

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.

<http://hdl.handle.net/2066/103931>

Please be advised that this information was generated on 2018-07-08 and may be subject to change.



TOP TO THE RESCUE

**TASK-ORIENTED PROGRAMMING
FOR INCIDENT RESPONSE APPLICATIONS**

BAS LIJNSE

TOP to the Rescue

Task-Oriented Programming for Incident Response Applications

Bas Lijnse

This research has been supported by the Netherlands Defence Academy through their project “Dynamic Workflows for Planning of Military Operations and Crisis Management”, with the exception of chapter 5. This chapter was supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs, Agriculture and Innovation (project number 07729).

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

ISBN 978-90-820259-0-3
IPA Dissertation Series 2013-4

Copyright © 2013 B. Lijnse

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

A free digital copy of this dissertation can be obtained from <http://www.baslijnse.nl/projects/top-to-the-rescue/>



TOP to the Rescue

Task-Oriented Programming for Incident Response Applications

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op woensdag 27 maart 2013
om 10:30 uur precies
door

Bas Lijnse

geboren op 13 januari 1984
te Valkenisse

Promotor:

Prof. dr. ir. M.J. Plasmeijer

Copromotor:

Dr. J.M. Jansen

(Nederlandse Defensie Academie)

Manuscriptcommissie:

Prof. dr. J.H. Geuvers

Prof. dr. S.D. Swierstra

Prof. dr. M.P. Haselkorn

(Universiteit Utrecht)

(University of Washington, Seattle)

Write stuff that matters

Contents

1	Introduction	1
1.1	Programming	1
1.2	Programming with Tasks	2
1.3	Programming with iTasks	3
1.4	Programming for Incident Response Tasks	5
1.5	Scope and Organization of this Thesis	7
1.6	Future Directions	12
1.7	TOP to the Rescue	12
I	Task-Oriented Programming with iTasks	15
2	iTasks for End-users	17
2.1	Introduction	17
2.2	Declarative Workflow Specification	18
2.3	The Revised iTask System	20
2.4	Dynamic Generic Web-Interfaces	28
2.5	Related Work	36
2.6	Conclusions	37
3	Getting a Grip on Tasks that Coordinate Tasks	39
3.1	Introduction	39
3.2	The iTask Core System	41
3.3	The Expressive Power and Limitations of the Combinators	45
3.4	Redesigning the Core System	51
3.5	Conclusions	53
4	Task-Oriented Programming in a Pure Functional Language	55
4.1	Introduction	55
4.2	The TOP Paradigm	57

4.3	A Formal Foundation of TOP	62
4.4	Practical TOP	78
4.5	Related Work	83
4.6	Conclusions and Future Work	84
II	Types and Information Models	87
5	Between Types and Tables	89
5.1	Introduction	89
5.2	Motivating Example	92
5.3	Types and Tables	92
5.4	Generic CRUD Operations	99
5.5	Implementation in Clean	104
5.6	Related Work	107
5.7	Conclusions & Future Work	108
6	CCL: A Lightweight ORM Embedding in Clean	109
6.1	Introduction	109
6.2	A CCL Example	111
6.3	Defining Conceptual Models with CCL	111
6.4	Defining Clean Types with CCL	115
6.5	Discussion	116
6.6	Related Work	118
6.7	Conclusions	118
III	The Netherlands Coast Guard Case	119
7	Towards Dynamic Workflow Support for Crisis Management	121
7.1	Introduction	121
7.2	The iTask System	122
7.3	iTasks for Crisis Management?	126
7.4	Conclusions	129
8	Capturing the Netherlands Coast Guard's SAR Workflow with iTasks	131
8.1	Introduction	131
8.2	Literature Review	133

8.3	Research Questions	134
8.4	Methodology	135
8.5	Results	136
8.6	Discussion	143
8.7	Conclusions & Future Work	145
9	Incidone: A Task-Oriented Incident Coordination Tool	147
9.1	Introduction	147
9.2	A Watch Officer's View of Incidone	148
9.3	A Programmer's View of Incidone	151
9.4	Status Quo and Future Work	154
9.5	Conclusion	155
	Bibliography	157
	Summary	167
	Samenvatting	169
	Acknowledgements	171
	Curriculum Vitae	173

1 Introduction

This dissertation is about Task-Oriented Programming (TOP) and, to some extent, its potential to support incident response applications. Task-Oriented Programming is a new programming paradigm that emphasizes “tasks” as central concept for constructing programs. Although tasks are a common notion in daily life, in order to use them as building block in programs we need to be more precise. If we have a formal notion of tasks, are able to define atomic tasks, and have means to define tasks in terms of other tasks, we can think about programs in terms of tasks.

TOP emerged during an exploration of the use of functional programming techniques to define workflow management systems, with a focus on applications in dynamic high-autonomy environments such as crisis management. A specific area of interest has been the coordination of coast guard search and rescue incidents. This thesis consists of a selection of articles in the fields of functional programming, information modeling, and crisis management information systems that contributed to our understanding of what programming with tasks is all about.

1.1 Programming

To understand the motivations behind a programming paradigm based on tasks, we first have to have a clear view of what “programming” is. To many people, including experienced programmers, “programming” has become synonymous to writing code in a mainstream programming language like Java, C, Python, or Javascript that can be executed by a computer. This view is rather narrow as it only captures a common form of programming, instead of its essence. There are many other ways to program a computer besides writing code in a mainstream language. Apart from the obvious alternative of writing code in a less common language, or a domain specific language, one can use other means such as drawing diagrams. The first digital computers were even programmed by rearranging wires and flipping switches. What all these different forms of programming have in common, is that they enable one to specify a computer’s future behavior to make it support a specific task. This capability

is what sets computers apart from other machines. Computers are not made for a single task, but can be used to support any task that can be captured by a program. This makes it possible that a machine used to prepare documents with, can at the same time be a video player, a communication device and even a musical instrument.

Although programming can have many forms that essentially accomplish the same thing, this does not imply that form does not matter. The amount of effort it takes to program a computer for a task, and therefore the feasibility of using a computer to support it, depends on it. If we, for example, had to specify a video conference application at the bare-metal level of detail that was necessary in early machines, it would be a herculean effort to write such a program. Modern video conference applications, to reuse the example, are not only possible because we have laptops with built-in cameras, high-resolutions screens and enough memory and processing power. They also rely on the networking protocols and user interface API's provided by an operating system, the video encoding, decoding and compression algorithms provided by reusable libraries, and on programming languages that enable the integration of all these components. Contemporary application programming is less about expressing algorithms that express computations, but more about gluing together existing libraries, components and subsystems. With the abundance of libraries, API's, languages and components that are available, it is easy to get distracted by the cool features and capabilities offered by them and get absorbed by the technical details involved in gluing them together. Even to the extent that we lose track of the task the program is supposed to support. Design choices are then not based on a rational tradeoff between programming effort and an expected improvement in task performance, but on whatever is easy to accomplish with the language and components used.

1.2 Programming with Tasks

To make rational design decisions when programming, it is important to understand the task you aim to support. However, if a task is complex and has no clearly defined boundaries or outcome, this is much easier said than done. A common strategy for dealing with complexity is to divide and conquer. Divide a complex task into smaller and simpler subtasks. In programming this is also a proven strategy, but programs are not necessarily divided along the same lines. Depending on the choice of programming paradigm and language, we divide our programs in different ways. We can, for example, organize programs into hierarchical classes of objects, modules with related functions, or groups of logical sentences.

The Task-Oriented Programming paradigm is based on structuring programs around the task they support. If a task can be divided into smaller subtasks,

a program to support the task can be defined by specifying a composition of subprograms that support those subtasks. In order to write programs in this way we need a number of ingredients:

- *A formal notion of the task concept.* Our informal use of the word task is too ambiguous to serve as a building block in programs. We need a more precise definition of how a component, or subprogram, that supports a task behaves and how it can be interacted with. The definition we have come to use, which is specified in full detail in Chapter 4, can be paraphrased as: “A task is a specified piece of work aiming to produce a result of known *type*. When *executed*, tasks may produce (temporary) *results* that can be observed in a controlled way.”
- *A set of task primitives.* To be able to structure programs as decompositions of tasks, we need to provide primitive components that support those tasks that cannot be further decomposed. These basic task components serve as the smallest building blocks from which a task-oriented program is constructed. Such primitives can for example be, entering a piece of data, querying a database, or accessing a sensor.
- *A set of composition and abstraction methods.* In order to define complex tasks as compositions of subtasks, we need means to compose tasks and to abstract from recurring patterns. Tasks can for example be divided into a set of subtasks that are performed simultaneously, or into a sequence of subtasks that have to be performed one after another.
- *A method for sharing data between tasks.* If a task is decomposed into subtasks, data that is produced by one task is often consumed by others. To program in a task-oriented way, means of sharing data between tasks must be facilitated.

These four ingredients have to be provided by a framework or a programming language in order to write task-oriented programs. The specific way we have studied Task-Oriented Programming so far, has been in the context of a pure functional programming language. TOP has been facilitated by a framework that provides an embedded domain specific language for specifying task compositions.

1.3 Programming with iTasks

The ideas behind Task-Oriented Programming were not the result of a momentary flash of insight. They emerged during development of, and experimenting with, the iTask System (or iTasks for short), a concrete prototype system that uses tasks as its core concept. The iTask System started as a library in the

Clean programming language for specifying workflow in a functional programming language[61]. Clean [69] is a statically typed pure functional programming language similar to Haskell [59]. In a pure functional language, programs are made up of side-effect free functions, meaning that the result of a function depends only on its arguments. Furthermore, functions can be higher-order, meaning that they can take functions as arguments or return a function as result. These properties make it possible to precisely define subprograms that support independent tasks as abstract state-transition functions with a specific interface. Task-level primitives can be defined by black-box implementations of this interface, and composition can be accomplished through higher-order functions, called combinators, that take such state-transition functions as arguments. The original iTask System was based on the notion of tasks that *finish* with a *typed* result. Tasks existed either in a fluent “in progress” state in which they could be worked on, or a “finished” final state in which they had an immutable value. When a task finished, its result was used to compute the next step in the workflow. This provided a mix of control-flow with data-flow that could be used to express dynamic data-driven workflows. The use of typed task results was not only a safety measure to prevent programming errors, it was also used to generate dynamic web pages using type generic functions. Influenced by the literature on workflow management systems, the iTask System evolved from a library to a programmable workflow management system. This system exploited the advances in the programmability of web-browsers to enable a client-server architecture with rich clients in the browser (see Chapter 2). In this revised architecture, common workflow management capabilities, such as a worklist, were provided by the iTask web client. The server applications that coordinated workflows were generated from task expressions. After applying the iTask System in real-world cases [42, 77], we found that the workflow management system approach did not cover relevant aspects of supporting tasks. Although workflows were programmable and tasks as seen from an organisation’s perspective could be expressed, the management of workflows by an end-user could not. The ways in which a user can select tasks to work on, can monitor ongoing tasks, or can manage delegated work, had become a fixed feature set of the workflow management system. Although this is not different from the state of the art in workflow management systems today, it turned out to be limiting in completely defining a program in terms of the task it supports. Additionally, the cases inspired a change in definition of the task concept used in the iTask System. The emphasis on “finished” tasks was loosened to focus instead on “current values” of tasks. This made it easier to take things like draft results into account, such that preemptive action can be taken, or decisions can be made based on incomplete results. Driven by these insights the iTask System evolved into a general-purpose TOP framework, which can be used to create multi-user web-based applications. In this

framework the common workflow management components such as worklists, which were previously hardcoded, are now expressed as tasks themselves. Development work on both the iTask System itself, and on applications made with it, has been a significant chunk of the effort behind this dissertation. Although it is not strictly necessary to have a full implementation of a TOP framework to illustrate the paradigm, its development was only possible by having a tangible system that expressed our understanding at all times. By having an implementation we could test its limitations through examples and experiment iteratively with new ideas. In some cases it even sparked improvements in the Clean programming language. In order to implement the task primitives as generic as possible, we needed to use some of Clean's language constructs, such as type generic functions and dynamic types, to an extent that had not been done before.

1.4 Programming for Incident Response Tasks

Understanding the task a program is meant to support is important for any program. The difficulty of understanding it, or the consequences of getting it wrong are not the same for every program though. Therefore, task-oriented programming is more suited for some applications than others. If the task is clear, and can be completely automated, the TOP paradigm does not add much value compared to a normal functional program. For example, a compiler or an image conversion program automate clear tasks. The inputs and outputs are well defined and while it is certainly not easy to describe the algorithms that transform input to output, compiling a program, or rotating a photo can be seen as a single automated task. Task-Oriented Programming is best suited for those tasks that cannot be fully automated, but can be supported by a mix of interactive and automated tasks that have to be coordinated to achieve the combined goal.

The most extreme examples of such tasks are those in which the conditions under which the task has to be accomplished are unpredictable, where people have to work together, and there is time pressure to get it done. Tasks like responding to life-or-death critical incidents in a rescue operation, or the management of crises in general, belong to this category. In order to support these tasks with computer systems it is important to understand how the subtasks of all those involved contribute to the whole, in order to design programs that assist humans without getting in the way. A challenge in the design of systems to support such tasks is that they have to handle inherently unpredictable scenarios. However, just having unpredictable scenarios does not imply that it is impossible to support humans in dealing with them. It just means that a certain amount of flexibility has to be built-in to reorganize supported tasks to adapt to each unique scenario. The key to doing this is to make sure that

high-level control-flow aspects of the system are designed to be a supported human task instead of a hardcoded automated task. In other words, coordinating tasks is also a task that has to be supported. In this way, users stay in control at all times.

To make sure that the TOP paradigm captures a wide range of real-world tasks we have continuously reflected on its expressiveness for such cases throughout the development of the iTask System and the TOP paradigm. By looking at cases where the task as a whole is a complex combination of tasks with a central role for humans, but assisted by automation, we have learned lessons that influenced the TOP paradigm a great deal. Most of these seem obvious when pointed out, but can nevertheless have negative consequences if they are forgotten when developing programs. For example:

- From a user's perspective, there is no distinction between which parts of a system are infrastructure and which parts are application specific. Hence, if you use a workflow management system, a groupware system, or a database system to support a given task, generic functionality like browsing a tasklist, or managing users and assigning permissions in that system, add overhead for the user that cannot be ignored.
- Managing, planning, and tracking tasks is also a task. If you design a workflow in which work is delegated, the total amount of work increases, because now someone has to do the work, and someone has to track whether it gets done.
- Tasks may not always have a clear "final result". The context may determine when something is finished. Most automated tasks have a clear beginning and end. A computation takes some processing time and then yields its result. Downloading a file completes at some point. Human tasks or combinations of human and automated tasks do not always have such a clear end result. Writing a book or a paper is done when you consider it done, or when you run out of time. But you can always keep working on it if you like.
- Even when results are not final, other tasks can use them to take proactive action. Sometimes when you are waiting for a result, you know you are definitely too late to act on it if you start acting only once the result is available. Therefore, you need to be able to observe temporary results to act pro-actively, such that once the final result comes in, you can deal with it.

Although it is not possible to exactly pinpoint how lessons such as these have influenced the TOP paradigm and the design of the iTask System, there is no doubt that they have. Studying incident response tasks, and crisis management in general has provided a frame of reference for the more technical

challenges. Because of this context, TOP and specifically the way it is embodied in iTasks, has evolved towards a paradigm for writings programs that can capture collaborative computer-supported high-autonomy tasks.

1.5 Scope and Organization of this Thesis

With the exception of this introduction, all chapters in this book are independent articles that have been published in peer-reviewed conference proceedings. Each chapter is self-contained, so there is some unavoidable overlap in introductions and preliminaries sections. The upside of this approach is that it is easier to select only those chapters that have your interest. The chapters are not presented in the chronological order in which they were published, but are grouped into three parts that correspond to three interrelated research themes, or tracks, that have been investigated simultaneously. These parts are: “Task-Oriented Programming with iTasks”, “Information Models and Data Types” and “The Netherlands Coast Guard Case”. The remainder of this section explains the role each of these themes has played in our understanding of programming with tasks. For each of the chapters a motivation is given why they were selected to be presented and what the contribution of the author of this dissertation has been.

Part I: Task-Oriented Programming with iTasks

This first part contains my contributions to the development of the iTask System and the TOP paradigm, which has been my main research focus. The three articles in this theme illustrate the status quo early in the project, more or less half-way and at the end of the project. When viewed together as a series they show the progressed insight into Task-Oriented Programming and the simultaneous evolution of the iTask System with it.

Chapter 2: iTasks for End-users

This chapter describes the transition of the original iTask system to the workflow management system approach. It introduced the client-server architecture with a significant role for the client system beyond displaying generated web pages. It also outlines improved generic techniques for providing a sufficient user experience based solely on task compositions and data types.

The chapter is based on the paper “*iTasks 2: iTasks for End-users*”[44]. It marked my first major contribution to the design and implementation of iTasks. I independently conducted almost all research, and all writing of this paper. Steffen Michels assisted in exploring incremental user interface updates.

**Chapter 3:
Getting a Grip on Tasks that Coordinate Tasks**

This chapter outlines the status of the iTask System just before its transition to a general-purpose TOP framework. It discusses new possibilities for reducing the number of core concepts for combining tasks, but also outlines the limitations of the then current task approach. It discusses the need to generalize the approach to cover a wider range of interactive applications beyond workflow management. The chapter is based on the invited paper “*Getting a Grip on Tasks that Coordinate Tasks*” [67]. It reports on the combined research of all authors. This included significant development of the iTask System performed by me. I also contributed to the writing of the paper.

**Chapter 4:
Task-Oriented Programming in a Pure Functional Language**

This chapter is the latest paper about the iTask System included in this dissertation. It is the first paper that positions TOP as a programming paradigm with iTasks as its supporting framework. In this chapter the core concepts of Task-Oriented Programming are explained and their semantics are formally defined by a rewrite system expressed in Clean. The chapter is based on the paper “*Task-Oriented Programming in a Pure Functional Language*” [68]. Just as Chapter 3, this paper reflects the complete team effort on the development of the iTask System. In the research that led up to this paper, I have explored the new task definition in which tasks no longer have a single final result but may yield intermediate values inspired by examples from the case studies I conducted. To explore its consequences I implemented a redesigned iTask System based on the new semantics.

During this process I also advocated the use of a separate name for the style of programming enabled by the iTask System. I first coined the term Task-Oriented Programming for referring to the programming paradigm, whereas iTasks refers to the framework that makes it possible.

More publications in this theme

The chapters in this part were selected because they best illustrate the progression of insight into Task-Oriented Programming with iTasks. In addition to these, I also contributed to other publications about the development of iTasks that are not included in this dissertation. These publications are:

**iTasks for a change:
Type-safe run-time change in dynamically evolving workflows**

In this paper[64] we explored an extension to iTasks for run time-change of partially completed tasks. We showed how arbitrary self-contained tasks could be

replaced by other tasks while providing safety guarantees on the execution of the modified compositions. I extended the iTask System with the capabilities demonstrated in the paper and co-authored the paper.

Defining multi-user web applications with iTasks

I assisted in a course on programming web applications with iTasks at the Central European Functional Programming Summer School 2011. This paper [65] provides an introduction to the iTask System targeted at intermediate level functional programmers with examples and exercises. It was used as lecture notes during this course. In addition to assisting during the summer school I also co-authored the lecture notes.

Part II: Types and Information Models

The second research theme to which I contributed, covers the relation between local data used in concrete tasks, and global data that is shared throughout an application. Although the chapters in this part are not directly related to TOP, they facilitate programming information systems in a task-oriented way. In information systems, tasks are often similar and can be abstracted into generic task patterns, but there is large variety in the data used in tasks. In iTasks the data used by tasks is typed, therefore a large collection of related data types is specified when defining an information system. The two chapters in this part deal on the one hand with making use of this type information to automate conversions to and from relational databases, and on the other hand with reducing the need to specify types by abstracting to underlying conceptual information models.

Chapter 5: Between Types and Tables

This chapter reports on research I initially did for my master's thesis with the same title. It explains how type-driven generic programming can be used to automatically map data between relational databases and values in a typed programming language when both representations are derived from the same conceptual information model.

The chapter is based on the paper "*Between types and tables - Using generic programming for automated mapping between data types and relational databases*" [45]. A shortened popularized version of this paper has been published as *Tussen Types en Tabellen...* [41] in *Optimize*, a Dutch magazine for database professionals. For my master's thesis I received the "Aia Master Thesis Award" for best computer science master's thesis of my graduation year at the Radboud University. I independently conducted all research and writing of both papers.

**Chapter 6:
CCL: A Lightweight ORM Embedding in Clean**

This chapter outlines an experimental language for expressing conceptual information models textually, such that it can be used as a Clean language extension. From these conceptual model specifications Clean types can be derived which makes specification of collections of conceptually related types more concise. The chapter is based on the paper “*CCL: A Lightweight ORM Embedding in Clean*” [46]. I independently conducted all research and writing of this paper.

Part III: The Netherlands Coast Guard Case

The third and final theme in my research has been a continuous reflection on the TOP paradigm and the iTask System by investigating it in the context of relevant real-world applications. The most influential of these has been a case study of the operations of the Netherlands Coast Guard. In this part, a series of three articles illustrate the influence of this case.

**Chapter 7:
Towards Dynamic Workflow Support for Crisis Management**

This chapter discusses the suitability of the iTasks approach to workflow support for crisis management applications. The discussion is based on the second generation iTask System as described in Chapter 2 before I conducted any case studies in this domain. The chapter is based on the short paper “*Towards Dynamic Workflow Support for Crisis Management*” [30]. I conducted the research as well as the writing for this paper together with Jan Martin Jansen.

**Chapter 8:
Capturing the Netherlands Coast Guards SAR Workflow with iTasks**

This chapter reports on our investigation to what extent the Search and Rescue operations of the Netherlands Coast Guard can be expressed with the iTask System’s formalism. This case study turned out to be very helpful in the further development of the iTask System and the TOP paradigm.

The chapter is based on the paper “*Capturing the Netherlands Coast Guards SAR Workflow with iTasks*” [42]. It was awarded the “Mike Melishkin Award” for best student paper at the ISCRAM 2011 conference. I independently performed the majority of the research and writing, with some help from Jan Martin Jansen and Ruud Nanne who provided valuable knowledge about maritime terminology and practice.

Chapter 9:**Incidone: A Task-Oriented Incident Coordination Tool**

This final chapter reports on the project that was started to make the lessons learned from the Coast Guard case in Chapter 8 more visible. It presents a draft design of a tool that is being developed with the latest iTask System described in Chapter 4 to support incident response operations such as Search and Rescue. The chapter is based on the “work in progress” paper “*Incidone: A Task-Oriented Incident Coordination Tool*” [43]. I independently designed and implemented the Incidone tool and authored the paper.

More publications in this theme

The three chapters in this section were selected because they show the use of TOP in a challenging application domain. However, the crisis management/incident coordination context is not the only domain in which we explored the use of TOP with iTasks. The following papers to which I also contributed but are not included in this dissertation illustrate the use of iTasks for other applications:

An iTask case study: a conference management system

This paper [63] explains the original iTask System by means of an example application: a conference management system. It served as leading example for a lecture about iTasks at the Advanced Functional Programming Summer School 2008. Together with Thomas van Noort I implemented the featured conference management system.

Web based dynamic workflow systems for C2 of military operations

This paper [31] provides a similar domain-wide discussion as Chapter 7, but focused on military operations instead of crisis management. I conducted the research as well as the writing of this paper together with Jan Martin Jansen and Tim Grant.

Managing COPD Exacerbations with Telemedicine

This paper [77] reports on the development of a prototype telecare system for Chronic Obstructive Pulmonary Disease (COPD) patients. Patient data is collected through smartphones equipped with additional sensors and reported back to a central backend system. A Bayesian model interprets this data and predicts the risk of exacerbation. I implemented the complete telecare system with the exception of the Bayesian model. I used iTasks for the backend that coordinates the data collection and distribution.

1.6 Future Directions

The work presented in this dissertation shows the progress made in understanding the possibilities and limitations of programming using the task concept. Task-Oriented Programming now has a solid foundation that is grounded in an understanding of high-autonomy response tasks.

However, there are also new fundamental questions to be answered. Although TOP is formally defined now, it is not yet clear how this can be used for analyzing, reasoning about, and proving properties of task-oriented programs for example. Another interesting question is to what extent we can distribute task-oriented programs over different organizations, locations and machines. From an organizational viewpoint, it might be interesting to know how to visualize or verbalize the knowledge that is captured by task-oriented programs, or what the effects of using different decompositions with the same result are. Perhaps the most interesting question is how useful Task-Oriented Programming is in real-world applications. The current tool support for TOP in the form of the iTask System is a research framework, not a mature product. First the tools need to mature in terms of robustness, performance and scalability to an acceptable level. Only then real-world programs can be developed and deployed such that this question be answered. With the development of the Incidone tool outlined in Chapter 9, we take a small step in this direction.

1.7 TOP to the Rescue

The work presented in this dissertation contributes knowledge to the fields of functional programming, information modeling, and crisis management information systems. Its purpose is to provide insights, not to make a value judgment. However, programming is as much an art as it is a science, and what constitutes “good” programming has been the subject of heated debates. As a programmer instead of a scientist I have of course my personal preferences and opinions on the subject. And while I am aware that the following is outside the scope of scientific argument, I think it is valuable to end this chapter with some points that I have found especially interesting while bringing TOP to the Rescue.

Effortless Interactive Tasks

An interesting property of focusing on tasks is that it forces programmers to think differently about the role of humans in interactive systems. The idea of tasks that have to be done, can be as easily applied to things humans have to do as well as computers. The difference when we are making a computer program is how we choose to enable execution of tasks. If we have a task that we want to be fully automated, we have to specify an algorithm that defines an

input-output relation. If we intend a task to be done by a human, we have to specify an interaction as a combination of viewing, manipulating, and entering data that enables her to do the task and to make its result available. TOP therefore acknowledges humans as resources that can work on tasks, instead of implicit originators of streams of mouse clicks and keystrokes.

Variable Levels of Specification Detail

Because the task concept applies to a wide range of abstraction levels and can be applied to both automated and manual tasks, it creates freedom for a programmer to choose the appropriate level of detail for every task. If a task is straightforward, it can be expressed in great detail with much automation. Conversely if a task requires human creativity, or there are many ways to accomplish it, it is just as easy to specify an open-ended task that defines only the interface in terms of the type of data that is produced. Both of these extremes can exist together in a task-oriented program. This makes it possible to iteratively develop applications by starting with programs with little automation that just store and distribute results produced by people. Then gradually, once the workflow is better understood, more detail can be added.

Instant Executable Interactive Programs

Just as you can vary the detail of task decomposition, the high-level task primitives enabled by generic algorithms make it possible to choose the amount of control you want to have over data storage or user interaction. This means that with minimal effort you can have an executable program for any task. You always get a working system immediately. If you then decide you want a prettier or more ergonomic user interface, you can put more effort into fine-tuning it, but if it is not important you do not have to. The same holds for data storage or exchange. A generic workable solution is always available, but if you want more control, you can consider whether it is a worth the effort for every subtask in your program.

Doubly Functional Programs

The term “functional” has multiple meanings in software development. In functional programming it refers to the pure mathematical notion of side-effect free computations. In functional specification, as a requirements engineering method, it refers to the specification of what a system is supposed to do, free from technical details. In both cases we are interested in *what*, rather than *how*. Task-oriented programs leverage the power of first-class functions to define interactive systems that are written at the level of abstraction of functional specifications.

Part I

Task-Oriented Programming with iTasks

2 iTasks for End-users

Workflow management systems (WFMSs) are systems that generate, coordinate and monitor tasks performed by human workers in collaboration with automated (information) systems. The iTask system (iTasks) is a WFMS that uses a combinator language embedded in the pure and lazy functional language Clean for the specification of highly dynamic workflows. iTask workflow specifications are declarative in the sense that they only specify (business) processes and the types of data involved. They abstract from user interface and storage issues, which are handled generically by the workflow engine.

Earlier work has focused on the development of the iTask combinator language. The workflow language was implemented as an engine that evaluated task combinator expressions and generated interactive web pages. Although suitable for its original purpose, this architecture has proven to be less so for generating practically usable workflow support systems.

In this paper we present a new implementation of the iTask system that implements the combinator library using a service based architecture that exposes the workflow and a user friendly Ajax client. Because user interface issues are outside the scope of workflow specifications, and cannot be specified explicitly, it is crucial that the generic operationalization of the declarative interaction primitives is of adequate quality. We explain the novel generic libraries we have developed for this purpose.

2.1 Introduction

Workflow management systems (WFMSs) are systems that generate, coordinate and monitor tasks performed by human workers in collaboration with automated (information) systems. Many contemporary WFMSs suffer from lack of flexibility. This is partially caused by the *static* nature of the languages used for modeling the business processes they coordinate. To address this limitation the iTask system has been developed. This system uses a function combinator library embedded in the pure and lazy functional programming language Clean to model business processes, and allows specification of highly dynamic workflows. The iTask system uses *declarative* specifications of tasks. Task specifications define what has to be done, by whom and when. However, they do not specify

how tasks are presented to users, how results are entered, or how progress is visualized. These operational details are taken care of fully automatically.

Earlier work [37, 61, 63, 66] has focused primarily on the benefits of the iTask system for programmers. Its goal has been to develop and extend the iTask combinator library to be able to express powerful, yet concise, specifications of arbitrary business processes. For this purpose a prototype implementation of the iTask engine with a minimum level of usability that could be used to simulate workflow scenarios by expert users has been sufficient.

In this paper we present a new implementation of the iTask system that uses a service based architecture to enable a practically applicable interface for end-users. Since user interaction is considered a declarative aspect of the iTask language and outside the scope of a workflow specification, it is *critical* for the usefulness of the iTask system that the generic framework performs adequately in this area. We show how we operationalize workflow specifications in such a way that, for end-users, selecting and working on tasks is no more difficult than the use of an average e-mail client.

The contributions of this paper are the following:

- We present a new implementation of the iTask system. We discuss its new service based architecture and key features, and how it compares to previous implementations.
- We explain the declarative nature of the iTask system. We discuss *what is* specified by iTask expressions, and *what is not*. We show *how* workflow specifications are operationalized by the iTask engine.
- We present a novel generic web interface library in Clean. This library provides *type-driven* Html visualizations of data as well as editable Ajax forms for manipulating data.

The remainder of this paper is organized as follows: First we cover the concept of declarative workflow specification in the iTask system in Section 2.2. Then an architectural overview of the iTask system is given in Section 2.3. The generic web-interface library is explained in Section 2.4. We discuss related work in Section 2.5 after which final concluding remarks are given in Section 2.6.

2.2 Declarative Workflow Specification

The iTask combinator language is designed for declarative specification of workflows. This means that the specifications describe *what* has to be done, not *how*. However, one cannot speak of a language being declarative without specifying at which level of granularity. The level of abstraction of a domain determines whether a specification can be classified as declarative at that level. Since

this level is not always immediately clear, especially in workflow languages, we elaborate on it some more in this section.

2.2.1 When Is a Workflow Specification Declarative?

The iTask system is based on the idea that in workflow support systems, the only differences that really matter between two systems are: 1) The (business) process they support, and 2) The data that is exchanged between actors. Everything else that is needed to build these systems can be generic. The iTask system provides both a specification language to describe the processes and data, as well as a framework that provides the generic foundation that operationalizes them.

In this context, we classify a specification as declarative when everything in it specifies either data or process. Contrary to what is sometimes called declarative workflow, a process can be specified very rigidly but still be considered declarative with respect to this definition. A specification that also specifies issues such as presentation, or storage is considered not declarative in this context. A quick glance at the signature of one of the iTask primitives for interacting with users in Figure 2.1 illustrates this best. For instance, the `enterInformation` primitive yields a task that asks a user to provide some information. This primitive describes the action that is needed to achieve some goal, but leaves entirely open *how* information is entered.

2.2.2 The iTask Workflow Language

Above we have already loosely mentioned the iTask specification language, yet we have not explained how it is defined and implemented. The iTask language is a domain specific language embedded in the pure and lazy functional programming language `Clean`. It is essentially an API of functions and (monadic) function combinators that is used to construct complex functions that when evaluated compute the tasks that have to be done. However, from the point of view of a workflow programmer, the combinator API is just a collection of primitives and operators that are used to define workflows in a syntax that just happens to have a striking resemblance to `Clean`.

The central concept of iTask workflow specifications is that everything is a task that produces a *typed* result once it is done. Tasks are represented by the abstract `Clean` type `:: Task a`, where `a` is the type of the result of the task. Although everything is a task, we can still make a distinction between *basic tasks* and *combined tasks*. Basic tasks are the smallest units of work like entering some data in a form, or reading a piece of data from a database. From these basic tasks, larger more complex tasks are constructed using *task combinators*. For example the monadic bind combinator (`>>=`), where the result of the first task is passed to a function that computes the second. By combining tasks

sequentially, in parallel or conditionally, tasks of unlimited complexity can be constructed. A short excerpt with common tasks and combinators from the iTask API is shown in Figure 2.1¹. The full API consists of many more basic tasks and combinators, like for instance, for interacting with users, generic storage and retrieval, access to meta-data of other workflows and users. Examples of iTask workflow specifications have been given in [61, 63].

2.2.3 Implementation Consequences

As can be seen in the API in Figure 2.1, workflow specifications in the iTask system define nothing more than data and process. However, a complete executable workflow system is generated from just that and nothing else. A major consequence of this design is that this generic foundation that is used to generate a working system from these high level specifications must be of such quality, that there is no need to further hack or tweak the system after generation. When this is not the case the risk exists that clever programmers will find ways to abuse the workflow language to force for example a specific interface layout. This clutters the workflow definitions and makes them no longer declarative.

Of course there are domains where generic solutions are far inferior to specialized instances. Entering a location for example, is easier by putting a marker on a map than by entering coordinates in a form. For these situations the iTask system provides the possibility to define custom domain libraries that contain data types and task primitives along with specializations of the generics. This enables the use of custom code when necessary without cluttering the workflow specifications.

2.3 The Revised iTask System

As mentioned in Section 2.1 the original iTask system was used primarily to explore the design of a workflow language based on function combinators. However, experiments with building applications beyond the level of toy examples showed that much hacking and tweaking was necessary to build somewhat usable applications. Examples of such tweaking are: the use of multiple variants of essentially the same task: `chooseTaskWithButtons` and `chooseTaskWithRadios`, or the use of presentation oriented data types such as `HtmlTextArea` instead of just `String`. To be able to generate iTask applications at the level of usability that may be expected from contemporary web-based information and workflow systems, without cluttering the workflow specifications with presentation issues, a major redesign of the iTask engine was necessary.

¹Context restrictions on overloaded types have been omitted for clarity

— Basic tasks —

```

1 // Ask a user to enter information.
2 enterInformation    :: question → Task a
3 // Ask a user to enter information while subject information is shown
4 enterInformationAbout :: question s → Task a
5 // Show a message to a user
6 showMessage        :: message → Task Void
7 // Show a message and subject information to a user
8 showMessageAbout   :: message s → Task Void
9 // Create a value in the data store
10 dbCreateItem      ::                Task a
11 // Read a value from the data store
12 dbReadItem        :: !(DBRef a) → Task (Maybe a)

```

— Task combinators —

```

13 // Lift a value to the task domain
14 return          :: a → Task a
15 // Bind two tasks sequentially
16 (>>=) infixl 1 :: (Task a) (a → Task b) → Task b
17 // Assign a task to another user
18 (@:) infixr 5 :: UserId (Task a) → Task a
19 // Execute two tasks in parallel
20 (-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b)
21 // Execute two tasks in parallel, finish as soon as one yields a result
22 (-||-) infixr 3 :: (Task a) (Task a) → Task a
23 // Execute all tasks in parallel
24 allTasks        :: ([Task a] → Task [a])
25 // Execute all tasks in parallel, finish as soon as one yields a result
26 anyTask         :: ([Task a] → Task a)

```

Figure 2.1: A short excerpt from the iTask API

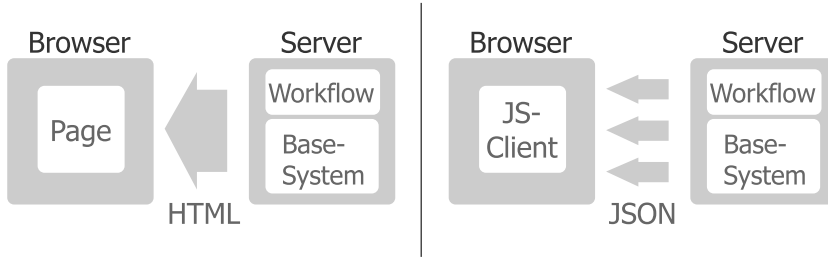


Figure 2.2: Architecture old (left) and new (right) iTask system

2.3.1 Original Architecture

Originally the architecture of the iTask system as presented in [62, 61] was that of a simple web application that dynamically generated `Html` pages. The content of these pages was generated by a program compiled from an iTask workflow specification and a generic base system. This architecture is depicted graphically in the left diagram of Figure 2.2. Page content generation was performed by application of a workflow definition to an initial state which yielded an output state that accumulated `Html` code. The abstract type `Task a` of task primitives and combinators was defined as $\text{Task } a ::= *TSt \rightarrow (a, *TSt)$ which is `Clean`'s notation for a function that takes a unique state of type `TSt` and returns a value of type `a` and new state. Additionally to generating the `Html` code for the tasks to display on the page, `TSt` also accumulated ad-hoc meta-data about tasks, which was used to generate the navigation components for switching between tasks. When users triggered some event in the generated page, like clicking a button or changing the content of a textbox, the event was sent to the server by reloading the entire page, and used to generate the updated page. This was necessary because each event could potentially cause the workflow to be reduced or the user interface to be different.

