Separation of Concerns in iTasks

Implementing a Command & Control System in a Pure Functional Language

– Research article, extended abstract –

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{j.stutterheim, p.achten, rinus}@cs.ru.nl

1 Introduction

Command and Control (C2) systems are tools that enable gathering information from distributed people, sensors, and other information sources. Their goal is to inform and coordinate all participating distributed operating parties in an optimal way such that they can accomplish their goals. For example, the Dutch Coast Guard uses C2 systems to coordinate Search and Rescue actions for people in problems on the North Sea. The Royal Netherlands Navy is currently modernising its C2 systems in order to improve its capabilities during missions. Technologies like the Internet, modern programming languages, and modern sensors will enable future C2 systems to support more complex missions more easily. Exactly how such technologies will be utilized is still subject to operational research, however.

In order to explore the possibilities modern technologies offer, the Royal Netherlands Navy is currently investing in prototype C2 system development. A key requirement to these prototypes is that one can rapidly iterate over new revisions and quickly add, remove, or change features. At the same time, these prototypes need to support complex ways of information gathering and the cooperation between distributed groups of people, but also cooperation between people and machines.

One of the tools the Royal Netherlands Navy chose to construct these prototypes is iTasks [3]. iTasks is a shallowly embedded domain-specific language (eDSL) that implements the Task-Oriented Programming paradigm (TOP) in the purely functional programming language Clean [4]. In TOP, programs are specified in terms of *tasks*. Tasks are represented by the monadic type `(Task a)`. iTasks' combinators are used to compose multi-user web-based applications. Common technical issues related to distributed client-server settings, such as communication, synchronization, user interface generation, and interaction handling, are handled automatically by applying advanced functional programming techniques. These include type driven generic functions, and the ability to store, load and communicate closures in a type safe way using Clean's dynamic system.

Building multi-user collaborative applications is non-trivial in any programming language. Doing so in a pure functional programming language is non-trivial in particular. Traditionally, one of the weak spots of pure functional programming languages has been I/O. Advanced concepts like monads [5] and uniqueness typing [1] had to be developed to deal with side effects in a pure language. Yet, managing I/O between multiple users is at the core of C2 systems. iTasks hides the gritty details of performing I/O in a distributed setting. At the same time, all aspects of an application, such as business logic, the (graphical) user interface, and storage, are implemented in a fully orthogonal way, enabling a separation of concerns.

In this paper we present a first step towards a prototype C2 system we have built for the Royal Netherlands Navy using iTasks. This system has two major features. First, it allows a lay person to model the lay-out of a ship and location of its main systems in a user-friendly graphical editor. Secondly, we can use these ship models to simulate fire-fighting and damage-control (FFDC) scenarios, both fully simulated and using human-in-the-loop testing. These scenarios provide insights in how the lay-out of a ship and the location of its systems influence the way calamities can be resolved. We also show how iTasks is used to solve common challenges encountered when applying pure functional languages to write interactive applications. These challenges include not only handling I/O, but also client/server communication, user-interface generation, and the software development process itself.

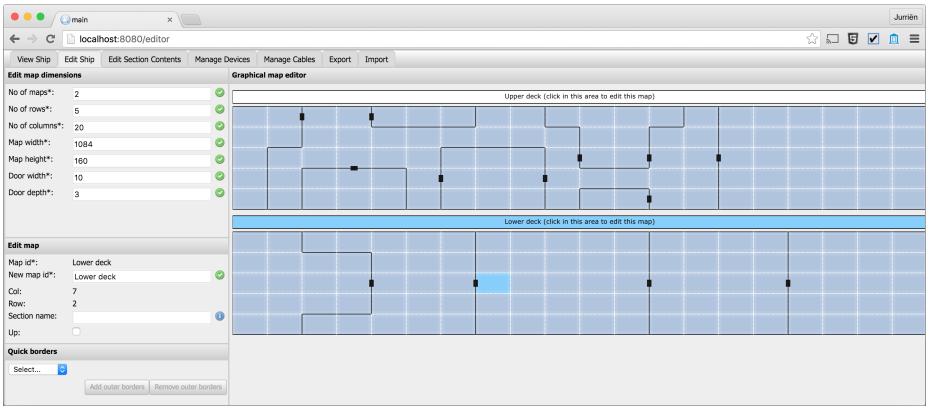
The rest of this abstract is structured as follows. Section 2 shows a demonstration of the C2 system. Section 3 introduces the core iTask concepts that are used to model TOP applications. In Section 4 we show how we have implemented the system, highlighting the challenges we have encountered along the way. Section 5 will discuss related work. Finally, for now, Section 6 discusses our results so far and concludes with some future work.

