. `doEvent` is a task that places an event in the event queue of iTasks. This is used by agents to produce events. We use `matchAgentEvent` to perform a match on an `AgentEvent`. The

task result is the result of the pattern match. `matchEditor` and `matchAction` produce representations when a tag or action name matches respectively. For simplicity, we assume that all actions have some unique name in our semantics.

```
doEvent :: Event → Task Void
doEvent ev = newTask eval
where
    eval tn t evt st=:{world}
        #world = queueEvent ev world
        = (Reduct (ValRes tn (Val Void Stable)) (doEvent ev), [], 
                {st&world=world, invalidateAgent=True})

matchAgentEvent :: (Responses → Maybe a) → Task a
matchAgentEvent p = newTask eval
where
    eval tn t (AgentEvent tsr) st=:{invalidateAgent = False}
         = case (p tsr) of
           Nothing = (Reduct (ExcRes AgentException) (matchAgentEvent p), [], st)
           Just x  = (Reduct (ValRes t (Val x Stable)) (matchAgentEvent p), [], st)
    eval tn t e st = (Reduct (ValRes t NoVal) (matchAgentEvent p), [], st)

findEditorTask :: String → Responses → Maybe (TaskNo, EditorResponse)
findEditorTask tag rsp
        = case (filter (eq tag) rsp) of
          [(t, EditorResponse e):_] = Just (t, e)
          _                         = Nothing
where
    eq tag (_, r) = case r of
                    EditorResponse e =
                        case e.tag of
                        Just t  = t==tag
                        Nothing = False
                    _ = False

findAction :: String → Responses → Maybe (TaskNo, (Action, Bool))
findAction action rsp
    = case (filter hasAction rsp) of
      [(t, ActionResponse as):_] = Just (t, (hd (filter pred as)))
      _     = Nothing
where
    pred (Action t, _) = t==action
    hasAction (_, ActionResponse as) = any pred as

matchEditor :: String → Task (TaskNo, EditorResponse)
matchEditor tag = matchAgentEvent (findEditorTask tag)

matchAction :: String → Task (TaskNo, (Action, Bool))
matchAction tag = matchAgentEvent (findAction tag)
```

Note that the `invalidateAgent` flag is set by `doEvent`. This indicates that the agent has invalidated the task representation that it is currently inspecting. In more clear terms, it means that the agent is no longer perceiving the current environment. The `invalidateAgent` machinery is in place to keep agents consistent when performing agent tasks sequentially. We explain this by a simple example:

```
someAgent = writeInformation "Number" 12 >>| readInformation "Number"
```

After the agent decides to enter a number in some task, one expects that reading the number from the same task would yield the same result. However, according to the iTasks semantics, `a >>| b` passes the same event to $a$ and $b$. This means that $b$ is inspecting the same task representation from `AgentEvent` as $a$, which would lead to inconsistency in agents. We use the simple `invalidateAgent` flag solution to overcome this issue. A better implementation could involve some changes to the core combinators of iTasks to alter the `AgentEvent` when needed.

`takeAction` can be used to take an action as defined with a step combinator ($>>*$). The task attempts to find an action with a particular action name. If the action exists and is enabled, an `ActionEvent` is immediately sent by using the `eventConsumer` function in `State`. `readInformation` and `writeInformation` use similar patterns.

```
takeAction :: String → Task Void
takeAction tag = matchAction tag
                 >>* [OnValue (λaction.
                               case action of
                               Val (_, (_, enabled)) _ = enabled
                               _                       = False)
                         λ(Val (taskId, (a, enabled)) _).
                          doEvent (ActionEvent taskId a)
                     ]

readInformation :: String → Task a | iTask a
readInformation tag = matchEditor tag
                      @ λ(_, edt).
                        (return o de_serialize o fst) edt.editValue

writeInformation :: String → a → Task Void | iTask a
writeInformation tag value = matchEditor tag
                             @ λ(taskNo, _).
                               doEvent (EditEvent taskNo (serialize value))
```

## 4.6 Towards automatically deriving agents

Now that we have a method to define agents with iTasks, we can explore how we can automatically derive basic agents from task specifications. Derived agents do

not perform any interesting reasoning nor follow any best strategies. They can be seen as *stubs* that can be expanded to include these features. Derived agents can also be used as simple test cases for task models. In order to demonstrate the possibility, we create a context free grammar representing a simple subset of the iTasks constructs. We acknowledge that this subset is limited. The type $T$ represents a task in iTasks, optionally associated with tag $\tau$. Both `writeInformation` and `takeAction` correspond to agent combinators that we defined earlier. The `writeInformation` combinator is parametrized with a tag $\tau$ to indicate the 'target' task. We assume that $d$ is a unique description for a step action involving user interaction. We inductively define the function $BasicAgent : T \to T$ over elements in the grammar that non-deterministically derives a simple agent from a given task specification.

$\langle T \rangle$ ::= `enterInformation`$_\tau$
| `updateInformation`$_\tau$
| `writeInformation` $\tau$
| `takeAction` d
| $\langle T \rangle$ `>>*` $\langle Step \rangle+$
| $\langle T \rangle$ `>>|` $\langle T \rangle$
| $\langle T \rangle$ `-||-` $\langle T \rangle$
| $\langle T \rangle$ `-&&-` $\langle T \rangle$

$\langle Step \rangle$ ::= `Action` d $\langle T \rangle$

$$
\begin{aligned}
&BasicAgent[\![\texttt{enterInformation}_\tau]\!] &&= \texttt{writeInformation}\ \tau \\
&BasicAgent[\![\texttt{updateInformation}_\tau]\!] &&= \texttt{writeInformation}\ \tau \\
&BasicAgent[\![t\ \texttt{>>*}\ \texttt{steps}]\!] &&= \left.\begin{array}{l} BasicAgent[\![t]\!] \\ \texttt{>>|}\ \texttt{takeAction}\ d \\ \texttt{>>|}\ BasicAgent[\![t']\!] \end{array}\right\} \text{if } \texttt{Action}\ d\ t' \in \texttt{steps} \\
&BasicAgent[\![t_1\ \texttt{>>|}\ t_2]\!] &&= BasicAgent[\![t_1]\!]\ \texttt{>>|}\ BasicAgent[\![t_2]\!] \\
&BasicAgent[\![t_1\ \texttt{-||-}\ t_2]\!] &&= \left\{\begin{array}{l} BasicAgent[\![t_1]\!] \\ BasicAgent[\![t_2]\!] \end{array}\right. \\
&BasicAgent[\![t_1\ \texttt{-\&\&-}\ t_2]\!] &&= BasicAgent[\![t_1]\!]\ \texttt{-\&\&-}\ BasicAgent[\![t_2]\!]
\end{aligned}
$$

An agent that is produced by *BasicAgent* simply writes information to the interaction tasks `enterInformation` and `updateInformation`. With the step combinator, *BasicAgent* picks a possible *step* non-deterministically. The step action is taken with `takeAction` followed by *BasicAgent* of $t'$. With the `-||-` combinator, one task must be completed. One of the two tasks is chosen by *BasicAgent* non-deterministically. For the `-&&-` combinator, the agent must complete both tasks.

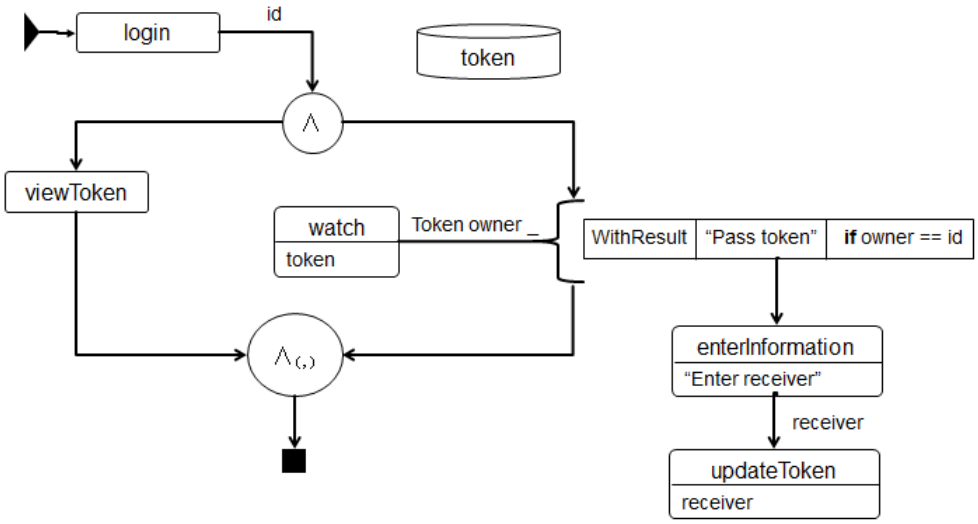


Figure 9: Abstract visualization of the task model

In practice, deriving agents from task specifications is constrained. For example, in order to perform tasks in a task model, some path condition must be satisfied in order to cover all tasks. An agent would need to produce input that satisfies this path condition, which is not always decidable.