2.3.2 Fundamental Problems

The original architecture, though suitable for showing the expressive power of the combinators, suffered from some scalability problems. When used in a more realistic setting, this architecture has a number of fundamental problems.

1. The first issue is one of separation of concerns. The original implementation of the task combinators as functions that both compute the advancement in a workflow *and* the representation of that workflow as a user interface only works for small examples. As soon as you want to define more intricate workflow combinators or put higher demands on the

user interface, the implementations of the workflow combinators quickly becomes too complex to manage.

2. Another problem, which is related to the previous issue, is that in the original architecture the only way to interact with iTask workflows was through the web interface. There was no easy means of integrating with other systems. The obvious solution would be to add some flavor of remote procedure calling to the system, but this would then also have to be handled *within* the combinators, making them even more complex.
3. The final issue, which may appear trivial, is the necessity to reload an entire page after each event. This approach is not only costly in terms of network overhead, it also inherently limits the possibilities for building a decent user interface. Essential local state, such as cursor focus, is lost during a page reload which makes filling out a simple form using just the keyboard nearly impossible.

2.3.3 Improved Architecture

To solve the problems described in the previous section, a drastic redesign of the iTask system was needed. The only way to address them was to re-implement the iTask combinator language on top of a different architecture.

The architecture of the new iTask implementation is a web-service based client-server architecture and is shown in head to head comparison with the old architecture in Figure 2.2 and illustrated in more detail in Figure 2.3. The major difference between the old and new architecture is that the new server system does not generate web pages. Instead, it evaluates workflow specifications with stored state of workflow instances to generate datastructures called *Task Trees*. These represent the current state of workflows at the task level. These trees contain structural information: how tasks are composed of subtasks, meta-data: for example, which user is assigned to which task, and task content: a definition of work that has to be done. For interactive tasks, the content is a high-level user interface definition that can be automatically generated, which will be explained in Section 2.4. Task trees can be queried and manipulated by a client program through a set of JSON (JavaScript Object Notation: A lightweight data-interchange format) web services.

The overview shown in Figure 2.3 illustrates how the various components in the server correspond with components in the client. The workflow specifications are queried directly through the workflow directory service. The authentication service queries the user store. All other services use the task trees as intermediate representation. In the next section, the computation of task trees and the individual services are explained in more detail.

The iTask system provides a default web based Ajax client system, described in Section 2.3.5, that lets users browse their task list, start new workflow instances

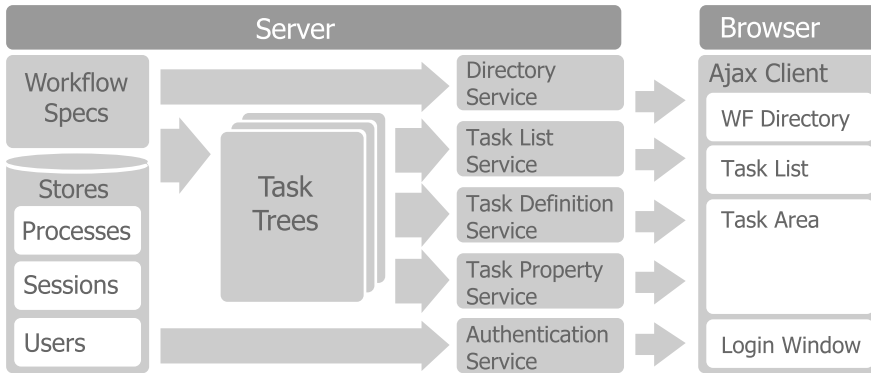


Figure 2.3: A detailed architecture overview

and work on multiple tasks concurrently. However, because the service based architecture nicely separates the computation of workflow state from presentation, and communication is based on open web standards, it is also easy to integrate with external systems. For example, we have also built a special purpose client written in Python that monitors a filesystem for new documents and starts a new workflow for processing that simply uses the same services as the standard client.

2.3.4 The Server System

The server system manages a database with the state of all active workflow instances (processes) and user and session information. It offers interaction with the workflow instances through JSON webservices. Requests to these services are HTTP requests that use HTTP POST variables to pass arguments. Responses are JSON encoded data structures. The server system is generated by compiling a Clean program that evaluates the `startEngine` function defined by the `iTask` base system. This function takes a list of workflow specifications as its argument. The `iTask` system provides two implementations of the `startEngine` function. One implements a simple HTTP server, which is useful for development and testing. The other implements the server system as a CGI application for use with third party web server software.

Task Tree Computation

The core task of the server system is to compute and update representations of the current states of executing workflow processes. The central internal repres-

entation of the state of a workflow instance that is computed by a combinator expression is a data structure called *Task Tree*. It is a tree structure where the leaves are the atomic tasks that have to be performed, and the nodes are compositions of other tasks. It is the primary interface between the workflow specifications and the rest of the framework and is queried to generate task lists and user interface definitions. Task trees are defined by the following Clean data type:

```

1 :: TaskTree
2 = // A stand-alone unit of work with meta-data
3   TTMainTask      TaskInfo TaskProperties [TaskTree]
4   // A task composed of a sequence of tasks
5   | TTSequenceTask  TaskInfo              [TaskTree]
6   // A task composed of a set tasks to be executed in parallel
7   | TTParallelTask  TaskInfo              [TaskTree]
8   // A task that interacts with a user
9   | TTInteractiveTask TaskInfo (Either TUIDef [TUIUpdate])
10  // A task that monitors an external event source
11  | TTMonitorTask    TaskInfo [HtmlTag]
12  // A completed task
13  | TTFinishedTask   TaskInfo
14
15 // Shared node information: task identifiers, labels, debug info etc.
16 :: TaskInfo
17 // Task meta-data for main tasks, assigned user, priority etc.
18 :: TaskProperties

```

Every function of type `Task` generates a (sub) task tree. Combined tasks use their argument tasks to compute the required sub task trees. Because an explanation of task tree generation is impossible without examining the combinators in detail, we will restrict ourselves to a demonstration of their use by means of an example. Let's consider the following simple workflow specification:

```

1 bugReport :: Task Void
2 bugReport = reportBug >>= fixBug
3 where
4   reportBug :: Task BugReport
5   reportBug = enterInformation "Please describe the bug you have found"
6
7   fixBug :: BugReport → Task Void
8   fixBug bug = "bas" @: (showMessageAbout "Please fix the following bug" bug)

```

Figure 2.4 graphically illustrates two task trees that reflect the state of this workflow at two moments during execution. The tree on the left is produced during the execution of the first `reportBug` task. The `bind (>>=)` combinator only has a left branch, which is the `TTInteractiveTask` that contains a user interface

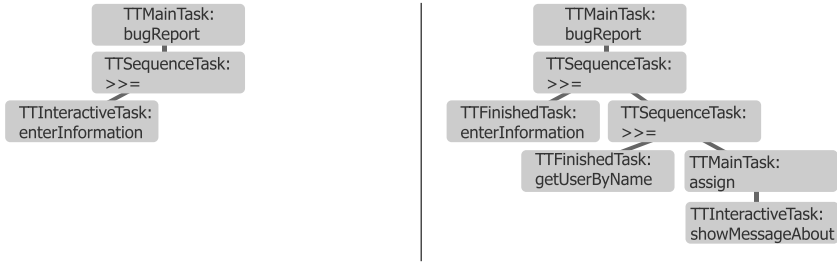


Figure 2.4: Task tree during `reportBug` (left) and `fixBug` (right)

definition for the bug report form. The tree on the right is produced during the execution of `fixBug`. At this point the leftmost branch is reduced to a `TTFinishedTask` and the `@:` has been expanded to a subtree consisting of a bind of some `getUserByName` task, that is finished, and a `TTMainTask` containing the `TTInteractiveTask` with the interface definition for showing the bug report.

The Authentication Service

The `iTask` server maintains a user and role database such that (parts of) workflows can be restricted to users with special roles, and roles may be used to find the right type of worker to do a certain task. The server handles authentication of clients and keeps a database of authenticated time-limited sessions. This service consist of two methods, `/handlers/authenticate` which accepts a username and password and yields a session key to access the other services, and `/handlers/deauthenticate` that can be passed a session key to explicitly terminate a session.

The Workflow Directory Service

In order to initiate new workflow instances, the `iTask` server offers a directory service to browse the available workflow definitions. The server maintains a hierarchic directory of available workflows that are filtered by the roles of a user. The `/handlers/new/list` method yields the list of possible workflows and subdirectories for any given node in the hierarchy. The `/handlers/new/start` method starts a new instance of a workflow definition and returns a task identification number for the top level task of that workflow instance.

The Tasklist Service

Users can find out if there is work for them through the tasklist service. The `/handlers/work/list` method yields a list of all *main tasks* assigned to the current

user along with the meta-data of those tasks. This list is an aggregation of all active tasks in all workflow instances the current user is involved in. Because tasks are often subtasks of other tasks, parent/child relation information is also available in the list entries to enable grouping in a client.

The Task Service

To actually get some work done, users will have to be able to work on tasks through some user interface. Because the tasks are highly dynamic, no fixed user interface can be used. Therefore, the iTask system uses a generic library to generate high-level user interface definitions that are interpreted by the client. The `/handlers/work/tab` method returns a tree structure that represents the current state of a workflow instance. This tree data is used by a client either to render an interface, or to adapt an already rendered interface. When a user updates an interactive control, this method is called with the event passed as argument. This yields a new tree that represents the updated state of the workflow after this event and possibly events from other users. This process is explained in more detail in Section 2.4.

The Property Service

To update the meta-data of a workflow instance, for example to reassign tasks to different users or change their priority, the service `/handlers/work/property` may be used. This service can set any of the meta-data properties of a workflow instance.

2.3.5 The Client System

Although the iTask system focuses on workflow specification and execution on the server, the average end-user will only interact with this server through a client. While the JSON service API is not limited to one specific client, the iTask system provides a default Javascript client built with the ExtJS framework. ExtJS is a Javascript library that facilitates construction of “desktop like” Ajax applications with multiple windows, different kinds of panels, and other GUI components in a web browser. The iTask client runs in any modern webbrowser and provides everything a user needs to view and work on tasks. Figure 2.5 shows a screenshot of the iTask client with multiple tasks opened. The client user interface is divided into three primary areas in a layout that is common in e-mail client applications. This similarity is chosen deliberately to ease the learning of the application. The area on the left of the screen shows a folder hierarchy that accesses the workflow directory service. New workflow instances can be started by clicking the available flows in the folders. The top right area shows a user’s *task list*, and the final main area is the lower right *task area*. In

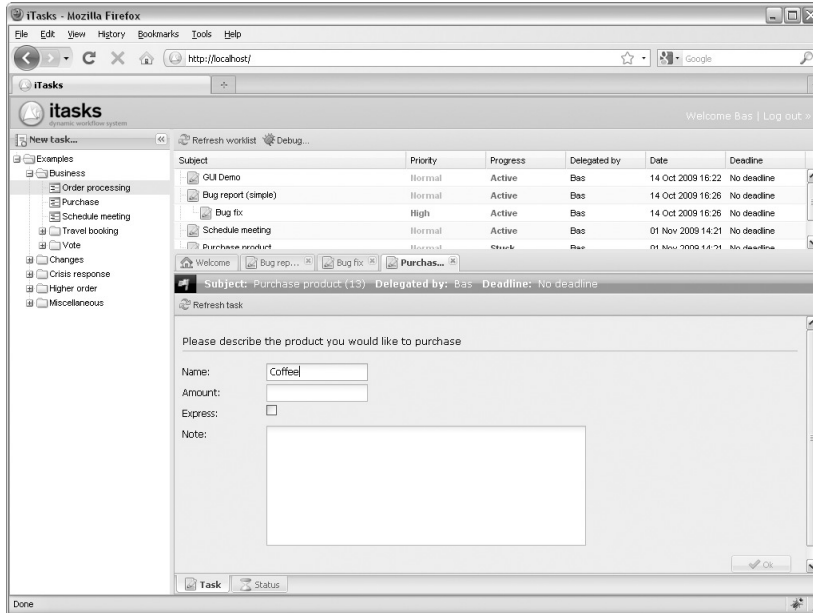


Figure 2.5: The iTask client interface

this part of the interface area users can work on multiple tasks concurrently in a series of tabs. New tabs are opened by clicking items in the task list. The most interesting feature of the client application is its ability to dynamically render and update arbitrary user interfaces defined by the server. It constructs the user interfaces required to work on tasks by interpreting a definition it receives from the server. It then monitors all interactive elements in the interface and synchronizes changes to them with the server in the background. The server processes the user input and responds by sending instructions to adapt the user interface when necessary. A big advantage of this design is that the server is kept synchronized with the client. This way the server can provide immediate feedback or dynamically extend an interface without the need for page refreshes. It also means that tasks can be reassigned at any moment without losing any work.

2.4 Dynamic Generic Web-Interfaces

One of the primary reasons for redesigning the iTask using a different architecture was to improve the user experience for end-users. In this section we show how the new iTask system makes use of the new architecture to operationalize

Please describe the bug you have found

Application:

Version (optional):

Date:

Occurs at: Startup Shutdown Other

Severity:

Description:

Figure 2.6: An automatically generated form

the declarative user interaction primitives of the specification language. For basic tasks like `enterInformation` or `displayMessageAbout` to be operationalized, the iTask system needs to be able to generate forms for entering data and visualizations of data to display. Because user interface issues are an aspect that is abstracted from in the iTask specification language, it is *essential* that its implementation is able to generate satisfactory user interfaces. For *any* type that someone defines in a workflow specification, the system needs to be able to generate forms and renderings that have to have the following properties:

- They need to be layed out in a visually and ergonomically pleasing way.
- They need to react responsively and consistently. The cursor should follow a logical path when using the keyboard to navigate through a form and there must never be unexplainable loss or change of focus.
- They must communicate clearly what is optional and what is mandatory. The forms must ensure that mandatory input is entered.
- They must be able to adapt dynamically depending on the choice of constructor for algebraic data types. It is, for example, simply impossible to generate a static form for entering a list, because the number of elements is unbounded.

The redesign of the iTask system with a service based architecture and stand-alone (Javascript) client as explained in Section 2.3 removes the implicit usability limitations of the original iTask system. It enables a new approach to dynamic interface generation that uses *type generic functions* as can be defined

in Clean [3] on the server and an interpreter in the client that is able to meet the demands stated above.

Figure 2.6 shows the user interface that is generated for the `BugReport` type used in the `enterInformation` task of the `bugReport` example in Section 2.3.4:

```
1 :: BugReport =
2   { application  :: String
3     , version    :: Maybe String
4     , date       :: Date
5     , occursAt  :: BugOccurance
6     , severity   :: BugSeverity
7     , description :: Note
8   }
9 :: BugSeverity = Low | Medium | High | Critical
10 :: BugOccurance = Startup | Shutdown | Other Note
```

The demands stated above are all applicable to this relatively simple type already. It contains both optional and mandatory parts, it has to adapt dynamically when the `Other` constructor is chosen and it has a wide variety of input elements that have to be arranged in a pleasing layout. An attentive reader may even spot that different input controls are used to select a constructor in Figure 2.6 for `BugOccurance` and `BugSeverity`. This choice is not specified explicitly, but is decided by a layout heuristic in the interface generation.

2.4.1 Key Concepts

The `iTask` system generically provides generic user interfaces through the interplay between two type generic functions. The first one, `gVisualize`, generates visualizations of values that are rendered by the client. The second one, `gUpdate`, maps updates in the rendered visualization back to changes in the corresponding values. Before explaining these functions in detail, we first introduce the key concepts underlying their design.

Visualizations

Visualizations in the `iTask` system are a combination of pretty printing and user interface generation. The idea behind this concept is that they are both just ways of presenting values to users, whether it is purely informational or for (interactive) editing purposes. The generic user interface library therefore integrates both in a single generic function. Furthermore, most types of visualizations can be coerced into other types of visualizations. For example: a value visualized as text can be easily coerced to an `Html` visualization, or vice versa. The library offers functions for such coercions. There are six types of visualizations currently supported as expressed by the following type:

```

1 :: VisualizationType
2   = VEditorDefinition
3   | VEditorUpdate
4   | VHtmlDisplay
5   | VTextDisplay
6   | VHtmlLabel
7   | VTextLabel

```

And four actual visualizations:

```

1 :: Visualization
2   = TextFragment String
3   | HtmlFragment [HtmlTag]
4   | TUIFragment TUIDef
5   | TUIUpdate TUIUpdate

```

The `VHtmlDisplay` and `VTextDisplay` constructors are pretty print visualizations in either plain text or `Html`. The `VHtmlLabel` and `VTextLabel` constructors are summaries of a value in at most one line of text or `Html`. Labels and display visualizations use the same constructor in the `Visualization` type. The `VEditorDefinition` and `VEditorUpdate` visualizations are explained in the next two subsections.

User Interface Definitions

When a value is to be visualized as an editor, it is represented as a high-level definition of a user interface. These `TUIDef` definitions are delivered in serialized form to a client as part of a `TTInteractiveTask` node of a task tree. A client can use this definition as a guideline for rendering an actual user interface. The `TUIDef` type is defined as follows:

```

1 :: TUIDef
2   = TUIButton TUIButton
3   | TUINumberField TUINumberField
4   | TUITextField TUITextField
5   | TUITextArea TUITextArea
6   | TUIComboBox TUIComboBox
7   | TUICheckBox TUICheckBox
8   ...
9   | TUIPanel TUIPanel
10  ...
11 :: TUIButton =
12  { name      :: String
13    , id      :: String
14    , text    :: String
15    , value   :: String
16    , disabled :: Bool
17    , iconCls :: String
18  }

```

```
19 :: TUIPanel =
20   { layout      :: String
21   , items       :: [TUIDef]
22   , buttons     :: [TUIDef]
23   ...
24   }
```

Components can be simple controls such as buttons described by the `TUIButton` type on line 11, or containers of other components such as the `TUIPanel` type on line 19 that contains two containers for components: One for its main content, and one additional container for action buttons (e.g. "Ok" or "Cancel").

User Interface Updates

To enable dynamic user interfaces that adapt without replacing an entire GUI, we need a representation of incremental updates. This is a visualization of the difference between two values expressed as a series of updates to an existing user interface.

```
1 :: TUIUpdate
2 = TUIAdd TUIId UIDef
3   | TUIRemove TUIId
4   | TUIReplace TUIId UIDef
5   | TUISetValue TUIId String
6   | TUISetEnabled TUIId Bool
7 :: TUIId ::= String
```

New components can be added, existing ones removed or replaced, values can be set and components can be disabled or enabled. The `TUIId` is a string that uniquely identifies the components in the interface that the operation targets. The one exception to this rule is the `TUIAdd` case, where the `TUIId` references the component *after* which the new component will have to be placed.

User interface updates are computed by a local structural comparison while traversing an old and new datastructure simultaneously. This ensures that only substructures that have changed are being updated.

Data Paths

In order to enable updating of values, it is necessary to identify substructures of datastructure. A `DataPath` is a list of integers (`::DataPath ::= [Int]`) that are indexes within constructors (of arity > 0) when a datastructure is being traversed. Figure 2.7 show some example `DataPaths` for a simple binary tree. `DataPaths` are a compact, yet robust identification of substructures within a datastructure.

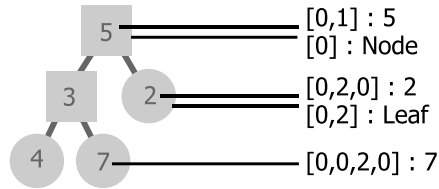


Figure 2.7: Data paths for a value of type `::Tree = Node Tree Int Tree | Leaf Int`

Data Masks

When a datastructure is edited, it is possible that during this editing, parts of the structure are temporarily in an “invalid” state. For example when an element is added to a list: between the structural extension of the list and the user entering the value of the new element, the list is in a state in which one of its elements has a value, but that is not entered by the user. To indicate which parts of a datastructure have been accessed by a user we use the `DataMask` concept. A `DataMask` is simply a list of all paths that have been accessed by a user (`::DataMask ::= [DataPath]`). This additional information is used to enhance usability by treating components that have not been touched by a user different from those that the user has already touched. For example, validation of only those fields in a form that have already been filled out.

2.4.2 The Big Picture

With the key concepts explained, we can now sketch the big picture of how user interfaces of interactive tasks are handled. This process consists of three main steps:

1. An initial user interface definition (`TUIDef`) representing the current value of a datastructure and its mask is generated by a generic function on the server. This definition is rendered by the client and event handlers are attached to interactive components to notify value changes.
2. When a user changes an interactive component, an encoding of this change and the datapath of the component are sent back to the server and interpreted by another type generic function that updates the datastructure and mask to reflect the change.
3. The updated datastructure is compared to its previous value and if there is a structural difference, a list of `TUIUpdate` is computed and sent back to the client. The client interprets these instructions and modifies the interface accordingly.

In the next section we will explain some of the machinery behind those steps. For reasons of brevity we do not go into implementation details, but explain the key datastructures and type signatures of key functions instead.

2.4.3 Low Level Machinery

The core machinery of the library consist of two generic functions: `gVisualize` and `gUpdate`. Instances of these functions for concrete types can be automatically derived. Because these functions have been designed favoring pragmatism over elegance, the library exposes them through a set of wrapper functions:

```
1 //Visualization wrappers (under condition that gVisualize exists for type a)
2 visualizeAsEditor      :: String DataMask a → ([TUIDef],Bool)
3                       | gVisualize{[*]} a
4 visualizeAsHtmlDisplay :: a → [HtmlTag]
5                       | gVisualize{[*]} a
6 determineEditorUpdates :: String DataMask DataMask a a → ([TUIUpdate],Bool)
7                       | gVisualize{[*]} a
8 ...
9 //Update wrappers (under condition that gUpdate exists for type a)
10 updateValueAndMask    :: String String a DataMask *World → (a,DataMask,*World)
11                       | gUpdate{[*]} a
12 ...
```

Tasks such as `enterInformation` use the `visualizeAsEditor` wrapper to create the content of a `TTInteractiveTask` node in the task tree. All interactive components are given an identifier derived from their data path within the data structure. This enables the client to send back updates when such a component is updated. When a client sends an event to the server, the `updateValueAndMask` wrapper is used to process the update. Its first two arguments are a string representation of the data path, and a string representation of the update. The last parameter is the unique world. `Clean` uses uniqueness typing to facilitate stateful functions by threading an abstract `World` value. The main reason that updates are impure, is that it enables impure specializations for specific types. For example when updating a `Maybe Date` from `Nothing` to `Just`, the current date can be set as value. After updating a value and mask, the `determineEditorUpdates` wrapper is used to create task content containing an incremental update for the client GUI. Although the generic functions are never called directly, and for normal use only *derived* for types, we conclude this section with a brief overview of their

type signatures and arguments to give an impression of what goes on under the hood.

```

1 generic gVisualize a ::
2   (VisualizationValue a) (VisualizationValue a) VSt → ([Visualization], VSt)
3
4 :: VisualizationValue a = Value a DataMask | VBlank
5 :: VSt =
6   { vizType           :: VisualizationType
7     , idPrefix        :: String
8     , label           :: Maybe String
9     , currentPath     :: DataPath
10    , useLabels        :: Bool
11    , onlyBody         :: Bool
12    , optional         :: Bool
13    , valid            :: Bool
14    }

```

The first two arguments are wrapped values of type `a` with their mask, or an undefined blank. The last argument that is both input and output of `gVisualize` is the visualization state. This state contains all parameters relevant to the visualization and is used to keep track of global properties. The `optional` field in the structure is used to mark parts of editor visualizations as optional. A specialization of `gVisualize` for the `Maybe a` type sets this field to true, and then produces a visualization of type of `a`. When a visualization of an optional value that is `Nothing` needed, there is no value of type `a` available. In that case `VBlank` values are used. The `valid` field of `vSt` is used to validate mandatory fields. It is updated at each interactive element and set to `False` when a non-optional field is not masked. This validation is used to disable completion of a task until its form has been filled out completely.

```

1 generic gUpdate a      :: a *UST → (a, *UST)
2 :: *UST =
3   { mode              :: UpdateMode
4     , searchPath      :: DataPath
5     , currentPath     :: DataPath
6     , update          :: String
7     , consPath        :: [ConsPos]
8     , mask            :: DataMask
9     , world           :: *World
10  }
11 :: UpdateMode = UDSearch | UDCreate | UDMask | UDDone

```

The `gUpdate` function traverses a datastructure recursively and at each point transforms the value and state according to one of four modes. In `UDSearch` mode, the `currentPath` path field is compared to the `searchPath` field and `update` is

applied when they are equal. The mode is then set to `UDDone` and the `mask` field is updated to include the value of `currentPath.` In `UDDone` mode, the function does nothing and is just an identity function. When a constructor of an algebraic data type is updated to one that has a non-zero arity, the `gUpdate` function needs to be able to produce default values for the substructures of the constructor. It uses its `UDCreate` mode to create these values. In this mode, the `gUpdate` ignores its input value and returns a default value. The last mode is the `UDMask` mode, which adds the paths of all substructures to the `mask` as it traverses the datastructure. This is used to compute a complete mask of a datastructure.

2.5 Related Work

The `iTask` system is a workflow management system, and is therefore comparable with other WFMSs. However, unlike many contemporary WFMSs (e.g. YAWL, WebSphere, Staffware, Flower, Bonita), the `iTask` system does not use a graphical formalism for the specification of workflows, but uses a compact combinator language embedded in a general purpose functional language instead. Although the `iTask` system is a WFMS, many web applications can be considered workflow support systems in some way or another. Therefore one could also view the `iTask` system as a more general framework for (rapid) development of web applications. This makes it comparable with other web development frameworks found in functional languages like WASH/CGI [75] and HAppS [29] in Haskell, XCaml in OCaml, or the frameworks available in dynamic scripting languages like Rails [70] in Ruby or Django [13] in Python. While these frameworks aim to simplify the development of any type of web application, the `iTask` system will only be suitable for applications that can be sensibly organized around tasks.

The final body of work that may be classified as related is not so much related to the `iTask` system itself but rather to its new generic web visualization library. Other functional web GUI libraries exist like the WUI combinator library in Curry [24], or the `iData` toolkit [60] in Clean that powered previous `iTask` implementations. The new `iTask` visualization library differs from those libraries in that it makes use of an active Ajax client, in this case built with the ExtJS framework [16]. This gives the generated editors more control over the browser than is possible with plain `Html` forms, hence enabling the generation of more powerful “desktop-like” user interfaces. However, the `iTask` client is a single application that interprets instructions generated on the server and is not to be confused with client side application frameworks such as Flapjax [53]. Such frameworks could be used as a replacement for ExtJS in alternative `iTask` clients.

2.6 Conclusions

In this paper we have presented a new implementation of the iTask system. This new implementation uses a service-based architecture combined with an active client. This approach enables the generation of more user-friendly interfaces for end-users without compromising the declarative nature of the iTask language. Although seemingly superficial, improved usability is a crucial aspect of the implementation of the iTask workflow language, because the iTask system generates executable systems solely from a workflow specification and *nothing else*. Hence, the generation quality largely determines the usefulness of the language. A direct consequence of, and a primary motivation for, this work is that it enables case study and pilot research to validate the effectiveness of the function combinator approach to workflow modeling used by the iTask system in scenarios with real end-users. Not surprisingly, such realistic case studies in the context of supporting disaster response operations are planned for the coming years.

More information, examples and downloads of the iTask system can be found at: <http://itasks.cs.ru.nl/>.

3 Getting a Grip on Tasks that Coordinate Tasks

Workflow management systems (WFMS) are software systems that coordinate the tasks human workers and computers have to perform to achieve a certain goal. The tasks to do and their interdependencies are described in a Workflow Description Language (WDL). Work can be organized in many, many ways and in the literature already more than hundred of useful workflow patterns for WDL's have been identified. The iTask system is not a WFMS, but a combinator library for the functional language Clean to support the construction of WFMS applications. Workflows can be described in a compositional style, using pure functions and combinators as self-contained building blocks. Thanks to the expressive power of the underlying functional language, complex workflows can be defined on top of just a handful of core task combinators. However, it is not sufficient to define the tasks that need to be done. We also need to express the way these tasks are being supervised, managed and visualized. In this paper we report on our current research effort to extend the iTask system such that the coordination of work can be defined as special tasks in the system as well. We take the opportunity to redesign editors which share information and the combinators for defining GUI interfaces for tasks, such as buttons, menu's and windows. Even though the expressiveness of the resulting system increases significantly, the number of core combinators can be reduced. We argue that only two general Swiss-Army-Knife higher order functions are needed to obtain the desired functionality. This simplifies the implementation significantly and increases the maintainability of the system. We discuss the design space and decisions that lead to these two general functions for constructing tasks.

3.1 Introduction

Workflow management systems (WFMS) are software systems that coordinate, generate, and monitor tasks performed by human workers and computers. A concrete workflow ensures that essential actions are performed in the right order. The purpose of the iTask system [61] is to support the construction of WFMS applications. It distinguishes itself from traditional WFMSs. First, the iTask system is actually a monadic combinator library in the pure and lazy functional programming language Clean. The constructed WFMS application

is embedded in `Clean` where the combinators are used to define how tasks can be composed. Tasks are defined by higher-order functions which are pure and self contained. Second, most WFMSs take a workflow description specified in a workflow description language (WDL) and generate a partial workflow application that still requires substantial coding effort. An `iTask` specification on the other hand denotes a full-fledged, web-based, multi-user workflow application. It strongly supports the view that a WDL should be considered as a complete specification language rather than a partial description language. Third, despite the fact that an `iTask` specification denotes a complete workflow application, the workflow engineer is not confronted with boilerplate programming (data storage and retrieval, GUI rendering, form interaction, and so on) because this is all dealt with using generic programming techniques. Fourth, the structure of an `iTask` workflow evolves dynamically, depending on user-input and results of subtasks. Fifth, in addition to the host language features, the `iTask` system adds higher-order tasks (workflow units that create and accept other workflow units) and recursion to the modelling repertoire of workflow engineers. Sixth, in contrast with the large catalogue of common workflow patterns [1], `iTask` workflows are captured by means of a small number of core combinator functions.

In this paper we reflect on these core combinators and the functionality they offer. Although complex workflows can be defined in a declarative style, one would like to have more flexibility in controlling the tasks one is working on. For instance, when a task is delegated, someone might want to monitor its progress. In the current system the delegator gets this information and she also obtains the power to change the properties of the delegated task, such as its priority, or, she can move the task to the desk of someone else. This is often useful, but is not always what is wanted. Perhaps one would like to inform other people involved as well. One also would like to define what kind of information is shown to a particular person and define what a manager can do with the tasks she is viewing. Controlling tasks can be seen as tasks as well and one would like to have combinators to programme their behaviour. In particular one would like to define control interfaces that show what goes on and which can be used to manage the tasks involved. The extended `iTask` system described in [55] appears to be a good starting point for defining such interfaces. In that paper we extended the system with new combinators given the ability to define GUIs for tasks. Furthermore we showed that it is possible to share information between tasks. Tasks can communicate with each other by modifying shared information.

Adding all these extensions to the `iTask` system can easily lead to a system with an excessive number of core combinators. This leads to high maintenance costs and hampers formal reasoning. Fortunately, the desired functionality can be obtained with only a very few powerful combinators with which the simplicity

of the system can be retained and the maintainability can be improved. It is the thesis of this paper that we can do with only two new general purpose elements.

The remainder of this paper is organized as follows. First, we describe the current core `iTask` system (Section 3.2) and explain its usage and shortcomings by means of small, yet illustrative examples (Section 3.3). Based on this analysis we identify the requirements that should be satisfied by the new `iTask` system and argue that they can be realized with only two new general constructs with which all current constructs can be defined (Section 3.4). In the conclusions we briefly discuss the current situation (Section 3.5).

3.2 The `iTask` Core System

In this section we give a brief overview of the `iTask` system. The kernel of the `iTask` system consists of basic tasks and combinator functions for combining tasks. On top of them, an API is defined. This API offers some notational convenience and allows to define workflows in a more verbose style to enhance the communication with domain experts, which are not likely to be functional programmers. In this paper we abstract from this API and focus on the `iTask` kernel.

3.2.1 Basic Tasks

Basic tasks are units of work of the opaque, parameterized type `Task a`: the type parameter `a` is the type of the result value that is committed to the workflow when the task has finished. In principle, any unit of work in daily life can be modeled as a basic task. It can be a call to a service on a web server, a system call, or a form to be filled in by a worker in a browser. The latter is the most interesting one in this context, since human interaction is a key feature of any workflow system. In this paper we restrict ourselves to this basic task only, for which we provide the function `edit`.

```
1 edit :: String a → Task a | iTask a
```

Note that in `Clean` the arity of functions is shown explicitly by separating argument types by spaces instead of `→`. An editor is created with `edit prompt va`. The `prompt` argument provides the worker with information about the purpose of this task. When applied to an initial `va` of some type `a`, it creates a GUI in which the worker can inspect and alter the given value arbitrarily many times. An editor can create and handle such a GUI for any first-order type `a`. It uses a set of type indexed generic functions (hence the context restriction `| iTask a`) which are derived by the compiler automatically. The `iTask` system guarantees that only values of type `a` can be created. This continues until the worker decides to commit the value to the workflow, which terminates the task `edit prompt va`.

Example

What follows is a very small `iTask` example: a task `enterInt` using the `edit` function to create a form for filling in an `Integer` number. The generic function `initialValue` yields an initial value for any first order type.

```

1 module example
2
3 import iTask
4
5 Start world = startEngine [workflow "Integer form" enterInt] world
6
7 enterInt :: Task Int
8 enterInt = edit "Please, fill in form" initialValue

```

To turn this task description into an executable WFMS one has to import the `iTask` library. The main function in `Clean` is called `Start` which obtains a (unique) world as argument which is used for the pure communication with the impure outside world. The library function `startEngine` takes a list of workflow specifications and creates a web-based workflow system for it (see Figure 3.1). Any task can be promoted to become a workflow with the function `workflow`. Such a workflow is added to the list of workflows in the left workflow start-pane. A workflow can be started by the worker arbitrary many times just by clicking on its icon. The tasks a worker has to do appear in the task-list pane, similar to a list of incoming emails. By clicking on an item in the list, a task-pane is opened allowing a worker to work on her tasks in arbitrary order.

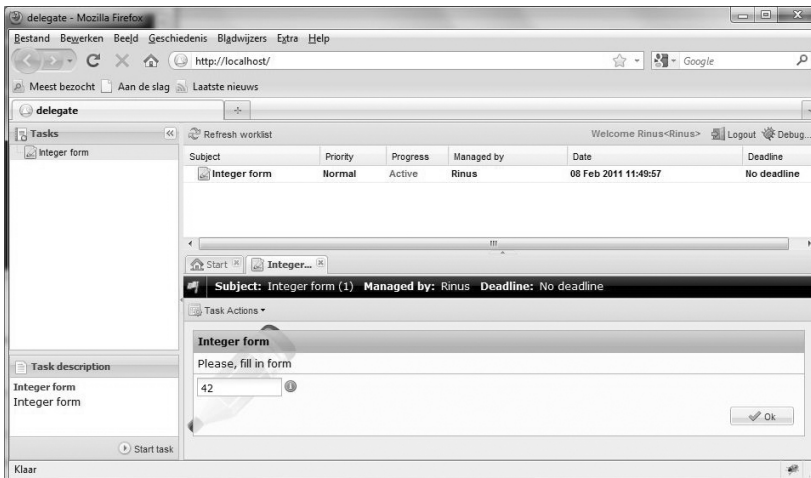


Figure 3.1: A screen-shot of the `iTask` browser interface

The `edit` function can be used on any first-order type. In the example below we show the form it creates for a list of `Persons`, where `Person` is some user-defined record type. One only has to change the type of the application of `edit`. Furthermore one has to ask the compiler to **derive** the generic functions for the types involved. The resulting form is shown in Figure 3.2.

Figure 3.2: The form generated for editing a list of `Persons`

```

1 module example2
2
3 import iTask
4
5 :: Person = { firstName  :: String
6              , surName   :: String
7              , dateOfBirth :: Date
8              , gender    :: Gender
9              }
10 :: Gender = Male | Female
11
12 derive class iTask Person, Gender
13
14 Start world = startEngine [workflow "Person form" enterPersons] world
15
16 enterPersons :: Task [Person]
17 enterPersons = edit "Please, fill in the form" initialValue

```


3.2.2 Core Combinators

In this paper we focus on the `iTask` core combinators. The semantics is formally defined in [64].

```

1 // assigning properties to a task:
2 (@:)   infixl 5 :: p (Task a) → Task a | property p & iTask a
3
4 // sequencing of tasks with a monad:
5 (>>=)  infixl 1 :: (Task a) (a → Task b) → Task b | iTask b
6 return      :: a                → Task a | iTask a
7
8 // defining parallel tasks: parallel-or and parallel-and:
9 (-||-)  infixr 3 :: (Task a) (Task a) → Task a      | iTask a
10 (-&&-)  infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a
11                                           & iTask b

```

Properties can be assigned to a task by using the `@:` combinator. In general, the properties that can be set are captured by the type class `property`. Examples of predefined properties are: an identification of a worker or group of workers to which the task is assigned, the priority of a task, and its deadline. One can assign user-defined properties (type `String`) as well. The properties are inherited by all subtasks of a task, unless other properties are assigned to them via a `@:` operator. We call `@:`-annotated tasks *main tasks*.