2 C2 System Demonstration

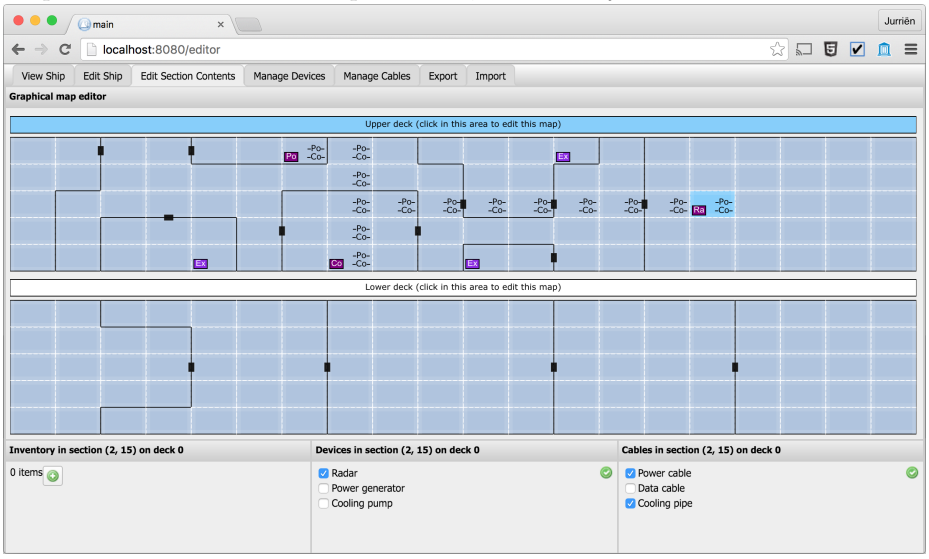
In this section we demonstrate the prototype C2 system we have developed and highlight some of its more complex aspects. This demonstration is split in two parts: the first part in Section 2.1 shows how one can construct part of a ship using the ship designer. The second part in Section 2.2 shows how we can simulate FFDC scenarios on the ship model.

2.1 Ship Editor

The screenshots below shows the ship designer. In the bar on the left side of the screen a user can specify the number of decks and the ship's dimensions. Each deck can be divided into a grid. Each grid cell wall can be absent, solid, or solid with a door. A user need only click on a wall to cycle between those three states. Any change made in this editor view is immediately propagated to the rest of the application, including the possibly already running FFDC simulation.



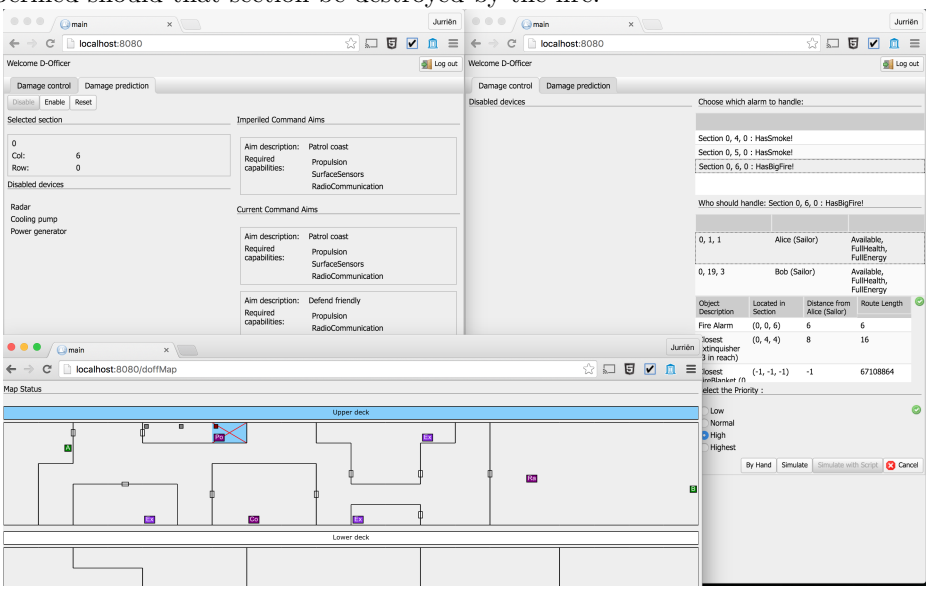
Items, such as fire extinguishers, can be added to each individual cell. These can be picked up by the people on board. Devices and cables between those devices can be defined and added to the ship as well. This enables the user to construct a network of devices. We can use this network specification to dynamically reason about the resource production and consumption of each device, and detect problems with resource requirements, should they arise.