## 4.7 An example

We show the interplay between agent tasks and a task specification with a small example. Assume that we have a token that is shared by a number of agents. The token is assigned to one agent at any time. The agent with ownership of the token can pass the token to another agent. We model the token as a share that is shared by all agents.

We first give the definition of the task model. The idea is that we tag each interaction task (strongly typed) such that agents can identify them. We define the `TokenTags` to serve this purpose. We define a function `viewToken` that shows information about who currently owns the token and a list of previous actions. We do so by using the `viewSharedInformation` combinator; it produces a task that presents information contained in a share. As the information in a share changes, the task will automatically update the user interface accordingly.

```
1 //Token is assigned to an agent (Int) and stores history
2 :: Token = Token Int [(Int, Int)]
3
4 //We use the following tags to identify the tasks in this task model
5 :: TokenTags = EnterId | EnterReceiver | PassToken
6
```

```
token :: Shared Token
token = sharedStore "token" (Token 1 [])

//View token information
viewToken :: Task Token
viewToken = viewSharedInformation "Token" [ViewWith view] token
where
    view (Token current history) =
            ("Agent " + toString current + " has the token. "
            ,map (λ(f, t). "Agent " + toString f + " gave agent "
                           + toString t + " the token") history
            )

//Enable the possibility of passing the token
handleToken :: Int → Task Token
handleToken myId =  watch token
                    >>* [WithResult (Action "Pass token" [])
                                    (λ(Token owner _). owner==myId)
                                    enterReceiver
                        ]
                    <<: PassToken
where
    //enter the receiving agent
    enterReceiver _ = enterInformation "Enter receiver" [] <<: EnterReceiver
                      >>= updateToken
    //update the token to contain the new owner and add history
    updateToken n   = update (λ(Token _ xs). Token n (xs++[(myId, n)])) token

login :: Task Int
login = enterInformation "Enter agent id" []
        <<: EnterId

Start :: *World → *World
Start world = startEngine [publish "/" WebApp (λ_→ task)] world
where
    task = login >>= λid. forever (viewToken -&&- handleToken id)
```

The `handleToken` function provides the opportunity to pass the token if the agent is the owner. This is expressed with the predicate `λ(Token owner _). owner==myId`. If the agent is indeed the owner, a receiving agent can be entered. This is expressed with the `enterInformation` combinator. Subsequently, the token is updated accordingly.

Next, we define the agents. We use the function `agentTask` to represent the agent behaviour. All agents are assigned an integer to uniquely identify them. The function `agentTask` takes such an identifier along with the identifier of an agent that will the receive the token once it is obtained. First, an agent will enter its identifier with the `writeInformation` combinator. Entering the identifier is completed by taking the "Continue" action with `takeAction`. Then, it will forever

pass the token to the receiving agent if possible. The combinator `takeAction` will only proceed if the action is enabled for the agent.

Note that there is a duality between the task model above and the agents we define below. We use `writeInformation` in the agents to update the `enterInformation` task in the task model. For every step combinator in the task model (`>>|`, `>>=` and ≫*), we use the appropriate `takeAction` combinator in the agents.

We will now show the agent specification:

```
1  //the task of the agent is to pass the token to some other agent
2  agentTask :: Int Int → Task Void
3  agentTask id target = writeInformation EnterId id
4                        >>| takeAction EnterId "Continue"
5                        >>| forever (
6                              takeAction PassToken "Pass token"
7                              >>| writeInformation EnterReceiver target
8                              >>| takeAction EnterReceiver "Continue"
9                             )
10
11 //Run 10 agents in parallel, each passing the token to each other
12 allAgents :: Task Void
13 allAgents = parallel Void
14                      (map (λ(id, target).
15                          (Embedded, λ_. runTokenAgent id target))
16                      (zip2 [1..9] [2..10] ++ [(10, 1)]))
17             @ const Void
18
19 //Start the iTasks engine with the allAgents task
20 Start :: *World → *World
21 Start world = startEngine [publish "/" WebApp (λ_→ allAgents)] world
22
23 //Run an agent with the runAgent combinator
24 runTokenAgent :: Int Int → Task Void
25 runTokenAgent id target = getSessionId server port
26                           >>= λsessionId.
27                               runAgent (taskProducer sessionId)
28                                        (eventConsumer sessionId)
29                                        (agentTask id target)
30 where
31     taskProducer  = webServiceTaskProducer  server port interval
32     eventConsumer = webServiceEventConsumer server port
33     server        = "localhost"
34     port          = 80
35     interval      = 2
```

# 5 Case study: damage control with iTasks

In this case study, we define a prototype in iTasks that models tasks performed by navy patrol vessel personnel in crisis situations. The goal of the case study is to show that we can define agents for complex task models using the method as presented in this work as proof of concept. Furthermore, we discuss how Cognitive Work Analysis (CWA) is used as context for requirements and how it relates to the implementation. We show how agents can be defined and how they are used to obtain results from simulations. We verify a number of claims about the prototype by analysing the results of the simulations.

It is important to consider that the purpose of this prototype is not to be deployed on a vessel for immediate use, but rather to demonstrate the expressiveness of iTasks along with the working of agents. Furthermore, the constructed task model serves as discussion for evolving requirements and claims. It yields something that is executable and forces domain experts and analysts to be exact about the tasks that must be supported by future information systems. To summarize, we can do the following with the formal task model:

- We can execute the task specification, forming discussion material for domain experts and decision makers. These discussions can lead to changes in the task model or requirements.
- We can define agents that simulate humans performing tasks. From simulations, we can gather results to verify claims about the task model and test the proposed manning and automation. These results can also serve as discussion material or can lead to changes in the task model.
- We can train the human crew of vessels by human-in-the-loop simulations.

## 5.1 Problem definition

The most important goal of operational maintenance on a patrol vessel is to preserve the *command aim*: the current mission of the vessel. Dealing with calamities (i.e. the *internal battle*) is required to achieve this goal. The command aim influences the way calamities are prioritized. When the command aim involves maintaining a certain velocity to intercept a hostile vessel, a fire in a compartment close to the engine room has a higher priority than a small leak in the messdeck.

Dealing with calamities requires communication among the various roles on a vessel. Communication is supported by many different devices. Devices include hand-helds, headsets, mobile phones, broadcasting and more. A problem common on these vessels is that communication is repeated throughout the chain of command: an order is given with a certain priority and written down by the first officer, who then communicates manually with a second officer responsible for a particular task (e.g. maintenance of combat systems) who in turn

communicates the order by headset to personnel on the ship. These communication links can be optimized by using digital communication, where each role is immediately aware of relevant data without the need for repetition. iTasks is especially suitable to define these sorts of models, in which complicated interactions between roles on a ship can be defined using tasks and (new) data is immediately available to those who need it.

The requirements of the prototype are based on decision ladders that are part of a CWA analysis. This CWA analysis was performed by TNO on a Dutch navy vessel and serves as a foundation for projects to increase efficiency of operational maintenance. We used this CWA analysis to investigate whether we can automatically derive iTasks applications from CWA products. It turned out that this is not possible, because of the lack of detail in these products. However, we did establish that CWA is suitable as *foundation*, a context that can be used to formulate requirements.