To compose tasks sequentially, the monadic combinators `return` and `>>=` [79] are provided. The task `return va` succeeds immediately and commits its value `va`. In `ta >>= λva → tb`, the task `ta` is evaluated first. When this task commits its value, say `va`, it is passed to `tb` to compute the next task to be executed.

To compose tasks in parallel, the combinators `-||-` and `-&&-` are provided. A task constructed using `-||-` is finished as soon as either one of its subtasks is finished, returning the result of that task. The combinator `-&&-` is finished as soon as both subtasks are finished, and pairs their results.

The `iTask` system actually offers a couple of additional core combinators: namely for managing workflow processes and for exception handling. However, their functionality is not important for this paper.

In the next section we give some examples of `iTask` workflow specification using the core combinators described above, and use these examples to explain how the combinators can be further improved.

3.3 The Expressive Power and Limitations of the Combinators

With the iTask core combinators, complex workflows can be defined which cannot be expressed in traditional WFMS. However, there is still room for improvement. We present some examples, explain what their effect is, and discuss what is missing.

3.3.1 Coordination of Tasks

Our first example is a higher order task: we define a task which delegates a given task to someone else. Notice that in most classical WFMSs, such higher order tasks cannot be defined.

```

1 delegate :: (Task a) → Task a | iTask a
2 delegate task
3   =           selectUsers
4   >>= λworker → worker @: task
5   >>= λresult → edit "Check result:" result
6
7 Start world = startEngine [workflow "Delegate" (delegate enterPersons)] world

```

In the `delegate` task there are initially two people involved: a *delegator* delegating work, and a *worker*, to whom the work is delegated. There are three sub-task steps in this `delegate` task, sequentialized by the `bind` combinator (`>>=`). When the task `delegate` is performed by the delegator, she first has to select a worker. The library function `selectUsers` displays a list of administrated users the delegator can select from. In the next step the task to delegate is assigned (`@:`) to the chosen worker. When the worker has finished the task, its result is shown to the delegator again. She can correct the returned result with the `editor`, and when she is happy and finishes, the whole delegation task is completed as well. The `delegate` workflow is a clear and concise specification which can be applied to *any* task. In this particular example, `enterPersons` is the task being delegated to the worker.

Yet there is something missing in this specification as well. It is clear what the worker has to do (see Figure 3.2), but what do we show to the delegator in the meantime? When the work is being delegated, the delegator might be interested in how the work is proceeding. Perhaps she also wants to do something with the information she sees.

In the first version of the iTask system, we just informed the delegator in the task-pane that her delegation task is waiting for the worker to finish. In the current version, we have used the capability to change tasks under execution [64] to turn this passive role of the delegator into an active one. The delegator is offered a predefined control screen as displayed in Figure 3.3.



Figure 3.3: Default task pane of a waiting task

It gives the delegator information about the properties of the delegated task, such as its priority, who is working on it and the last time the task has been worked on. Moreover it gives the delegator the capability to *coordinate* the delegated task: she can change its priority and she can replace the worker by another one, including herself. When a change is made, the worker sees this change immediately when an event is committed to the server. When the task is transformed to some other worker, he can continue with the work of the previous worker, since all the work done so far is retained.

In this example we observe that there are actually *two* tasks involved when a main task is created. The task being assigned is explicitly defined by the workflow engineer, but the coordinating task is predefined and fixed in the iTask library. Since the iTask system is intended as a system for constructing WFMS's, it would be better if the coordinating meta-task is not fixed by the underlying system but can be defined as a task by the workflow engineer as well. This implies that such a task must continuously be provided with the actual status information of the main tasks involved.

With this status information one can display a view to whom it concerns. For instance, in the current system, the delegator is by default turned into a manager, but perhaps this is not desirable at all. It might also be possible that several people are interested in the progress of the delegated task. Consider the following example:

```

1 Start world
2 = startEngine [workflow "Delegate 2"
3               (delegate (delegate enterPersons))] world

```

Here we delegate the delegation-task thus introducing potentially two workers who might be interested in the progress of the actual work.

Hence we need to provide the programmer with a combinator with which she can define arbitrary coordination tasks which have a view on the ongoing work and enable to change their properties as wanted.

3.3.2 Sharing of Information

In the previous section we have identified the need to allow coordination tasks to continuously monitor and alter task (properties). This need extends in a natural way to arbitrary tasks and arbitrary information. Michels *et al.* [55] show that some tasks do need to constantly exchange information while one is working on it. This extends the data flow that is dictated by the current workflow combinators in which information is only propagated to tasks once the information-producing task has been terminated and thus committed its value to the workflow. An appealing example is chat.

```

1 :: Note = Note String
2
3 :: View m v = { viewFrom :: m → v, viewTo :: v → m → m }
4
5 chat :: User User → Task Void
6 chat user1 user2
7   =       createDB (Note "", Note "")
8   >>= λref → (user1 @: editShare ("Chat with "++user2) a ref)
9             -&&-
10          (user2 @: editShare ("Chat with "++user1) b ref)
11 >>= λ_ →   return Void
12 where
13   a = { viewFrom = λ(note1, note2) → (Display note2,note1)
14         , viewTo  = λ(_,note1)  ( _, note2) → (note1, note2)
15         }
16   b = { viewFrom = λ(note1, note2) → (Display note1,note2)
17         , viewTo  = λ(_,note2)  (note1, _) → (note1, note2)
18         }

```

In this example, two workers chat with each other continuously. To make this possible, the text produced by both (of type `(Note, Note)`) is stored in a database. It serves as a shared model, and both workers may change the model at the same time. Each worker, however, has its own view on the shared model. For this purpose one has to describe the mapping between model and view (`viewFrom`) and backwards (`viewTo`) also known as a lens [7]. This implements the well known model-view-controller paradigm [38]. In the chat example, the predefined type `Display` is used to prevent one worker to change the information typed in by the other. For a screen shot see Figure 3.4. Hence, an editing conflict caused when more than one worker changes the same information at the same time cannot arise in this particular example due to the well chosen view. In general such editing conflicts are possible. The system can prevent the database for becoming inconsistent by producing an error message when somebody is trying to change data which is not up-to-date.

To make this all possible, we need a new kind of editor, like `editShare`, which reads in the current model stored in the database, and converts this information

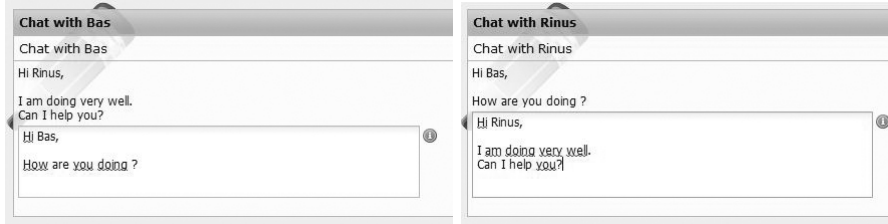


Figure 3.4: Workers Rinus and Bas chatting with each other

to a view to show in the client. Every time a worker makes a change, the view is converted back to the model and stored in the database. The view others have on this information has to be updated accordingly.

The `editShare` editor has the following type:

```
1 editShare :: String (View m v) (Ref m) → Task (Maybe m)
2                                     | iTask m & iTask v
```

It requires a `String` for prompting, a view of type `v` on a model of type `m` and a reference to the database where the shared model of type `m` is stored. Only if the `editShare` editor finished in a valid state, the final value `mv` is returned as `Just mv`, otherwise `Nothing` is returned.

The ability to share information and provide a specific view on this information is required to define tasks that coordinate other tasks. In this case the state information of the tasks to coordinate has to be provided to the coordinating tasks.

3.3.3 Adding GUI Elements

With the standard core editor a form can be generated for any first order type. One can change the values in the form as often as desired. The system ensures that the form can only be filled with values of the demanded type. The standard “OK” button can therefore only be pressed when all required parts of the form have been filled in properly.

Clearly one would like to have the possibility to attach an arbitrary number of buttons to an editor instead of just one. Michels *et al* [55] extend the `iTask` system and enrich editors with GUI elements such as buttons and menus. Furthermore, several editors can be active at the same time each running in their own window. An example using such an enriched editor, `editA` (for *edit action*), is (see also Figure 3.5):

```

1 editA :: String [(Action, (Maybe a) → Bool)] a
2     → Task (Action, Maybe a) | iTask a
3
4 enterPerson :: Task (Maybe Person)
5 enterPerson
6 = editA "Please, fill in form" myActions initialValue
7 >>= λ(event,mbr) → return mbr
8 where
9   myActions      = [(ActionOk,ifValid), (ActionCancel,always)]
10
11   ifValid (Just _) = True
12   ifValid _       = False
13
14   always _        = True

```

The idea is to attach a list of action-predicate pairs to an editor, as shown in the type of `editA`. The predicate defines when the corresponding action can be chosen. An action such as “OK” can only be chosen if a complete form has been filled in, but one can also specify that the entered value has to satisfy additional requirements. Other actions, such as `Cancel`, should always



Figure 3.5: Editor for type `Person` with `Ok` and `Cancel` buttons.

be possible, regardless what has been entered. It can therefore no longer be guaranteed that such an editor will always return a proper value. Hence, `editA` returns the chosen action and `Maybe a` value. In `enterPerson` two actions are attached to the editor: `ActionOk` which can only be chosen if the form has been filled in completely (guarded with `ifValid`), and `ActionCancel` which can always be chosen (guarded with `always`).

It is better to structure actions using menus when there are many of them. This can be done via task annotations using the operator `<<@`. Here one can define a mapping between actions and menu items. For the actions not mentioned in this mapping buttons are generated. Here we adapt `enterPerson` to use a menu.

```
1 enterPerson :: Task (Maybe Person)
2 enterPerson
3 =   editA "Please, fill in form" myActions initialValue
4     <<@ myMenu
5     >>= λ(event,mbr) → return mbr
6 where
7   myActions = [(ActionOk,ifValid), (ActionCancel,always)]
8   myMenu    = [Menu "Edit"
9                [MenuItem ActionCancel (Just cancelHotkey)
10               ] ]
11   cancelHotkey = {key=C, ctrl=True, alt=False, shift=False}
```

In [55] Michels *et al* propose a special combinator for creating multiple editors in parallel each running in their own window. As we will see in Section 3.4, we conjecture that all the different types of editors presented so far can be combined into one. Similarly, we can also combine the different ways of creating parallel tasks in one combinator.

3.3.4 Swiss-Army-Knife Parallel Combinator

The need for more functionality does not necessarily imply that more combinators are required. By using higher order functions, Swiss-Army-Knife combinators can be defined, that strongly reduce the number of needed core combinators. In the current `iTask` system, the parallel combinator is one such example:

```
1 parallel :: ([a]→Bool) ([a]→b) ([a]→b) [Task a] → Task b | iTask a & iTask b
```

For instance, the core combinators `-||-` and `-&&-` (Section 3.2.2) can be replaced by suitable parametrization of `parallel`. The function `parallel predOK someDone allDone taskList` takes a list of tasks (`taskList`) to be executed in parallel, a predicate (`predOK`), and two conversion functions (`someDone` and `allDone`). Whenever a member of `taskList` is finished, its result is collected in a list `results` of type `[a]`, maintaining the order of tasks. Now `predOK results` is computed to determine whether `parallel` should complete, in which case the result is computed by `someDone results`. When all parallel tasks have run to completion, and `predOK` is still not satisfied, then `parallel` also completes, but now with result `allDone results`. We can define `-||-` and `-&&-` as follows:

```
1 (-||-) infixr 3 :: (Task a) (Task a) → Task a | iTask a
2 (-||-) ta1 ta2 = parallel (not o isEmpty) first undef [ta1, ta2]
3 where
4   first [a] = a
5
6 (-&&-) infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b
7 (-&&-) ta tb = parallel (const False) undef all [ta >>= Left, tb >>= Right]
8 where
9   all [Left a,Right b] = (a,b)
```

Although a Swiss-Army-Knife combinator such as `parallel` can be used to define many different kinds of parallel behaviours, there is room for improvement here as well. With `predOK` one can freely define when the parallel tasks can be stopped, but perhaps one also needs to be able to start new tasks dynamically, because more work is required.

Also there seem to be different categories of work to be done in parallel. One category is formed by tasks different people work on in parallel, such as in the chat example. But one can also think of parallel tasks one person works on, each task running in its own window being part of one and the same GUI-application.

3.4 Redesigning the Core System

In the previous section we have identified some shortcomings of the current `iTask` core system as presented in Section 3.2. One would like to have a more general basic task editor that can be used for ordinary tasks as well as for the coordination of tasks. Furthermore, one would like to have a more general applicable combinator for defining parallel tasks. In this section we argue that it is possible to identify two such general purpose new elements.

3.4.1 Basic Editor Task Revisited

The basic `edit` task, as defined in Section 3.2.1, lacks some functionality needed to define coordination tasks. In Section 3.3.2 and Section 3.3.3, we gave examples of extended basic tasks with additional functionality.

As for task combinators the different variants of editor tasks can be seen as special cases of one Swiss-Army-Knife editor task. For instance, a task not using actions is actually a special case only using the `Ok` action which can only be chosen if the editor is in a valid state. Working on local data can be seen as a special case of working on a shared database, which is only used by a single task and deleted afterwards. A last example is that tasks not using a custom defined view, actually use the identity view (defined as `{viewFrom = id, viewTo = const}`). Summarized a Swiss-Army-Knife editor task has to meet the following criteria:

1. Edited data can be shared by an arbitrary number of editor tasks, which are possibly carried out by different workers. The system ensures that the data is kept in a consistent state by detecting and reporting edit conflicts.
2. It is possible to edit only a part of the data given to the task. Also the representation shown to the worker might be different than the original data model. This can be achieved by using functionally defined views.
3. An arbitrary number of actions can be attached to each editor task. They are triggered either by buttons or menus which structure is given by

annotating the task. A predicate is used to define when an action can be triggered. The task only returns a value if the editor stopped in a valid state.

We are currently implementing all tasks for user interaction in the `iTask` library (which also includes special tasks not discussed in this paper, for example for making choices), using the same underlying general editor implementation. In the actual implementation some optimizations might be considered. For example, not storing a separate database if it is only used by a single task.

3.4.2 Core Combinators Revisited

In Section 3.3.4, we have shown how one single combinator, `parallel`, can be used to create the derived combinators `-||-` and `-&&-`. We have argued in Section 3.3.1 that delegating work is also a form of parallel task creation. The current shortcoming of delegation is that the `iTask` system has predefined behaviour to control and coordinate these tasks. The workflow engineer should be able to specify the means of control as (arbitrarily many) additional tasks that coordinate these tasks. We hypothesize that these forms of parallel behaviour can be captured with a single, more general combinator. The combinator needs to meet the following criteria:

1. The number of tasks in the current `parallel` combinator remains constant, and `parallel` can only enforce early termination, not the extension of new tasks. The number of tasks in a parallel setting should not be fixed once and for all, but should adapt to the needs of the current situation.
2. The tasks within the current `parallel` combinator simply perform their duty and as such do not interfere with each other (except ofcourse when using shared communication). Next to these *regular* tasks we introduce *control* tasks. These are also tasks, but, being control tasks, they ‘edit’ the collection of parallel tasks. In this way, we can replace the predefined behaviour of task delegation and instead leave it to the workflow engineer whether or not to use a predefined control delegation-task or introduce a (number of) custom control task(s).
3. Because the number of both regular and control tasks varies during the evaluation of a parallel group, we need to *share* information about the state of the parallel group. Access to this state is restricted to control tasks only, which is easily achieved using the strong type system.
4. In the current `parallel` combinator, control is limited to *either* early completion (computed by `predOK`) in which case the final task result was computed by `someDone` *or* full completion in which case the final result was computed by `allDone`. In the more general case, we need to decide how to

continue whenever a regular or control task runs to completion. Again, this should not be computed by the regular tasks. Instead, we need a function that knows which task has completed, and hence has a result value that needs to be accumulated in the shared state. In addition, this function can decide what should happen with the group of parallel (control and regular) tasks: tasks can be suspended and resumed, they can be removed, replaced, and new (control and regular) tasks can be added to the group of parallel tasks. It is clear that this functionality subsumes the current behaviour of `parallel`, and adds behaviour that was inexpressible before.

5. The final part that should be abstracted from is the arrangement, or layout, of the generated GUIs of the (control and regular) tasks. In the current `iTask` system a distinction is made between a parallel form for tasks that can, in principle, each be delegated to other workers and a parallel form for tasks which GUI should be merged into one single presentation. In order to abstract from this, it is better to parameterize the new parallel combinator with a function that describes how the component GUIs of (control and regular) tasks should merged.

We are currently experimenting with a single `parallel` combinator that meets the above criteria. With this combinator we hope to express all other task combinators as special cases. This should aid the development of a formal framework of the new `iTask` system. Note that, for efficiency reasons, an actual implementation may need to resort to specialized implementations.

3.5 Conclusions

The original `iTask` system offers a lot of functionality on a high level of abstraction liberating the programmer from worrying about many implementation details. The concept of an `iTask` task was a unit of work performed somewhere which, when finished, yielded a value of a certain type which is used to dynamically determine which other tasks to do next. With a fixed but small and powerful set of combinators complex work patterns can be captured.

In this paper we have argued that nevertheless more expressive power is needed. The purpose of the `iTask` library is not only to provide a concise formalism for defining tasks, but also to support the construction of WFMS's. From an `iTask` specification, an executable distributed web enabled WFMS is generated. The library should therefore have as little as possible predefined behaviour. In addition to the tasks that need to be done one also wants to be able to define the view and control managers have on the work that is going on. Furthermore, web browsers nowadays offer much more functionality than a couple of years

ago. Instead of offering a simple form to be filled in, complete full-fledged GUI applications can be run in a web browser.

Currently we are redefining and re-implementing the `iTask` library. We had to change the tasks concept enabling a task to share information with others while the work is going on. In addition to regular tasks, special control tasks are added. Tasks can become a complete GUI application, offering buttons, menus, dialogues and multiple windows.

At this stage we have not yet tested the new system with non-toy examples, but we hypothesize that we can capture *all* current `iTask` combinators *and* the above mentioned shortcomings with only *two* constructs: one very general editor and one Swiss-Army-Knife combinator for creating parallel tasks. These two should suffice to construct all other combinators.

4 Task-Oriented Programming in a Pure Functional Language

Task-Oriented Programming (*TOP*) is a novel programming paradigm for the construction of distributed systems where users work together on the internet. When multiple users collaborate, they need to interact with each other frequently. *TOP* supports the definition of tasks that react to the progress made by others. With *TOP*, complex multi-user interactions can be programmed in a declarative style just by defining the tasks that have to be accomplished, thus eliminating the need to worry about the implementation detail that commonly frustrates the development of applications for this domain. *TOP* builds on four core concepts: tasks that represent computations or work to do which have an observable value that may change over time, data sharing enabling tasks to observe each other while the work is in progress, generic type driven generation of user interaction, and special combinators for sequential and parallel task composition. The semantics of these core concepts is defined in this paper. As an example we present the *iTask3* framework, which embeds *TOP* in the functional programming language *Clean*.

4.1 Introduction

When humans and software systems collaborate to achieve a certain goal they interact with each other frequently and in various ways. Constructing software systems that support human tasks in a flexible way is hard. In order to do their work properly human beings need to be well informed about the progress made by others. We lack a formalism in which this aspect of work is specified at a high level of abstraction.

In this paper we introduce *Task-Oriented Programming (TOP)*, a novel programming paradigm to define interactive systems using *tasks* as the main abstraction. *TOP* provides advanced features for task collaboration. We choose tasks as *unit of application logic* for three reasons. First, they cover many phenomena that have to be dealt with when constructing systems in a natural and intuitive way. In daily life we use this notion to describe activities that have to be done by persons to achieve a certain goal. In computer systems, running processes are also commonly called tasks. On a programming language scale, a function, a remote procedure, a method, or a web service, can all be seen as

tasks that can be executed. Second, in daily life it is common practice to split work into parallel and sequential sub-tasks and at the same time, during execution, not to be very strict about their termination behavior and production of results. Progress of work can be guaranteed even though some, or all, sub-tasks produce partial results. This contrasts with the usual concept of computational tasks that are interpreted as well-defined units of work that take some arguments, take some time to complete, and terminate with a result. Third, tasks abstract from the operational details of the work that they describe, assuming that the processor of the task knows how to perform it. The processor must deal with a plethora of issues: generate and handle interactive web pages, communicate with browsers, interact with web services in the cloud, interface with databases, and so on. Application logic is polluted with the management of side effects, the handling of complicated I/O like communication over the web, and the sharing of information with all users and system components. In this pandemonium of technical details one needs to read between the lines to figure out what a program intends to accomplish. Using tasks as abstraction prevents this. For these reasons, we conjecture and show that in the TOP paradigm specifying what the task *is* that needs to be done, and *how* it can be divided into simpler tasks is sufficient to create the desired application.

We present a foundation for Task-Oriented Programming in a pure functional language. We formalize the notion of tasks as abstract descriptions of interactive persistent units of work. Tasks produce typed, observable, results but have an abstract implementation. When observed by other tasks, a task can either have no (meaningful) value, have a value that is a temporary result that may change, or have a stable final result. We show how to program using this notion of tasks by defining a set of primitive tasks, a model for sharing data between tasks, and a set of operators for composing tasks. Because higher-order function composition provides powerful composition already, only a small set of operators is necessary. These are sequential composition, parallel composition, and the conversion of task results.

Most notably, we make the following contributions:

- We introduce *Task-Oriented Programming* as a paradigm for programming interactive multi-user systems composed of interacting tasks.
- We present *tasks* as abstract units of work with observable intermediate values and continuous access to shared information.
- We present combinators for composition and transformation of tasks and formally define their semantics.
- We demonstrate real-world TOP in Clean using the redesigned and extended iTask3 framework.

The remainder of this paper is organized as follows: in Section 4.2 we informally explain the TOP paradigm by defining its concepts and a non-trivial example in Clean with the `iTask3` framework. In Section 4.3 we formalize the foundations of TOP component-wise: tasks and their evaluation, sharing information, user interaction, and sequential and parallel task composition. In Section 4.4 we reflect on the pragmatic issues that need to be dealt with in frameworks that facilitate real-world TOP programming. After a discussion of related work in Section 4.5, we conclude in Section 4.6 .

For readability, we use Clean* [20] which is a dialect of Clean that adapts a number of Haskell language features. In this paper we deploy *curried function types* (Clean function types have arity), and the *unit type* ().

4.2 The TOP Paradigm

Task-Oriented Programming extends pure Functional Programming with a notion of *tasks* and operations for composing programs from tasks. Complex interactive multi-user systems are specified as decompositions of the tasks they aim to support.

4.2.1 TOP Concepts

Tasks: Tasks are abstract descriptions of interactive persistent units of work that have a typed value. When a task is *executed*, by a TOP framework, it has an opaque persistent state. Other tasks can observe the *current* value of a task in a carefully controlled way. When an executing task is observed, there are three possibilities:

1. **The task has no value observable for others:** This does not mean that no progress is made, but just means that no value of the right type can be produced that is ready for observation.
2. **The task has an unstable value:** When a task has an unstable value, it has a value of the correct type but this result may be different after handling an event. It is even possible that the next time the task is observed it has no value.
3. **The task has a stable value:** The task has a clear final result. This implies that if the task is observed again, it will always have the same value.

Tasks may be interactive. Such tasks process events and update their internal state. However, this event processing is abstracted from in Task-Oriented programs. The effects of events are only visible as changes in task results.

Many-to-many Communication with Shared Data: When multiple tasks are executed simultaneously, they may need to share data between them. How and where this data is stored however, is often completely irrelevant to the task. What matters is that the data is available and that it is shared. Thus, when one task modifies shared data, the other tasks can observe this change. In TOP we abstract from how and where data is stored and define *Shared Data Sources* (SDS) as typed abstract interfaces which can be read, written and updated atomically.

Generic Interaction: The smallest tasks into which an interactive system can be divided are single interactions, either between the system and its users or between the system and another system. Single interactions can be entering or updating some data, making a choice or just viewing some information. In TOP we abstract from how such interactions are realized unless it is essential to the task. A TOP framework *generates* user interfaces *generically* for any type of data used by tasks. This means that it is not necessary to design a user interface and program event handling just to enter or view some information. It is possible to specify interactions in more detail, but it is not needed to get a working program.

Task Composition: TOP introduces the notion of tasks as first-class values, but also leverages first-class functions from pure functional programming. This means that only a small carefully designed set of core combinator functions is needed from which complex patterns can be constructed.

1. **Sequential composition:** TOP uses *dynamic* sequential composition. Because task values are observable, sequential compositions are not defined by blindly executing one task after another. They are defined by composing an initial task with a set of functions that *compute* possible next steps from the observed value of the initial task.
2. **Parallel composition:** Parallel composition is defined as executing a set of tasks simultaneously. Tasks in a parallel set have read-only access to a shared data source that reflects the current values of all sibling tasks in the set. In this way tasks can monitor each other's progress and react accordingly.
3. **Value transformation:** Task domains can be converted by pure functions in order to combine tasks in a type consistent way.

4.2.2 An Example of TOP in Clean

To illustrate Task-Oriented Programming in practice, we present a non-trivial example that uses the novel iTask3 framework. In the example, one specific

user, the *coordinator*, has to collaborate with an arbitrary number of users to find a meeting date and time. Figure 4.1 displays that this task consists of three sub-tasks. This figure consists of actual screenshots of the user interfaces generated by the iTask3 framework.

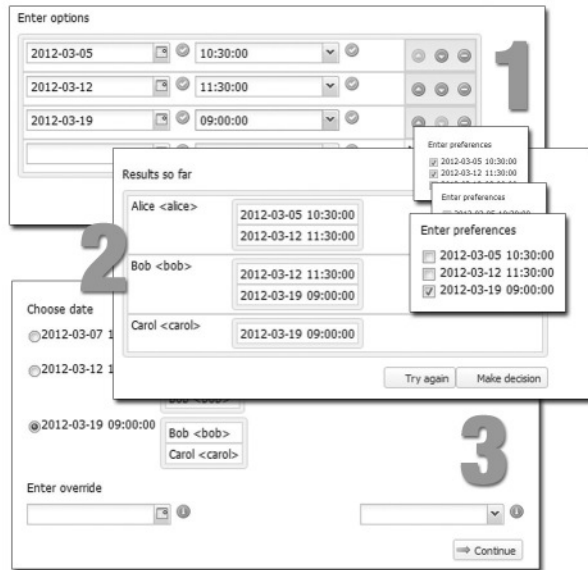


Figure 4.1: Selecting possible dates for a meeting

In sub-task *one*, the coordinator creates a number of date-time pairs. While doing so, he or she can rearrange their order, insert new date-time pairs, or remove them. Once satisfied, the coordinator confirms the work by pressing the *Continue* button, and *steps* into sub-task *two*.

This sub-task *two* consists of a number of tasks running in parallel. The users (Alice, Bob, and Carol in this example) are all asked to make a selection of the proposed date-time pairs (the *Enter preferences* windows). Meanwhile, the coordinator can monitor and follow the selections being made (the *Results so far* window). At any time, the coordinator can either choose to restart the entire task all over again, by pressing the *Try again* button. He or she can also select a date-time pair that is suitable for (the majority of) all users by pressing the *Make decision* button. In the first case, they *step* into the plan meeting task afresh, and in the latter case, they *step* into sub-task *three*.

In sub-task *three* the system provides the coordinator with an overview of available users per date-time pair, thus helping him or her to make a good decision. The coordinator can also decide not to pick any of the candidate

date-time pairs and override them with a proposed alternative. Once satisfied with a choice, the coordinator terminates the entire task by pressing *Continue*, and returns a *stable* date-time value.

In the remainder of this section we show how to specify this example in a Task-Oriented way. Figure 4.2 displays the complete specification. It contains TOP-notions explained in detail further on in this paper. The key point of this example is to show how Task-Oriented Programming aids to create a specification that closely matches the description that is shown above. The semantics of the used concepts are defined in Section 4.3.

The entire task of the coordinator is described by `planMeeting`. Its type (line 1) expresses that given a list of users, it is a task that produces a date-time pair. `User` and `DateTime` are predefined data types. `User` represents a registered user. `DateTime` is just a pair of `Date` (day-month-year triplet) and `Time` (hours-minutes-seconds triplet) which also happen to be predefined.

As discussed, the main structure of `planMeeting` consists of three subsequent sub-tasks (line 3), which are glued together by means of the *step* combinator `>>*`. The second argument of `>>*` enumerates the potential subsequent *task steps* that can be stepped into while the first argument task is in progress. Hence, the first sub-task, `enterDateTimeOptions`, is followed by `askPreferences`, which in turn is followed by *either* `tryAgain` *or* `decide`. Entering user information (performed by `enterDateTimeOptions`, `select`, and `pick`) is an example of a task that may or may not have a *task value*. This depends on the input provided by the user. The potential task steps which can follow can observe the task value and define whether or not sufficient information is provided to step into the next task. In case of the transition from the first sub-task to the second sub-task, this requires an *action* from the coordinator (line 10). This is only sensible if the previous task has a task value, which is tested by the predicate `hasValue`. In that case, the current task value is retrieved (`getValue`) and used to step into the next sub-task, which is to ask all users to choose preferred date-time pairs. The *observable* task value is accessible in the step combinator to determine the next task steps chosen. The task value and its access functions are straightforward: `hasValue` tests for the `Val` data constructor, and `getValue` returns that value if present:

```
1 :: Value a    = NoVal    | Val a Stability
2 :: Stability  = Unstable | Stable
3
4 hasValue :: Value a → Bool
5 hasValue (Val _ _) = True
6 hasValue _         = False
7
8 getValue :: Value a → a
9 getValue (Val a _) = a
```

```

1 planMeeting :: [User] → Task DateTime
2 planMeeting users
3   = enterDateTimeOptions >>* [askPreferences users] >>* [tryAgain users, decide]
4
5 enterDateTimeOptions :: Task [DateTime]
6 enterDateTimeOptions = enterInformation "Enter options" []
7
8 askPreferences :: [User] → TaskStep [DateTime] [(User,[DateTime])]
9 askPreferences users
10  = OnAction (Action "Continue") hasValue (ask users o getValue)
11
12 ask :: [User] → [DateTime] → Task [(User,[DateTime])]
13 ask users options
14  = parallel "Collect possibilities"
15    [ (Embedded, monitor)
16      : [(Detached (worker u),select u options) \\ u ← users]]
17      @ λanswers → [a \\ (_,Val a _) ← answers]
18
19 monitor :: ParallelTask a | iTask a
20 monitor all_results
21  = viewSharedInformation "Results so far" []
22    (mapRead t1 (taskListState all_results))
23    @? λ_ → NoVal
24
25 select :: User → [DateTime] → ParallelTask (User,[DateTime])
26 select user options _
27  = enterMultipleChoice "Enter preferences" [] options
28    @ λchoice → (user,choice)
29
30 tryAgain :: [User] → TaskStep [(User,[DateTime])] DateTime
31 tryAgain users
32  = OnAction (Action "Try again") (const True) (const (planMeeting users))
33
34 decide :: TaskStep [(User,[DateTime])] DateTime
35 decide = OnAction (Action "Make decision") hasValue (pick o getValue)
36
37 pick :: [(User,[DateTime])] → Task DateTime
38 pick user_dates
39  = (enterChoice "Choose date" [] (transpose user_dates) @ fst)
40    -| -
41    (enterInformation "Enter override" [])
42  >>* [OnAction (Action "Continue") hasValue (return o getValue)]

```

Figure 4.2: Complete task specification of the `planMeeting` example

Tasks with `Stable` values are terminated and can no longer produce a different task value. Hence *task values* are first-class citizens in Task-Oriented Programming. Two task transformer functions provide access: `@?` alters the task value of the preceding task, and `@` is similar, but only if a `Val` is present:

```
1 (@?) infixl 1 :: Task a → (Value a → Value b)
2                               → Task b | iTask a & iTask b
3 (@)  infixl 1 :: Task a → (a → b) → Task b | iTask a & iTask b
```

The second sub-task of the coordinator is to ask all users *in parallel* to make a selection of the created date-time pairs. In addition, the coordinator *constantly monitors* their progress. Parallel composition of tasks is defined with the `parallel` combinator. It is used explicitly in the `ask` task, and implicitly (by means of the derived parallel-or combinator `-||-` that provides a shorter notation for the common case of choice between two alternative tasks) in the `pick` task. Parallel composition is a core concept in Task-Oriented Programming. The second argument of `parallel` enumerates the sub-tasks that need to be evaluated in parallel. The progress is *shared* between all sub-tasks. Relevant to the example is the function `taskListState`, which transforms this shared state to share the current *task values*. This is used by the `monitor` task (lines 19-23) to create a view on the current task values of the users. The `monitor` task uses `@?` to explicitly state that its task value never contains a concrete value. These can be provided only by the `select` sub-tasks. They offer their user the means to make a multiple-choice of the provided date-time pairs, and use `@` to attach the user to identify who made that specific selection (lines 27-28).

Finally, the last sub-task can be stepped into when the coordinator either decides to start all over again (lines 30-32) or pick a value (lines 34-35). The first action step is always valid (`const True`, line 32) and the second action step only when the previous task actually has a value (line 35). The derived combinator `-||-` evaluates its two task arguments in parallel, and has a task value that is either stable (if one or both sub-tasks have one) or unstable (if one or both have one) or none. Hence, the action step can only occur when the coordinator has either selected one of the suggested date-time pairs or chosen to override them.

This example demonstrates how a TOP approach can lead to a concise specification in which tasks are glued together and overall progress can be achieved even though the tasks themselves might not terminate or consume too much time.

4.3 A Formal Foundation of TOP

In this section we introduce and semantically define the core concepts of Task-Oriented Programming. These are *task values*, *tasks* and their *evaluation* (Section 4.3.1), *many-to-many* communication (Section 4.3.2), *user-interaction*

(Section 4.3.3), *sequential* task composition (Section 4.3.4), and *parallel* task composition (Section 4.3.5).

Except for Section 4.3.1, every section has the same structure: we first introduce the core concept and illustrate it by means of the `iTask3` system, and then formally define the operational semantics using *rewrite semantics*. The rewrite rules are specified in `Clean*`. Such a way of formal specification of semantics is somewhat unusual, but this approach has certain advantages over traditional ones [36]. The specification is well-defined, concise, compositional, executable, and can express even complicated language constructs as the ones introduced in this paper. Since we are dealing with constructs embedded in a functional language it is an advantage to describe their semantics as pure functions in a functional language as well. We have experimented with several alternative definitions which can easily introduce errors that remain overlooked. It is an advantage that the descriptions are checked by the compiler and that we have been able to test their correct working by applying it to concrete examples. Furthermore, the formal semantics is very suited and also used as blue print for the actual implementation and can serve as a reference implementation for implementations in other programming languages as well. In order to distinguish semantic definitions from `iTask3` API and code snippets, we display semantic definitions as *framed* verbatim text, and `iTask3` fragments as *unframed* verbatim text.

4.3.1 Tasks and their Evaluation

In this section we define *task results* and *task values* (Section 4.3.1), *tasks* (Section 4.3.1), their *evaluation* (Section 4.3.1), and a number of *task transformer functions* (Section 4.3.1).

Task Results and Task Values

A task of type `Task a` is a description of work which progress can be inspected by a *task value* of type `Value a` (Section 4.2.2). Tasks handle events. Events have a time stamp, for which we use an increasing counter, making it possible to determine the temporal order of events. The *task result* of handling an event may be a new task value. Semantically, we extend the task value with the time stamp of the event that caused the creation of that task value. Tasks that run into an exceptional situation have as task result an exception value instead of a task value. The domains of task results and task values capture these situations:

```

1 :: TaskResult a =      ValRes TimeStamp (Value a)
2                       |  ∃e: ExcRes e & iTask e
3 :: TimeStamp      ::= Int
4 :: Value a        =  NoVal    | Val a Stability
5 :: Stability      =   Unstable | Stable

```

The task value of a task result can be in three different states: there can be no value at all (`NoVal`), there can be an `Unstable` value which may vary over time, or the value is `Stable` and fixed. To illustrate, consider the task of writing a paper p . At time t_0 you have no paper at all (`ValRes t0 NoVal`). After a while, at time t_1 there may be a draft paper p_1 , which is updated many times at subsequent time stamps $t_2 \dots t_n$ with draft papers $p_2 \dots p_n$ (`ValRes ti (Val pi Unstable)`). You may even start all over again (`ValRes tn+1 NoVal`). At a certain point in time, t_{n+k} say, when you decide that the paper is finished the task has result `ValRes tn+k (Val pn+k Stable)` meaning that the paper can no longer be altered.