2.2 FFDC Simulation

In simulation mode, the ship that was just designed in the ship editor can be used to simulation FFDC scenarios. The screenshot below shows an FFDC scenario from the perspective of the Damage Control Officer (DC-Off). Three windows are displayed. Typically, these would be spread out across more than one screen. The map in the lower-left corner provides the DC-Off with a live overview of the ship. Two crew members, Alice and Bob, are on board. They can be recognized

as the green squares with their initials in them. At the selected square (light blue) is a room in which two smoke alarms and a fire alarm went off. The fire is in the same room as the power generator (Po). It is the DC-Off's job to coordinate Alice or Bob to extinguish the fire using the limited resources on board. This can be done with the controls in the right-hand side window. To support the DC-Off in reasoning about the consequences of the fire in the power generator room, he can selectively "disable" selected parts of the ship. The window in the top-left corner will then show which *command aims* (the ship's current missions) will be imperilled should that section be destroyed by the fire.



3 Implementing a C2 System in iTasks

Task Oriented Programming (TOP) such as offered by iTasks offers an Embedded Domain Specific Language to support the development of distributed systems, such as C2 systems. Being embedded in the host language Clean, it offers all the advantages we know and love from pure functional languages. Hence, a large part of a TOP application consists of algorithms defined in the familiar way, by using *Algebraic Data Types* (ADT) and *pure functions* on these types. To make distributed running applications, we need more, however.

In order to communicate information between the different distributively working stake holders, TOP employs *Shared Data Sources* (SDS). An SDS is an ADT which value can be shared between the tasks executed by the stake holders. A sophisticated publish-subscribe system is attached to each SDS. A TOP run-time system knows which task is currently depending on which SDS. Whenever someone changes an SDS, all other parties interested in the change will be informed automatically. It is not guaranteed that everyone immediately sees every change, but eventually all parties are informed about the latest state

of affairs. As a consequence, the programmer does not have to worry about the communication and synchronization which is needed to inform each other. The style of programming is very declarative. In a way, it resembles a distributed spread-sheet.

In real-life all kinds of things might happen that influence the work that has to be done. In TOP, *tasks* play the central role. They define the work that has to be done, the tasks to do, and how these tasks depend on each other. A task is "just" a function returning a value of type `(Task a)`. Tasks are reactive: a task returns a value of some type `a`. But this value is not fixed: it may *change* over time. It reflects how work takes place in real-life: the status of the work is constantly changing while the work takes place. When the work is finished, it is fixed and will no longer change. Using *Task Combinators*, tasks can be composed into sub-tasks. Tasks can observe the progress of other tasks. Hence, the current value of a specific task might influence how other tasks proceed. There are only two core combinators for combining tasks: a sequential one (the `step` combinator), and the `parallel` combinator. In combination with the Shared Data Stores, any imaginable distributed collaboration can be expressed.

In TOP, tasks can be assigned to end-users. There are predefined basic tasks for end-users, called *editors*, that enable end-users to input or change data. One can have editors for any first order type. Using type-driven generic programming [2], we can, for any first-order type `a`, generate a *Graphical User Interface* automatically for displaying and updating values of that type. For instance, with an editor of type `Int` an end-user is presented with a form in which he can type in only an integer value. For a record type a form editor is generated. With the `step` combinator, one can further ensure that values entered are only accepted when certain predicates hold. Editors are very useful for rapid prototyping, because one gets interactive user-interfaces for free. In case the resulting GUI is inappropriate or requires tuning, it is possible to completely customize how values of a particular type have to be shown and can be edited by the user. This used in the C2 demo described above to draw a value of type `ShipMap` as a graphical interactive map.

Shared values and task definitions make it possible to define distributed running applications without a need to worry about technical realizations, communication, synchronization, and graphical user interfaces for doing interactions. The looks of the applications can be improved later on without a need to change other parts of the code.

In the next sections we will present part of the code of the C2 demo to illustrate that indeed complicated distributed applications can be defined in a declarative way, with a very good separation of concerns.

4 Implementation

This section will be implemented in the final paper.

5 Related Work

This section will be implemented in the final paper.

6 Conclusion and Future Work

This section will be implemented in the final paper.

References

1. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. In: Mathematical Structures in Computer Science. vol. 6, pp. 579–612 (1996)
2. Jansson, P., Jeuring, J.: PolyP — a polytypic programming language extension. In: Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 470–482. ACM Press (1997)
3. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: 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. pp. 195–206. ACM, Leuven, Belgium (Sep 2012)
4. Plasmeijer, R., van Eekelen, M.: Clean language report (version 2.1) (2002), http://clean.cs.ru.nl
5. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. pp. 24–52. Springer-Verlag, London, UK, UK (1995), http://dl.acm.org/citation.cfm?id=647698.734146