The decision ladder in figure 10 describes the general process of handling the internal battle on a patrol vessel. The goal of the ladder is described at the top of the figure: the command aim must be maintained at all times. The left hand side of the ladder describes detection and observation, while the right hand side describes planning and taking action. The top of the ladder describes prioritizing calamities which is a responsibility of the top of the command chain. The decision ladder is invoked by detection of an anomaly in the environment, which leads to an alert situation. The first step is to construct an internal picture of the anomaly (e.g. gather information using sensors) and determining the possible consequences to the command aim. Officers in command determine the priorities and allocate personnel to handle the calamity on the ship.

This case study involves the roles of damage officer (D-Officer), team leader and seaman. Agents with these roles need to cooperate in order to resolve calamities on a ship, such as fires, leaks and damaged systems. The D-Officer is responsible of coordinating team leaders and constructing plans to repair damage in the vessel. The D-Officer corresponds to the top level of the decision ladder: this role prioritizes calamities according to the command aim. In reality, more officers are concerned with prioritizing. In this case study, we limit this task to the D-Officer. Team leaders are responsible for assembling teams of seamen to attack fires and monitoring the attacking process. Seamen perform the labour work: they must repair leaks, fix systems and extinguish fires. In addition, seamen are responsible for performing *blanket searches*, searching the vessel for calamities and reporting them to the bridge. Seamen and team leaders are involved in both sides of the decision ladder: they detect calamities and repair damage on the vessel.

We obtained the requirements from interviews with domain experts; they were also the authors of the CWA analysis. We picked these requirements because of the relative complexity: they require communication and cooperation between different roles, they include a monitoring and resource allocation problem and the amount of requirements fitted the time available for the case study. The

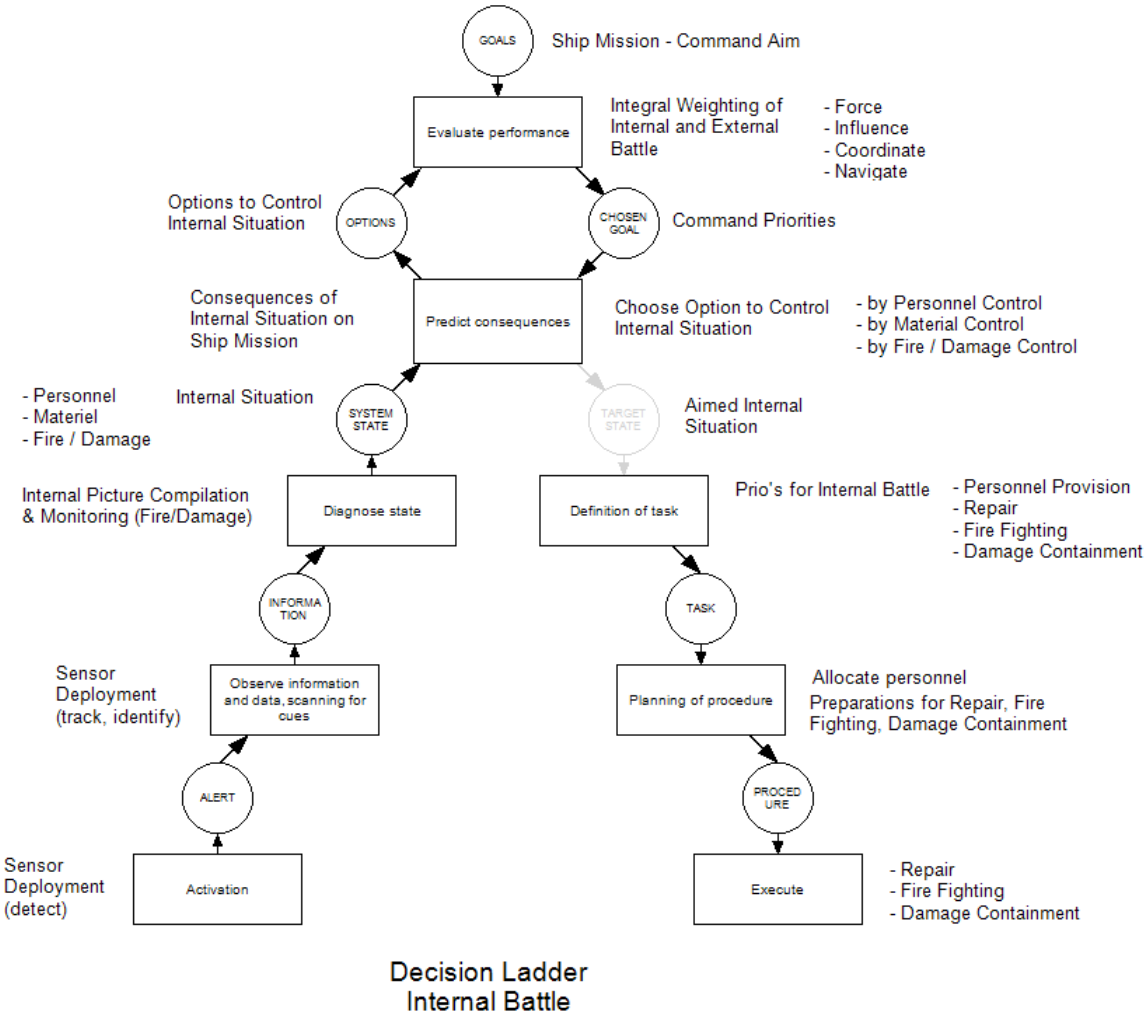


Figure 10: Decision ladder from [29]

decision ladder was used during interviews as the primary source of discussion. We experimented with different requirements and ideas to use iTasks. For instance, in earlier prototypes, we supported switching between automated and human sensor tasks. A sensor is an example of an automated (autonomous) task, while a blanket search is a sensor task performed by humans.

1. The system should enable the D-Officer to initiate and monitor blanket searches.
2. The system should enable the D-Officer to handle fire alerts by constructing attack plans containing the following details:

- An attack post from which the attack team initiates the attack.
- A team leader which will lead and coordinate the attack.
- An attack route from the attack post to the location of the fire.

3. The system should enable the D-Officer to monitor ongoing attacks of fires.
4. The system should enable the D-Officer to send engineers to repair damaged systems and leaks.
5. The system should enable the team leaders to (re-)allocate seamen to attack fires.
6. The system should enable the team leaders to monitor ongoing attacks of fires.
7. The system should enable the seamen to perform blanket searches, possibly reporting calamities.
8. The system should enable the seamen to assist fire teams.
9. The system should enable the seaman to repair leaks and damaged systems.
10. It must be possible to manage a scenario:
    - Simulating fires, leaks and damaged on a virtual ship.
    - Monitoring the activities performed by users in the prototype.

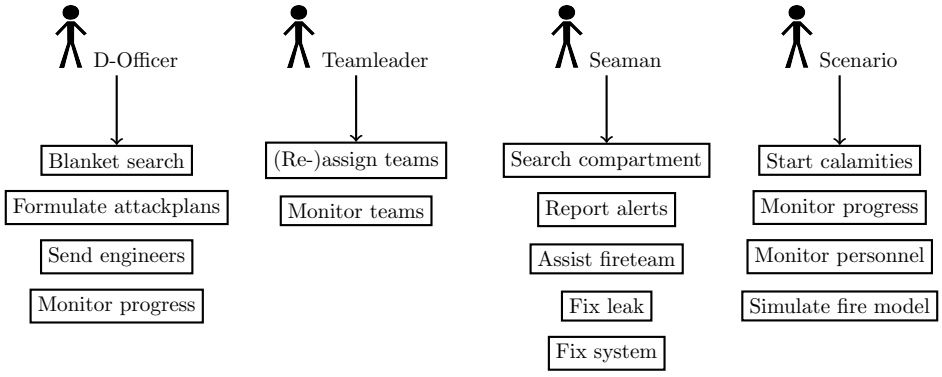


Figure 11: Overview of roles and tasks

## 5.2 Building the iTasks prototype

Given the described requirements, we designed a task model in iTasks. In this chapter, we discuss the implementation of the model.