Some tasks never produce a stable value. Examples are the interactive tasks (`enterInformation`, `viewSharedInformation`, `enterChoice`, `enterMultipleChoice`) that were used in Section 4.2.2: a user can create, change or delete a value as many times as wanted. Typical examples of tasks that produce a `Stable` value are ordinary functions, or system and web service calls. A task can raise an exception value (`ExcRes e`) in case it is known that it can no longer produce a meaningful value (for instance when a call to a web service turns out to be unavailable). Any value can be thrown as exception and inspected by an exception handler (Section 4.3.4), using existential quantification \exists and the type class context restriction `& iTask e`. Tasks with stable values or exception values have no visualization but memorize their task result forever. The other tasks require a visualization to support further interaction with the user.

Tasks

Semantically, we define a task to be a state transforming function that reacts to an event, rewrites itself to a `reduct`, and accumulates responses to users:

```

1 :: Task a    ::= Event → *State → *(Reduct a, Reponses, *State)
2 :: Event    = RefreshEvent
3              | EditEvent   TaskNo Dynamic // Section 4.3.3
4              | ActionEvent TaskNo Action  // Section 4.3.4
5 :: *State   = { taskNo    :: TaskNo
6                 , timeStamp :: TimeStamp // Section 4.3.3
7                 , mem      :: [Dynamic]  // Section 4.3.2
8                 , world    :: *World
9                 }
10 :: Reduct a = Reduct (TaskResult a) (Task a)
11 :: TaskNo   ::= Int
12 :: Responses ::= [(TaskNo, Response)] // Section 4.3.3

```

We distinguish three sorts of events: a `RefreshEvent`, e.g. when an user wants to refresh a web page, an `EditEvent`, e.g. a new value that is committed intended for an interactive task (Section 4.3.3), and an `ActionEvent` which is used to tell the step combinator which task to do next (Section 4.3.4). The latter two cases identify the task that is required to handle the event. The interactive task and

step task are provided with a fresh identification value and current time stamp, using the semantic function `newTask`:

```

1 newTask :: (TaskNo → TimeStamp → Task a) → Task a
2 newTask ta ev st = {taskNo = no, timeStamp = t}
3   = ta no t ev {st & taskNo = no+1}

```

Fresh task identification numbers are generated by keeping track of the latest assigned number in the State. The State extends the external environment of type `*World` with internal administration and is passed around in a single-threaded way which is enforced by the uniqueness attribute `*`.

The `reduct` contains both the *latest task result* and a *continuation* of type `Task a`, which is the remaining part of the work that still has to be done. This continuation can be further evaluated in the future when the next event arrives. The responses collect all responses of all subtasks the task is composed of. They are used to update every client with the proper information about the latest state of affairs. A client can use this information to adjust the page in the browser or in an app.

In the remainder of this paper we define semantic task functions for the core basic tasks and task combinators, thus explaining how these elements rewrite to the next `reduct`.

Task Evaluation

A TOP application consists of one top level task, the main task, which has to be evaluated. The work continues until either an exception escapes handling, or the work at hand has obtained a stable task value.

```

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

```

In `Clean(*)`, passing around multiple unique environments explicitly, such as `st (:: *State)` and `world (:: *World)`, is syntactically supported by means of the non-recursive `#let` definitions. The main task is recursively rewritten by the function `rewrite`. Rewriting is triggered by an event. We abstract from the behaviour of clients and just assume that they send events and handle responses. We assume that all events are collected in a queue. In `getNextEvent` (line 9) the next event is fetched from this queue. If there are no events, the system waits until there is one. The current time is stored in the state (lines 10-11) to ensure that all tasks which update their value in this rewrite round, will get the same time stamp. Hereafter (line 12), the main task `ta` is evaluated given the event and current state. Any sub-task defined in the main task is a task as well, and can be evaluated in the same way: just apply the corresponding task function to the current event and the current state. Rewriting stops when the main task has delivered a stable value (line 14), or an uncaught exception is raised (line 15). Otherwise, the main task is not finished yet, and the continuation task returned in the `reduct` defines the remaining work which has to be done. First the accumulated responses are sent to the clients (`informClients`, line 17) to inform them about the latest state-of-affairs. We abstract in the semantics from the way this is done. Rewriting continues with the continuation `nta` and the updated state.

Utility Functions for Converting Tasks

The semantic function `stable`, when applied to a time stamp `t` and value `va`, defines a task that has reached a stable value:

```
1 stable :: TimeStamp → a → Task a
2 stable t va _ st
3 = (Reduct (ValRes t (Val va Stable)) (stable t va), [], st)
```

Notice that the continuation of the task `stable t va` in the `reduct` is exactly the same function `stable t va`. It is a kind of fixed point task, which, whenever it is evaluated in some future, always returns the same `reduct` (value and continuation). With this semantic function, we can define the semantic function of the core task `return`:

```
1 return :: a → Task a
2 return va ev st = {timeStamp = t} = stable t va ev st
```

Here, `return` has a similar role as the `return` function in a monadic setting: it lifts an arbitrary value `va` of type `a` to the task domain.

Raising an exception is similar, except that the task result is always an exception value:

```
1 throw :: e → Task e | iTask e
2 throw e _ st = (Reduct (ExcRes e) (throw e), [], st)
```

With operator $@?$ and a function f of type $\text{Value } a \rightarrow \text{Value } b$ a task ta of type $\text{Task } a$ can be converted to a task of type $\text{Task } b$:

```

1 (@?) infixl 1 :: Task a → (Value a → Value b)
2                               → Task b | iTask a & iTask b
3 (@?) ta f ev st
4 = case ta ev st of
5   (Reduct (ValRes t aval) nta,rsp,nst)
6   → case f aval of
7     Val b Stable
8     → stable t b ev nst
9     bval → (Reduct (ValRes t bval) (nta @? f),rsp,nst)
10  (Reduct (ExcRes e) __,__,nst)
11  → throw e ev nst
12
13 (@) infixl 1 :: Task a → (a → b) → Task b | iTask a & iTask b
14 (@) ta f = t @? λaval → case aval of
15   NoVal   = NoVal
16   Val a s = Val (f a) s
    
```

First the task ta is evaluated (line 4). Exceptions raised by ta are simply propagated (lines 10-11). The resulting task value, if any, is converted by function f . If this results in a stable value, then the entire task becomes stable with the current time stamp (lines 7-8). Notice that this has as consequence that the original task ta is no longer needed. If the result is not stable, the original task may change its value over time, and we need to apply the conversion function to values produced in the future as well. Therefore, the current result $bval$ of the conversion is stored in the reduct with the continuation $nta @? f$ which takes care of the conversion of the new task values produced in the future (line 9). The derived operator $@$ uses $@?$ to transform task values only when a concrete value is present.

4.3.2 Many-to-many Communication

For collaborating tasks it is important to keep each other up-to-date with the latest developments while the work is going on. Hence we need to be able to share information between tasks and support many-to-many communication. How and where this data is stored, is completely irrelevant to the tasks. What matters is that the data is available and that it is shared. To achieve this abstraction we use the concept of multi-purpose Shared Data Sources (SDS) [54]. SDSs are typed, abstract interfaces which can be read, written and updated atomically.

A SDS can represent a shared file, a shared structured database, reveal the current users of a system, or it can be a physical entity, like the current time or temperature. In general, a SDS abstracts from any shared entity that holds a value that varies over time.


```

1 :: RWShared r w
2
3 :: ROShared r  ::= RWShared r  ()
4 :: WOShared w  ::= RWShared () w
5 :: Shared a    ::= RWShared a  a

```

A SDS has abstract type `RWShared r w`. *Reading* its current value returns a value of type `r`, and *writing* is done with a new value of type `w`. Read-only shared objects (`ROShared r`) only support reading as type `r`, write-only shared objects (`WOShared w`) only support writing as type `w`, and `Shared a` objects demand that the read and write values have the same type `a`.

As an example, we show a few shares that are offered by the `iTask3` system to create SDSs:

```

1 sharedFile  :: Path → a → Shared a | iTask a
2 currentTime :: ROShared Time
3 currentUsers :: ROShared [User]

```

With `(sharedFile fname content)` a task is described that associates a file identified by `fname` with an initial value of type `a`. A task gains access to the current time and registered users with the tasks `currentTime` and `currentUsers`.

SDSs provide many-to-many communication both between tasks and other applications. We make a difference between external and internal SDSs. External SDSs are abstractions of external objects such as files and databases and can be accessed anywhere in the application. For the internal communication between tasks only, one can create a shared memory SDS of type `Shared a` which has a limited scope. A task `ta` can be parameterized with a freshly created shared memory SDS `sa` of type `Shared a` that has some initial value `va` using the combinator `withShared va (λsa → ta)`:

```

1 withShared :: a → (Shared a → Task b) → Task b | iTask a

```

In this way, a shared memory is created which can only be accessed by the sub-tasks defined within `ta`. For an example of its use, see Section 4.4.

To write a value to a SDS, one can connect a task `ta` with a SDS `s` using a function `f` with the combinator `ta @> (f,s)`:

```

1 (@>) infixl 1 :: Task a
2             → (Value a → r → Maybe w, RWShared r w)
3             → Task a | iTask a

```

This enforces `f` to be repeatedly applied to the current task value of `ta` (if any) and the currently read value of `s`, the result of which is the new value (if any) that is written to `s`. The combinators `withShared` and `@>` are defined in Section 4.3.2.

SDSs integrate smoothly with interactive tasks. For every basic interactive task (such as `enterChoice` and `enterMultipleChoice`) a *shared* version (such as

`enterSharedChoice` and `enterSharedMultipleChoice`) is provided that expects a SDS instead of a common value. This is discussed in Section 4.3.3 in more detail. In this way tasks can monitor and alter SDSs.

Semantics of Memory Shared between Tasks

To explain the semantics of SDSs, we restrict ourselves to their use for offering shared memory between (parallel) tasks. These SDSs cannot be accessed by external applications. Hence the semantic definition does not need to handle concurrency and atomicity issues: there is only one `rewrite` function (Section 4.3.1) that handles rewriting of all tasks defined in an application.

Shared memory cells are stored in the `State`, in record field `mem` of type `[Dynamic]`. Each SDS memory cell can be used to store a value of arbitrary type, hence `mem` is modeled as a heterogeneous list using Clean's built-in dynamic types [78, 80]. Any shared value of any type can be stored in a value of type `Dynamic`, together with a representation of its type (using the function `serialize :: a → Dynamic | iTask a`). It can be fetched from this store any time later, using a dynamic type pattern match that guarantees that no type errors can occur at run-time (using the function `de_serialize :: Dynamic → a | iTask a`). We define a SDS creation function, and two functions to update a SDS:

```

1 :: RWShared r w = { get :: *State → *(r,*State)
2                   , set :: w → *State → *State
3                   }
4
5 createShared :: a → *State → *(Shared a,*State) | iTask a
6 createShared a st={mem}
7 = ({get = get,set = set},{st & mem = mem ++ [serialize a]})
8 where
9   idx          = length mem
10  get  st={mem} = (de_serialize (mem!!idx),st)
11  set a st={mem} = {st & mem = updateAt idx (serialize a) mem}
12
13 updateShared :: (r → w) → RWShared r w → *State → *(w,*State)
14 updateShared f sh_a st
15 # (rv,st) = sh_a.get st
16 # wv      = f rv
17 = (wv,sh_a.set wv st)
18
19 updateMaybeShared :: (r → Maybe w) → RWShared r w → *State
20                               → *(Maybe w,*State)
21 updateMaybeShared f sh_rw st
22 # (readv,st) = sh_rw.get st
23 = case f readv of
24   Nothing = (Nothing,st)
25   Just wv = (Just wv,sh_rw.set wv st)

```

A SDS is represented by two access functions `get` and `set` that retrieve and store the required information from and to the state. Creating a shared value with `createShared` appends an initial serialized value to the list of memory locations (line 7), and returns two dedicated `get` and `set` functions that access this new memory location. The SDS update functions both obtain the current read value of the SDS argument (line 15 and 22). However, `updateShared` always updates the SDS with a new value, and `updateMaybeShared` does this only if the argument function actually produces a new value. With these internal functions, we can define `withShared` and `@>`:

```

1 withShared :: a → (Shared a → Task b) → Task b | iTask a
2 withShared va tfun ev st
3 # (sh_a,st) = createShared va st
4 = tfun sh_a ev st
5
6 (@>) infixl 1 :: Task a
7     → (Value a → r → Maybe w, RWShared r w)
8     → Task a | iTask a
9 (@>) ta (f,sh_rw) = update NoVal ta
10 where
11   update otval ta ev st
12 = case ta ev st of
13   (Reduct (ExcRes e) nta, _, nst)
14     → throw e ev nst
15   (Reduct (ValRes ts nval) nta,rsp,nst)
16     → ( Reduct (ValRes ts nval) (update nval nta)
17         , rsp
18         , if (ntval==otval)
19             nst
20             (snd (updateMaybeShared (f nval) sh_rw nst)))
21   )

```

`withShared` creates a fresh SDS for its argument task function and applies it to obtain the proper task. The combinator `@>` memorizes the previous task value (initially `NoVal`) and the current task continuation (initially the task argument `ta`) (line 9 and 16). As usual, at each event the current task continuation is evaluated (line 12). Exceptions are propagated (lines 13-14). The only difference is that if the new task value `ntval` is different from the memorized task value `otval`, then the SDS is updated using the argument function of `@>` and the function `updateMaybeShared` (line 20). This function only updates the SDS if a new value is computed. In this way unnecessary updates of shared data are avoided. Because `@>` keeps checking the SDS using the most recent task value, this leads to reactive behavior: every time the watched task is changing its value, the shared memory also gets updated conditionally, as described above.

4.3.3 User Interaction

In Task-Oriented Programming user-interactions are defined as tasks that allow a user to enter and modify a visualized value of some type. Such an interactive task is called an *editor*. The type of the value to be edited plays a central role. By using type indexed generic functions [27, 2] this visualization is generated fully automatically for any (first order) type. This way one can focus on defining tasks, without having to deal with the complexities of web protocols and formats.

Interaction tasks follow a model-view pattern where the value of the task is the model and the visualization is the view. Events in the view are processed by the TOP framework to update the model. Conversely, when the model changes the view is updated automatically by the TOP framework.

Interaction tasks are all alike, yet different. In this section we define the semantics of one core editor task (Section 4.3.3). However, to improve readability TOP frameworks can offer a range of predefined interaction tasks derived from this core editor. A few examples from the `iTask3` framework are:

```

1 enterInformation      :: d → [EnterOpt m]
2                      → Task m      | descr d & iTask m
3 updateInformation   :: d → [UpdateOpt m m] → m
4                      → Task m      | descr d & iTask m
5 viewInformation     :: d → [ViewOpt m] → m
6                      → Task m      | descr d & iTask m
7 updateSharedInformation :: d → [UpdateOpt r w] → RWS shared r w
8                      → Task w      | descr d
9                      & iTask r & iTask w
10 viewSharedInformation :: d → [ViewOpt r] → RWS shared r w
11                      → Task r      | descr d & iTask r

```

With `enterInformation` an editor for type `m` is created, no initial value needs to be given. The `update-editor` variants allow editing of a given local, respectively shared, value. The `view-editor` variants only display the value of a given local, or shared, value. There are many more similar editor functions predefined in the library, with names like `enterChoice`, `enterSharedChoice`, `updateChoice`, `updateSharedChoice`, `enterSharedMultipleChoice`, and so on.

The overloaded argument `d` of class `descr` in these tasks is description of the task. This can be a simple string, or a more elaborate description. Although the generated view is certainly good enough for rapid prototyping, more fine-grained control is sometimes desirable. Therefore, the `EnterOpt`, `UpdateOpt` and `ViewOpt` arguments provide hooks for fine-tuning interactions.

```

1 :: ViewOpt a      = ∃v: ViewWith (a → v)          & iTask v
2 :: EnterOpt a     = ∃v: EnterWith (v → a)         & iTask v
3 :: UpdateOpt a b = ∃v: UpdateWith (a → v) (a v → b) & iTask v

```

By defining a mapping, a different type `v` can be used to view, enter or update information. In Section 4.4 we discuss in more detail how this and other pragmatic issues are dealt with.

Semantics of a Task Editor

The `iTask3` library provides many different editor task functions because this clarifies in the task descriptions what kind of interaction is required, and aids in creating the desired user interface. However, both in the implementation and the semantics all editor task variants can be created and handled by one single function. To understand how it works we restrict ourselves to a simplified version in which we omit the view list details because these are just trivial mapping functions. Before we discuss this function `edit` we first have a look at the use of `Events` and `Responses`. Due to the model-view nature of editor tasks, every user manipulation of an editor task of a value of type `a` can be expressed as sending a new value `new` of type `a` from the client to the server. If we wrap this value-type pair into a `Dynamic` and include the task identification number, `no` say, then this amounts to the `(EditEvent no (dynamic new :: a))` event. The unique task number is used to map a task described in the code to the corresponding interactive view generated in the client, and is used to label the events and corresponding responses.

The responses of the server tell the client what interface should be rendered to the user.

```

1 :: Response      = EditorResponse EditorResponse
2                  | ActionResponse ActionResponse // Section 4.3.4
3 :: EditorResponse = { description :: String
4                      , editValue  :: EditValue
5                      , editing    :: EditMode
6                      }
7 :: EditValue     ::= (LocalVal, SharedVal)
8 :: LocalVal      ::= Dynamic
9 :: SharedVal     ::= Dynamic
10 :: EditMode     = Editing | Displaying

```

The response to an editor task executed on a client informs the client about the latest state of the editor (`EditorResponse`) and contains, in serialized form, the current local value to edit and a shared value to show. With these `Events` and `Responses`, we can define the semantics of the editor task combinator which updates a local value of type `l` while displaying the latest value `r` stored in an SDS of type `RWShared r w`.

```

1 edit :: String → l → RWShared r w → (l → r → Maybe a)
2                                     → Task a | iTask l & iTask r
3 edit descr lv sh_rw cv = newTask (edit1 lv)
4 where
5   edit1 lv tn t ev st
6   # (nt,nlv) = case ev of
7       EditEvent tid dyn
8         → if (tid==tn)
9           (st.timeStamp,de_serialize dyn)
10          (t,lv)
11        _ → (t,lv)
12   # (sr,st) = sh_rw.get st
13   = ( Reduct (ValRes nt (toValue (cv nlv sr))) (edit1 nlv tn nt)
14     , [(tn,EditorResponse
15         { description = descr
16           , editing    = Editing
17           , editValue  = (serialize nlv, serialize sr)
18         }
19       )]
20     , st
21   )
22 where
23   toValue :: Maybe a → Value a
24   toValue (Just a) = Val a Unstable
25   toValue Nothing  = NoVal

```

The `edit` function, and its continuation in the `reduct` (line 13), is defined in terms of `edit1` that keeps track of the latest local value edited, the unique task number given to this interactive task, and the time the latest modification has been made.

Editor tasks always have an unstable value (if any). They return a response containing the latest information on the state of the editor (lines 14-19 and 23-25). It includes the latest value of the data stored in shared memory (line 12) which might have been changed by some other task (e.g. using the `&>` operator). Only when the received event is an edit event intended for this editor (the task numbers match), the local value is updated with the new value received from the client (line 9). The new task value is computed using the most recent local and shared value (line 13).

4.3.4 Sequential Tasks

Once a task is started, it stays alive until it is no longer needed. Its value, which might change over time, can be inspected while the work is going on in order to decide whether or not to step to a next task. The task *step* operator `>>*` does exactly this.

```
1 (>>*) infixl 1 :: Task a → [TaskStep a b] → Task b | iTask a & iTask b
2 :: TaskStep a b
3   = OnAction Action (Predicate a) (NextTask a b)
4   | OnValue      (Predicate a) (NextTask a b)
5   | ∃e: OnException (e → Task b) & iTask e
6
7 :: Predicate a ::= Value a → Bool
8 :: NextTask a b ::= Value a → Task b
9
10 :: Action = Action String | ActionOk | ActionCancel | ...
```

The step operator is similar to an ordinary monadic “bind-operator” in the sense that it defines a sequence between two tasks. The first operand, a task of type `Task a`, is evaluated. Its current task value can be inspected to decide whether the next task can be stepped into. If so, the evaluation of the first task is abandoned and the application proceeds with the chosen task step. The step operator can offer several tasks to continue with in the list, but only one task step can be stepped into.

There are three categories of task steps: those that require the user to actively select an action (`OnAction`), those that inspect the current task value (if any) (`OnValue`), and those that handle exceptions (`OnException`). `OnAction` task steps are labeled with an `Action` that is presented to the user as a button or menu item. For frequently used action names such as `Ok` and `Cancel`, the `Action` data type enumerates a number of special combinators to enable the client to use special icons. The predicate determines which action steps are available at all. Selection of an action by the user causes the corresponding alternative to be continued with. `OnValue` task steps inspect the current task value to determine whether or not a task step can be performed. Finally, `OnException` task steps handle an exception only if their argument function matches the type of the exception. Uncaught exceptions are propagated by `>>*`.

It sometimes can be the case that none of the candidate task steps can be chosen. However, task values change over time, hence also the candidates that can be chosen change over time.

We illustrate the use of `>>*` with two examples.

```
1 palindrome :: Task (Maybe String)
2 palindrome = enterInformation "Enter a palindrome" []
3             >>* [ OnAction ActionOk    ifPalindrome
4                  (return o Just o getValue)
5                  , OnAction ActionCancel (const True)
6                  (const (return Nothing))
7             ]
```

The `palindrome` task prompts the user to enter a palindrome. As usual, the user can enter a string and change it over time. With `>>*` two possible action task steps are added. The user can choose action `Ok`, but only when the entered

string is indeed a palindrome. If `Ok` is chosen, `Just p` is returned, where `p` is the entered and checked palindrome. At any time, the user can choose `Cancel`, and the task returns `Nothing`.

In the second example we implement a traditional monadic bind operator `>>=` to demonstrate the general nature of `>>*`:

```
1 (>>=) infixl 1 :: Task a → (a → Task b) → Task b | iTask a & iTask b
2 (>>=) ta atb = ta >>* [OnValue isStable (atb o getValue)]
```

Task evaluation starts with the first argument `ta`. *Only* when this task produces a *stable* value `a`, evaluation continues with `atb a`. For this reason, `>>=` is less suited in the domain of tasks that may not produce a stable result.

Semantics of the Step Combinator

First we finalize the details of `ActionResponses`. The client is informed by `>>*` about the current set of actions and whether they are enabled or disabled. This information is collected in the `ActionResponse` list and added to the response accumulator.

1	:: ActionResponse ::= [(Action, Enabled)]
2	:: Enabled ::= Bool

The client may react by sending an action event `ActionEvent taskno action` telling which action is triggered by the user.

The complete semantic definition of `>>*` is given in Figure 4.3. It is rather long because it needs to handle all `TaskStep` cases and prioritize them properly. However, each of these cases is rather straightforward. The step combinator is handled by `step1` which memorizes the current task description in its first argument (initially task `ta`, line 2, and in the reduct `nta`, line 19). The semantic function `newTask` (Section 4.3.1) provides it with a unique task number for communication with the client and current time stamp `t` (line 2). The current task description is evaluated first (line 5), resulting in a new task value that is inspected to decide which task step can be stepped into. Triggers (line 6) take priority over actions (line 7). If no task step is applicable, then we proceed with `step1` again, but now parameterized with the calculated reduced task (line 8).

A trigger is a task step that can continue without interference of the user. These are the `OnException` and `OnValue` task steps. In case of an exception, an exception handler is searched for (line 11 and 25). If none is defined, then the exception propagates (line 11). In case of a task value, all available `OnValue` task steps are searched for (line 12 and 26).

The actions are selected only if the event is an action event for this task (line 14 and 15). In that case all available `OnAction` task steps are searched for


```

1 (>>*) infixl 1 :: Task a → [TaskStep a b] → Task b | iTask a & iTask b
2 (>>*) ta steps = newTask (step1 ta)
3 where
4   step1 ta tn t ev st
5   # (Reduct tval nta, rsp, st) = ta ev st
6   = hd ( findTriggers tval
7         ++ findActions tval ev
8         ++ [step1' tval nta rsp]
9         ) ev st
10  where
11    findTriggers (ExcRes e) = catchers e ++ [throw e]
12    findTriggers (ValRes _ v) = values v
13
14    findActions (ValRes _ v) (ActionEvent tid act)
15    | tid==tn          = actions act v
16    findActions _ _   = []
17
18    step1' (ValRes _ v) nta rsp _ st
19    = (Reduct no_tval (step1 nta tn t), nrsp ++ rsp, st)
20  where
21    no_tval = ValRes t NoVal
22    as      = [(a,p v) \\ OnAction a p _← steps]
23    nrsp    = if (isEmpty as) [] [(tn, ActionResponse as)]
24
25  catchers e = [etb e \\ OnException etb← steps]
26  values v = [atb v \\ OnValue p atb← steps | p v]
27  actions act v = [atb v \\ OnAction a p atb← steps | act==a && p v]

```

Figure 4.3: The complete semantic definition of `>>*`.

that match the received action and that are available, as determined by their predicate (lines 27).

Finally, when no task step can be selected a reduct is made by `step1'` that waits for a new event (line 19). All actions are collected in the response accumulator (line 22 and 23).

4.3.5 Parallel Tasks

Tasks can often be divided into parallel sub tasks if there is no specific predetermined order in which the sub tasks have to be done. It might not even be required that all sub tasks contribute sensibly to a stable result. All variants of parallel composition can be handled by a single `parallel` combinator:

```

1 parallel :: d → [(ParallelTaskType, ParallelTask a)]
2           → Task [(TimeStamp, Value a) | descr d & iTask a
3
4 :: ParallelTaskType = Embedded | Detached ManagementMeta
5 :: ManagementMeta = { worker :: Maybe User
6                     , role   :: Maybe Role
7                     , ... }
8 :: ParallelTask    a ::= SharedTaskList a → Task a
9 :: SharedTaskList a ::= RShared (TaskList a)
10 :: TaskList       a = { state  :: [Value a]
11                       , ... }

```

We distinguish two sorts of parallel sub-tasks: `Detached` tasks get distributed to different users and `Embedded` tasks are executed by the current user. The client may present these tasks in different ways. `Detached` tasks need a window of their own while `Embedded` tasks may be visualized in an existing window. With the `ManagementMeta` structure properties can be set such as which `worker` must perform the sub-task, or which `role` he should have.

Whatever its sort, every parallel sub-task can inspect each others progress. Of each parallel sub-tasks its current task value and some other system information is collected in a shared task list. The parallel sub-tasks have *read-only* access to this task list. The `parallel` combinator also delivers all task values in a list of type `[(TimeStamp, Value a)]`. Hence, the progress of every parallel sub-task can also be monitored constantly from the “outside”. For instance, a parallel task can be monitored with the step combinator `>>*` to decide if the parallel task as a whole can be terminated because its sub tasks have made sufficient progress for doing the next step. It is also possible to observe the task and convert its value to some other type using the conversion operator `@?` (Section 4.3.1).

For completeness, we remark that the shared task list is also used to allow dynamic creation and deletion of parallel sub-tasks. We do not discuss this further in this paper.

In the `iTask3` library `parallel` is used to predefine several frequently used task patterns. In Section 4.2.2 the `-||-` combinator was used to start tasks in `parallel`.

```

1 (-||-) infix 3 :: Task a → Task a → Task a | iTask a
2 (-||-) a b
3   = parallel () [(Embedded, const a), (Embedded, const b)] @? first
4 where
5   first NoVal = NoVal
6   first (Value vs _)
7     = hd ( [v \\< ( _, v =: (Val _ Stable)) ← vs]
8           ++ [v \\< ( _, v =: (Val _ _)) ← sortBy newer vs]
9           ++ [NoVal] )
10  newer (t1, _) (t2, _) = t1 > t2

```

The `first` function inspects the progress of both parallel sub-tasks to determine the task value of the composition. The first sub-task to produce a stable task value turns the composition into a stable task with that value. If no sub-task has produced a stable value, then the most recent unstable task value, if any, is the observable result, or no task value is observable at all.

Semantics of the Parallel Combinator

In the semantic description we ignore the meta information assigned to detached tasks and therefore do not distinguish embedded tasks from detached tasks. As another non-essential simplification, we define the shared task list as a *read-write* SDS instead of a *read-only* SDS. The shared task list is a finite map from process ids to task reducts:

```

1 :: SharedTaskList a ::= RWShared (TaskList a)
2 :: TaskList       a ::= [(Pid a, Reduct a)]
3 :: Pid           a ::= Int

```

The complete semantic definition of `parallel` is given in Figure 4.4. The semantic function `parallel'` (lines 12-21) defines the purpose of the `parallel` combinator: to evaluate each and every sub-task (line 14) until either an exception has been thrown (line 15), or all sub-tasks have become stable (line 19). While this is not the case, `parallel'` proceeds to rewrite to itself (line 20-21).

Both `parallel'` and its sub-tasks require access to their progress, which is stored in the shared task list which is created as the first step of the `parallel` combinator (line 3 and lines 6-10). Initially, the task list consists of all initial parallel sub-tasks that have access to the shared task list.

The semantic functions `evalParTasks` and `evalParTask` define the evaluation of the parallel sub tasks: `evalParTasks` collects the current list of sub-tasks (line 26) and applies `evalParTask` to each and every sub-task (line 27). Evaluation of a sub-task (line 34) might result in an exception (line 36), in which case the exception is propagated throughout the evaluation of all sub-tasks (line 39). If a sub-task does not result in an exception, then its new reduct is stored in the shared task list (line 37), thus allowing the other sub-tasks to inspect its progress (`updateEM (pid,newr)` updates any existing element `(pid,_)` in the shared task list with `(pid,newr)`). The responses of the evaluated sub-task are collected and returned (line 38).

4.4 Practical TOP

Although the TOP paradigm adopts functional programming's emphasis of *what* over *how*, some pragmatic issues remain unavoidable in practical TOP programming. In this section we discuss pragmatics issues that we encountered

```

1 parallel :: [ParallelTask a] → Task [(TimeStamp,Value a)] | iTask a
2 parallel ptas ev st
3 # (stt,st) = createTList ptas st
4 = parallel' stt ev st
5
6 createTList :: [ParallelTask a] → *State → *(SharedTaskList a,*State) | iTask a
7 createTList ptas st=: {timeStamp = t}
8 # (stt,st) = createShared [] st
9 = (stt,stt.set [ (pid,Reduct (ValRes t NoVal) (pta stt))
10                \\ pta←ptas & pid← [0..] ] st)
11
12 parallel' :: SharedTaskList a → Task [(TimeStamp,Value a)] | iTask a
13 parallel' stt ev st
14 = case evalParTasks stt ev st of
15   (Left (ExcRes e),st) = throw e ev st
16   (Right rsp,st)
17   # (values,st) = get_task_values stt st
18   # maxt       = foldr max 0 (map fst values)
19   | all (isStable o snd) values = stable maxt values ev st
20   | otherwise
21   = (Reduct (ValRes maxt (Val values Unstable)) (parallel' stt),rsp,st)
22
23 evalParTasks :: SharedTaskList a → Event → *State
24              → *(Either (TaskResult a) Responses,*State) | iTask a
25 evalParTasks stt ev st
26 # (tt,st) = stt.get st
27 = foldl (evalParTask stt ev) (Right [],st) tt
28
29 evalParTask :: SharedTaskList a → Event
30             → *(Either (TaskResult a) Responses,*State)
31             → (Pid a,Reduct a)
32             → *(Either (TaskResult a) Responses,*State)
33 evalParTask stt ev (Right rsp,st) (pid,Reduct _ ta)
34 # (newr,nrsp,st) = ta ev st
35 # (Reduct ntval nta) = newr
36 | isExcRes ntval = (Left ntval,st)
37 # (_,st) = updateShared (updateFM (pid,newr)) stt st
38 = (Right (nrsp ++ rsp),st)
39 evalParTask _ _ (Left e,st) _ = (Left e,st)
40
41 get_task_values :: SharedTaskList a → *State → *([(TimeStamp, Value a)],*State)
42 get_task_values stt st
43 # (tt,st) = stt.get st
44 = ([(t,val) \\ (_,Reduct (ValRes t val) _) ← tt],st)

```

Figure 4.4: The complete semantic definition of parallel.



Figure 4.5: A very simple text editor

in the implementation of the TOP concept in the *iTask3* toolkit, and show examples of *iTask3* programs to illustrate its use in real-world applications.

4.4.1 Pragmatic Issues

Custom Interaction: TOP programs focus on defining decompositions of tasks without worrying how interactions of basic tasks are implemented by the TOP framework. The underlying implementation has to take care of that. The *iTask3* system follows the semantic definitions, with additional support for customization for obtaining practical applicable applications. The interactive applications that are generated by default by the system suffice for rapid prototyping. However, aesthetic and ergonomic properties of these interactions affect the ease of use and attractiveness of a system. For example, the task of choosing a file from a file system is performed more easily by navigating a tree structure than by selecting an item from a long list of all files.

To allow for such task specific optimization, all interaction tasks in the *iTask3* framework have a *views* parameter, in which optional mappings between the task's domain and another arbitrary domain can be defined. The library provides types that represent abstract user interface controls with which customized interactions can be composed. Here is an example (see Figure 4.5):

```

1 :: Statistics = { lineCount :: Int, wordCount :: Int }
2 derive class iTask Statistics
3
4 simpleEdit :: Task Note
5 simpleEdit = withShared (Note "") edit
6 where
7   edit note
8   = updateSharedInformation "Enter text:" [] note
9     - | | -
10    viewSharedInformation "Statistics:" [ViewWith stat] note <<@ horizontal
11
12   stat (Note txt) = { lineCount = length lines
13                     , wordCount = length words
14                     }
15 where lines      = split Newline txt
16       words      = split " " (replaceSubString Newline " " txt)

```

By default, if a value of the predefined type `Note` is used in an `iTask3` editor, a text box is presented to the user on the client to enter text. In `simpleEdit` we create a shared memory for a value of this type `Note` with initial value `Note ""` and we define two interactive tasks on this shared value. The first task allows the user to update the initial text (line 8), while the second gives a view on the shared text that is fine-tuned with `viewWith` which, in this case, converts the text into a value of type `Statistics`. As a result, while entering text, the user sees the corresponding statistics.

Customized Layout: For task compositions a similar need for customization exist. Depending on the composition, it may be more appealing or easier to use when tasks are divided over tabs or windows than when tasks are shown side-by-side. To customize layout, the `iTask3` framework provides an annotation operator (`<<@`) that can be used to annotate tasks with custom layout functions or post-layout processing functions. Such functions combine a set of abstract GUI definitions into a single definition. By default a heuristic layout function is used to provide a sensible default. Post-processing functions modify a GUI definition after a task is layed out. Such modifications are for example changing its size, adding margins or changing to a horizontal layout as is done with the `<<@ horizontal` annotation in the `simple editor`. It is defined as:

```
1 horizontal = AfterLayout (tweakUI (setDirection Horizontal))
```

Localization: Another pragmatic aspect one may need to deal with is localization. Because task definitions contain many prompts, hints and other texts, one needs to deal with localization of such texts without compromising the readability of task definitions. Furthermore, localization may also be required on the task level. To comply with local law and regulations, different task definitions may have to be used in different countries. The `iTask3` framework does not offer any special support for localization, but one can make use of the standard modular structure of `Clean` to create different local versions.

Third Party Formats and Protocols: To integrate TOP applications with other applications, the gap between the domain of tasks and the formats or protocols required to interact with these systems must be bridged. With TOP one does not escape writing the parsing, formatting and communication code that is necessary for such integrations, but it can be separated from the application code by moving it to task libraries.

4.4.2 Examples

A Generic Work List: A major leap in the development of TOP as a general paradigm was the insight that, from a user's point of view, interaction with a

“Work List”, in which users can work on tasks assigned to them, is actually part of the work that has to be done. Work list handling e.g. as offered by an email application or a workflow system is commonly hard coded in the systems used. In iTask3 this functionality is defined in the system itself as “just” any other task. Figure 4.6 shows the generic work list task we offer as a standard

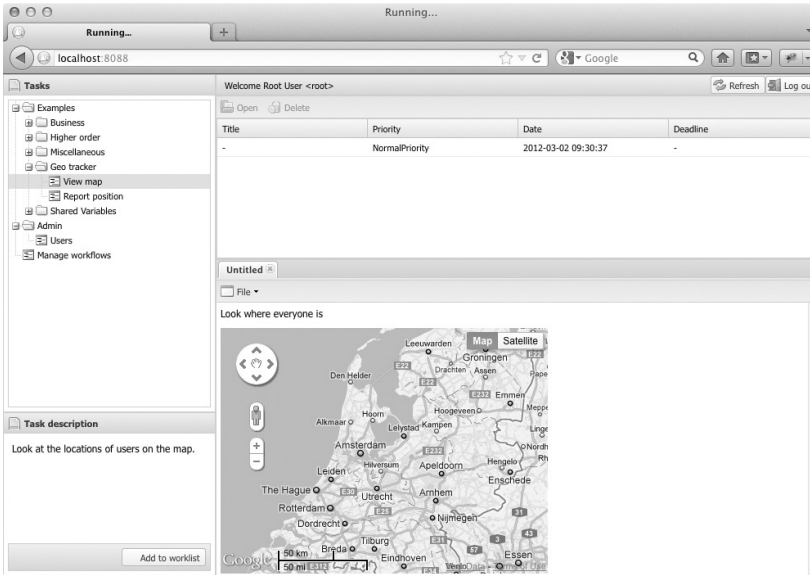


Figure 4.6: A generic WFMS Work List

example. In the left panel a tree of tasks that can be started is displayed. The tasks to do are displayed in the upper-right pane, similar to an inbox in a email application. The user can work on several tasks at the same time in the lower right pane, by opening them in separate tabs. This complete work list application is defined in less than 200 lines of TOP code.

The Incidone Incident Coordination Tool: The Coast Guard case study [30, 42] not only fueled the refinement of the task concept and the TOP paradigm, it also lead to the development of the Incidone tool [43]. A preview of this tool for supporting Coast Guard operations is shown in Figure 4.7. It is being developed using the iTask3 framework to illustrate the use of TOP for crisis management applications. In this tool immediate information sharing between team members working together is crucial to handle incidents properly.

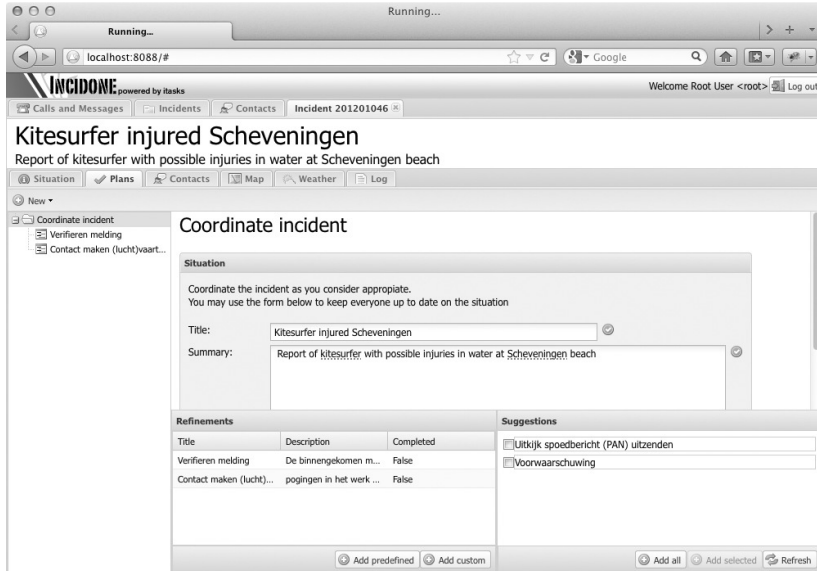


Figure 4.7: The Incidone Tool

4.5 Related Work

The TOP paradigm emerged during continued work on the iTask system. In its first incarnation [61], iTask1, the notion of tasks was introduced for the specification of dedicated workflow management systems. In iTask1 and its successor iTask2 [44], a task is an opaque unit of work that, once completed, yields a result from which subsequent tasks can be computed. When deploying these systems for real-world applications, viz. in telecare [77] and modeling the dynamic task of coordinating Coast Guard Search and Rescue operations [30, 42] we experienced that this concept of task is not adequate to express the coordination of tasks where teams constantly need to be informed about the progress made by others. The search for better abstraction has resulted in the TOP approach and task concept as introduced in this paper.

Task-Oriented programming touches on two broad areas of research. First the programming of interactive multi-user (web) applications, and second the specification of tasks.

There are many languages, libraries and frameworks for programming multi-user web applications. Some academic, and many more in the open-source and proprietary commercial software markets. Examples from the academic functional programming community include: the Haskell cgi library [51]; the Curry approach [24]; writing xml applications [14] in *SMLserver* [15]; WashCGI

[75]; the Hop [72, 47] web programming language; Links [10] and formlets [11]. All these solutions address the technical challenges of creating multi-user web applications. Naturally, these challenges also need to be addressed within the TOP approach. The principal difference between TOP and these web technologies is the emphasis on using tasks both as modeling and programming unit to abstract from these issues, including coordination of tasks that may or may not have a value.

Tasks are an ambiguous notion used in different fields, such as Workflow Management Systems (WFMS), human-computer interaction, and ergonomics. Although the iTask1 system was influenced and partially motivated by the use of tasks in WFMSs [1], iTask3 has evolved to the more general TOP approach of structuring software systems. As such, it is more similar in spirit to the Web-WorkFlow project [26], which is an object oriented approach that breaks down the logic into separate clauses instead of functions. Cognitive Task Analysis methods [12] seek to understand how people accomplish tasks. Their results are useful in the design of software systems, but they are not software development methods. In Robotics the notion of task and even the “Task-Oriented Programming” moniker are also used. In this field it is used to indicate a level of autonomy at which robots are programmed. To the best of our knowledge, TOP as a paradigm for interactive multi-user systems, rooted in functional programming is a novel approach, distinct from other uses of the notion of tasks in the fields mentioned above.

4.6 Conclusions and Future Work

In this paper we introduced Task-Oriented Programming, a paradigm for programming interactive multi-user applications in a pure functional language. The distinguishing feature of TOP is the ability to concisely describe and implement collaboration and complex interaction of tasks. This is achieved by four core concepts: 1) *Tasks observe intermediate values of other tasks* and react on these values before the other tasks are completely finished. 2) *Tasks running in parallel communicate via shared data sources*. Shared data sources enable useful lightweight communication between related tasks. By restricting the use of shared data sources we avoid an overly complex semantics. 3) *Tasks interact with users based on arbitrary typed data*, the interface required for this type is derived by type driven generic programming. 4) *Tasks are composed to more complex tasks using a small set of combinators*. The step combinator \gg subsumes the classic monad bind operator $\gg=$. The presented operational semantics specifies the constructs unambiguously. The development of this semantics was an important anchor point during the design of TOP.

TOP is embedded in Clean by offering a newly developed iTask3 library. We have used TOP successfully for the development of a prototype implementation

of a Search and Rescue decision support system for the Dutch Coast Guard. The coordination of such rescue operations requires up-to-date information of subtasks, this is precisely the goal of TOP. In collaboration with Dutch industry we started to investigate and validate the suitability of the TOP paradigm to handle specific complex real world distributed application areas.

Part II

**Types and Information
Models**

5 Between Types and Tables

In today's digital society, information systems play an important role in many organizations. While their construction is a well understood software engineering process, it still requires much engineering effort. The de facto storage mechanism in information systems is the relational database. Although the representation of data in these databases is optimized for efficient storage, it is less suitable for use in the software components that manipulate the data. Therefore, much of the construction of an information system consists of programming translations between the database and a more convenient representation in the software.

In this paper we present an approach which automates this work for data entry applications, by providing generic versions of the elementary CRUD (Create, Read, Update, Delete) operations. In the spirit of model based development we use Object Role Models, which are normally used to design databases, to derive not only a database, but also a set of data types in Clean to hold data during manipulation. These types represent all information related to a conceptual entity as a single value, and contain enough information about the database to enable automatic mapping. For data entry applications this means that all database operations can be handled by a single generic function.

To illustrate the viability of our approach, a prototype library, which performs this mapping, and an example information system have been implemented.

5.1 Introduction

In today's digital society, information systems play an important role in many organizations. Many administrative business processes are supported by these systems, while others have even been entirely automated. While the construction of such systems has become a more or less standardized software engineering process, the required amount of effort remains high. Because each organisation has different business processes, information systems need to be tailored or custom made for each individual organisation.

One of the primary functions of information systems is to create, manipulate and view (large) persistent shared collections of data. The de facto storage mechanism for these data structures is the relational database, in which all

information is represented in tables with records that reference other records. Although this representation is optimized for redundancy free storage of data, it is less suited for direct manipulation of that data. The reason for this is that conceptually elementary units are often split up into multiple database records. For example, in a small business system, a project consisting of a name and a number of tasks is broken down into one record for the project and a number of records for the tasks which each reference the project.

In data entry applications it is more convenient for developers to do operations on conceptual units instead of single database records. To reuse the example, adding a project instead of adding a project record and a number of task records. Therefore, in the programming language we use to build the data entry components, we need data structures that represent conceptual units rather than database records. While it is easy to construct a type in most modern languages to represent a conceptual unit as a single data structure, using any type more complex than a single database record means that some translation is required whenever data enters or leaves the database. As a result, since each system has a unique database design, a lot of boiler plate code has to be written to achieve this translation. This translation code is all very similar except for the types and tables they translate between. Even when a DSEL is used to abstract the database interaction from low level SQL, one still has to define the mapping for each new type. This repetitive programming work is not only mind numbing for developers, it is also time consuming and error-prone. Over the years several tools and libraries have been developed to solve this issue with varying degrees of success and practical use. We discuss these approaches in detail in Section 5.6.

In this paper we present a novel approach based on generic programming in Clean that provides generic versions of the elementary CRUD (Create, Read, Update, Delete) operations that abstract over types and tables. These operations map changes in data structures that reflect the conceptual unit structure of entities, to changes in a relational database. The main prerequisite for enabling this, is that all necessary information about the entities' database representations can be inferred from the types of these data structures. In the spirit of model based development, we do this by deriving both the data types and a relational database from the same high level data model. The language we use for these models is Object Role Modeling (ORM). In this graphic modelling language one can specify what information is to be stored in an information system by expressing facts about the modelled domain. Since ORM has a formally defined syntax and semantics, it enables the derivation of a set of database tables, as done in the standard Rmap algorithm [49], or a set of types in our approach.

Our approach consists of four mappings between representations on different levels of abstraction that are depicted in Figure 5.1. The first step (1) is a

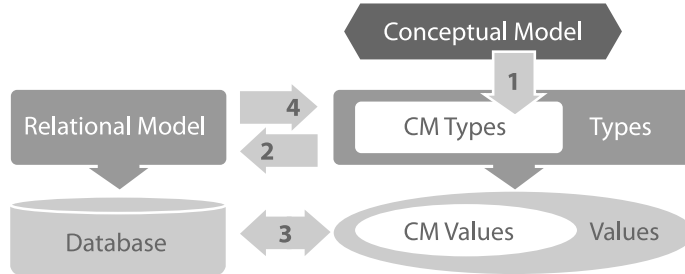


Figure 5.1: The four steps in our method.

mapping from ORM models on a conceptual level to a set of Clean types on the type level. From these types we derive a matching relational model for storage in a database (2). Our generic library is then used at the value level (3) to do CRUD operations on values of the representations where it automatically maps the values to the database. For many existing databases it is also possible to reverse engineer a set of representation types from a relational model (4).

The key idea behind our approach is that it addresses the representations of data for storage and manipulation as two sides of the same coin. Instead of focusing on either using databases as storage for Clean values, or on Clean values as interface to a storage representation, we consider Clean values and databases as different representations of the same high-level concepts.

Although our approach involves many stages of the software engineering process, we consider the following to be the main contributions of this paper:

- We introduce a structured method to derive Clean data types from ORM models, that allow capturing all information about a conceptual entity in a single data structure. The details of this process can be found in Section 5.3.
- We present a generic library which provides the four CRUD operations as functions on single data structures. These operations also work when the representations in the database span multiple tables. Especially in data entry applications, this library can replace much boiler plate code. The CRUD operations are covered in Section 5.4 and the implementation of the library is discussed in Section 5.5.

5.2 Motivating Example

To illustrate the various steps in our approach, and to provide some feeling about how it can be applied, we will make use of a running example throughout the remainder of this paper. This example is a simple project management system for a typical small business, which stores information about the following conceptual entities:

- **Projects** are abstract entities which are identified by a unique project number and have a textual description. Projects are containers for tasks and can be worked on by employees. A project can be a sub project of another project and can have sub projects of its own.
- **Tasks** are units of work that have to be done for a certain project. They are identified by a unique task number and have a textual description. The system also keeps track of whether a task is finished or not.
- **Employees** are workers that are identified by a unique name and have a description. They can be assigned to work on projects. An employee can work on several projects at a time and multiple employees may work on the same project.

5.2.1 ORM Formalization

To enable our generic mapping we need to make the above specification more precise. Using ORM [22], we can make a formal conceptual model of the example as shown in Figure 5.2. Using ORM, one models *facts* about *objects*. Facts are expressed as semi-natural language sentences. For example: “**Employee** *a* works on **Project** *b*”. An ORM model abstracts over concrete facts about concrete objects by defining *fact types* (the boxes) and *object types* (the circles). Unlike other data modeling languages like ER [9] or UML[76], ORM does not differentiate between relations and attributes, but considers only facts. ORM also models several basic constraints on the roles that objects have in facts. One can express uniqueness, meaning that a fact about some combination of objects occurs at most once, and mandatory role constraints which enforce that a fact about a certain object must occur at least once. In Figure 5.2, these constraint are depicted as arrows spanning unique combinations of roles, and dots on roles that are mandatory.

5.3 Types and Tables

The key idea on which our approach is based is that, in data entry applications, we want to manipulate single data structures that represent a conceptual unit.

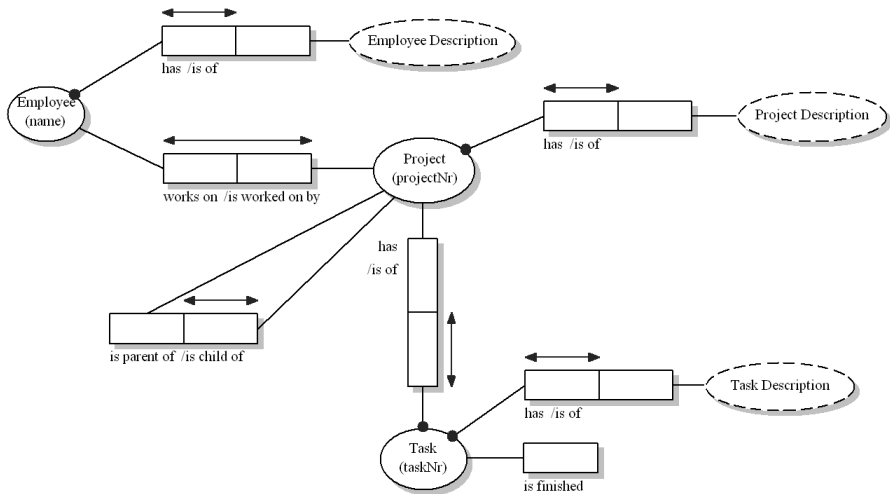


Figure 5.2: A simple ORM model for a project management system

What we do not want, is to manually specify the queries required to build such data structures, or to specify how to update the database after the data structure has been altered. Unfortunately this is often necessary because, since types in data entry applications are often defined ad-hoc for a separately designed database, the relation between types and tables is unclear and inconsistent. We improve this situation by using a structured process. Since a relational database, and the Clean types used for manipulating it, are simply two different representations of the same abstract entities, the obvious thing to do is define a high level specification of these abstract entities and use it to derive both representations. In the next section we show that when enough information about the storage representation of objects can be inferred from the types of their corresponding Clean representation, we can define an automated mapping once and for all using generic functions. In this section we show how we can obtain a set of types and tables for which this property holds.

5.3.1 Object Role Models

Instead of defining our own language for defining conceptual entities, we use an existing language from the information modeling field: Object Role Modeling. However, for reasons of simplicity, our approach only considers ORM models that satisfy the following constraints:

- The model only contains entity types, value types and unary and binary fact types.

- Each entity type can be identified by a single value.
- Uniqueness constraints on single facts and mandatory role constraints are the only constraints used.
- Each fact type has at least one uniqueness constraint.
- Uniqueness constraints spanning two roles are only used for facts concerning two entity types.

Although this subset of ORM neglects some advanced ORM constructs, like subtyping or n-ary fact types, it has roughly the same expressive power as the widely used Entity Relationship (ER) [9] modeling language, and is sufficient for most common information systems. Nonetheless, we still use ORM instead of ER because it allows extension of our method to even more expressive conceptual models in the future.

5.3.2 Representation Types

Although a solid conceptual model is the basis of a well-designed information system, from a programmers perspective however, we are more interested in the concrete representation as types in our (Clean) applications.

Conceptual entities can have different types of relations and constraints. When we want to represent conceptual objects as single Clean data structures we need types that can contain all facts about an entity and also retain information about constraints and relations. This is achieved by defining a subset of Clean's record types with meaningful field names. This set is defined as follows:

- **Entity Records**

Clean records are used as the primary construct to represent conceptual entities. These records have the same name as the entity type they represent, and have fields for every fact type concerning an entity. The names of these fields have a mandatory structure which can have the following three forms:

- `<entity name>.<value name>`

- This form is used for values or entities that have a *one-to-one relationship* with this entity. The entity name is a unique name for this entity type, typically the same as the name of the record type. The first field of an entity record must always have this form and is assumed to be a unique identifier for the current entity.

- `<entity name>_ofwhich_<match name>`

- This form is used for embedding relations between two entities where the relation between the two entities is defined such that the value of

the match name of one of the entities is equal to the identity value of another entity. This form is used for *one-to-many relations* between entities. The entity name is the identifier of the “many” part of the relationship. The current entity is the “one” side of the relation.

- `<relation name>.<select name>_ofwhich.<match name>`
This form is used for *many-to-many relationships* between entity types. The relation name is a unique name for this relation and is used by both entity records that have a role in the relation. The select and match names are role identifiers for both parts of the relation.

The types that fields in an entity record are allowed to have, are limited as well. They can be of scalar type, another entity or identification record type, or `Maybe` or list of scalar or entity or identification record type.

- **Identification Records**

Because we do not always want to store or load an entire database, we need a representation for references to entities that stay in the database. We represent these references as identification records. These are records that have the same name as the entity record they identify, with an “ID” suffix. These records contain exactly one field which has the same name and type as the corresponding entity record.

- **Scalar Types**

Value types in ORM are mapped to the basic scalar types in Clean: `Int`, `Bool`, `Char`, `String` and `Real`.

- **List and Maybe types**

When the uniqueness and total role constraints on a fact type define that a fact can be optional, or can have multiple instances, we use Clean’s list (`[a]`) and `Maybe` (`::Maybe a = Nothing | Just a`) type to wrap the type of the object involved in the fact. It is important to note that the order of lists is considered to have no meaning in these types. Storage of an entity record which contains a list does therefore not guarantee that this list has the same order when read again.

Using these types, the ORM model of our project management system (Figure 5.2) can be represented by the set of Clean types given below.

```

1 :: Employee = { employee_name           :: String
2                , employee_description  :: String
3                , projectworkers_project_ofwhich_employee :: [ProjectID]
4                }
5 :: EmployeeID = { employee_name         :: String
6                 }
```

```
7 :: Project = { project_projectNr      :: Int
8               , project_description   :: String
9               , project_parent        :: (Maybe ProjectID)
10              , task_ofwhich_project  :: [Task]
11              , project_ofwhich_parent :: [ProjectID]
12              , projectworkers_employee_ofwhich_project :: [EmployeeID]
13              }
14 :: ProjectID = { project_projectNr      :: Int
15                }
16 :: Task = { task_taskNr      :: Int
17            , task_project    :: ProjectID
18            , task_description :: String
19            , task_done        :: Bool
20            }
21 :: TaskID = { task_taskNr      :: Int
22              }
```

An interesting property of these types is that, unlike database records these Clean records can also contain nested representations of related objects.

5.3.3 From ORM To Representation Types

To make sure that a set of representation types represent the right concepts, we systematically derive the types from an ORM model (mapping (1) in Figure 5.1). The algorithm to perform this mapping groups fact types in a similar fashion as the standard Rmap [49] algorithm and is summarized below. A more elaborate description can be found in [40].

1. For each entity type in the ORM, define an entity and identification record in Clean. They both have one field, which will have the name and type of the primary identification of the entity in ORM.
2. Add fields to the entity records. Each entity record will get a field for all the fact types in which it plays a role. The types and names of the fields are determined based on the object types and constraints in the model.
 - When the entity type is related to another entity type, the type of the field is the identification record for that entity. When it is related to a value type, the field will have a scalar value. The name of the field may be freely chosen but has to be prefixed with a globally unique entity identifier. The obvious choice for this is the name of the entity type.
 - When the fact type is unary, the field's type will be `Bool`.
 - When there is no mandatory role constraint on the role an entity is playing, the field's type will be a `Maybe` type.

- When there is no uniqueness constraint on the role an entity is playing the field's type be a list type.
 - Each field name is prefixed with a grouping identifier. If a fact type can be attributed completely to the entity we are defining the type for, we use the name of the entity as prefix. If not, we choose a unique prefix for that fact type, that is to be used in the entity records of both entities playing a role in the fact type.
3. Optionally replace identification record types in record fields to entity record types. This allows the direct embedding of related entities in the data structure of an entity. One has to be careful however to not introduce "inclusion cycles". When an included related entity embeds the original entity again, a cycle exists which will cause endless recursion during execution.

Because step 3. is optional, and the choice between inclusion or reference depends on the intended use of the representation types, this transformation can only be automated by an interactive process or annotation of the ORM model.

5.3.4 From Representation Types to Tables

The next step in our approach is getting from a set of representation types to a relational model (mapping (2) in Figure 5.1). The obvious way would be to map from ORM directly to a relational model as is done in the standard Rmap algorithm [49]. However, since the representation types are already very close to the relational structure, it is easier to derive the tables from these types. A summary of the mapping process is given below. A more detailed version can be found in [40].

1. Define tables for all entities. In these tables all record fields are grouped that have the same entity name as the first (identification) field of the record. The types of the columns are the types of the record fields in the case of scalar types. In the case of entity or identification records the column gets the type of the first field of these record types. When a record field's type is a `Maybe` type, the corresponding column is allowed to have `NULL` values.
2. Define tables for all many-to-many relations. For all many-to-many relationships find the pairs of relation names and define a two-column table by that name. The names of the two columns are the entity names found in the record fields in the representation types.

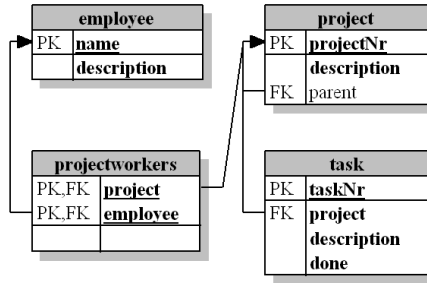


Figure 5.3: The derived tables of the ORM model in Figure 5.2

3. Add foreign key constraints. Everywhere an entity or identification type in the record field is mapped to a column in a table, a foreign key constraint is defined that references the primary key of the table of the corresponding entity type.

When this algorithm is applied to the set of representation types of Section 5.3.2, we get the set of database tables depicted in Figure 5.3. Since this algorithm is completely deterministic it can be easily automated.

With this mapping, we have done all the preparatory work that is required to use our generic library. For new information systems, this is all the initial work one has to do: Define an ORM model, derive a set of representation types and derive a relational model from those types.

5.3.5 Reverse engineering: From Tables to Representation Types

In situations where we already have a database, that we want to interface with, we still want to be able to use our generic library. In many situations we are able to reverse engineer a set of representation types from an existing relational model to make this possible.

The process itself (mapping (4) in Figure 5.1) is a rather trivial inverse operation of the method to derive a relational model from the representation types. However, this is only possible under certain conditions:

- The relational model must only contain tables indexed on a single column primary key that represent entities and two column tables with a primary key spanning both columns that represent additional facts. When this condition holds, there exists a set of representation types from which we could have derived the existing database.

- We must know which columns are used as references, and what entities they reference. Since the use of foreign keys is not obligatory, it is not always possible to infer the references in a relational model. We can only define a set of representation types if we know which conceptual entities are related and how.

When these conditions hold, which often do for simple information systems, we are able to use our library even in situations where no ORM model of the existing system is available. When these conditions do not hold for the complete database, but do hold for a part of the database, it is still possible to define a set of types for that part. Such partial use of the generic mapping, can still save a lot of work.

5.4 Generic CRUD Operations

Although having Clean types and database tables that have a clear relation with a formal conceptual model is a merit on its own, the point of that exercise was to enable generic CRUD operations.

What we want to achieve is a set of four generic functions that are available for every possible representation type and enable us to manipulate the entities in the databases of our information systems. Ideally the type definitions of this set would look somewhat like the following code:

```
1 create :: entity db → (ref, db)
2 read  :: ref  db  → (entity, db)
3 update :: entity db → db
4 delete :: ref  db  → db
```

Here `ref`, `entity` and `db` are type variables for respectively the identification record type, the entity record type and a database cursor type. Obviously it is not possible to create such a completely polymorphic set of functions, but we can come very close using generics and overloading.

In this section we show how two of these operations, `read` and `update`, work by means of an example. The other two are similar and are not covered for the sake of brevity. A full detailed description of all four operations can be found in [40]. In the example we will assume the conceptual model of Figure 5.2, the types of Section 5.3.2, and the database tables of Figure 5.3.

5.4.1 Reading objects

The first operation we will show is the generic `read`. Suppose we have a database with the following information about some project:

- It has `projectNr` 84, description “Spring brochure” and the project has no parent project and no sub projects.

- A task is defined for this project with taskNr 481 and description “Draft text” which is not done yet.
- Another task is defined with taskNr 487 and description “Call printer about price” which is also not done yet.
- Employees “john” and “bob” are working on this project.

All of this information can be read into a single Clean value of type `Project` in just one line of code¹:

```
1 (mbError, mbProject, cur) = gsql_read {ProjectID|project_projectNr = 84} cur
```

If all goes well, this will give us the following data structure:

```
1 { Project | project_projectNr = 84
2   , project_description = ‘‘Spring brochure‘‘
3   , project_parent = Nothing
4   , task_ofwhich_project
5 = [ { Task | task_taskNr = 481, task_project = 84
6     , task_description = ‘‘Draft text‘‘, task_done = False
7     }
8   , { Task | task_taskNr = 487, task_project = 84
9     , task_description = ‘‘Call printer about price‘‘, task_done = False
10    } ]
11 , project_ofwhich_parent = []
12 , projectworkers_employee_ofwhich_project
13 = [ {EmployeeID | employee_name = ‘‘john‘‘ }
14   , {EmployeeID | employee_name = ‘‘bob‘‘ } ]
15 }
```

This single line of code gives us a lot for free. If we had to write a `read_product` function by hand, it would have required three different SQL queries plus a conversion from flat lists of SQL data values to the nested `Project` structure. To achieve this generically, two problems have to be solved: 1. How do we find the information in the database? And 2. how do we construct a value, in this case of type `Project`? The first problem is solved by interpreting the field names of record types and translating them to SQL queries. The results of these queries are then systematically concatenated to produce a stream of values (tokens) which is a serialized representation of the value we want to construct. This reduces the second problem to deserialization of that representation. Instead of describing the read operation at an abstract level, it is easier to see what happens by following it step by step when used to read the `Project` described above.

¹In this code the variable `cur` is a unique database cursor used to query the database.

1. The first step we take is serialization of the `ProjecID` value to create an initial token stream. Thus in this case, the read operation is started with initial stream `[84]`².
2. The next step is to apply the instantiation of the generic read operation for the `Project` type. When the read operation is applied to read an entity record, the first thing that is done is to expand the token stream by reading additional data for all fields of the record. The head of the token stream is used to match database records and the SQL queries are constructed from the information encoded in the field names. For example, the data for the field `project_description` is retrieved with the SQL query: `SELECT description FROM project WHERE projectNr = 84`. When an optional field is empty a `NULL` token is added to the stream and when a field has multiple values a terminator (`TRM`) token is added after the last value.

So for the example project, the token stream has the value after expansion: `[84, "Spring brochure", NULL, 481, 487, TRM, TRM, "john", "bob", TRM]`

3. With the data for all project fields read, the read operation is applied recursively to construct the record fields. When the read operation is instantiated for basic types or identification records no additional data is read. Instead, tokens are consumed to construct values. So after the values of the first three fields (`84`, `"Spring brochure"` and `Nothing`) are constructed the token stream has the value: `[481, 487, TRM, TRM, "john", "bob", TRM]`
4. The instantiation of the read operation for lists will repeatedly apply the read operation for its element type until the head of the token stream is a terminator. So in this case, the create operation for type `Task` will be called twice. Because `Task`, like `Project`, is an entity record type, we read additional data again. After expansion of the first task the stream has value: `[481, 84, "Draft text", false, 487, TRM, TRM, "john", "bob", TRM]`

When the list of both tasks is read and constructed the stream is reduced to: `[TRM, "john", "bob", TRM]`

5. Thus the process continues, and when recursion is completed for all fields we have an empty token stream and can construct the `Project` record.

²To illustrate the intermediate values of the token stream we use an ad-hoc untyped list notation. This is **not** Clean syntax.

5.4.2 Local changes with global meaning

Once all facts about an object are read into a Clean data structure, we can change it in a program. Because this structure is not just some convenient grouping of values for computation, but has a meaningful relationship with both the underlying conceptual model and the relational model in the database, we can interpret changes to this data structure as changes on the conceptual level.

To illustrate this we make some changes to the example `Project` of the previous section and consider their meaning on the conceptual level.

- We change the value of the `project_description` field to ‘‘Summer brochure’’. The meaning of this change is simple. Since each project has exactly one description, this new description will replace the old value in the database.
- We change the value of the field `task_done` of the first `Task` in the list to `True`.

The meaning of this change is simple as well. Since each task is either done or not, this new value will replace the value in the database. So although the task is embedded in the project value, it is still a separate object on the conceptual level which facts can be changed.

- We remove the second `Task` from the list.

The meaning of this change is less obvious. Since tasks and projects are both conceptual objects that happen to be related, does a removal from the list mean that the conceptual task object and all its facts are removed? Or does it mean that just the relation between the task and project is removed? For the representation types, this choice is dependent on the used type. For entity records, like `Task`, we will interpret removal of the list as complete removal of the object. For identification records, like `TaskID`, we will only remove the relation between objects. Thus in this case task 487 will be deleted completely.

- We add a new `Task` defined as:

```
1 { task_taskNr = 0, task_project = {ProjectID | project_projectNr = 0}
2   , task_description = ‘‘Check online prices’’, task_done = False
3   }
4
```

This change means that a new task for this project has to be created. The interesting parts however are the `task_taskNr` and `task_project` fields. Each task is related to exactly one project. We have specified in the task record that this is project 0. But this task is created as part of the list

of tasks of project 84. When new objects are created in the context of another object we will let the context take precedence and ignore the specified identification. Hence, this change means that a new task is created which is related to project 84, not 0.

The `task_taskNr` field is also interesting. For the identification of new objects we interpret the specified value (0) as a suggestion, but leave it up to the database to determine the actual value. This enables the use of auto incrementing counters which are commonly used in databases.

- We remove ‘‘john’’ from the list in `projectworkers_employee_ofwhich_project`. Because the `projectworkers_employee_ofwhich_project` field is a list of identification records, we will interpret the removal of ‘‘john’’ from this list as ‘‘john no longer works on this project’’ and not as complete removal of the employee named ‘‘john’’ from the database.

5.4.3 Updating objects

In the previous section we have made quite a few changes to our local representation of the project, but all of these changes can be applied to the global representation in the database at once with just the following single line of Clean code:

```
1 (mbError, mbProjectId, cur) = gsql_update project cur
```

This single line saves us even more programming work than the generic read function. To apply all the changes by hand would in this case require six custom crafted SQL queries and the necessary conversion code.

As with the read operation, we illustrate the generic update by following its operation step by step.

1. The update operation for entity records is done in three recursive passes. In the first pass we consider only the fields that are single basic values or identity records. In this case the fields that start with `project_`. The update operation on basic values and identification records does no database interaction, but just serializes values to produce the token stream. After this first pass the token stream has the value: [84, ‘‘Summer brochure’’, NULL].
2. After this pass we update the database record for this project. Because new objects can be added (like the new task) we verify that the update query did indeed modify a record in the database. If not, we create a new record. After this update/create we know the definitive identification of this project (84) and are ready for the next pass.

3. In the second pass we will do a recursive update of the remaining record fields. To make sure that the identification context object takes precedence when updating nested objects we pass along special override tokens (OVR) that specify for which fields in the nested entity records the context must be used instead of its value. In this case the second pass is started with token stream: [OVR task_project ⇒ 84, OVR projectworkers_project ⇒ 84]. The override tokens are used during serialization in the first update pass of a nested entity record. When the second pass finishes the resulting token stream has value: [481, 532, TRM, TRM, "bob", TRM]. The value 532 is an automatically assigned identification for the newly created task.
4. In the third and final pass, the token stream of the second pass is compared with the token stream that a (non-recursive) read operation is for this project produces to determine which list elements have been removed. For these values, the generic delete operation is used to remove them from them from the database.
5. After these three passes, the identification value of the current record is added to the token stream it was started with. In this case returning a token stream of value: [84].
6. The final step is to deserialize the token stream to produce a ProjectID value.

5.4.4 Shared consequences

An interesting property of the previously illustrated generic operations is that changes in one object have consequences for related objects. Because facts are conceptually shared between objects, the operations maintain that shared structure in the database. If we would have read the `Employee` record of ‘john’ before going through the example, the list in the `projectworkers_project_ofwhich_employee` would have contained the value `{ProjectID|project_projectNr=84}`. If we would read it again after updating the project, this value would no longer occur in the list.

5.5 Implementation in Clean

To validate the generic operations, we have implemented the operations described in the previous section as a prototype library in Clean called “GenSQL”. This library contains about 950 lines of Clean code of which roughly 500 are used for the definition of the main generic function. The rest constitutes about fifty helper functions. Because of its large size, it is not possible to present the generic function in detail. The design of the library as a whole is therefore presented instead³.

³Full sources of both the library and the demo application can be found at:

5.5.1 Jack of All Trades

Because the generics mechanism in Clean has some limitations, the implementation of the operations in the GenSQL library has a somewhat unusual design. In Clean it is not possible to call other generic functions of unknown type in the definition of a generic function. The different CRUD operations however, do have some overlap in their functionality. The update operation, for instance, uses the delete operation during a garbage collect step. Because of the limitation we are not able to isolate this overlap in a separate generic function. To deal with this limitation of the generics mechanism, all operations have been combined into one “Jack of all trades” function. The type signature of this function, `gSQL`, is as follows:

```
1 generic gSQL t ::
2   GSQLMode GSQLPass (Maybe t) [GSQLFieldInfo] [GSQLToken] *cur   →
3   ((Maybe GSQLError), (Maybe t), [GSQLFieldInfo], [GSQLToken], *cur) |SQLCursor cur
```

The first two arguments of this function are the mode and pass of the operation we want `gSQL` to perform. The mode is one of the four operations `GSQLRead`, `GSQLCreate`, `GSQLUpdate`, `GSQLDelete`, the type information `mode GSQLInfo` or `GSQLInit`. The latter serializes a reference value to the token list in order to start a read or delete operation. The `GSQLPass` type is simply a synonym for `Int`.

The next three arguments are the data structures on which the `gSQL` function operates. All three are both input and output parameters and depending on the mode, are either produced or consumed. The first argument is an optional value of type `t`. During the read and delete operations, this argument is `Nothing` in the input and `Just` in the output because values are constructed from the token list. During the create, update, info and init operations, the argument is `Just` in the input because values are serialized to the token or info list. The second argument is the token list to which data structures are serialized. The third argument is the info list. In this list, type information about record fields is accumulated. The last argument of the `gSQL` function is a unique database cursor which has to be in the `SQLCursor` type class⁴. This is a handle which is used to interact with the database. The return type of the `gSQL` function is a tuple which contains an optional error an optional value of type `t`, the token list, the info list and the database cursor.

Although this “Jack of all trades” function is large, it is clearly divided into separate cases for the different types and modes to keep it readable and maintainable.

<http://www.st.cs.ru.nl/papers/2009/gensql-prototype.tgz>

⁴A | in a type signature is Clean notation for specifying class constraints

5.5.2 Convenient wrappers

Because of the all-in-one design of the `gSQL` function, it is not very practical to use. For the read and delete operations, it even has to be called twice. First in the init mode to prepare the token list, and then in the read or delete mode to do the actual work.

To hide all of this nastiness from the programmer, the GenSQL library provides wrapper functions for each of the four operations. These wrappers have the following type signature.

```
1 gsql_read  :: a *cur → (Maybe GSQLError, Maybe b, *cur)
2     | gSQL{[*]} a & gSQL{[*]} b & SQLCursor cur
3 gsql_create :: b *cur → (Maybe GSQLError, Maybe a, *cur)
4     | gSQL{[*]} a & gSQL{[*]} b & SQLCursor cur
5 gsql_update :: b *cur → (Maybe GSQLError, Maybe a, *cur)
6     | gSQL{[*]} a & gSQL{[*]} b & SQLCursor cur
7 gsql_delete :: a *cur → (Maybe GSQLError, Maybe b, *cur)
8     | gSQL{[*]} a & gSQL{[*]} b & SQLCursor cur
```

Thanks to Clean's overloading mechanism we can use these wrapper functions for any entity for which we have derived `gSQL` for its identification (a) and entity record (b) type.

5.5.3 Project management example system

In order to test and demonstrate our generic library, we have also implemented the project management system from Section 5.2. This system is a CGI web application written in Clean which runs within an external (Apache) web server and stores its information in a (MySQL) relational database using the GenSQL library. Figure 5.4 shows the prototype application while updating a project.

5.5.4 Performance

The generic mapping function relieves the programmer of writing much boilerplate code and SQL queries. It is however important to realize that there is a cost associated with this convenience.

First of all there is some overhead cost in space and time consumption of Clean's generic mechanism. However when optimization techniques [4] are applied by the compiler this can be completely removed.