### 5.2.1 Handling fire alerts

When a fire is reported by a seaman during a blanket search, the D-Officer should be able to respond by formulating an attackplan (requirement 2). This is defined as follows in the task model:

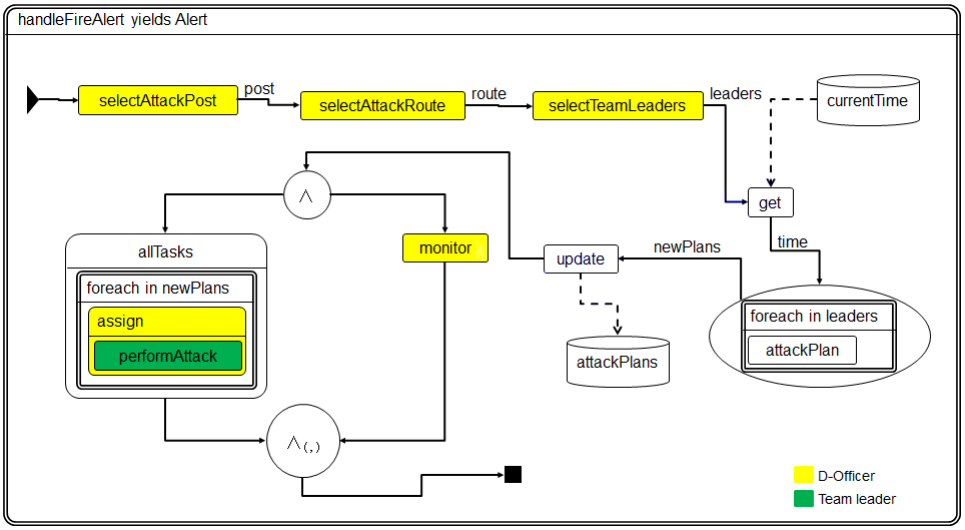


Figure 12: An abstracted GiN visualisation of handleFireAlert

```
:: AlertKind = Fire | Leak | DamagedSystem

:: Alert =
    { location      :: Compartment
    , description   :: String
    , sensor        :: Int
    , type          :: AlertKind
    , time          :: Time
    , uniqueId      :: Int
    }

handleFireAlert :: Alert → Task Void
handleFireAlert alert
    =               selectAttackPost alert
    >>= λpost.      selectAttackRoute alert post
    >>= λroute.     selectTeamLeaders alert
    >>= λleaders.   get currentTime
    >>= λtime.      return [attackPlan time leader post route
                                \\{ShipUserView|user=Hidden leader} ← leaders]
    >>= λnewPlans.  update (λplans. plans ++ newPlans) attackPlans
    >>| monitor ||- allTasks [assign (attackInfo plan.leader)
                                     (performAttack plan attackPlans)
                              \\plan ← newPlans
```

```
24                              ]
25      @ const Void
```

The D-Officer performs various tasks in sequence to formulate attack plans. The sequence is displayed in figure 12. First, an attackpost is selected from which the attack will commence. Then, a route is established from the attackpost to the location of the fire. After that, the team leader is selected. This results in a set of plans 'newPlans' that is stored in a share called 'attackPlans'. Each plan is assigned to a team leader with a task. Then, the D-Officer monitors the compartment (showing information about the current temperature and other information) and directs the team leaders to assemble teams and attack the fire according to the plans.

### 5.2.2 Starting blanket searches

A blanket search (requirement 1) can be seen as a task that produces a list of alerts (possibly empty) containing reported calamities by seamen during the search. The blanket search is divided amongst the available seamen on the ship. The D-Officer monitors the percentage of compartments that is searched. Note that the type of the function is quite general. A list of alerts can also be produced by sensors that automate the reporting process. In a more advanced prototype, the implementation of the function could use sensors as primary source of alerts and fall back on a blanket search whenever sensor systems have failed.

```
1 blanketSearch :: Task [Alert]
2 blanketSearch = get (shipUserViewsWithRole "seaman")
3                 >>= λusers. parallel "Blanket search"
4                                   [(Embedded, monitor):divideWork users]
5                 @? collect
```

The function `divideWork` divides the number of search tasks evenly amongst the available seamen, returning a list of delegated tasks each possibly producing an alert. The function `monitor` counts the number of results produced and shows the corresponding percentage. `collect` collects all the reported alerts.

### 5.2.3 Searching compartments

Seamen must be able to search compartments in a blanket search (requirement 7). A compartment search can be seen as a task that takes an representation of a compartment and possibly produces an alert:

```
1 :: CompartmentEnv =
2     { identifier    :: String
3     , smoke         :: Bool
```

```
    , temperature   :: (Int, [(Time, Int)]) //current temperature, previous temperature measures
    , systems       :: [System]
    , leak          :: Bool
    }

searchCompartment :: Compartment → Task (Maybe Alert)
searchCompartment compartment
    =   viewSharedInformation ("Sensor", "Report fire, leak or damaged systems")
                [ViewWith view] (getCompartmentEnv compartment)
    >>* actions compartment <<: CompartmentInfo compartmentId
where
    view            = toCompartmentView
    compartmentId   = compartment.Compartment.identifier
    actions c = [Always (Action "Report fire" [])    (alert Fire)
                ,Always (Action "Report leak" [])    (alert Leak)
                ,Always (Action "Report system damage" [])
                    (alert DamagedSystem)
                ,Always (Action "No observations" [])    (return Nothing)
                ]
```

The function `getCompartmentEnv` returns a 'live view' of a compartment, including the current temperature and whether the compartment is filled with smoke. The seaman can respond by reporting an alert or indicating that there are no calamities occurring. Note that we use the `<<:` operator to *tag* the task with information that is used by software agents defined later in this work.

### 5.2.4 Executing attack plans

Team leaders must be able to execute attack plans (requirement 5 and 6). We implemented this by giving the team leader a live view of both the attack plan (which can be updated when seamen retreat while extinguishing fires) and the compartment (current temperature). Parallel to this view, the team leader can assign a team of seamen. The team leader is responsible for ending a fire alert.

```
attackFire :: (Shared AttackPlan) → Task Void
attackFire plan
    =   get plan
        >>= λp. viewSharedInformation "Attackplan" [ViewWith view]
                    (plan |+| getCompartmentEnv p.AttackPlan.target)
        -&&-    assignAttackTeam plan
        >>*     [Always (Action "End fire alert" []) (return Void)]
        <<:     LeadFireTeam (compartmentId p)

assignAttackTeam :: (Shared AttackPlan) → Task [ShipUserView]
assignAttackTeam plan
    =   get plan
```

```
13      >>= λp. enterSharedMultipleChoice "Assign attackteam" []
14                  (shipUserViewsWithRole "seaman")
15      >>*     [WithResult (Action "Assign" []) (const True) (assignTasks p)]
16      <<:     AssignFireTeam (compartmentId p)
```

We use the `viewSharedInformation` combinator to produce a view of both the attack plan and the compartment. This view is produced by composing two shares with the `|+|` operator. With the step combinator `>>*`, we define an action to end the fire alert at any time. With the function `assignAttackTeam`, the team leader can assign a team by selecting available seamen with the `enterSharedMultipleChoice` combinator.

## 5.3 Using the iTasks agent framework

### 5.3.1 Introduction

We defined software agents for the role of D-Officer, team leader and seaman in the prototype. Using these agents, it is possible to perform simulations with iTasks without any human intervention. Later in this work, we present some results of simulations performed in the case study. The software agents also provide an opportunity to do human-in-the-loop experiments where humans and software agents cooperate to perform tasks. This is especially useful in training scenarios.

In this case study, we created some functionality to be able to simulate the physical costs of tasks performed by agents. We implemented this functionality by using the features available in the iTasks framework. This is an example of a reusable building block: it can be reused in development of future agents. In existing agent-based tools, it is difficult to implement these kinds of extensions, because agent languages often lack the constructs required to create abstractions. Since we define agents in Clean with iTasks, we can exploit the full power of all the functionality available. Note that we compare our method to languages specifically designed to define agents; we do not intend to comment on or disregard the abstraction and reuse capabilities of technologies such as Java.

### 5.3.2 Task costs

To demonstrate the extensibility of the agent framework, we added an element of realism by associating costs to certain tasks that agents perform. We do so by defining a new function that takes an agent task and a cost, producing a function that takes an initial agent state to construct a task that produces a new agent state. This resembles the inner belief state that agents maintain. In the world of iTasks, we can define agents by tasks that consume and produce agent states.

```
:: Fatigue  = Fatigue   Int
:: Health   = Health    Int

:: AgentState s =
    { fatigue   :: Fatigue
    , health    :: Health
    , id        :: String
    , state     :: s
    }

//Duration minSecs maxSecs
:: Duration = Duration Int Int | NoDuration

:: Costs = Costs Fatigue Health Duration

(<<-) infixl 2 :: (Task a) Costs → (AgentState s) → Task (AgentState s)
    | iTask a & iTask s
```

The agent state produced by `<<-` is updated each time interval, subtracting the costs from the supplied state. This simulates physical costs when performing tasks. Optionally, the time it takes to perform a certain task can be simulated by supplying a duration. We used this functionality to add realism to the software agent simulating the seaman role in the prototype.