Secondly there is a cost in the amount of database queries that are performed. The current implementation of the generic operations is not optimized to minimize the amount of queries. Each retrieval or update of an object does a separate query. When an object has many facts with embedded related objects this will result in linearly many queries. Theoretically however, there is no reason why the generic operations would require more queries than handwritten versions.



Figure 5.4: Screenshot of the project edit page

5.6 Related Work

At first glance, our library appears very similar to Object Relational Mapping [19] libraries in object oriented languages. These libraries achieve persistence of objects in an OO language by mapping them to a relational database. Although both approaches relieve programmers of the burden of writing boilerplate data conversion code, there is an important difference: our approach treats a subset of all Clean types as a meaningful model of an underlying redundancy free database. This allows us to easily map binary fact types to the entity records of both sides without duplicating any information in the database. In object relational mapping where objects are made persistent, we can only avoid duplication by mapping binary relations between objects to only one side of the relation. Based on this property, object relational mapping is more similar to generic persistence libraries [73] than to the method presented in this paper. Also related to our work are other methods and tools that use conceptual data models to generate parts of an information system like user interfaces [34], or even complete applications [48]. These tools reduce the effort required to build an information system as well, but are often all-or-nothing solutions that do a certain trick well, but have no solution when you want something a little different. Of course you can always make changes to the generated code, but this means you can only generate once, or have to manually merge your changes

upon regeneration. Because our approach is designed as a generic library, and generic programming is an integral part of the Clean language, we can combine a generic solution for common situations together with handwritten code for exceptional situations in one coherent and type safe solution.

The final related area of research is that of abstraction from SQL by embedding a query language inside another language. This approach is used in the HaskellDB library in Haskell [39, 6], in the LINQ library in C# [52], and more recently, using dependent types in a database library for Agda [57]. While these approaches make the programming of data operations easier and type safe, they do not reduce the amount of work one has to do. When using our library, a developer no longer needs to define queries at all, thus eliminating the need for easier and safer ways of defining them. These libraries could however, be used complementary to ours to get a generic solution for the common CRUD operations, and type safety for the exceptional custom queries.

5.7 Conclusions & Future Work

In this paper we have shown that given the right choice of data types and database tables, it is possible to use generic programming to automate the mapping between entities stored in a database and their representation in Clean.

To do so, we have shifted the focus from both the database and the data types, towards the conceptual level of ORM models. By deriving not only a database, but also a set of types from these models, we enable an automatic mapping between them. This means that by just making an ORM model of a perceived system, you get a database design, a set of types for convenient manipulation, and the machinery for doing CRUD operations on values of those types for free. This relieves a Clean programmer of dealing with how changes in a database must be expressed in SQL, and instead enables the manipulation of a database in a more familiar fashion: manipulation of a local data structure.

We have shown the viability of this approach by means of a prototype library and its use in an example information system. While not ready for production systems yet, this library is already useful for rapid prototyping. But, with optimization of the library, and additional generic operations for handling sets of entities, much of the construction effort of information systems can be reduced to just the definition of ORM models.

What remains to be done is extension of our approach to the complete ORM language. While we selected a subset which is useful for many domains, we have ignored some constructs that make ORM more powerful than, for example, ER. We have yet to investigate how these can be integrated in the current approach. Another area where further work can be done is to explore how the mechanism for locally manipulating parts of a global shared data structure can be used to facilitate sharing in a functional language. Could it for instance be used to implement a heap on top of an in-memory SQL engine?

6 CCL: A Lightweight ORM Embedding in Clean

Agile software development advocates a rapid iterative process where working systems are delivered at each iteration. For information systems, this drive to produce something working soon, makes it tempting to skip conceptual domain modeling. The long term benefits of developing an explicit conceptual model are traded for the short term benefit of reduced overhead. A possible way to reconcile conceptual modeling with a code-centric agile process is by embedding it in a programming language. We investigate this approach with CCL, a compact textual notation for embedding Object-Role Models in the functional language Clean. CCL enables specification of Clean types as derivatives of conceptual types. Together with its compact notation, this means that defining data types with CCL as intermediary requires no more programming effort than defining data types directly. Moreover, because embedded ORM is still ORM, mappings to other ORM representations remain possible at any time.

6.1 Introduction

The foundation of a successful information system is a solid understanding of the domain it represents. A way of capturing such understanding on a conceptual level is through the use of Object-Role Models (ORM [23]). They allow modelers to express the conceptual relationships in a domain without committing to the level of detail that implementation data structures require. A common approach to ORM is to make models with dedicated tools as analysis and design activity, and use them to bootstrap development of information systems by generating relational schemas and template code. An implicit assumption of this approach is that the development process has a requirements analysis or design phase prior to a construction or programming phase. With the increasing popularity of Agile development [18] this assumption does no longer always hold. Agile development advocates an iterative process with short cycles in which working systems are delivered at each iteration and stakeholders are actively involved in the development. In such a process, with a focus on delivering working systems in a short amount of time, it is tempting to skip conceptual modeling, because the overhead of using dedicated modeling tools while already

programming an information system may outweigh the short-term benefits. Unfortunately this means that the long-term benefit of an explicit conceptual model that can be used in communication with stakeholders is also lost.

A possible way to reconcile conceptual modeling with a code-centric Agile development process is by embedding ORM in a programming language. At first glance, this appears to be a compromise. For ORM modelers, it means that conceptual models have to be specified textually instead of graphically in order to let it be embeddable in the structured text of a programming language. For programmers this means that they cannot define data types directly, but that they need to specify a conceptual model of the domain first. Yet, we expect that embedding ORM in a programming language can have several benefits: First, conceptual relations between different data structures used in the code of an information system are often implicit. By making them explicit, data types can be specified more compactly as derivatives of shared conceptual types, giving programmers an immediate short-term benefit. It also gives compilers additional opportunities to check code and detect programming mistakes.

Second, because models are integrated in the source code of information systems there is a direct link between the model and the working information system. This means there cannot be discrepancies between the ORM models and the behavior of the information system caused by misinterpretation of the models during implementation. Moreover, embedded ORM models are still ORM models. This makes it possible to maintain bidirectional mappings to representations used by other ORM notations and tools.

In this paper we investigate the embedded ORM approach with CCL (Concepts in CLean), a compact notation for embedding ORM models in the pure functional programming language Clean [69]. Clean is a statically typed language where data types play an important role. Not only are they used to detect programming errors, but they are also for type-driven generic programming, a technique in which data types are used to parameterize algorithms. Contemporary Clean frameworks like iTasks [32] aim to improve agility by enabling large parts of information systems to be generated from abstract patterns that depend on types. This makes specification of data types a key element of agile Clean programming.

The main contributions of this paper are:

- We provide a new compact textual notation of a core subset of ORM.
- We demonstrate how this notation is used to embed ORM in a programming language.
- We extend the Clean language with the possibility to specify conceptual types underlying multiple data types.

The remainder of this paper is organized as follows: We start with an example of CCL in Section 6.2. We then continue with an explanation of the notation of

ORM constructs for defining conceptual models in CCL in Section 6.3 and for defining derivative data types in Section 6.4. We reflect on the benefits as well as the scope and limitations of our work in Section 6.5 and put it the context of related work in Section 6.6. In Section 6.7, we end with concluding remarks and an outlook on future work.

6.2 A CCL Example

Before going into technical details of the CCL notation, we present a simple, yet nontrivial example. We model a personal audio catalogue of digital audio files like for example a collection of ripped CD's or an iTunes library. The central concept is an album, which can be a music album or an audiobook. Additionally songs, artists and authors of audiobooks are modeled.

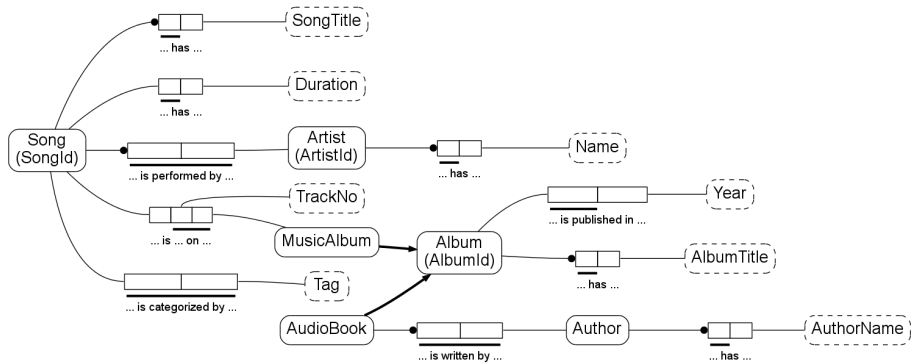


Figure 6.1: ORM diagram of a personal audio collection

Figure 6.1 shows the ORM2 diagram of such an audio collection which is defined by the CCL code in Figure 6.2. This definition consists of three parts. A series of object type definitions consisting of entity type definitions and value type definitions, a series of fact type definitions and a series of fact container type definitions that define Clean data types in terms of the conceptual definitions. In the next section we explain this definition in more detail.

6.3 Defining Conceptual Models with CCL

Conceptual models are defined as structured text in CCL source modules, text files with `.ccl` as extension. Each CCL module starts with a module header defining its module name.

```
concept module AudioCollection
```

```

1 //Module header
2 concept module AudioCollection
3 // Entity types
4 $$ Album
5 $$ AudioBook [Album]
6 $$ Author
7 $$ MusicAlbum [Album]
8 $$ Artist
9 $$ Song
10 // Value types
11 $$ Name = String
12 $$ SongId = Int
13 $$ SongTitle = String
14 $$ AlbumId = Int
15 $$ AlbumTitle = String
16 $$ ArtistId = Int
17 $$ Year = Int
18 $$ Duration = Time
19 $$ TrackNo = Int
20 $$ Tag = String
21 $$ AuthorName = String
22 // Fact type definitions
23 ## album_id =
24 << !Album >> has << AlbumId >>
25 ## album_title =
26 << !Album >> has AlbumTitle
27 ## album_year =
28 << Album >> is published in Year
29 ## song_id =
30 << !Song >> has << SongId >>
31 ## title =
32 << !Song >> has SongTitle
33 ## duration =
34 << Song >> has Duration
35 ## songs =
36 Song is << TrackNo on MusicAlbum >>
37 ## performed_by =
38 << !Song is performed by Artist >>
39 ## tags =
40 << Song is categorized by Tag >>
41 ## artist_id =
42 << !Artist >> has << ArtistId >>
43 ## artist_name =
44 << !Artist >> has Name
45 ## author_name =
46 << !Author >> has AuthorName
47 ## author =
48 << !AudioBook is written by Author >>
49 // Fact container types
50 #: Album = Album {...}
51 #: Artist = Artist {...}
52 #: Song = Song {...}

```

Figure 6.2: CCL Definition of Personal Audio Collection

By giving each module a name we make it compatible with the module system of Clean such that we can refer to its content from within other modules. The module header is followed by a series of declarations. These can be *object type* declarations, *fact type* or declarations, that define the model, or *fact container type* declarations that link the conceptual level to concrete first-order data structures. In Table 6.1 an overview of the CCL declarations is listed together with examples and their corresponding ORM diagrams.

6.3.1 Entity Types

Entity types are defined by declaring a name for a concept. Their declarations consist of an *object type marker*, two dollar signs, followed by an *object type name*. The object type marker is a symbol (\$\$) that indicates we are defining an object type declaration. The object type name must consist of alphanumeric characters only and must start with a capital letter.


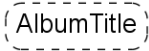
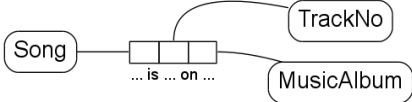
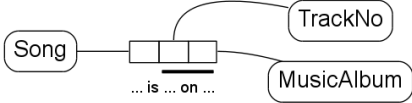
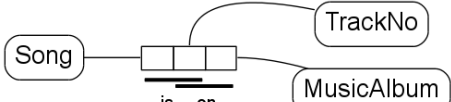


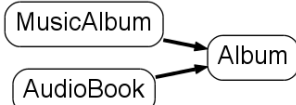
Entity Types	
\$\$ Song	
Value Types	
\$\$ AlbumTitle = String	
Fact Types	
## Song is TrackNo on MusicAlbum or ## songs = Song is TrackNo on MusicAlbum	
Uniqueness Constraints	
## Song is << TrackNo on MusicAlbum >>	
## <1< Song is <2< TrackNo >1> on MusicAlbum >2>	
Total Roles	
## !Song has SongTitle	
Primary Roles	
\$\$ SongId = Int ## << !Song >> has << SongId >>	
Subtypes	
\$\$ Album \$\$ MusicAlbum [Album] \$\$ AudioBook [Album]	

Table 6.1: CCL Language Constructs

Optionally a list of super types can be specified after the object type name to declare subtyping constraints. This list starts with an open bracket, is followed by entity type names separated by commas and ends with a closing bracket.

6.3.2 Value Types

Value types are defined by assigning a name to a first-order Clean type. They are declared the same way as object types, but with the addition of an equals sign, followed by the name of a first-order Clean type. Specification of super types is not allowed for value types.

6.3.3 Fact Types

Fact type declarations are defined by a *fact type marker* (`##`) followed by a sentence consisting of capitalized words and all lowercase words. The capitalized words are interpreted as names of object types.

It is allowed to reference object types that are not explicitly declared by an object type declaration. Undefined references are interpreted as implicit entity type declarations. It is recommended to explicitly declare entity types, but by allowing implicit use it is possible to construct a model by supplying fact types only. The CCL compiler can detect implicit references and issue warnings.

Optionally one can assign names to fact types by adding a *fact type name* followed by an equals sign between the fact type marker and the sentence. Assigning a name to a fact type makes it possible to reference the fact type in the declaration of *fact container types*.

6.3.4 Uniqueness Constraints

Uniqueness constraints are expressed by annotations in fact type declarations. Unique sets of roles can be marked by enclosing parts of the sentence with double angle brackets (`<< >>`). When multiple uniqueness constraints within the same fact type overlap making their definition becomes ambiguous. This can be resolved by labeling the constraints with a unique number. This number is placed between the angle brackets. Non-adjacent uniqueness can be specified by using the same label on multiple annotations.

6.3.5 Mandatory and Primary Roles

Mandatory role constraints are specified by prefixing object type references in a fact type declaration with an exclamation mark (`!`).

To indicate that a mandatory role is not only mandatory, but that the associated value type(s) may be used as reference for an entity type, we use an annotation to mark a role as the *primary* role. Because it is essentially a

stronger version of a mandatory constraint, we use a double exclamation mark as annotation (!!).

This annotation does not necessarily have to be used together with uniqueness constraints, but if it is used in a binary fact type between an entity type and a value type with two unique roles, the value type is depicted with the shorthand notation for standard names in the derived diagram. Each entity type can have only one primary role annotation.

6.4 Defining Clean Types with CCL

To manipulate facts in Clean programs, we need to represent them in compound data structures. Because Clean is a strong statically typed language that uses Burstall-style algebraic data types to type compound structures, we need to define (Clean) types of such data structures representing facts. To achieve this, CCL offers notation to define Clean types in terms of CCL fact types, which are automatically expanded. This way, the specification of CCL fact types reduces the specification effort of Clean types, because a single fact type can contribute to the definition of multiple Clean types if it relates to multiple entity types. Even for the simple model in Section 6.2 the CCL definition is more concise than the expanded Clean data types derived from it.

6.4.1 Fact Container Types

To define Clean types that can contain composite structures of facts, CCL offers so called *fact container types*. Their notation is similar to the notation for *record types* in Clean. For example:

```
#: SongSummary = Song {song_id,title,songs}
```

Which expands to the following Clean record type:

```
1 :: SongSummary =
2   { song_id :: SongId
3     , title :: SongTitle
4     , songs :: [(TrackNo,AlbumId)]
5   }
```

A fact container type definition starts with a *container type marker* (#:), followed by a type name and an equals sign. The righthand side of the equals sign consists of a *focus entity* and a selection of facts that get mapped to field names. The focus entity is the name of a conceptual entity type, that we collect facts about. It is implicit in all the facts that the type contains. The selection of facts is a comma separated list of named facts in which the focus entity has a role. Each fact maps to a field in the expanded record type. The types of the fields (in Clean denoted with the double colon) do not have to be specified

because they can be inferred from the conceptual model. For value types this is simply their Clean type. For entity types, the type of the fact type with a primary role annotation is used. This inference makes the type specifications more concise.

6.4.2 Complete Container Types

To make the definition of data types even more concise, a shorthand notation is available for defining fact container types that contain *all* facts about an entity. To define a type that expands to include all named facts a focus entity has role in, one can use the following notation:

```
#: Album = Album {..}
```

6.4.3 Explicit Field Type Specifications

If more control over the data types of fields in fact container types is desired, selected facts may be annotated with type information. To not just reference related entities, but to include facts about them one could for example specify:

```
#: SongSummary = Song {song_id,title,songs :: (TrackNo,Album)}
```

The name of a fact in the selection is annotated by a double colon followed by a comma separated list of Clean types enclosed in parenthesis for all roles except the role of the focus entity. For binary facts the parenthesis may be omitted because only one role remains.

6.5 Discussion

6.5.1 Conceptual Modeling

CCL is a lightweight embedding of ORM that provides notation for a core set of ORM constructs only. The reason for this is partially intentional and partially practical. Covering all ORM constraints fully would make the CCL language more complex while the additional value is uncertain. Because we are embedding ORM in a general purpose programming language, we don't need to be complete. Many constraints can also be expressed alternatively in Clean and addition would only duplicate existing functionality. The supported ORM subset adds the structural conceptual level that could not be expressed explicitly in Clean. Exploration of the effects of additional constraints remains a topic for future research.

Although unnecessary for the generation of Clean code, the CCL notation uses complete sentences to define fact types. For constraints however, it uses annotations instead of a more verbose verbal form. This approach aims to

provide a notation that is concise but still contains all information necessary for verbalization. The notation is therefore not as close to natural language as for example SBVR, but still starts from stating facts about a universe of discourse.

6.5.2 Effects on Agility

The use of CCL as intermediate step in the definition of collections of data types, may improve agility of a Clean programmer in two ways. First, it reduces the amount of code that has to be written, because CCL makes the definition of types more compact. Secondly, it automatically keeps the data types that represent entities that share fact types consistent. This makes it easier to incrementally extend the system under development, because changes that involve multiple conceptual entities can be made in one place.

A possible negative effect may be that too much of a domain is modeled. If in CCL more types are defined than are used in the current iteration of an information system, time is spent on something that does not contribute to the working system. Luckily this can easily be detected by the compiler.

6.5.3 Implementation

To be able to test and investigate the use of CCL in information systems developed with Clean, we have implemented a basic compiler. This is a proof-of-concept compiler that serves as a preprocessor to the Clean compiler. The primary purpose of our compiler is to compile CCL to Clean. Although CCL's syntax is designed to avoid conflict with Clean's syntax such that CCL can be mixed with Clean in the same module, we currently only support separate CCL modules that can transparently be compiled to Clean using the Clean IDE's preprocessing support.

The secondary purpose is to generate representations of the conceptual models for communication with domain experts. Currently we support the generation of ORM2 diagrams, but one could also think of verbalizations, or a combination of both in hyperlinked documents. Because, CCL does not allow the specification of diagram layout, we use the popular open-source Grapviz tool, to visualize CCL. The CCL compiler generates a graph structure in the DOT language which is rendered by Graphviz. All diagrams shown in Section 6.2 and Section 6.3 have been generated from CCL in this way.

Both the CCL notation, and the CCL compiler have many opportunities for improvement. Obvious additional targets are the generation of relational schema's for storage, and access functions for conversion between flat relational structures and CCL fact container types. Another area in which the CCL compiler could be improved, is interoperability with formats from tools such as Norma, or languages as CQL or SBVR.

6.6 Related Work

It is obvious that CCL as presented in this paper is not intended to replace any of the current ORM notations and tools, but rather to introduce ORM in a new context where explicit conceptual models have added value. Therefore it may be less obvious where to position CCL in the ORM literature. Unlike most concrete ORM notations, like the earlier NIAM [56], FCO-IM [5] or more recent ORM2 [23], CCL is a textual language with a formal concrete syntax, instead of a graphical language. In its textual approach it is closer to SBVR vocabularies [21] or CQL [25], but less natural language oriented. CCL is a pure data definition language. Where languages as RIDL [50] and LISA-D [74], as well as CQL incorporate the querying and manipulation of information, CCL defines only the conceptual structure. Manipulation of data is already provided by the host language Clean. In its aim to accommodate an agile process and use of a text oriented approach, CCL has some overlap with CQL. However, because CCL is an embedded language it has a different focus. CCL's syntax emphasizes concise notation, to align with the host language, whereas CQL chooses a linguistic approach for better alignment with domain experts. CCL is less expressive as CQL, because it is just a lightweight embedding, not a standalone language. With regard to information system development in Clean, CCL can be related to earlier work on automated mapping between relational tables and Clean data types [45]. That approach relied on the encoding of conceptual relations in Clean types because no explicit ORM model could be expressed. CCL could be used to improve this mapping.

6.7 Conclusions

In this paper we have presented CCL, a lightweight embedding of ORM in the functional programming language Clean. We have shown that it is possible to define conceptual models from within a programming language without additional overhead. Because derivation of data types from conceptual types enables a more concise specification of collections of data types that model a domain, the additional effort of specifying the conceptual types pays off immediately. Moreover, because a tightly integrated conceptual model is developed from within the programming language, ORM diagrams or verbalizations can be extracted from an information system's source code at any time.

Part III

The Netherlands Coast Guard Case

7 Towards Dynamic Workflow Support for Crisis Management

Current process support technology for crisis management is often limited to either sharing of information or hard-coded process support through dedicated systems. Workflow management systems have the potential to improve crisis response operations by automating coordination aspects. Unfortunately most contemporary systems can only support static workflows, hence yielding inflexible support systems. Recent work on the use of functional programming techniques for workflow modeling has led to the development of the iTask system. It uses function combination to model dynamic data-driven processes and generates executable workflow support systems. Because of its focus on dynamic processes it appears promising for development of flexible crisis response systems. In this paper we present an initial discussion of the potential of the iTask system for crisis management applications. We give an overview of the iTask system, and discuss to what extent it meets the requirements of the crisis management domain.

7.1 Introduction

Crisis management operations involve cooperation and collaboration between large numbers of diverse organizations (e.g. police, firemen, rescue workers, medics). Activities in these operations are highly dynamic and situation dependent. To cooperate and collaborate, activities by diverse organizations must be synchronized (or at least deconflicted) with one another. At first glance, Workflow Management Systems appear to have potential to support. WFMSs are computer applications that coordinate, generate, and monitor tasks to be performed by human workers and computers. Every activity in a crisis management operation can be considered a task. Activities can depend on each other and must be performed in sequence, while other activities may be carried out in parallel. The workflow system can be used to support the distribution and monitoring of these activities. But there are some serious problems, as already acknowledged by Fahland and Woith [17] and Sell and Braun [71]. First, contemporary workflow systems are commonly rather rigid because they only model the static flow of control. Second, the activities to be conducted for tackling a crisis often cannot be captured in a predefined plan. Only a rough

sketch of the actions to be taken can be given. Plans can be further refined only at runtime, when more information becomes available. Most workflow systems cannot deal with this.

Recent work on the use of functional programming techniques for workflow modeling has led to the development of the iTask system [61]. The iTask system is a domain specific workflow language embedded in the functional programming language Clean, enabling the creation of data-driven dynamic workflow systems. It supports data dependent behavior of tasks, where the new tasks to do may depend on the results of previous tasks. The iTask system also allows for on-the-fly adaptation of tasks.

7.2 The iTask System

The iTask system (itasks.cs.ru.nl) is a domain specific workflow language embedded in the functional programming language Clean. It enables the creation of dynamic workflow systems. In the iTask system a workflow consists of a combination of tasks to be performed by humans and/or automated processes. From iTask specifications complete web-based workflow applications are generated. The system is based on open web-standards and can be accessed by anyone who has access to Internet, including many mobile devices. The iTask system is built upon a few simple concepts. The main concept is that of a typed task. A task is a unit of work to be performed by a worker or computer (or a combination of both) that produces a result of a certain type. A task can be a single (black box) step, or a composition of other tasks. The result of one task can be used as the input for subsequent tasks, and therefore these new tasks are dynamically dependent on this result. iTask allows for the data dependent sequential and parallel execution of tasks where information is automatically transported between tasks. Result types are not limited to simple data such as integers, records, etc., but can also be documents, or even new tasks.

7.2.1 Programming Workflows

The iTask standard library offers several functions for creating basic units of work. An important example is the generic task where a user is asked to supply information. The generation of a web-form to enter information and the processing of its result are handled fully automatically by the system. In this way data entry tasks are created in just a single line of code. Figure 7.1 shows the code and the generated form for an *Incident* data type. This code comes from an example application that dynamically allocates ambulances from multiple ambulance posts based on location, number of injured and availability in case of an incident.

An obvious advantage of such compact definition of data entry tasks is that it

— Code —

```

1 ::Incident = { type      :: IncidentType
2               , time    :: Time
3               , nrInjured :: Int
4               , description :: String
5               , location  :: Location
6               }
7
8 ::IncidentType = Accident | Fire
9               | Fight   | Other String
10
11 ::Location    = {street::String,place::String}
12
13 enterIncident :: Task Incident
14 enterIncident = enterInformation "Describe the incident"

```

— Generic User Interface —

The image shows a graphical user interface window titled "Describe the incident". The window contains several input fields and a button. The fields are: "Type:" with a dropdown menu showing "Accident"; "Time:" with a dropdown menu showing "21:30:00"; "Nr injured:" with a text input field containing "2"; "Description:" with a text input field containing "Car collision"; and a "Location" section containing "Street:" with a text input field containing "Heyendaalseweg 135" and "Place:" with a text input field containing "Nijmegen". At the bottom right of the window is an "OK" button with a checkmark icon.

Figure 7.1: A Generic Data Entry Task for Incident Data.

enables readable and easily modifiable workflow specifications. But there are some less obvious, but more important ones: First, the separation of declarative task definition and generic implementation enables different implementations for different devices. Second, because interfaces can be automatically generated, the system can automatically provide a fall back based on manual data entry for every task. Even for tasks that were designed to receive their input through an automated process.

Other examples of basic task functions are: listing all users of the system (if necessary grouped by their role); tasks that return at a predefined moment in time or after an amount of time; tasks that communicate with other applications or web services (for the exchange of information).

New tasks can be composed from other tasks by using combinator functions. We distinguish between combinators that say something about the order in which tasks have to be performed and combinators that say something about an individual task: who has to perform it; where to store information about the task, etc.

In contrast to most workflow specification languages, information is passed explicitly from one task to another in the iTask system. In a sequential composition of two tasks, the first task is activated first and when it finishes, the result is passed to a second task, which takes this result as its input. In code, this is denoted by:

```
1 first_task >>= second_task_function
```

`t>>f` (or `t` followed by `f`) integrates computation and sequential ordering in a single pattern. In this way the second task can dynamically adapt to the result of the first task. In other workflow formalisms it is harder to specify a function that acts on the result of a preceding task because only control is passed between tasks.

The `parallel` combinator can be used for executing tasks in parallel. It can be parameterized with predicates such that many patterns of parallel composition can be expressed using this single combinator. For example: `or-`, `and-` and `ad-hoc` (conditional) parallelism.

7.2.2 Dealing with Dynamic Behavior: Exceptions and Change

Several authors [71, 17] already indicated that workflows need to be adaptive to be of use for crisis management operations. iTask offers three constructs to achieve this:

The sequence (`>>=`) combinator is used to make tasks dynamically dependent on results of previous tasks.

The iTask *exception* mechanism can be used in case the normal course of actions is affected, and a new procedure should be started. A task may throw an exception in case an exceptional situation occurs. The entire workflow the task

is part of is now stopped (if there are parallel tasks in it, the users participating in these tasks are informed). The exception is passed to an exception handler that can start a new task using information raised in the exception. Exceptions enable the separation of uncommon borderline cases from the regular workflow. The *change* concept is complementary to that of the exception. A change is something that is triggered from outside the specified workflow. Tasks on which people are working can be replaced on-the-fly with other tasks. An example of a change is the replacement of a complex process by a simple to-do list, in case the user has determined that the process is inappropriate for the current situation, or the replacement of a task by manual entry of a result that is obtained outside the workflow system.

7.2.3 Working on Tasks

When a workflow specification is compiled, a server executable is generated that coordinates tasks through a set of web services. It serves task lists, a workflow catalogue, and high level user interface definitions of concrete tasks. Users can access these services with a generic web based application (Figure 7.2).

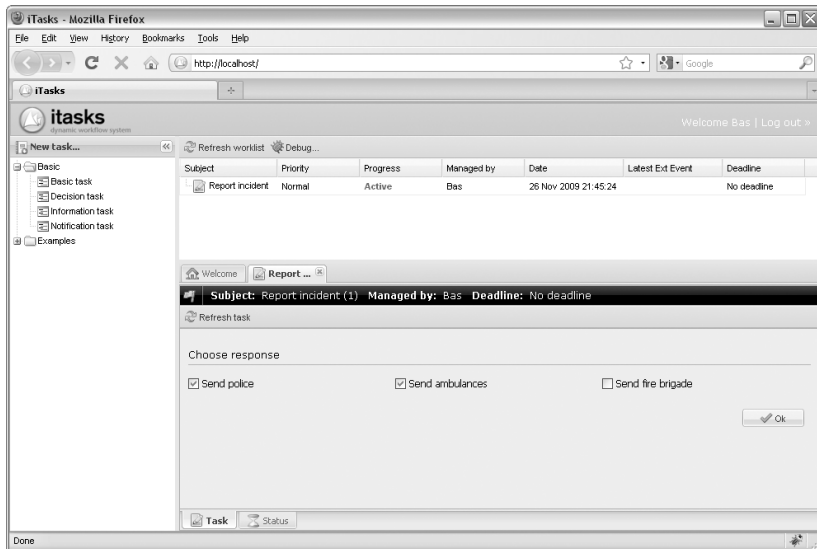


Figure 7.2: A Screenshot of the iTask Client Application

Tasks that a user needs to perform are presented in the task list inbox displayed in the upper right pane. The state of a selected task is displayed in the lower right task pane. Tasks can be selected in any order, or simultaneously, allowing

a user to determine a preferred order of execution. All work is immediately synced with the server.

Users can start new workflows by selecting them in the left workflow pane. In general any number of workflows can be started.

7.3 iTasks for Crisis Management?

Although the iTask system has not specifically been designed with crisis management applications in mind, we contend that it is a potentially valuable tool for building crisis management systems. To find out what is demanded from crisis management systems, such that we can identify challenges for further development, we turned to the literature for requirements. Unfortunately, different authors use different requirements when proposing technology for this domain (e.g.[28, 71]). What we needed was an independently defined set of general requirements. Jul in [35] provides such a set in the form of five design requirements distilled from an analysis of the crisis domain in general. In this chapter we discuss the strengths and weaknesses of the iTask system in light of each of these requirements:

- **Design Requirement #1:** Response technology should seek to support just-in-time learning, first, of the task the tool is intended to support, second, of the needs and goals of the present operation, and, third, of disaster management practices in general.
- **Design Requirement #2:** Response technology, even when focused on agent-driven tasks, should seek to aid response-driven tasks, such as planning, coordination and resource management.
- **Design Requirement #3:** All response technology should actively nurture cooperation, collaboration and partnership formation.
- **Design Requirement #4:** Response technology, while imposing standard structures and procedures, must, insofar as possible, allow flexibility and deviation in their application.
- **Design Requirement #5:** Response technology should aim for graceful augmentation, allowing the technology to be integrated in or removed from the users activities with a minimum of disruption.

7.3.1 Requirement #1: Just-in-time Learning

Because people do not need to know what they will have to do in advance, the step-by-step guidance through standard procedures by a workflow system is essentially just-in-time learning of those procedures. The workflow specification

guides people through procedures they might have never done before. Once users learn how to use the interface to find out what tasks they have to do, and how they can select tasks to work on, they can rely on the system to tell them what needs to be done. To ease the initial learning curve, the iTask user interface has been designed to resemble an e-mail client as much as possible. Users can simply think of the system as a special e-mail system where all messages in their inbox happen to be requests to do something.

A weakness of the iTask system is that the goals and instructions of tasks are communicated primarily through text as defined in the workflow models. When a user is presented with a task having instructions he or she cannot understand, or even worse, can misunderstand, there are no built-in ways to easily resolve that knowledge gap. The learnability of the tasks is therefore almost completely determined by the degree to which the workflow models supply enough information. Of course, this problem also exists for paper handbooks and contingency plans. Interactive workflow systems have an opportunity to do more, e.g. to provide access to information sources, or to provide easy communication to ask peers help.

7.3.2 Requirement #2: Response Driven Tasks

A workflow system, by definition, supports response driven tasks, since its sole purpose is to automate the coordination and execution of standard procedures. It has the additional advantage over hard-coded support systems of having inspectable models that, at run-time, can be queried to get information about what is going on. The dynamic data-driven workflow models that are used by the iTask system have the additional potential of enabling flexible resource allocation and planning. Data that becomes available as a result of performed tasks can be used for the (re)distribution of resources or for planning/scheduling of other tasks. However, currently available resource allocation combinators in the iTask systems standard library are purely algorithmic. It is possible to integrate stochastic or other predictive models to distribute tasks and resources, or to support decision making at crucial points in a workflow. Having such tasks available in a library of the workflow language could further improve the support of response driven tasks.

7.3.3 Requirement #3: Cooperation and Collaboration

Cooperation and collaboration are supported in iTask workflow models by (re) assigning tasks to users and routing the task results from one user to another. Tasks can be delegated and tracked. It is also possible to define workflows that add new users to the system, who then immediately can get tasks assigned to them.

The multi-user features of the iTask system make it possible to define workflow models that involve multiple users. However, to assume that therefore it “*nurtures cooperation, collaboration and partnership formation*” would be too shortsighted. There are still many things that should be facilitated to promote cooperation, regardless of the concrete tasks at hand, such as for example, integrated communication capabilities (chat, voice, video) to enable users to discuss the tasks they are working on, or formation of ad-hoc teams of users. A more fundamental challenge will be a shift to multi-user tasks. Currently, tasks are always assigned to, and managed by, a single individual. Relations, both formal and informal, between users are not modeled in the iTask system. In daily life, however, it is not uncommon to work together on a task without exactly dividing it into discrete subtasks, or to have shared responsibility for a task. When concurrently working on tasks by multiple users is possible, one could leverage the fact that tasks can produce new tasks, to cooperatively define and execute a workflow with a group of people.

7.3.4 Requirement #4: Flexibility

Flexibility is a feature of the iTask system that pointed us to the potential usefulness of dynamic workflows for crisis management in the first place. Because iTask workflow specifications support the modeling of dynamic processes at multiple levels, it is potentially capable of complete compliance with this fourth requirement. However, although it is technically possible to define very flexible workflow models, the usefulness of this expressive power is constrained by the interface through which it is exposed to end-users. An important research challenge will be to develop generically applicable problem solving patterns that can be applied when normal procedures do not apply. More research is also needed on what information is required by users to become aware that there is a need for deviation from standard procedures, and what is required to decide what course of action is to be selected to resolve the issue.

7.3.5 Requirement #5: Graceful Augmentation

Removal of a workflow system during the execution of an operation guided by it, will always cause disruption. However, it is possible to meet this requirement as closely as possible by reducing the amount of disruption if (a part of) the system is temporarily removed. The current iTask system does not specifically address this issue, because network infrastructure has been assumed to be available. However, it has been shown that it is possible to run parts of workflows offline [66], by transferring part of the workflow computation to the client system. A last resort would be to use the system in a controlled environment such as a command post while communicating tasks through other channels. In this case it could still have advantages over written handbooks, because iTask workflow specifications are active and dynamic.

7.4 Conclusions

In this paper we presented dynamic workflow modeling, as implemented in the iTask system, as a candidate platform for developing applications to support crisis management operations. Although this system has not been designed for crisis management, we contend that, because of its unique features like: data driven, parameterizable workflows and extensive support of dynamic behavior, it has potential value in this domain. We have explored this potential through a discussion of iTasks strengths and weaknesses in light of five key design requirements for crisis response technology defined by Jul in [35]. The aim of this discussion has been to identify challenges for further research, which most notably are: collaboration and effective use of flexible workflows. By addressing these challenges, we hope to develop the iTask system into a valuable tool for building systems that flexibly support people under demanding circumstances.

8 Capturing the Netherlands Coast Guard's SAR Workflow with iTasks

The dynamic nature of crisis response operations and the rigidity of workflow modelling languages are two things that do not go well together. A recent alternative approach to workflow management systems that allows for more flexibility is the iTask system. It uses an embedded functional language for the specification of workflow models that integrates control-flow with data-flow in dynamic data-dependent workflow specifications. Although there is a variety of publications about the iTask workflow definition language (WDL) and its implementation, its applications have been limited to academic toy examples. To explore the iTasks WDL for crisis response applications, we have developed an iTask specification of the Search And Rescue (SAR) workflow of the Netherlands Coast Guard. In this specification we capture the mix of standard procedures and creative improvisation of which a SAR operation exists.