### 5.3.3 Agent behaviour

The software agents we defined in this case study use various behaviours. We consider the term *behaviour* as a composition of agent tasks expressed in iTasks: observation, reasoning and acting using the agent (and iTasks) primitives we defined earlier. A behaviour is a program used to model a task performer working on tasks that are specified in the central iTasks model. Each agent can have a different behaviour and therefore a different manner of performing the same task.

A more complicated central task model that provides alternatives to achieve goals will allow for more freedom in agent behaviour, simply because there are many ways to perform a task. A simple task model that uses fixed workflows and procedures will often result in less complicated agents. However, agent developers are free to use as much complexity as desired.

Although it is possible to implement advanced behaviours involving (machine) learning and other interesting algorithms, we only implemented some basic behaviours. Most of the reasoning that the agents perform is straight-forward: favouring tasks with high priority, selecting the engineer with the least amount of assigned work to fix a leak, dividing seamen evenly amongst fires, etcetera. We will now discuss some of the implemented behaviours.

For both the seaman agent and the team leader agent, we use a *dashboard style* model in the central task model. This dashboard consists of a list of tasks to be performed together with an overview of tasks that are currently being performed. The agents need to explicitly accept tasks in the task list to start them. The agent decides when to work on a task and when not. For the seaman agent, we implemented the behaviour such that only one task is performed at a time. This is an example of a case where the agent is more restrictive than the central task model.

Although it is up to agents when to work on tasks, we sometimes need a mechanism in the central task model to be able to respond when an agent decides to pause or cancel a task, which would normally be an implicit action. We had to make this an explicit action in some cases. An example of this is agents extinguishing fires: since this is associated with a physical cost, they must be able to recover and thus pause the extinguishing task. The team leader needs to be notified in order to judge whether more personnel is required on the team. This is another example of continuously dividing tasks and functionality between agents and the central task model.

**Seaman agent** The seaman agent is a worker drone: it does not perform many interesting reasoning steps. The goal of the agent is simply to empty a priority queue of tasks that are assigned by the D-Officer and team leaders. The task assignment is specified in the central task model, while the seaman agent is responsible for performing these tasks. The agent is only capable of performing one task at a time, to mimic a real life situation where it is not possible to extinguish a fire and perform a blanket search at the same time. However, the central task model does allow seaman agents to perform multiple tasks at the same time. We made the choice to restrict the behaviour of the agent.

The implemented agent has a notion of task costs as defined earlier. These costs are agent-specific and are not defined in the central task model. Extinguishing fires and repairing leaks take time and impact the physical state of a human in the real world. As a result, the seaman agent needs to rest once in a while to recover from the physical impact, negatively impacting the time it takes to attack a fire. We modelled the option to retreat from extinguishing a fire in the central task model.

While the seaman agent is attacking a fire, it might be the case that a task with a higher priority is ordered by an officer. This might be required in a situation where a new calamity impacts the command aim. In this case, the agent leaves the fire team to accept the task with a higher priority. This feature is modelled in the agent, making use of the flexibility of the central task model.

**Team leader agent** The team leader agent is responsible for managing attack teams to extinguish fires. The agent receives its orders from the D-Officer. The team leader agent is capable of leading multiple teams at the same

time, requiring only one instance of this agent during a simulation, although the central task model allows for multiple team leader agents. The agent monitors the amount of fires it needs to extinguish, continuously reassigning teams to maintain a balanced amount of seaman for each attack. The central task model provides the option to reassign teams and choose team members, while the agent is responsible for actually choosing these teams and team members. Once the smoke in a compartment has disappeared and the temperature is at an acceptable level, the agent notifies the D-Officer and the men in the fire team to end the attack and stop the alert.

**D-Officer agent** The D-Officer has the responsibility of defining attack plans to extinguish fires. Furthermore, this role allocates engineers to fix damaged systems and leaks. The D-Officer maintains and monitors these tasks. Blanket searches are also initiated by this role. The D-Officer initiates the blanket search, but the central task model automatically divides the search tasks among the available seamen. This is a design choice: it is also possible to change the task model such that the D-Officer manually divides the tasks among available resources. The agent representing this role needs to handle each incoming alert reported by seamen. Thus, it handles an arbitrary amount of tasks in parallel at any time. When the alert constitutes a fire calamity, the officer formulates an attack plan by selecting the attack post followed by a team leader and attack route. When an alert describes a system failure or a leak, an engineer is allocated. The D-Officer chooses an engineer with the least amount of assigned tasks.

### 5.3.4 Implementation

**Searching compartments** The software agent representing the seaman role should be able to investigate compartments and report any abnormalities. As shown in 5.2.3, the agent is represented with a live view of the compartment that includes information about smoke development and the current temperature. The agent should decide what alerts should be reported based on this information.

```
1  inspectCompartment :: String → SeamanState → Task SeamanState
2  inspectCompartment compartment = inspect <<- Costs (Fatigue 5)
3                                                     (Health 0)
4                                                     (Duration 5 20)
5  where
6      compTag = CompartmentInfo compartment
7      inspect = readInformation compTag
8                >>= decide
9      decide comp
10     | comp.CompartmentView.leak     = takeAction compTag "Report leak"
11     | comp.CompartmentView.smoke    = takeAction compTag "Report fire"
12     | hasDamagedSystem comp
```

```
13          = takeAction compTag "Report damaged system"
14          = takeAction compTag "No observations"
15      hasDamagedSystem {CompartmentView|systems}
16          = any (λ{System|state}. case state of
17                                    SysDestroyed = True
18                                    _            = False) systems
```

We assign costs to the *inspect* task with the `<<-` operator we defined before. Inspecting a compartment takes between 5 to 20 seconds and negatively impacts the fatigue of the agent.

**Assisting fire teams** The seaman agent is also responsible for assisting fire teams in attacking fires. At any moment, the agent is capable of interrupting the assist by a "Pause" action. It is also capable of cancelling the assist through the "Stop" action. As explained in the strategy of the seaman agent, the agent will assist the team as long as *a*) it is physically capable of doing so and *b*) if no task with a higher priority is assigned to the agent.

```
1  assistFireTeam :: TaskView String SeamanState → Task SeamanState
2  assistFireTeam tv compartment state
3      =   determineBestTask -&&- assistTeam state
4      >>* [WhenValid (isTired o snd)
5                     (interruptTired o snd)
6          ,WhenValid (λ(tv‘, _). tv.TaskView.priority < tv‘.TaskView.priority)
7                     (interruptPriority o snd)
8          ,WhenStable (return o snd)
9          ,CatchAll (const (recover state)))
10         ]
11 where
12     compTag = CompartmentInfo compartment
13
14     assistTeam = readInformation compTag
15                  >>* [WhenValid (λcp.
16                                    cp.CompartmentView.temperature ≤ 20)
17                                 (λ_. takeAction compTag "Stop")
18                      ]
19                  <<- Costs (Fatigue 2) (Health 2) NoDuration
20
21     interruptTired s =  takeAction compTag "Pause"
22                         >>| recovery s
23
24     interruptPriority s = takeAction compTag "Pause" @ const s
```

We use the `-&&-` and the `>>*` to monitor both the assigned task list and the current physical state of the agent. If the agent is tired, we interrupt the assist with the "Pause" action and start a recovery process. The assist task is reassigned and picked up again when the agent is fully recovered. At the same time, the agent checks whether there is a task with a higher

priority than the current task. If this is the case, the agent interrupts the assist. While the agent is assisting, it is continuously monitoring the temperature in the compartment. The agent stops the assist when the temperature has reached an acceptable level.