8.1 Introduction

Workflow management systems (WFMS) are not particularly well known for their flexibility. Hence, the thought of using a WFMS to support the coordination of Coast Guard Search and Rescue (SAR) operations may seem to be a doomed endeavour from its onset. Many contemporary WFMSs use a flow-diagram graphical language interpreted by a workflow engine to orchestrate or dictate the work to be done. The implicit assumption of such specification languages is that all activities and the order in which they will be executed can be specified in advance. That this assumption does not hold for SAR operations at the Netherlands Coast Guard is best illustrated by the following quote from the OPPLAN-SAR, their primary operational procedure document (freely translated from Dutch):

“Because we know from experience that no two SAR incidents are alike, it is impossible to define a fixed, extensive, always applicable procedure”

Jansen et al. [30] claim that, by contrast to other WFMSs, the iTask system's approach to workflow specification is expressive enough to capture the dynamic

nature of tasks required for crisis management. To date, this claim has not been fully tested. Although the iTask system has been subject of extensive research in the field of programming language design [61], no work exists that explores its applicability outside the realm of academic toys.

The purpose of this study is to explore the strengths, weaknesses, and other properties of the iTask workflow definition language (WDL) when it is used to specify a real-world crisis response workflow. The Netherlands Coast Guard manages small and large crises on a daily basis. Their SAR operations provide an interesting case, because they consist of a mix of following standard procedures and ad-hoc crisis management. It is also a convenient case to study because their procedures are well documented and all incidents are logged.

8.1.1 The iTask System

The iTask system (iTasks) is a workflow language embedded in the functional programming language Clean (clean.cs.ru.nl). It enables the creation of dynamic workflow systems. In the iTask system a workflow consists of a combination of tasks to be performed by humans and/or automated processes. From iTask specifications complete web-based workflow applications are generated. The applications are based on open web-standards and can be accessed by anyone who has access to Internet, including many mobile devices. iTasks is a textual formalism (i.e., a programming language) and offers a much higher degree of flexibility than graphical formalisms that are in use for specifying workflow systems. The iTask system is built upon a few core concepts. The main concept is that of a typed task. A task is a unit of work to be performed by a worker or computer (or a combination of both) that produces a result of a certain type. A task can be a single (black-box) step, or a composition of other tasks. The result of one task can be used as the input for subsequent tasks, and therefore these new tasks depend dynamically on this result. iTasks allows sequential and parallel execution of tasks, with information automatically being transported between tasks. Intermediate results of tasks that are executed in parallel can be used to decide whether the execution of other tasks running in parallel should be stopped or altered. Result types are not limited to simple data such as integers, records, etc., but can also be documents, or even new tasks. Tasks can be explicitly allocated to persons, and dynamically reallocated if necessary.

Two concepts especially contribute to the languages expressiveness. The first is that tasks can be higher order. This means that the result of a task can be a new task. For example, a task may use the output of one or several other tasks to construct a new set of tasks and the way they must be executed: sequentially, in parallel, or a combination of both. As a consequence, the workflow specification cannot be completely determined beforehand, but is constructed iteratively during execution. The second is that tasks can be parameterized by data

types. This makes it possible to define generic tasks or task structures that are independent of the specific result type of the task. This allows them to be used in multiple contexts.

The iTask implementation is a research prototype, because its WDL is still evolving. Development of the WDL focuses on the exploration of workflow specification concepts for applications in dynamic domains, such as crisis management, command & control, and medical support systems. The prototype status means that the core WDL concepts are available, and workflow specifications can be compiled to complete executable workflow support systems. However, there is no large standard library as could be expected from a production WFMS. Features without scientific interest are added by demand.

8.1.2 The Netherlands Coast Guard

The Netherlands Coast Guard is an independent civil organization with its own responsibilities and competences. The Coast Guard functions as a central reporting, information, and coordination centre in its role as the National Maritime and Aeronautical Rescue Centre (Joint RCC). Its main area of operations is the North Sea. This is one of the busier shipping routes in the world, populated with a crowded combination of commercial and private vessels. Resources must be deployed at short notice when an accident or incident at sea occurs. These resources consist of vessels, airplanes, helicopters, and rescue team stations. Each has different sponsors, different lines of communication, and different procedures, making the communication and coordination of crisis response complex.

One of the main responsibilities of the Coast Guard is execution of the SAR service. This service is responsible for searching for aircraft, ships, and oil drilling platforms in distress within the North Sea and in Dutch coastal waters and for rescuing their crews and passengers.

The Coast Guard currently uses a variety of communication systems (radio, telephone, telex, etc.) and systems for information sharing, for information retrieval (databases, documentation), and for information logging. Logged information is used both for information sharing during operations and for evaluation afterwards. The Coast Guard's current systems offer only modest workflow support in the form of simple action plans, digital procedure documentation, and predefined forms.

8.2 Literature Review

Several authors discuss the use of WFMSs to handle crisis response operations. They all recognize that workflow systems have the potential to offer better support than the use of printed document procedures only. They agree that

adaptability is a key issue to be solved if these systems are to be really useful for Crisis Response Operations.

Sell and Braun [71] defined a number of generic requirements that a WFMS should fulfil in order to be useful for crisis response operations. According to them, the WFMS should:

1. support the management of resources;
2. always depict the current state of deployment;
3. allow the adaptation of the workflow before and during execution;
4. support the delegation of measures;
5. support the execution of workflows.

Based on these requirements, Sell and Braun propose a workflow data model that fulfils these requirements, but do not give an implementation of it. Jansen, Lijunse, and Plasmeijer [30] claim that iTasks provides concepts that are powerful enough to fulfill these requirements. In this paper we focus on requirements 2 and 3.

Fahland and Woith [17] also focus on the use of WFMSs for crisis management. They observe that routine processes, even if specifically designed for a situation, should never be enacted blindly. Rather, actions and processes should adapt their behaviour based on observations and available information. They propose specifying an adaptive process as a set of scenarios using Petri nets. Their operational model provides an adaptation operator that synthesizes and adapts system behaviour at run-time, based on these scenarios.

Peukert, Lincourt, and Zimmermann [58] developed the Collaborative Task Manager as a tool to model and execute workflows. This system enables modelling the exchange and reuse of user-defined task structures. It uses an email-based system for the distribution and delegation of tasks. It supports ad-hoc deviations from pre-defined plans. Due to the tools tracking functionality, previous ad-hoc processes can be analyzed on the back-end system to give guidance to the current process and to discover best practices.

8.3 Research Questions

Since the iTask system has not yet been applied in real-world applications, our research goal is to address a question that extends beyond the specific Coast Guard case. We wish to know if the iTask specification language can be used to specify workflow support systems for real-world crisis response operations, and if not, what is lacking.

We contribute to the broader question by developing a specification for a case that is both real-world and dynamic. Coordinating SAR operations provides us with a reference case, enabling us to answer the following research questions:

- **RQ1:** What are the properties of the specification?
How is it structured and why? Which features of the iTask WDL are used?
- **RQ2:** Is the iTask WDL expressive enough for this case?
Are there aspects that could not be specified using the WDLs primitives? If so, is there a fundamental reason why not? Or can the WDL be extended to include these aspects? Are there aspects that could only be specified using the WDLs dynamic features?

The primary purpose of answering these questions is to better understand the strengths and weaknesses of the iTask WDL as a method for capturing crisis response workflows. The effectiveness of execution of these workflows and the quality of the support systems generated from them is beyond the scope of this paper. As a bonus, the specification also gives us an insight into which aspects of the Coast Guards SAR work would benefit from further automation, and - perhaps just as importantly - which aspects should be left to skilled operators.

8.4 Methodology

To answer our research questions, we have conducted a qualitative explorative case study using document review complemented with observation and in-situ interviews. This specification is primarily based on information found in the following reviewed documents:

- The OPPLAN-SAR V7, which is the primary operational plan containing high-level procedures, contracts between involved authorities, other agreements, and background information.
- Internal operational procedure documents. These contain more detailed guidelines than described in the OPPLAN-SAR and can be considered as its operational implementation.
- Configuration databases from VISION, the Coast Guards current logging and incident management system. These databases provide insight into what information is collected during incidents.
- The Netherlands Coast Guard public website (www.kustwacht.nl).

We searched these documents for fragments containing procedure descriptions and compiled them into a single file. These fragments were then formalized using the WDL of iTasks release 10.8.

The documents are not followed blindly, but are used as guidelines. Well-trained officers have their individual interpretations of the procedures. Hence, we made four on-site visits to the command centre. The first visit was a guided tour of the organization, and included an in-depth demonstration of the communication and information systems currently in use. The other three visits were devoted to following an operational team during the course of a shift to observe the actual workflow of incident coordination and to interviewing the duty and watch officers in-situ during quiet moments. These interviews focused primarily on storytelling to give context to the documents.

Based on the increased understanding of the domain from these visits, another pass over the documents was made, and the fragments were integrated into a single specification.

8.5 Results

8.5.1 Properties of the Specification

Because of the size of the specification (± 2700 LOC), and because it contains proprietary information, it is impossible to cover it in full detail here. Instead, we present an overview of the specification, explaining the key parts and their relations, illustrated with examples taken from the specification. Details of the iTasks WDL can be found in [33].

Overview

At first glance, the process of managing a SAR operation appears to be simple. When a distress call is received, actions are taken immediately to collect more information and to assess the situation. This may vary from a simple request for medical advice to a full-scale disaster; in each case, the set of connected events is known as an incident. When the situation requires, rescue vessels and aircraft are dispatched to search for the originator of the distress call and to rescue any crew or passengers. The incident is over when the vessel or people have been located and rescued or when no reasonable hope of rescue exists. In real incidents, managing a SAR operation is a complex, highly parallel process consisting of many interdependent actions, communications, and decisions, all based on incomplete and uncertain information.

The iTasks SAR workflow specification consists of two main parts. The first part defines the tasks for responding to inbound communications. When a Coast Guard officer answers the telephone or receives a radio call, he/she does not know the topic of the conversation in advance. Any call can be related to an ongoing incident, or it can initiate a new one. Therefore, responding to inbound communication is specified separately from subsequent actions. The

that the creation of an incident is optional. Arrows denote computation. These inbound response tasks are tailored to the main communication systems, but information could also be received from unspecified sources via the public telephone network. For example, suppose that the friend of a watch officer gets into difficulties while sailing and calls the watch officer on his/her personal cell phone. For such cases, there is also a task defining the ad-hoc creation of a new incident:

```
1 createNewIncident    :: Task Incident
```

Part 2: Incident Coordination

nce an incident has been reported via one of the inbound communication channels it is the responsibility of the duty officer to coordinate a response operation. This involves taking a mixture of actions that are prescribed by standard procedures and ad-hoc actions guided by the duty officers continuous reassessment of the situation. In the iTask specification we have modelled the task of coordinating an incident as follows:

```
1 coordinateIncident :: Incident → Task Incident
2 coordinateIncident incident
3 = (manageInformation incident -||- coordinateActions incident)
4 >>= postIncidentActions
```

This states that the task of coordinating an incident consists of two steps in sequence (expressed by the `>>=` operator). In the first step, two tasks are executed in parallel: `manageInformation` and `coordinateActions` (combined using the `-||-` operator). Then, when the operation has been completed, a task `postIncidentActions` is executed. The `manageInformation` task states that the duty officer makes sense of the situation by gathering together and assessing the available information. At the same time, actions are taken by the duty and watch officers either to collect more information, to distribute information to others, or to instruct and command the rescue units. The aggregation of all these actions is captured by the `coordinateActions` task. Everything that has to be done after an incident, such as writing reports or evaluating the incident, is captured by the `postIncidentActions` task.

Part 2a: Information Management Although information management is a key task during SAR operations, its primary specification, the data model, is outside the scope of a WDL. From a workflow perspective, the browsing, viewing, and editing of information about an incident can be viewed as a single task. In the specification, we have defined this task only minimally for testing purposes using iTasks built-in object database tasks. However, because the data underlying the `manageInformation` task is accessed and modified by

the coordinateActions part of the specification, an interface to the available information is needed. This interface is specified as a collection of data types that define incident-related data. For example, there is a type Incident that represents the collection of all known information about an incident, and there is a type Contact that contains information about a party involved in the incident, such as a ship or an aircraft.

The definition of these types is given below:

```

1 :: Incident =
2   { incidentNo :: IncidentNo
3     , title     :: U String
4     , summary  :: U Note
5     , type     :: U IncidentType
6     , phase    :: U EmergencyPhase
7     , weather  :: U WeatherInfo
8     , contacts :: [Contact]
9     , log      :: [LogEntry]
10    , closed   :: Bool
11  }

1 :: Contact =
2   { contactNo :: ContactNo
3     , type     :: U ContactType
4     , name     :: U String
5     , position :: U GeoPosition
6     , contactOn :: U ContactMedium
7     , isCoastGuard :: U Bool
8     , inDistress :: U Bool
9     , canHelp   :: U Bool
10    , reportedIncident :: U Bool
11    , notes     :: U Note
12  }

```

Because information is likely to be incomplete or uncertain, some data is wrapped in a parameterized type U:

```
1 :: U a = Unknown | Known (a, Timestamp, Source) [(a, Timestamp, Source)]
```

This defines that values can be either unknown or known. For known values, the source from which the information came and the time that it became known are tracked. A list, expressed in code by [] brackets, of previous values is maintained to log changes over time.

Part 2a: Action Coordination The crux of the specification is the definition of the coordinateActions task. Based on interviews with officers and the reviewed documents, we established that the specification of the coordination workflow needs to comply with a set of constraining principles:

- The duty officer coordinates the operation, not the WFMS.
- Actions prescribed by standard procedures do not apply to every incident.
- The order in which actions are taken is not fixed, but subject to the duty officer's judgment.
- Ad-hoc actions defined during an operation are necessary to supplement standard actions.

From these principles, we conclude that a rigid specification of predefined tasks executed in a fixed order, as is common in workflow specifications, is not an

option. We need a formalization that retains more flexibility. We achieved this by exploiting the fact that the `iTasks` WDL is embedded in a pure functional language. We capture tasks that are executed in this context by a special data type `HSTask` (Hierarchical State Task) and use type abstraction and higher order functions to define an abstract recipe to compute concrete tasks from values of this type.

The `HSTask` data type is defined as follows:

```
1 :: HSTask s =
2   { meta      :: HSMeta
3     , activity :: HSActivity s
4     , refinement :: HSRefinement s
5   }
```

`HSTasks` specify a task in three parts. The meta part specifies meta-data that summarizes the activity to make it possible to choose which activities to start, the activity part defines how the task is to be completed, and the refinement part defines how to refine a task into smaller sub-tasks. The `HSTask` type is parameterized with the parameter `s` that defines the type of information that is available to complete the task. For the tasks in the SAR specification `s` is `Incident`. This means that all information about a specific incident is available during the activity.

```
1 :: HSMeta =
2   { name      :: String
3     , title    :: String
4     , description :: Note
5   }
```

The meta-data is straightforward except for the distinction between name and title. The name of the `HSTask` is a unique identifier that makes it possible to keep track of tasks that should be executed only once during an incident. The title defines a displayable label.

```
1 :: HSActivity s =
2   { interaction :: s → Task s
3     , procedures  :: [DocumentName]
4     , relevance   :: Maybe (s → Bool)
5   }
```

The `HSActivity` part of an `HSTask` defines how the task can be completed. For most tasks this is simply a choice between marking the task completed or cancelled. Other tasks embed their own small workflow, such as `outbound-CallAction` (explained below). The `procedures` field allows the specification of a set of procedure documents that contain instructions about the task in natural language. This documentation is made available for quick reference simultaneously with the interaction. The `relevance` field allows specification

of a predicate that tests whether the task is still relevant, given the current information. This is useful when the purpose of the task is to find out some information, but that information has already become available through other means. Based on this predicate, a warning message can be displayed when the task is no longer relevant.

```

1 :: HSRefinement s =
2   { suggested  :: Maybe (s → [HSTask s])
3     , alternative :: Maybe (s → [HSTask s])
4     , custom    :: s → Task (HSTask s, HSTaskWhen)
5   }

```

The `HSTasks` are called hierarchical because each task can be refined into a number of sub-tasks. Broadly defined tasks, like “ollect as much information about the vessel as possible, from any source”, need to be further refined during the incident. The `HSRefinement` part of an `HSTask` enables the specification of such refinement. The `suggested` and `alternative` fields contain (optional) functions that compute a set of predefined suggested and alternative `HSTasks` to choose from. Suppose for example that an INMARSAT number of the distressed vessel is known, then a suggested refinement could be “Search vessel info in INMARSAT database” while an alternative refinement could be “earch vessel info with Google”. To enable improvisation, the `custom` field specifies the workflow for creating custom refinements. The result of this task is a pair consisting of the refinement and an indication of when the task is to be executed. The `HSTaskWhen` type has the following values:

```

1 :: HSTaskWhen =
2   HSAreadyDone DateTime | HSNOW | HSAfterTime Time | HSATime DateTime

```

Because we observed that urgent actions are often taken first and administrated later, the `HSAreadyDone` task may be used to add actions that have already been completed.

In the specification, `HSTask` values are never constructed directly, but always via wrapper functions. This makes the definition of concrete actions as concise as possible. For example, consider the following definition for informing a medic at military airfield “De Kooy”¹:

```

1 informMedicMVKK :: HSTask Incident
2 informMedicMVKK = outboundCallAction "informMedicMVKK" "Medical service"
3   "Inform medic on duty at MVKK" (Just (QueryPhonebook "MVKK"))

```

The `outboundCallAction` task defines the workflow for all outgoing telephone calls. Although many telephone calls are made during an incident, it is possible to define a generic workflow that applies to all calls. This ensures that all calls are

¹Translated only in the paper. In the specification, we use the Dutch descriptions from the procedure documents.

logged, and that all parties involved in the incident are tracked. To illustrate how this is specified in the iTASK WDL we include its definition below:

```
1 outboundCallAction :: String String String (Maybe ContactHint) → HSTask Incident
2 outboundCallAction name title description mbHint
3   = { meta = meta name title description, activity = customActivity interaction
4     , refinement = basicRefinements }
5 where
6   interaction incident
7     = defineCaller mbHint incident >>= makeCall incident
8   makeCall incident caller
9     = connectCall caller >>= select (gotAnswer incident caller)
10                                (gotNoAnswer incident caller)
11
12   gotAnswer incident caller
13     = addLogMessage ("Called with " + visualize caller) incident
14   >>= linkContactToIncident caller
15   >>| requestConfirmation "Add information"
16       "Do you want to add new information to the incident?"
17   >>= conditional editIncidentInformation incident
18
19   gotNoAnswer incident caller
20     = addLogMessage ("Tried to call " + visualize caller) incident
21   >>| requestConfirmation "Try again" "Do you want to try again?"
22   >>= conditional (rescheduleCall caller) incident
```

The above example shows that detailed concrete tasks can be specified with a custom HSAActivity definition. At the same time, we can define flexible, less-detailed tasks. This is illustrated by the `deployOFFSARHeli` task definition below:

```
1 deployOFFSARHeli :: HSTask Incident
2 deployOFFSARHeli = multiProceduralAction
3   "deployOFFSARHeli" "Deploy OFFSAR Heli" "Deployment of OFFSAR helicopter"
4   ["deploy-units.txt", "deploy-offsar.txt"] suggested noAlternative
5 where
6   suggested incident = filter (completed incident)
7     [ informDCOFFSAR, informHangarOFFSAR, alertCrewOFFSAR
8     , requestBackupMedic, informMedicMVKK, informOperationsMVKK
9     , requestNOTAM, sendNOTAMRequest, requestCHCNetherlands
10    , informRACAlkmaar, informCSUSecurity, informKMAR, revokeNOTAM
11    ]
```

For this task, it is up to the duty officer to decide which of the suggested refinement actions are actually taken. In this case, no alternative actions are defined, but the suggested tasks and available procedure documentation provide support, while retaining flexibility.

8.5.2 Limitations of the Specification

Not all aspects of the Coast Guards SAR operations are included in the iTask specification. The following limitations can be identified from the SAR application:

- Resource allocation, i.e., the assignment of tasks to workers, is not specified. The workflow is specified without explicit task assignments because the duty and watch officer work as a team. Although they have different roles, each officer can pick up a task for an incident. Hence, work is divided on an ad-hoc basis. Assignment of tasks to teams instead of individuals is not supported in the iTask WDL.
- Planning and scheduling information for future tasks is only implicitly specified. In particular, the iTask WDL only allows the specification of deadlines for tasks, not specific start times.
- The information management task is defined minimally. To make the viewing and editing of all incident-related information manageable, the iTask system should be integrated with a full-fledged information system.
- The specification only contains tasks mentioned in the documents we reviewed. From our interviews and observations, we suspect that there is more structure than is currently documented.

8.6 Discussion

8.6.1 Properties of the Specification

With the results of the case study to hand, the first thing we should do is to reflect on what exactly has been specified. This is expressed in the first research question: What are the properties of the specification?

How is it structured and why?

The first thing that is interesting about the structure of the specification is that it is not a simple linear workflow with a clear beginning and end. Rather, it is a combination of two shorter workflows for dealing with new information and for coordinating the incident. The coordination workflow has no pre-defined end, because operations can be aborted at any time. It defines an ongoing (re)assessment of available information and actions that follow from that.

In this respect it is similar to the OODA [8] view of Command and Control operations. However, we do not specify the task as an explicit loop; in effect, Observe, Orient, Decide, and Act all happen concurrently.

The second interesting aspect is that action coordination is driven completely by human initiative. Actions are suggested but never started automatically. Specifying the workflow as a dependent set of suggestion functions is required to capture the need to adjust the workflow to the specific situation during the incident. Due to the loose organization of tasks, it is impossible to pass information from one task to another, as it is never certain whether a specific task will be executed. Therefore, these tasks have to retrieve the data they need from and store the data they produce into a centrally shared container. A final point is that, although the workflow to coordinate an incident is organized purely by task suggestions, the workflow of tasks at a more detailed level can be specified as a straightforward sequential/parallel composition of tasks. This allows for a mix of detailed and loosely defined tasks which never overspecify the task, while minimizing underspecification.

Which features of the iTasks WDL are used?

We observe that all iTask core combinators are used to define the inbound communication workflows, the incident response actions, and the generic behaviour of HSTasks. From the basic task primitives, tasks are used mostly for user interaction and data storage. Higher order tasks are used to specify ad-hoc tasks during an incident. The ability of iTask to treat tasks as data and wrap them together with meta-data in single values is used in selecting the suggested and alternative tasks.

8.6.2 Expressiveness of the iTask WDL

The Coast Guard workflow specification reveals intrinsic properties of the iTask WDL. Those aspects that could not be specified show its boundaries, while the use of specific features justifies their inclusion in the language. These topics are covered by the second research question: Is the iTask WDL expressive enough for this case?

Are there aspects that could not be specified using the WDL's primitives?

There are a few aspects of the SAR workflow that could not be specified with the WDL. Resource allocation is not modelled because the WDL only supports the assignment of tasks to individual users, not to multiple users at once. Because the specific assignment of tasks to individuals does not reflect the work on tasks as a team, resource allocation has been omitted from the specification altogether. Planning of future tasks has been included in the specification only implicitly in the form of a line of text added to a task description stating at what time a task should be done. The iTask WDL is only capable of specifying deadlines on tasks, not start times. Hence, the support system generated from

the specification does not have the possibility of drawing the users attention to the task at the scheduled start time.

Another aspect that cannot be defined using the WDL is the structure of the database underlying the `manageInformation` tasks. Technically, this is outside the scope of workflow definition. However, because shared information plays a big role in this case, it would be desirable to have database integration that goes beyond explicit query and update steps in the workflow.

If so, is there a fundamental reason why not? Or can the WDL be improved to include these?

No fundamental issue would preclude extending the iTask WDL to remove the first pair of limitations, i.e., task assignment to teams and scheduled start times. The only reason why these aspects have not been defined is the immaturity of the language. As the language evolves, we expect that each new case study will require some API extension. By contrast, database integration is a fundamental issue. This would make the language a hybrid workflow and information modelling language. However, methods exist for mapping transparently between Clean and databases [45] which could underlie such a hybrid language.

Are there aspects that could only be specified using the WDL's dynamic features?

It is easy to focus on the limitations of the language because they are clearly revealed by the case study. But, it is just as interesting to reflect on aspects that could be expressed. What this case study shows is that not all tasks can be known in advance, and that the order in which tasks are executed is not fixed. This can only be expressed by a language that determines tasks during execution. A formalism that is only capable of choice between fixed tasks quickly becomes unmanageable under these conditions as n tasks can be executed in $n!$ orderings. Another property of the work of the Coast Guard is that ad-hoc activities that are not defined in a procedure, but based on experience, common sense, and creative problem solving, are a normal part of the work. It follows that coming up with new tasks is part of the normal workflow. Higher-order task definition are therefore a necessary language feature.

8.7 Conclusions & Future Work

In this paper, we have explored the use of iTasks to capture crisis response workflow by means of a case study based on the Netherlands Coast Guards Search And Rescue operations. From this case, we found that, to attain the

required flexibility, the workflow needs to be specified as a collection of suggested and alternative actions based on current information, rather than as a statically pre-determined flow. The workflow is highly parallel, with most tasks depending on or contributing to a shared operational picture. Furthermore, we found that improvisation and performing ad-hoc actions are an essential part of the regular SAR workflow.

We concluded that the iTask WDL is expressive enough to capture the loose and parallel structure of tasks and, by using higher-order tasks, the definition of ad-hoc actions. We have captured this structure in a generic hierarchical model, which is reusable for other coordination tasks with improvisation aspects. Nonetheless, the emphasis on a shared operational picture around which the coordination of SAR revolves reveals an opportunity for future work on extending the iTask WDL. The current version of the WDL is designed to specify tasks and the data they use, not to specify information systems. We believe it would be possible to integrate the workflow definition language with a database modelling language using methods from [45]. This would enable a complete executable SAR support system to be specified using a single specification language.

9 Incidone: A Task-Oriented Incident Coordination Tool

Coordinating rescue operations for incidents at sea can be a complex task. In this paper we present an ongoing project that aims to develop an incident coordination tool to support it. This tool, Incidone, is based on the specification outlined by Lijnse et al in "Capturing the Netherlands Coast Guard SAR Workflow with iTasks" and is therefore modeled after, but not necessarily limited to, the workflow of the Netherlands Coast Guard. The unique feature of Incidone is that it is the first tool of its kind developed using the Task-Oriented Programming paradigm. Therefore, we present the tool both from the perspective of its intended end-users as well as from the perspective of a software developer. The primary goal of the Incidone project is to provide an example of this method to developers of similar crisis management applications.

9.1 Introduction

The sea can be an unpleasant place to be. When surprised by bad weather, or equipment failure, the sea can quickly turn into a hostile environment. Luckily, coast guards around the world provide assistance to those in need. They need to resolve incidents safely and in a timely manner, constraint by weather conditions and limited resources.

Designing and implementing software tools to support this task is a complex challenge. In their design, the conflicting requirements of flexibility to deal with unique incidents need to be weighted against the automation of simple, but time consuming, tasks. To explore the use of an experimental method of specifying workflow support systems, Lijnse et al defined a specification of the search and rescue tasks of the Netherlands Coast Guard [42]. This work provided insight in the capabilities and limitation of the specification language and tooling, but had little practical value because the specification was defined in an experimental language with a limited implementation. To make it possible to put the results of this case study in practice, and thereby making its potential added value testable, an improved specification needs to be developed together with an improved version of the specification language and its tooling. This motivated the development of the Incidone tool.

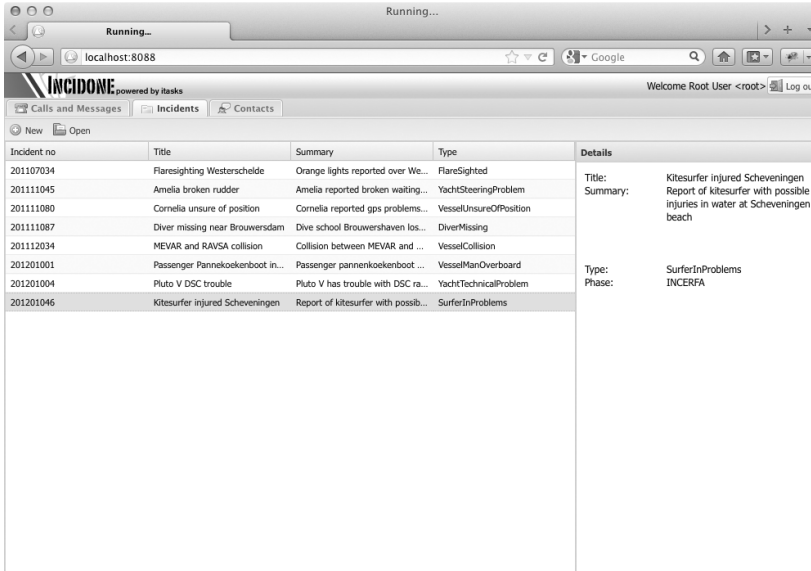


Figure 9.1: Keeping watch: Tracking incidents, contacts, and responding to calls

Although the development of an effective incident coordination tool is a respectable goal in itself, it is not our primary motivation. The primary goal of the Incidone project is to explore the use of an experimental software development paradigm called Task-Oriented Programming (TOP) for this type of applications. It serves as a test case for the iTask System [32], the framework that facilitates this type of development, and provides an example for software developers of similar tools.

The remainder of this paper is structured as follows: In the next section we give an overview of the tool from the perspective of its intended end-users. We then focus on the software aspect by illustrating parts of its source code. We end the paper with notes on the status of the project followed by some overall conclusions.

9.2 A Watch Officer's View of Incidone

The intended users of the Incidone tool are Coast Guard watch officers in a centralized command center. As their job title implies, the primary task of these people is to keep watch. That is, to proactively monitor and respond to incidents when they occur. We operationalize this task by providing three

initial overviews with the necessary information, arranged in a set of tabs as illustrated in Figure 9.1. These overviews are:

Calls & Messages: A watch officer must respond to distress calls, requests for medical assistance or other requests for help of any kind. Calls can come in by radio, or (satellite) telephone. Additionally messages may come in, such as DSC radio messages, Telex, or even e-mails. Messages may come from people or from automated alert systems like emergency transponders. The tool provides support for logging a call or message, to associate it with an incident, and to process it, tailored to the different types of communication media.

Incidents: During incidents information about the situation and what is being done to resolve it is collected. This overview provides a watch officer with information about incidents that are currently going on and with an entry point for zooming in on a specific incident.

Contacts: Because the Incidone tool is targeted at use in a centralized command center, all knowledge watch officers have of the area they are watching comes from contacts outside, or from automated systems like AIS (Automatic Identification System). To assist in maintaining a proper situational picture of the outside world, the tool provides support for collecting and sharing the whereabouts, status and background information of contacts. Contacts can either be persistent contacts like patrol vessels, or incidental ones such as a ship that broadcasted a distress call.

9.2.1 Responding to Calls and Messages

When a call comes in, the first two questions a watch officer has to answer are “who is calling?” and “what is this call about?” To make it easier to answer these questions the tool provides an interface (see Figure 9.2.) that offers a list of known contacts to choose from, in case the caller has called before, and a form to collect a name and contact information for first-time callers. Simultaneously it provides a list of current incidents to immediately link the call to, and provides a form to create a new incident record. Because it is impossible to anticipate what information a call may yield within its limited timeframe, the tool also supports freeform note taking for later processing.

Incoming messages are not bound to the same limited timeframe as calls, but the same questions apply: “who is the message from?” and “what is it about?” Depending on the type of message, different information may be present in the message to answer these questions. Therefore the tool provides a customized interface for each type of message for either linking it to an ongoing incident or creating a new incident record, as well as to identify its sender.

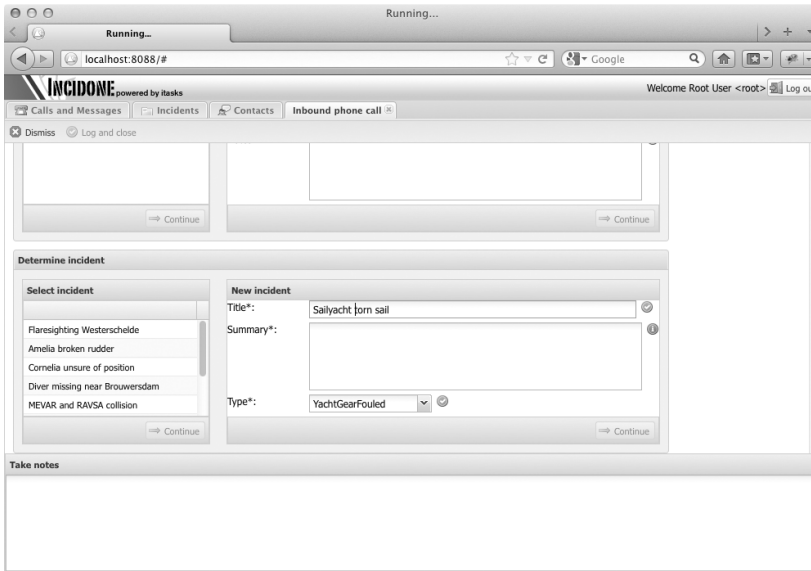


Figure 9.2: Responding to calls

9.2.2 Contribution to Ongoing Incidents

From the overview of incidents, which provides some general awareness, watch officers can “open” incidents to zoom in on a specific incident. This opens a new tab in which all information relevant to an incident is collected. The information for an incident is divided in six overviews. A summary of the situation, an overview of plans to resolve the incident and their progress, an overview of all contacts involved in this incident, a map on which all information with a geospatial aspect are plotted, an overview of the weather at the scene of the incident, and finally, a chronological log.

The most important overview for resolving incidents in a timely manner is the “Plans” overview. In Incidone, plans are hierarchical to-do lists that can be used to coordinate actions taken to resolve incidents. These to-do lists are not static lists, but are defined and refined as the incident unfolds by a coordinating watch officer. To support watch officers in this task, plans contain refinement functions that compute suggested sub-plans from the current available information about an incident combined with models of standard procedures. These suggestions can then be added to the plan in a single click. When new information is available, the suggestions are automatically recomputed. Suggestions are fully programmable. This means that any method for determining sugges-

ted actions can be plugged into the framework. Whether it is a simple business rule, or an integration with a third party system. Additionally, predefined sub-plans can contain optimized to-do items that embed a user-interface specifically designed for them. This makes it possible to optimize the workflow for common tasks as preparation for future incidents.

9.2.3 Tracking Contacts

Being confined to the limited viewpoint of a command center means that no direct information about the outside world is available and effort must be made to get a clear situational picture. The Incidone tool enables collection of information about “contacts” for two reasons. The first is to enable sharing information between watch officers such that the limited time windows of radio or phone contact are not wasted on redundant inquiry. The second reason why information about contacts is tracked, is to have an accurate representation of an incident available from which the plans can compute suggested actions. The tool makes no distinction between contacts that belong to the own organization, such as rescue vessels or patrol airplanes, and contacts that are in need of help. Needing help, or being available for deployment are tracked as attributes which can be assigned to any contact. This makes it possible to deal with any situation, even situations in which a unit deployed for a rescue gets in trouble itself and gets help from some vessel that happens to be around.

9.3 A Programmer’s View of Incidone

The previous section describes the Incidone tool from the perspective of its intended end-users. But this group of users is not the tool’s primary target audience. The source code of the tool is actually more interesting than its executable form because it is the first realistic crisis management application developed using the Task-Oriented Programming paradigm. It is intended as example and inspiration for programmers aspiring to develop similar tools. Incidone is written in the functional programming language Clean [69] using the iTasks framework [32] that facilitates programming in a Task-Oriented style. This means that programs are defined by functions that create tasks, compose tasks from other tasks, or modify tasks. Tasks are persistent units of work that, when executed, produce a result. Tasks can be simple basic tasks such as viewing a piece of text, filling out a form, querying a database or calling a third-party web-service. But tasks can also be complex dynamic compositions of tasks and functions where results of completed tasks are used to compute new compositions at runtime. In fact, a complete program is “just” a task. Task results are statically typed, which ensures that compositions are only possible when results used as parameters of other tasks “match”.

TOP programs emphasize the task that a piece of software is intended to support, and abstract from implementation details. Hence, TOP programs are concise yet accurate models of how to accomplish tasks an organization has prepared for. They provide an executable, interactive alternative to natural language plans and procedure documents.

9.3.1 A Short Impression of iTasks Code

In this short paper it is impossible to present much of the code that defines the Incidone tool. However, it is equally impossible to get an impression of a programming language without seeing code written in it. Therefore we will try to give a short impression by walking through two small fragments taken from Incidone's source.