An exception handler is in place to handle the situation when the agent attempts to perform an illegal action or read invalid information from the environment. This can happen when the team leader ends the fire alert: the task that is assigned to the seaman to assist the fire team is immediately removed from the active tasks. In that case, monitoring the compartment or taking some action is invalid, since the task with the particular tag (*compTag*) no longer exists. We handle this through the `CatchAll` handler.

**Handling alerts by the D-Officer** The D-Officer agents needs to continuously monitor and handle newly reported alerts.

```
1  //Continuously monitor and handle reported alerts
2  monitorAndHandleAlerts :: Task Void
3  monitorAndHandleAlerts =
4      repeatTask (
5          λknownAlerts. readOptions Alerts
6          >>* [WhenValid (any (λa. all ((=!=)a) alerts))
7                         (handleAlerts o newAlerts knownAlerts)]
8      ) (const False) []
9      @ const Void
10
11 newAlerts old new =  [a \\ a←new | all ((=!=)a) old]
12
13 //Handle new alerts
14 handleAlerts :: [AlertView] → Task [AlertView]
15 handleAlerts newAlerts
16     =  allTasks (map (λa. selectAlert a >>| handleAlert a) newAlerts)
17         @ const newAlerts
18
19 selectAlert :: Task Void
20 selectAlert a = chooseOption Alerts a >>| takeAction Alerts "Select"
21
22 handleAlert :: Task Void
23 handleAlert a=:{AlertView|type=Fire} = handleFire agentId a
24 handleAlert a=:{AlertView|type=Leak} = handleLeak agentId a
25 handleAlert a=:{AlertView|type=DamagedSystem} = handleDamage agentId a
```

The agent for the D-Officer performs this task by observing information from the task tagged with the tag `Alerts`. It does so by using the `readOptions` combinator, which is derived from the `readInformation` combinator:

```
//Read list of options from a task with some tag t
readOptions :: t → Task [v] | iTask v & Tag t
```

If the alert list contains any alerts that we did not handle before, we obtain the new alerts and handle them sequentially. We use the iTasks API function `repeatTask` to continue monitoring, keeping track of the list of alerts that we already handled:

```
//Iterate a task as long as a predicate is not valid.
repeatTask :: (a → Task a) (a → Bool) a → Task a | iTask a
```

Then, for each newly reported alert, the agent selects the appropriate task with the `handleAlert` function. Handling all new alerts is composed into a new task with the API function `allTasks`.

## 5.4 Performing simulations

To demonstrate the possibility of simulation with software agents in iTasks, we have set up a simple experiment with the prototype. We would like to validate the claim that 20 agents can extinguish two fires in under five minutes. This is a simple example of a claim that could be part of the requirements of a software system. We have also tested the claim for 5, 10 and 15 agents. Results of these simulations are not meant to be an accurate estimate of reality. However, they can be used to measure the effects of reducing the amount of agents or changes to the task model. At the very least, results can be used as discussion material for domain experts, possibly altering requirements or claims.

In our prototype, agents performing various roles need to cooperate in order to extinguish fires. The D-Officer needs to call for a blanket search, a seaman must report the fire, the D-Officer formulates an attack plan, etcetera. We would like to measure the effects (in terms of time costs) of reducing the amount of seamen. In order to do so, we added logging functionality to software agents. This allows us to collect and measure detailed information about the actions and reasoning that agents perform.

We have conducted the experiment as follows: we used a tool to perform various simulation runs using different amounts of seamen and fires. The fires are started at fixed points in the ship. We measure the amount of time taken from the initial blanket search order to the last 'end fire' call by the team leader. In some cases, the agents are not capable of extinguishing the fire(s), due to lack of agents. In that case, the fire spreads faster than the agents can deal with.

We defined a very basic model to simulate temperature development of fires in compartments, where the amount of agents currently attacking in the compartment impacts the speed in which the temperature drops:

$$t_i = t_{i-1} + 5 - 3.5n$$

The temperature $t$ is updated with each time interval $i$, considering the amount of agents $n$ that are currently attacking the fire. Of course, this model does not

have real-world accuracy, but it does take into account agents retreating and attacking fires repeatedly. The model can be refined easily if more accuracy is required.

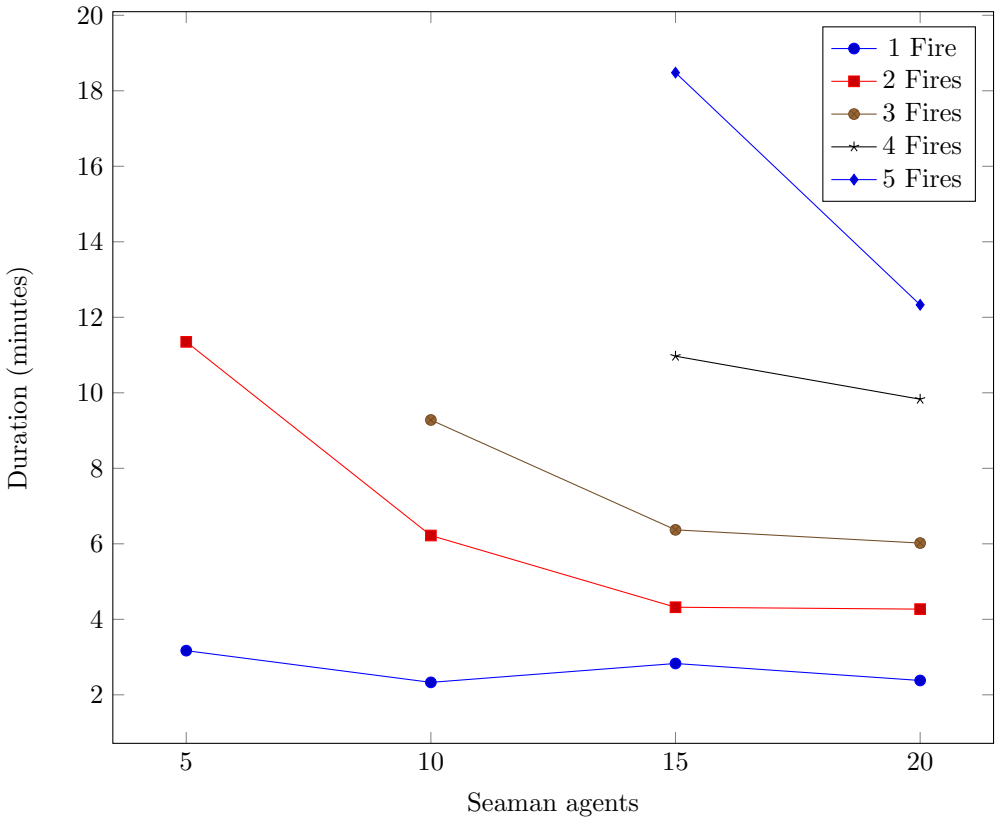


Figure 13: Time required to extinguish $n$ fires with $m$ seamen

Using these results, we can verify that 20 agents can extinguish 2 fires in under 5 minutes (4.27 to be precise). From these simulations, the results also show that 5 agents are only capable of attacking two fires at most. Furthermore, 10 agents can resolve at most 3 fires at the same time.

Based on the simulation results, we discovered a bottleneck in the task model regarding blanket searches and the strategy of the team leader agent. When a fire is discovered, the team leader immediately assigns all available seamen to resolve the fire as soon as possible. However, seamen are also needed to perform the blanket search. If the blanket search is terminated because the seamen are all ordered to attack a fire, other fires do not get discovered. This is a typical manning issue. One possible solution to this issue is to change the task model to allow delegation of tasks by seamen. In that case, we could let seamen delegate blanket search tasks to seamen who are not assigned high priority tasks. If

we then alter the strategy of the team leader agent to only assign a subset of available men, we could solve this issue.

One conclusion of these results could be that they are too inaccurate. In that case, it is possible to introduce a more complex temperature model. However, even with a simple model, we can already show the impact of increasing or decreasing the amount of agents with little effort.

## 5.5 Discussion

We discovered that the iTasks workflow specification language (WSL) was powerful enough to capture the requirements in our case study. Combined with some additional primitives, the WSL provides a suitable framework to define agents. We are able to express basic agents that reactively act based on changes in the environment (the task model). In addition, we used iTasks constructs to express physical costs when performing tasks and agent tiredness, showing the extensibility of the framework.

During development of the agent framework and agents for the prototype, we also observed several limitations. We discuss these limitations in this section.

**Real-time versus virtual-time simulations** Currently, the agent framework does not support simulations with virtual time. This feature would introduce a virtual clock, such that time costs can be simulated. We cannot do this by introducing a new combinator or expanding the `<<-` operator defined earlier, because time is a concept in both the central task model and the agent model. Awaiting some period of time should affect other tasks that are performed subsequently. The `<<-` combinator introduced earlier actually awaits the duration that is supplied. This increases the amount of time that simulations take, negatively impacting the benefits of computer simulations.

**No unified logging mechanism** In order to evaluate simulations, there must be a way to record all agent activity. We used a simple logging mechanism that writes some information annotated with a time stamp to a text file. However, it required us to clutter agents with logging expressions, making them less readable.

**Lacking common agent framework features** While the basic primitives for agents are presented in this work, primitives that are common to other agent frameworks are missing. For example, we had to implement a combinator to simulate task costs for our case-study. Another example of an important lacking feature is inter-agent communication: while iTasks provides enough primitives to implement this feature, we did not implement it because of time constraints. Being able to communicate is one of the requirements in an agent-based simulation framework according to Macal

[22]. The agents that we defined in the case-study only communicate implicitly through the environment (the central task model). It is possible that we could have defined more efficient strategies for agents when they can communicate directly with each other.

Arguably, we also lack features such as positioning agents in space and time. These features add more realism to the simulations and extend the set of scenarios that can be expressed with iTasks agents.

**Agent framework inefficiency** The efficiency of the agent framework can be improved. Right now, we experienced that it is difficult to simulate with more than 25 agents. This is due to the fact that each agent polls information from iTasks regularly. Recently, iTasks introduced the concept of push-based events, so that clients are informed about changes in their task models in a more efficient manner. In the future, the agent framework needs to use this feature in order to increase the efficiency.

**Type safety issues** We defined a dual for each interaction combinator in iTasks to construct agents. For example, the `enterInformation` task in iTasks can be performed with the `writeInformation` task in our agent framework. However, when developing agents, one needs to be careful what types are read from and written to tasks. There is no static check that verifies whether the types match. Although we made sure that types do not contain any unexpected UI information as described in section 4.4.1, mistakes are made easily, leading to runtime errors. It would help tremendously if some tool were available to analyse agents with respect to task models.

# 6 Conclusions

In this research, we attempted to use iTasks to improve the process of designing and building automation for complex domains. We defined our main research question as follows:

> How can we use iTasks as a tool to assist in designing automation for complex domains?

As this research question is quite broad, we decided to focus on performing agent-based simulations and using cognitive work analysis in conjunction with iTasks. We asked the following sub-questions:

1. How can we perform simulations with agents in iTasks?
2. How do agents in iTasks relate to agents in other frameworks?
3. How can iTasks be used to rapidly build prototypes in a non-trivial domain, based on products from task analysis methods and input from domain experts? In particular, how can cognitive work analysis assist in designing requirements for iTasks models?

We first explored the various techniques in chapter 2 and 3. We discussed the concepts of iTasks and CWA and how these techniques relate. We also discussed how CWA and iTasks can be positioned in a software development method in section 2.3. Then, we explored the topic of agent-based simulation. We discovered that there is no consensus on the meaning of the term *agent* and that there are many different tools available to perform agent-based simulations. We attempted to compare these tools to the agent framework that we proposed in this work to answer sub-question 2, although this is difficult due to the immaturity of our agent framework and the differences in concepts. From the comparisons, we can conclude that the main difference between our proposed method and other tools is that we use a central task model to specify tasks for teams of agents. Other simulation methods tend to only specify tasks in individual agents. We discuss the implication of having a central task model later on in this conclusion.

We showed how agent-based simulation can be performed with iTasks in section 4, which is related to the technical part of sub-question 1. This resulted in an integrated agent framework, requiring various additions and changes to the iTasks framework. We touched upon the duality relation of agents in section 4.3 and tasks and the possibility to automatically derive agents from iTasks task models in section 4.6. This dualism relation is unique when compared to other tools: future research must indicate whether it is useful in practice. Furthermore, we proposed a semantics for agents in iTasks in section 4.5.

Part of our thesis is a case study, developing a prototype for a non-trivial domain. The purpose of this case study is to be able to verify whether our method works and to actually use CWA in conjunction with iTasks for a practical problem. We can also use the results of the case study to help answer the practical part of our research questions. We developed a prototype for damage control on a navy patrol vessel, where various actors need to cooperate in order to resolve calamities on a vessel. We designed a formal task model for iTasks based on requirements that were specified using a decision ladder, which we partly show in section 5.2. In order to answer sub-question 3, we explained how these requirements relate to specific parts of the decision ladder in section 2.1 and 5.1. Concluding from the experience in our case study and the general discussion in section 2.1, we believe that some of the CWA products are suitable to derive requirements and claims, the decision ladders in particular.

As part of the case study, we developed a set of agents with the framework that we proposed in section 4. This captures the practical part of sub-question 1. In order to simulate tasks costs in terms of time and physical impact, we defined the `<<-` combinator in section 5.3.2. We discussed the behaviours of the agents in section 5.3.3. We believe that this section also reveals the interesting interplay between a central task model and agents performing tasks within this model: it is up to the developer to decide what tasks need to be defined centrally and decentrally. If the central task model is fixed and rigid, the agents are usually simple and not capable of interesting reasoning. If the central task model is

flexible and dynamic, the agents are usually more complicated and interesting. A choice can be made about the amount of freedom for agents: the central task model can be extremely restrictive or extremely permissive. A central model enables fine-grained control over task allocation and coordination, which can be useful in many domains. In section 5.4, we used these agents to verify the claim whether 20 seaman can resolve 2 fires in under 5 minutes. From these results, we can conclude that our method can be used to perform simulations that can be used to verify claims and requirements.

We believe that the concept of a task in iTasks and the system itself is enormously versatile; we were able to capture both agents and a moderately complex case study without any invasive changes to the core of the system. Furthermore, during development of our case study, we experienced that the notion of a task bridges a gap in communication with domain experts. It forces a level of detail and concreteness that is extremely valuable. By conducting the case study, we experienced that iTasks is a useful tool that can be positioned within a software development process.

## 6.1 Future work

In section 4.6, we touched upon the dualism property between tasks and agents to automatically derive simple agents from task models. Such functionality could have tremendous potential: a task model would be enough to automatically derive a working application together with simple software agents to perform human-in-the-loop tests. Similarly, a notion of agent soundness and agent completeness with respect to a task model could be defined. More research is required to investigate whether this is feasible.

We performed a case study in which we defined agents for an iTasks task model. We discovered that an interesting interplay exists between these agents and the central task model. Although we did discuss this interplay, more research is necessary to fully understand the implications of our proposed agent framework and whether it is useful for case studies in other domains. We believe that our method has potential in domains where there is a need for fine-grained task allocation and coordination.

GiN [11] needs to be updated with elements to support the latest iteration of iTasks. Providing a graphical notation for iTasks programs is essential to bridge a gap between developers and domain experts. Although a direct translation from GiN models to iTasks programs is interesting, it also clutters the diagrams with details that are not relevant to domain experts. It would be helpful if a visualization tool is also capable of omitting details (e.g. zoom functionality).

In section 5.5, we discussed the limitations of the agent framework. We discovered some of these limitations during implementation of the case study. We will now propose directions for future work to resolve these issues.

**Real-time versus virtual-time simulations** Currently, most tasks use the *currentTime* share or similar shares to obtain time information. These can be modified to support virtual time. It is possible to introduce a *virtualTime* share, such that time can be modified. During human-in-the-loop simulations, the clock needs to run at real-time. Therefore, it should be possible to change the behaviour of the clock (i.e. real-time or virtual-time) with a flag.

**No unified logging mechanism** A unified logging mechanism can be implemented by logging the events that agents produce. In order to distinguish agents, the concept of an agent identity is required in the agent framework. For the case study, we used a simple variant of identity that is not integrated in the framework. Still, in order to produce detailed information about actions performed by agents, a variant of custom logging directly implemented in agents needs to be supported.