The first fragment is a common task that anyone who has programmed a multi-user system will likely have written in his or her language of choice. The `doAuthenticated` function: This function ensures that only authenticated users can access the application. It is defined as follows:

```
1 doAuthenticated :: (Task a) → Task a | iTask a
2 doAuthenticated task
3   = enterCredentials >>= verifyCredentials >>= executeTask task
4 where
5   enterCredentials :: Task Credentials
6   enterCredentials
7     = enterInformation ("Log in", "Please enter your credentials") []
8
9   verifyCredentials :: Credentials → Task (Maybe User)
10  verifyCredentials {Credentials|username,password}
11    = authenticateUser username password
12
13  executeTask :: (Task a) (Maybe User) → Task a | iTask a
14  executeTask task (Just user) = workAs user task
15  executeTask task Nothing    = throw "Log in failed"
```

The `doAuthenticated` function is a function that takes any task of unspecified result type `a` and yields a task of the same type. Although `a` is a further unspecified type variable, the result type of `doAuthenticated` is guaranteed to be the same as that of its argument. The task being constructed in this function consists of three straightforward steps composed sequentially with the `>>=` operator. First a form for entering credentials needs to be filled out. This is specified by the local task definition `enterCredentials` in the **where** clause. This task produces a record structure with a name and password that are verified in `verifyCredentials` by the predefined task `authenticateUser`. The result of this task is pattern matched in the third step, the `executeTask` function. If the credentials are valid, and a `Just user` value is given, the final step consists of

executing the task that was passed as parameter with the identity of the user. If authentication has failed, and `Nothing` is given, an exception is thrown. This short example illustrates the Task-Oriented style of programming used to create the Incidone tool. It shows how programs are written through composition of tasks and abstraction to parameterized generic task functions.

9.3.2 Flexibility through Active Plans

The code we have shown thus far, although concise and focused, is still rather static in nature. It defines a task following a predefined structure. To support the type of Coast Guard operations for which the tool is designed we need to encode operational knowledge in the application. We could try to model standard operating procedures and contingency plans using the operators on tasks seen in the `doAuthenticated` function, but this would give us a rigid tool with little room to deal with incidents not foreseen by our plans. Instead we'll leverage the property of our framework that tasks are first-class citizens that we can use as building blocks.

Instead of directly defining monolithic task specifications to coordinate incidents, we define small self-contained tasks that are wrapped into data structures of type `ActivePlan`. These structures (formerly known as `HSTask` [42]) construct a loosely coupled framework in which tasks can be suggested based on available information but in which the final plan is composed by the watch officers. This way, straightforward tasks can be automated without forcing watch officers into a rigid workflow.

Active Plans are represented by the following data type:

```

1 :: ActivePlan context =
2   { title          :: String
3   , description    :: Maybe Note
4   , planId        :: Maybe PlanId
5   , completeWith  :: Maybe (CompleteTask context)
6   , refineWith    :: RefinementTasks context
7   }
8 :: CompleteTask context
9 ::= (Shared context) → Task Bool
10 :: RefinementTask context
11 ::= [PlanId] [PlanId] context → Task [ActivePlan context]
12 :: RefinementTasks context =
13   { suggested      :: Maybe (RefinementTask context)
14   , alternative    :: Maybe (RefinementTask context)
15   , custom         :: Maybe (context → Task (ActivePlan context))
16   }
```

Plans are parameterized with the type variable `context`, which is a shared context. In the Incidone tool this is instantiated with the `Incident` type for data

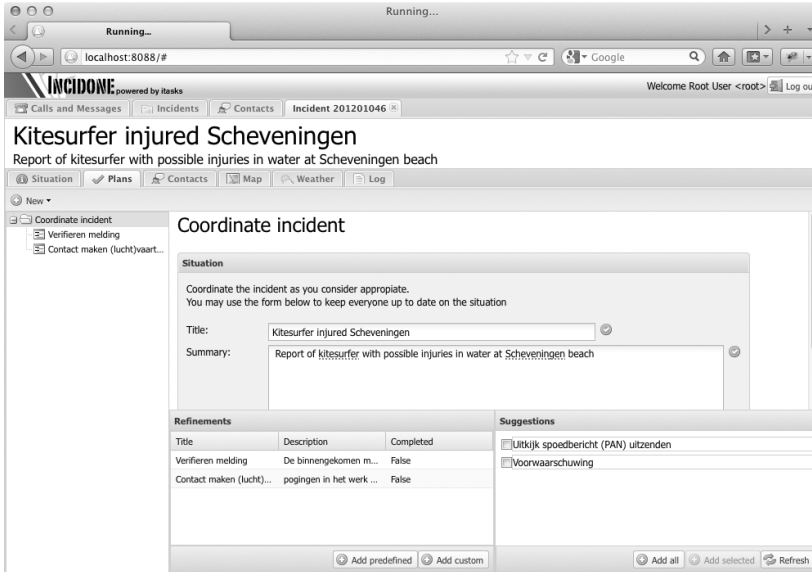


Figure 9.3: Working on incidents

structures containing all information relevant to an incident. The first three fields in the `ActivePlan` structure are basic meta-data. What makes this structure interesting are the `completeWith` and `refineWith` fields. The `completeWith` field wraps the task used to complete this plan. It is an optional (`Maybe`) field. If it is not defined, a default task is used to simply mark the plan as completed or canceled. The `refineWith` field contains a sub-structure with three tasks to refine this plan. The first two are used to determine suggested, or alternative predefined sub-plans. The third allows custom sub-plans to be defined. Each of these tasks is parameterized with the shared context as well as lists containing which plans are selected and/or completed.

Because the `iTasks` framework can only execute `Task` values, plans are put in motion by the `executePlan` function. This function interprets plans and creates the progress tracking, and suggestion tasks as shown in Figure 9.3.

9.4 Status Quo and Future Work

At the time of writing, a basic working version of the tool is available (from which the screenshots in Figures 9.1 to 9.3 have been taken). The code illustrated in this paper has been taken verbatim from the source code, but

may still change as the tool evolves. The overall structure and framework are completed, but there still are many “rough edges”. It cannot yet handle realistically large amounts of data, because searching and filtering tasks are not yet available and data persistence is not efficiently implemented. Additionally the user interface needs polishing to make better use of screen real estate and prevent unnecessary clicking or scrolling.

To effectively serve as an example, we also need to compile a realistic collection of demonstration content. This includes a database of predefined contacts, predefined plans, and a database with fictional contacts and incidents.

9.5 Conclusion

In this paper we have introduced a project in which an incident coordination tool for coast guard operations is developed. The distinguishing feature of this tool is that it is developed using the Task-Oriented Programming paradigm. We have presented the current capabilities of the tool, and provided an impression of its code. When completed, we intend to publish it with full source code as example of the paradigm for developers of similar crisis management applications.

Bibliography

- [1] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology, 2002. Cited on pages 40 and 84.
- [2] A. Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005. ISBN 3-540-67658-9. Cited on page 71.
- [3] A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, 2002. Cited on page 30.
- [4] A. Alimarine and S. Smetsers. Optimizing Generic Functions. In D. Kozen, editor, *The 7th International Conference, Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 16–31. Springer Verlag, Jul 2004. Cited on page 106.
- [5] G. Bakema, J. Zwart, and H. van der Lek. *Fully Communication Oriented Information Modelling*. FCO-IM Consultancy, 2002. Cited on page 118.
- [6] B. Bingert and A. Höckersten. Student paper: HaskellDB improved. In *Proceedings of 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM Press, 2004. Cited on page 108.
- [7] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM. Cited on page 47.
- [8] J. R. Boyd. The essence of winning and losing, 1996. Unpublished lecture notes. Cited on page 143.

- [9] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. Cited on pages 92 and 94.
- [10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, Nov. 2006. Springer-Verlag. Cited on page 84.
- [11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. An idiom’s guide to formlets. Technical report, The University of Edinburgh, UK, 2007. <http://groups.inf.ed.ac.uk/links/papers-/formlets-draft2007.pdf>. Cited on page 84.
- [12] B. Crandall, G. Klein, and R. R. Hoffman. *Working Minds: A practitioner’s guide to cognitive task analysis*. MIT Press, 2006. Cited on page 84.
- [13] Django framework. <http://www.djangoproject.com/>. Cited on page 36.
- [14] M. Elsman and K. Friis Larsen. Typing XHTML web applications in ML. In *Proceedings of the 6th International Symposium on the Practical Aspects of Declarative Programming, PADL '04*, volume 3057 of *Lecture Notes in Computer Science*, pages 224–238. Dallas, TX, USA, Springer-Verlag, June 2004. Cited on page 83.
- [15] M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Proceedings of the 5th International Symposium on the Practical Aspects of Declarative Programming, PADL '03*. New Orleans, LA, USA, Springer-Verlag, Jan. 2003. Cited on page 83.
- [16] ExtJS framework. <http://www.extjs.com/>. Cited on page 36.
- [17] D. Fahland and H. Woith. Towards process models for disaster response. In M. Leoni, S. Dustdar, and A. Hofstede, editors, *Proceedings of the First International Workshop on Process Management for Highly Dynamic and Pervasive Scenarios (PM4HDPS), co-located with 6th International Conference on Business Process Management (BPM'08)*., 2008. Cited on pages 121, 124, and 134.
- [18] M. Fowler and J. Highsmith. The agile manifesto. *Software Development*, 9(August):2835, 2001. Cited on page 109.

- [19] M. Fussel. Foundations of object-relational mapping. <http://www.chimu.com/publications/objectRelational/index.html>, 1997. Whitepaper. Cited on page 107.
- [20] J. van Groningen, T. van Noort, P. Achten, P. Koopman, and R. Plasmeijer. Exchanging sources between Clean and Haskell - A double-edged front end for the Clean compiler. In J. Gibbons, editor, *Proceedings of the Haskell Symposium, Haskell '10, Baltimore, MD, USA*, pages 49–60. ACM Press, 2010. Cited on page 57.
- [21] O. M. Group. The semantics of business vocabulary and business rules, 2009. <http://www.omg.org/spec/SBVR/1.0/>. Cited on page 118.
- [22] T. Halpin. *Information modeling and relational database: from conceptual analysis to logical design*. Morgan Kaufmann Publishers Inc, 2001. Cited on page 92.
- [23] T. Halpin. Orm 2. In R. Meersman, Z. Tari, and P. Herrero, editors, *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 676–687. Springer-Verlag, 2005. Cited on pages 109 and 118.
- [24] M. Hanus. High-level server side web scripting in Curry. In *Proceedings of the 3rd International Symposium on the Practical Aspects of Declarative Programming, PADL '01*, pages 76–92. Springer-Verlag, 2001. Cited on pages 36 and 83.
- [25] C. Heath. The constellation query language. In R. Meersman, P. Herrero, and T. Dillon, editors, *On The Move To Meaningful Internet Systems: OTM 2009 Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 682–691. Springer-Verlag, 2009. Cited on page 118.
- [26] Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS '08*, volume 5301 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2008. Cited on page 84.
- [27] R. Hinze. A new approach to generic functional programming. In T. Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000. Cited on page 71.
- [28] M. Ingmarsson, H. Eriksson, and N. Hallberg. Exploring development of service-oriented C2 systems for emergency response. In J. Landgren and

- S. Jul, editors, *Proceedings of the 6th International ISCRAM Conference*, Gothenburg, Sweden, 2009. ISCRAM Association. Cited on page 126.
- [29] A. Jacobson. Haskell application server, 2006. <http://happs.org/>. Cited on page 36.
- [30] J. Jansen, B. Lijnse, and R. Plasmeijer. Towards dynamic workflows for crisis management. In S. French, B. Tomaszewski, and C. Zobel, editors, *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management, ISCRAM '10*, Seattle, WA, USA, May 2010. Cited on pages 10, 82, 83, 131, and 134.
- [31] J. Jansen, B. Lijnse, R. Plasmeijer, and T. Grant. Web based dynamic workflow systems for C2 of military operations. In *Revised Selected Papers of the 15th International Command and Control Research and Technology Symposium, ICCRTS '10*, Santa Monica, CA, USA, June 2010. Cited on page 11.
- [32] J. Jansen, R. Plasmeijer, P. Koopman, and P. Achten. Embedding a web-based workflow management system in a functional language. In C. Brabrand and P. Moreau, editors, *Proceedings 10th Workshop on Language Descriptions, Tools and Applications, LDTA '10*, pages 79–93, Paphos, Cyprus, March 27-28 2010. Cited on pages 110, 148, and 151.
- [33] J. Jansen, R. Plasmeijer, P. Koopman, and P. Achten. Embedding a web-based workflow management system in a functional language. In C. Brabrand and P. Moreau, editors, *Proceedings 10th Workshop on Language Descriptions, Tools and Applications, LDTA '10*, pages 79–93, Paphos, Cyprus, March 27-28 2010. Cited on page 136.
- [34] C. Janssen, A. Weisbecker, and J. Ziegler. Generating user interfaces from data models and dialogue net specifications. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 418–423, New York, NY, USA, 1993. ACM. Cited on page 107.
- [35] S. Jul. Who's really on first? a domain-level user, task and context analysis for response technology. In B. v. d. Walle, P. Burghardt, and C. Nieuwenhuis, editors, *Proceedings of the 5th International ISCRAM Conference*, Delft, the Netherlands, 2007. Cited on pages 126 and 129.
- [36] P. Koopman, R. Plasmeijer, and P. Achten. *An Effective Methodology for Defining Consistent Semantics of Complex Systems*, volume 6299 of *LNCS*, pages 224–267. Springer-Verlag, Komarno, Slovakia, 25-130, May 2009. Cited on page 63.

- [37] P. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for iTasks. In S.-B. Scholz and O. Chitil, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '08, Hertfordshire, UK*, volume 5836 of *LNCS*, pages 212–232, Hatfield, UK, 2011. Springer. Cited on page 18.
- [38] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug. 1988. Cited on page 47.
- [39] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct 1999. Also appeared in *ACM SIGPLAN Notices* 35, 1, (Jan. 2000). Cited on page 108.
- [40] B. Lijnse. Between types and tables: Generic mapping between relational databases and data structures in clean. Master's thesis, University of Nijmegen, Jul 2008. Number 590. Cited on pages 96, 97, and 99.
- [41] B. Lijnse. Tussen types en tabellen... *Optimize: Onafhankelijk vaktijdschrift voor de Oracle-professional*, 13(4):20–25, Sept. 2010. Cited on page 9.
- [42] B. Lijnse, J. Jansen, R. Nanne, and R. Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In D. Mendonca and J. Dugdale, editors, *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM '11*, Lisbon, Portugal, May 2011. ISCRAM Association. Cited on pages 4, 10, 82, 83, 147, and 153.
- [43] B. Lijnse, J. Jansen, and R. Plasmeijer. Incidone: A task-oriented incident coordination tool. In L. Rothkrantz, J. Ristvej, and Z. Franco, editors, *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*, Vancouver, Canada, Apr. 2012. Cited on pages 11 and 82.
- [44] B. Lijnse and R. Plasmeijer. iTasks 2: iTasks for End-users. In M. Morazán and S. Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA*, volume 6041 of *LNCS*, pages 36–54. Springer-Verlag, 2010. Cited on pages 7 and 83.
- [45] B. Lijnse and R. Plasmeijer. Between types and tables - Using generic programming for automated mapping between data types and relational

- databases. In S. Scholz and O. Chitil, editors, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, volume 5836 of *LNCS*, pages 272–290, Hatfield, UK, 2011. Springer. Cited on pages 9, 118, 145, and 146.
- [46] B. Lijnse, P. van Bommel, and R. Plasmeijer. Ccl: A lightweight orm embedding in clean. In P. Herrero, H. Panetto, R. Meersman, and T. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2012 Workshops*, volume 7567 of *Lecture Notes in Computer Science*, pages 338–347, Rome, Italy, Sept. 2012. Springer. Cited on page 10.
- [47] F. Loitsch and M. Serrano. Hop client-side compilation. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '07*, pages 141–158, New York, NY, USA, 2-4, Apr. 2007. Interact. Cited on page 84.
- [48] E. Manoku, J. P. Zwart, and G. Bakema. A fact approach to automatic application development. *Journal of conceptual modeling*, Sep 2006. Cited on page 107.
- [49] J. McCormack, T. Halpin, and P. Ritson. Automated mapping of conceptual schemas to relational schemas. In *Proceedings of the Fifth International Conference CAiSE'93 on Advanced Information Systems Engineering*, volume 685 of *LNCS*, pages 432–448. Springer Verlag, 1993. Cited on pages 90, 96, and 97.
- [50] R. Meersman. The RIDL conceptual language. Technical report, International Centre for Information Analysis Services, Control Data Belgium Inc., 1982. Cited on page 118.
- [51] E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000. Cited on page 83.
- [52] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM. Cited on page 108.
- [53] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. Tech report, CS-09-04, Brown University, Providence, RI, 04 2009. Cited on page 36.
- [54] S. Michels and R. Plasmeijer. Uniform data sources in a functional language. Submitted for presentation at Symposium on Trends in Functional Programming, TFP '12, 2012. Cited on page 67.

- [55] S. Michels, R. Plasmeijer, and P. Achten. iTask as a new paradigm for building GUI applications. In J. Hage and M. Morazán, editors, *Proceedings of the 22nd International Symposium on the Implementation and Application of Functional Languages, IFL '10, Selected Papers*, volume 6647 of *LNCS*, pages 153–168, Alphen aan den Rijn, The Netherlands, 2011. Springer. Cited on pages 40, 47, 48, and 50.
- [56] G. Nijssen and T. Halpin. *Conceptual schema and relational database design: A fact oriented approach*. Prentice Hall, New York, 1989. Cited on page 118.
- [57] U. Norell. Dependently typed programming in agda. Technical Report ICIS-R08008, Radboud University Nijmegen, 2008. Cited on page 108.
- [58] H. Peukert, D. Lincourt, and B. Zimmermann. Support for agile planning & execution of coordinated actions. In *Proceedings of 14th ICCRTS C2 and Agility*, 2009. Cited on page 134.
- [59] S. Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. Cited on page 4.
- [60] R. Plasmeijer and P. Achten. iData for the world wide web - Programming interconnected web forms. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS '06*, volume 3945 of *LNCS*, pages 242–258, Fuji Susone, Japan, 24–26, Apr. 2006. Springer Verlag. Cited on page 36.
- [61] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 2007. ACM Press. Cited on pages 4, 18, 20, 22, 39, 83, 122, and 132.
- [62] R. Plasmeijer, P. Achten, and P. Koopman. An introduction to iTasks: defining interactive work flows for the web. In *Revised Selected Lectures of the 2nd Central European Functional Programming School, CEFP '07*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, June 2008. Springer-Verlag. Cited on page 22.
- [63] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, and T. van Noort. An iTask case study: a conference management system. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Revised Lectures of the International Summer School on Advanced Functional Programming, AFP '08, Heijen, The Netherlands*, volume 5832 of *LNCS*, pages 306–329. Springer-Verlag, 2008. Cited on pages 11, 18, and 20.

- [64] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. van Noort, and J. van Groningen. iTasks for a change - Type-safe run-time change in dynamically evolving workflows. In S. Khoo and J. Siek, editors, *Proceedings of the Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160. ACM Press, 2011. Cited on pages 8, 44, and 45.
- [65] R. Plasmeijer, P. Achten, B. Lijnse, and S. Michels. Defining multi-user web applications with iTasks. In V. Zsóka, Z. Horváth, and R. Plasmeijer, editors, *Proceedings of the 4th Central European Functional Programming School, CEFP '11, Revised Selected Papers*, volume 7241 of *LNCS*, pages 46–92, Eötvös Loránd University, Budapest, Hungary, 14–24, June 2012. Springer. Cited on page 9.
- [66] R. Plasmeijer, J. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP '08*, pages 56–66, Valencia, Spain, 15–17, July 2008. Cited on pages 18 and 128.
- [67] R. Plasmeijer, B. Lijnse, P. Achten, and S. Michels. Getting a grip on tasks that coordinate tasks. In *Proceedings Workshop on Language Descriptions, Tools, and Applications (LDTA)*, Saarbrücken, Germany, March 26–27 2011. Cited on page 8.
- [68] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM. Cited on page 8.
- [69] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). <http://clean.cs.ru.nl>, 2002. Cited on pages 4, 110, and 151.
- [70] Ruby on Rails. <http://rubyonrails.org/>. Cited on page 36.
- [71] C. Sell and I. Braun. Using a workflow management system to manage emergency plans. In J. Landgren and S. Jul, editors, *Proceedings of the 6th International ISCRAM Conference*, Gothenburg, Sweden, 2009. ISCRAM Association. Cited on pages 121, 124, 126, and 134.
- [72] M. Serrano, E. Gallezio, and F. Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the 11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '06*, pages 975–985, Portland, Oregon, USA, 22–26, Oct. 2006. Cited on page 84.

- [73] S. Smetsers, A. van Weelden, and R. Plasmeijer. Efficient and type-safe generic data storage. In *Proceedings of the 1st Workshop on Generative Technologies, WGT '08*, Budapest, Hungary, Apr 2008. Electronic Notes in Theoretical Computer Science. Cited on page 107.
- [74] A. ter Hofstede, H. Proper, and T. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489 – 523, 1993. Cited on page 118.
- [75] P. Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In S. Krishnamurthi and R. Ramakrishnan, editors, *Proceedings of the 4th International Symposium on the Practical Aspects of Declarative Programming, PADL '02*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208, Portland, OR, USA, 19–20, Jan. 2002. Springer-Verlag. Cited on pages 36 and 84.
- [76] UML version 2.2 specification. <http://www.omg.org/spec/UML/2.2/>, Feb 2009. Cited on page 92.
- [77] M. van der Heijden, B. Lijnse, P. Lucas, Y. Heijdra, and T. Schermer. Managing COPD exacerbations with telemedicine. In *13th Conference on Artificial Intelligence in Medicine, AIME '11*, volume 6747 of *LNCS*, pages 169–178, Bled, Slovenia, July 2011. Springer-Verlag. Cited on pages 4, 11, and 83.
- [78] M. Vervoort and R. Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In R. Peña and T. Arts, editors, *Revised Selected Papers of the 14th International Workshop on the Implementation of Functional Languages, IFL '02, Madrid, Spain*, volume 2670 of *LNCS*, pages 101–117. Springer-Verlag, 2003. Cited on page 69.
- [79] P. Wadler. Comprehending monads. In *Proceedings of the Conference on Lisp and Functional Programming, LFP '90, Nice, France*, pages 61–77, 1990. Cited on page 44.
- [80] A. van Weelden. *Putting types to good use*. PhD thesis, Radboud University Nijmegen, 17, Oct. 2007. ISBN 978-90-9022041-3. Cited on page 69.

Summary

When we use computers to accomplish a task, we usually do so to make our work easier, faster, cheaper, or simply because the task would be practically impossible without a computer. Because computers are multi-purpose machines, we need to program them for the task we intend to accomplish. Ideally this would be a straightforward process of expressing the task at hand as an algorithmically solvable problem, but in reality programming is a lot more complex. Contemporary application programming is more about gluing together existing libraries, components and subsystems than it is about expressing algorithms. With the abundance of libraries, API's, languages and components that are available, there are so many technical details to consider that it is easy to get absorbed by them instead of staying focused on the task at hand. In such programs one has to read between the lines to figure out what task is actually being accomplished. To what extent this matters depends on the application. Integration of the latest fashionable API or service may be what sells some applications, but for computer systems that support mission-critical tasks, overhead caused by a bad fit between the computer system and the task it is supposed to support matters more. To rationally design systems focused on optimally supporting critical tasks, it is important to understand the task you aim to support. However, if a task is complex and has no clearly defined boundaries or outcome, this is much easier said than done.

Task-Oriented Programming (TOP) is a new programming paradigm that uses "tasks" as central concept for constructing programs. Although tasks are a common notion in daily life, in order to use them as building block in programs we need to be more precise. With a formal notion of tasks, a way to express atomic tasks, and the means to define tasks in terms of other tasks, we can think about programs in terms of the tasks we need to do, instead of the necessary underlying technical details. In TOP, a task is a specified piece of work aiming to produce a result of known type. When executed, tasks produce (temporary) results that can be observed in a controlled way. As work progresses it can be continuously monitored and controlled by other tasks. Tasks can either be fully automated, or can be performed by humans with computer support. TOP is therefore best suitable for applications that cannot be fully automated. These can be supported by a mix of interactive and automated tasks that have to be

coordinated to achieve a combined goal though. The most extreme examples of these are those applications in which the conditions under which the task has to be accomplished are unpredictable, where people have to work together, and there is time pressure to get it done. Tasks such as responding to incidents with a rescue operation, or the management of crises in general, belong to this category. The ideas behind TOP emerged during development of the iTask System (or iTasks for short), a concrete prototype system that uses tasks as its core concept, when we applied it to such cases. With iTasks, we have continuously reflected on TOP to ensure that the paradigm is able to capture a wide range of real-world tasks.

This dissertation consists of three parts. Each covers a research theme that has been investigated simultaneously with the other themes. Together they present the research that I have contributed to, in order to understand the use of tasks to structure incident response applications. The first part, “Task-Oriented Programming with iTasks” (chapters 2, 3 and 4), illustrates the evolution of the iTask System and the emergence of the TOP paradigm, which has been my main research focus. The chapters report on the status quo and progress early in the project, more or less halfway and at the end of the project. As a series they show the progressed insight into TOP. The second part, “Information Models and Data Types” (chapters 5 and 6), contains contributions concerning the intersection of information models, for defining shared databases, and data types that define values in programs. Although these contributions extend beyond TOP, they enable programming information systems in a task-oriented way. This is because in such systems most tasks deal with storing, retrieving and sharing information. The third and final part, “The Netherlands Coast Guard Case” (chapters 7, 8 and 9), is concerned with TOP’s potential to be used for the development of incident response applications. The most influential application with which we reflected on TOP has been a case study of the search and rescue operations of the Netherlands Coast Guard. The chapters in this part cover that case.

With TOP we have a new approach for the design and development of interactive programs. We reduce programming to what matters most: understanding the task at hand.

Samenvatting

Wanneer we computers gebruiken om een taak te volbrengen, doen we dat meestal om ons werk makkelijker, sneller of goedkoper te maken, of simpelweg omdat de taak praktisch onmogelijk zou zijn zonder computer. Omdat computers universele machines zijn, moeten we ze programmeren voor specifieke taken. In het ideale geval zou dit het rechttoe-rechtaan uitdrukken zijn van de uit te voeren taak als een algorithmisch oplosbaar probleem. In werkelijkheid is programmeren echter een stuk complexer. Het huidige applicatie programmeren bestaat meer uit het aan elkaar knopen van bestaande bibliotheken, componenten en systemen, dan uit het uitdrukken van algorithmes. Maar door de overdaad aan bibliotheken, API's, programmeertalen en componenten die beschikbaar zijn, zijn er zoveel technische details om je mee bezig te houden dat het makkelijk is om erin te verdrinken in plaats van geconcentreerd te blijven op de taak die volbracht moet worden. In zulke programma's moet je tussen de regels door lezen wat men probeert te bereiken. Of, en in welke mate, dit een probleem is, hangt af van de toepassing. Integratie met de nieuwste hippe API of dienst is voor sommige applicaties een belangrijk verkoopargument. Maar voor computersystemen die kritieke taken ondersteunen weegt de overhead die ontstaat wanneer een systeem niet bij de uit te voeren taak aansluit zwaarder. Om rationeel systemen te ontwerpen gefocust op het optimaal ondersteunen van kritieke taken, is het belangrijk om die taken goed te begrijpen. Echter, wanneer een taak complex is en geen duidelijke grenzen of uitkomsten heeft, is dit makkelijker gezegd dan gedaan.

Taakgeoriënteerd Programmeren (Task-Oriented Programming oftewel TOP) is een nieuw programmeerparadigma dat “taken” gebruikt als centraal concept voor het construeren van programma's. Hoewel taken een bekend begrip zijn in het dagelijks leven, om ze als bouwstenen van programma's te gebruiken moeten we preciezer zijn in wat we ermee bedoelen. Door een formele notie van taken te definiëren samen met een manier om atomaire taken te beschrijven en middelen om taken uit te drukken in termen van andere taken, kunnen we over programma's nadenken in termen van de taken die we moeten doen in plaats van de noodzakelijke technische details om ze te realiseren. In TOP is een taak een vastgestelde eenheid werk gericht op het leveren van een resultaat van een bekend type. Wanneer taken uitgevoerd worden, produceren ze

(tijdelijke) resultaten die op gecontroleerde wijze geobserveerd kunnen worden. Terwijl het werk vordert, kan het continue gemonitord en gecontroleerd worden door andere taken. Taken kunnen ofwel volledig geautomatiseerd zijn, ofwel uitgevoerd worden door mensen met computerondersteuning. TOP is daarom vooral geschikt voor het soort toepassingen dat niet volledig geautomatiseerd kan worden. Deze kunnen wel ondersteund kunnen worden met een mix van interactieve en geautomatiseerde taken die op gecoördineerde wijze een gezamenlijke doel trachten te bereiken. De meest extreme voorbeelden hiervan zijn toepassingen waarbij taken onder onvoorspelbare omstandigheden en onder tijdsdruk uitgevoerd moet worden, en waarbij mensen samen moeten werken. Taken zoals het uitvoeren van een reddingsoperatie naar aanleiding van een incident, of het managen van crises in het algemeen, behoren tot deze categorie. De ideeën achter TOP zijn ontstaan toen we het iTask Systeem (kortweg iTasks), een prototype systeem dat taken gebruikt als kernconcept, gingen toepassen op dergelijke cases. Om te zorgen dat het TOP paradigma een breed scala aan reële toepassingen aan kan, hebben we continue met iTasks gereflecteerd op de mogelijkheden.

Dit proefschrift bestaat uit drie delen. Ieder deel behandelt een eigen onderzoeksthema's dat gelijktijdig met de andere thema's onderzocht is. Samen illustreren ze het onderzoek naar het gebruik van taken voor "incident response" toepassingen waaraan ik bij gedragen heb. Het eerste deel, "Task-Oriented Programming with iTasks" (hoofdstukken 2, 3 en 4), illustreert de evolutie van het iTask systeem en het ontstaan van het TOP paradigma. Dit thema was mijn primaire onderzoeksfocus. De hoofdstukken rapporteren de status en voortgang vroeg in het project, ongeveer halverwege het project, en aan het einde van het project. Samen geven ze het voorschrijdend inzicht in TOP weer. Het tweede deel, "Information Models and Data Types" (hoofdstukken 5 en 6), bevat bijdragen op het snijvlak tussen informatiemodellen, voor het beschrijven van gedeelde databases, en datatypes die lokale gegevens in programma's beschrijven. Alhoewel deze bijdragen breder zijn dan TOP alleen, maken ze het programmeren van informatiesystemen op een taakgeoriënteerde manier mogelijk. Dit is omdat de meeste taken in zulke systemen te maken hebben met het opslaan, terughalen en delen van informatie. Het derde en laatste deel, "The Netherlands Coast Guard Case" (hoofdstukken 7, 8 en 9), heeft betrekking op TOP's potentiële gebruik voor het ontwikkelen van "incident response" toepassingen. De belangrijkste toepassing waarmee we op TOP gereflecteerd hebben was een casus over de "Search and Rescue" operaties van de Nederlandse Kustwacht. De hoofdstukken in dit deel behandelen deze casus. Met TOP hebben we een nieuwe aanpak voor het ontwerpen en ontwikkelen van interactieve programma's. We reduceren programmeren tot de hoofdzaak: het begrijpen van de taak die je wilt volbrengen.

Acknowledgements

Although writing a dissertation is by definition a one-man project, I have never felt I had to do it alone. I have been lucky to have been surrounded by so many supporting people to whom I owe my gratitude, that it is impossible to mention each of them here without forgetting anyone. If you feel your name should be mentioned on this page, you are probably right and I apologize.

First of all, I want to thank my promotor Rinus and copromotor Jan Martin for their advice and support. I have met few PhD students who have received the amount of attention from their advisors as I have enjoyed the past few years.

I want to thank everyone at MBSD with whom I have worked together with, or who have had to endure my occasional rants simply because we shared an office. Thomas, Peter, Pieter, John, Erik, Steffen, Jeroen, László, Maarten, Sander, Marina, Freek, Martijn, Patrick, Stijn and Peter, thanks.

Similarly, I want to thank the people I have worked together with directly at the NLDA. Ruud, Tim, Dick and Fok, thanks. I also want to thank everyone who have made it possible for me to work both in Nijmegen and Den Helder. Thanks to them, and to the off-work hours company of people like Paul, Lanah and Ralph, my trips to Den Helder have been both enjoyable and productive. I am grateful to everyone at the Netherlands Coast Guard. People who enabled my research like KTZ Trimpe Burger, Gerrit, Andre and Jan, but most of all Duty officers Sjaco, Ron, Rolf and all the Watch officers who have accepted me in their environment and gave me an impression of their work.

I am also grateful to the countless people from the functional programming and ISCRAM communities with whom I have had valuable discussions at conferences, summerschools and other events. Especially ISCRAM with its encouraging attitude towards PhD students has meant a lot. Who knew how much influence a summerschool in Tilburg would have on my project.

I want to thank Kol Klaren and the members of SENECA for giving me a chance to take this research to the next level and show how it can be applied in an operational setting.

Finally I want to thank my parents and family for teaching me to have faith in myself, and to confidently follow my curiosity, Linda who is always there for me reminding me every now and then to relax and enjoy what I have, and of course Mara who already means more to me than this dissertation.

Curriculum Vitae

Bas Lijnse was born on January the 13th 1984 in Valkenisse, The Netherlands. He started his secondary education (VWO) in 1995 at the Christelijke Scholengemeenschap Walcheren in Middelburg and graduated in 2000 through state examination (Staatsexamen). From 2001 to 2008, he studied Informatics at the Radboud University Nijmegen where he received a Bachelors degree in 2007 and graduated with a Masters degree in 2008. During this period he had various part-time jobs as a software developer. From 2002 to 2004 at Hexon BV, and in 2004 also at Kalden Projects BV. In 2005 he co-founded EntiQ BV, a software development company that specialized in web-based information systems. In 2007 he left EntiQ to complete his studies. After graduation he remained at the Radboud University. He started as a scientific programmer between 2008 and 2009. From 2009 to 2012 he got funded by the Netherlands Defense Academy to pursue a PhD as junior researcher.

Titles in the IPA Dissertation Series since 2007

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

- A.H.J. Mathijssen.** *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17
- D. Jarnikov.** *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18
- M. A. Abam.** *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19
- W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01
- A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02
- M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03
- A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04
- N.C.W.M. Braspenning.** *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05
- M. Bravenboer.** *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06
- M. Torabi Dashti.** *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07
- I.S.M. de Jong.** *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08
- I. Hasuo.** *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09
- L.G.W.A. Cleophas.** *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10
- I.S. Zapreev.** *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11
- M. Farshi.** *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12
- G. Gulesir.** *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13
- F.D. Garcia.** *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

- P. E. A. Dürr.** *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15
- E.M. Bortnik.** *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16
- R.H. Mak.** *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17
- M. van der Horst.** *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18
- C.M. Gray.** *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19
- J.R. Calamé.** *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20
- E. Mumford.** *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21
- E.H. de Graaf.** *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22
- R. Brijder.** *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23
- A. Koprowski.** *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24
- U. Khadim.** *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25
- J. Markovski.** *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26
- H. Kastenber.** *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27
- I.R. Buhan.** *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28
- R.S. Marin-Perianu.** *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29
- M.H.G. Verhoef.** *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01
- M. de Mol.** *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science,

Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS*

using JML. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

- R.S.S. O'Connor.** *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19
- B. Ploeger.** *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20
- T. Han.** *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21
- R. Li.** *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22
- J.H.P. Kwisthout.** *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23
- T.K. Cocx.** *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24
- A.I. Baars.** *Embedded Compilers.* Faculty of Science, UU. 2009-25
- M.A.C. Dekker.** *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26
- J.F.J. Laros.** *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27
- C.J. Boogerd.** *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01
- M.R. Neuhäuser.** *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02
- J. Endrullis.** *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03
- T. Staijen.** *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04
- Y. Wang.** *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05
- J.K. Berendsen.** *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06
- A. Nugroho.** *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07
- A. Silva.** *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08
- J.S. de Bruin.** *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

- D. Costa.** *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10
- M.M. Jaghoori.** *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11
- R. Bakhshi.** *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01
- B.J. Arnoldus.** *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02
- E. Zambon.** *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03
- L. Astefanoaei.** *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04
- J. Proença.** *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05
- A. Morali.** *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06
- M. van der Bijl.** *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07
- C. Krause.** *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08
- M.E. Andrés.** *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09
- M. Atif.** *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10
- P.J.A. van Tilburg.** *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11
- Z. Protic.** *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12
- S. Georgievska.** *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13
- S. Malakuti.** *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14
- M. Raffelsieper.** *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

- C.P. Tsirogiannis.** *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16
- Y.-J. Moon.** *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17
- R. Middelkoop.** *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18
- M.F. van Amstel.** *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19
- A.N. Tamalet.** *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20
- H.J.S. Basten.** *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21
- M. Izadi.** *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22
- L.C.L. Kats.** *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23
- S. Kemper.** *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24
- J. Wang.** *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25
- A. Khosravi.** *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02
- Z. Hemel.** *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03
- T. Dimkov.** *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04
- S. Sedghi.** *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05
- F. Heidarian Dehkordi.** *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06
- K. Verbeek.** *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

- D.E. Nadales Agut.** *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08
- H. Rahmani.** *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09
- S.D. Vermolen.** *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10
- L.J.P. Engelen.** *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11
- F.P.M. Stappers.** *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12
- W. Heijstek.** *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13
- C. Kop.** *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14
- A. Osaiweran.** *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15
- W. Kuijper.** *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16
- H. Beohar.** *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01
- G. Igna.** *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02
- E. Zambon.** *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03
- B. Lijnse.** *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

Work it harder, make it better, do it faster, makes us stronger, more than ever hour after our work is never over.

– Daft Punk