**Lacking common agent framework features** Some important functionality is missing in the agent framework that we proposed; one of these features involves agent communication. Various communication protocols have been established for inter-agent communication [25]. We believe that any of these communication protocols can be implemented by using shares in iTasks: communication can be done by queuing messages with a mailbox communication style. Future research is needed to properly implement such functionality according to some agent communication language.

In this research, we did not focus on more detailed aspects that can be used to model humans. For instance, cognitive workload is used to simulate scenarios where cognitive lock-up occurs. This provides valuable information to reorganize task models. We recognize that this is important, but we did not have enough time to implement this feature. Furthermore, we did not investigate how these features can be expressed formally.

**Agent framework inefficiency** We discovered that simulation with more than 25 agents causes degraded performance. This is possibly caused by the polling mechanism that is in place to execute agents. Recently, a mechanism has been introduced in iTasks that allows push-based events. Performance could be improved if the agent framework is updated to support this mechanism. Furthermore, in order to reduce the amount of communication between the agent framework and iTasks, it is possible to combine task representations for multiple sessions in one HTTP request/response.

**Type safety issues** When defining agents for task models, one must make sure that the correct types are read from and written to tasks. If incorrect types are used, then runtime errors occur. We believe that the duality relation as explained in section 4.3 and 4.6 provides a starting point to research the possibility of building a tool to assist agent development. Although we acknowledge the undecidability of the soundness and completeness properties, there might be a subset of problems that is

decidable.

# References

[1] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(3):7280–7287, May 2002.

[2] Jeffrey M Bradshaw, Stuart Dutfield, Pete Benoit, and John D Woolley. Kaos: Toward an industrial-strength open agent architecture. *Software agents*, pages 375–418, 1997.

[3] Guido Bruinsma and Robert de Hoog. Exploring protocols for multidisciplinary disaster response using adaptive workflow simulation. In *International Conference on Information System for Crisis Response and Management (ISCRAM). Newark, New Jersey*, 2006.

[4] William J Clancey, Patricia Sachs, Maarten Sierhuis, and Ron Van Hoof. Brahms: Simulating practice for work systems design. *International Journal of Human-Computer Studies*, 49(6):831–865, 1998.

[5] Mehdi Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.

[6] Keith Decker. Taems: A framework for environment centered analysis & design of coordination mechanisms. *Foundations of distributed artificial intelligence*, pages 429–448, 1996.

[7] George S Fishman. Principles of discrete event simulation.[book review]. 1978.

[8] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent agents III agent theories, architectures, and languages*, pages 21–35. Springer, 1997.

[9] Maaike Harbers, Karel van den Bosch, and John-Jules Meyer. A methodology for developing self-explaining agents for virtual training. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1129–1130, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[10] Clinton Heinze, Simon Goss, Torgny Josefsson, Kerry Bennett, Sam Waugh, Ian Lloyd, Graeme Murray, and John Oldfield. Interchanging agents and humans in military simulation. *AI Magazine*, 23(2):37, 2002.

[11] Jeroen Henrix, Rinus Plasmeijer, and Peter Achten. Gin: a graphical language and tool for defining itask workflows. In *Proceedings of the 12th international conference on Trends in Functional Programming*, TFP'11, pages 163–178, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Bryan Horling, Victor Lesser, Regis Vincent, Tom Wagner, Anita Raja, Shelley Zhang, Keith Decker, and Alan Garvey. The TAEMS White Paper, 1999.

[13] Helen PN Hughes, Chris W Clegg, Mark A Robinson, and Richard M Crowder. Agent-based modelling and simulation: The potential contribution to organizational psychology. *Journal of Occupational and Organizational Psychology*, 85(3):487–502, 2012.

[14] Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, and Peter Achten. Embedding a web-based workflow management system in a functional language. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 7:1–7:8, New York, NY, USA, 2010. ACM.

[15] Daniel P Jenkins, Neville A Stanton, Paul M Salmon, Guy H Walker, and Laura Rafferty. Using the decision-ladder to add a formative element to naturalistic decision-making research. *Intl. Journal of Human–Computer Interaction*, 26(2-3):132–146, 2010.

[16] D.P. Jenkins, N.A. Stanton, G.H. Walker, P.M. Salmon, and M.S. Young. Applying cognitive work analysis to the design of rapidly reconfigurable interfaces in complex networks. *Theoretical Issues in Ergonomics Science*, 9(4):273–295, July 2008.

[17] W David Kelton and Averill M Law. *Simulation modeling and analysis*. McGraw Hill Boston, MA, 2000.

[18] Poole David L. and Mackworth Alan K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, New York, NY, USA, 2010.

[19] Marián Lekavý and Pavol Návrat. Expressivity of strips-like and htn-like planning. In *Proceedings of the 1st KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, KES-AMSTA '07, pages 121–130, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Bas Lijnse, Jan Martin Jansen, Ruud Nanne, and Rinus Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In *Proceedings of the 8th International ISCRAM Conference–Lisbon*, volume 1, 2011.

[21] Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, et al. Incidone: A task-oriented incident coordination tool. *Proc. ISCRAM 2012*, 2012.

[22] Charles M. Macal and Michael J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation*, WSC '05, pages 2–15. Winter Simulation Conference, 2005.

[23] A. Naiem, M. Reda, M. El-Beltagy, and I. El-Khodary. An agent based approach for modeling traffic flow. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–6, 2010.

[24] Mark A Neerincx and Jasper Lindenberg. Situated cognitive engineering for complex task environments. *Naturalistic decision making and macrocognition*, pages 373–390, 2008.

[25] Jeremy Pitt and Abe Mamdani. A protocol-based semantics for an agent communication language. In *IJCAI*, volume 99, pages 486–491, 1999.

[26] Rinus Plasmeijer, Peter Achten, Bas Lijnse, and Steffen Michels. Defining multi-user web applications with itasks. In *Proceedings of the 4th Summer School conference on Central European Functional Programming School*, CEFP'11, pages 46–92, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 195–206, New York, NY, USA, 2012. ACM.

[28] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP 12, pages 195–206, New York, NY, USA, 2012. ACM.

[29] Wilfried Post and Marleen Rakhorst-Oudendijk. Methods for sociotechnical system analysis applied to opv operational maintenance. *Unpublished*, 2013.

[30] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[31] J. Rasmussen, A.M. Pejtersen, and L.P. Goodstein. *Cognitive systems engineering*. Wiley Series in Systems Engineering and Management. Wiley, 1994.

[32] Kurt A Richardson. On the limits of bottom-up computer simulation: Towards a nonlinear modeling culture. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2003.

[33] Earl D. Sacerdoti. A structure for plans and behavior. Technical Report 109, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Aug 1975.

[34] Thomas B Sheridan and William L Verplank. Human and computer control of undersea teleoperators. Technical report, DTIC Document, 1978.

[35] Maarten Sierhuis, Jeffrey M Bradshaw, Alessandro Acquisti, Ron Van Hoof, Renia Jeffers, and Andrzej Uszok. Human-agent teamwork and adjustable

autonomy in practice. In *Proceedings of the seventh international symposium on artificial intelligence, robotics and automation in space (I-SAIRAS)*, 2003.

[36] Maarten Sierhuis, William J Clancey, and Ron JJ van Hoof. Brahms an agent-oriented language for work practice simulation and multi-agent systems development. In *Multi-Agent Programming:*, pages 73–117. Springer, 2009.

[37] Nanja JJM Smets, Jurriaan van Diggelen, Mark A Neerincx, Jeffrey M Bradshaw, Catholijn M Jonker, Lennard JV de Rijk, Pieter AM Senster, Ot ten Thije, and Maarten Sierhuis. Assessing human-agent teams for future space missions. *IEEE Intelligent Systems*, pages 46–53, 2010.

[38] Neville A. Stanton, Human Factors, Integration Defence, Technology Centre, and Ub Ph. Hierarchical task analysis: Developments, applications and extensions.

[39] Andrzej Uszok, Jeffrey Bradshaw, Renia Jeffers, Niranjan Suri, Patrick Hayes, Maggie Breedy, Larry Bunch, Matt Johnson, Shriniwas Kulkarni, and James Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 93–96. IEEE, 2003.

[40] K.J. Vicente. The abstraction hierarchy as a basis for interface design: an empirical evaluation. In *Systems, Man and Cybernetics, 1990. Conference Proceedings., IEEE International Conference on*, pages 657–659, 1990.