A graphical workflow editor for iTask

Jeroen Henrix

Thesis number
638

Supervisors
Peter Achten
Rinus Plasmeijer

Abstract

A workflow is a series of tasks that comprise a business process. In conventional information systems, the process structure is often implicitly encoded in the system. In contrast, workflow management systems (WFMSs) separate the process structure from the rest of the system. A WFMS is a software system which supports the enactment of business processes, based on a *workflow model*: a formalism which describes the workflow. Workflow models are traditionally expressed in graphical workflow definition languages.

iTask is a WFMS, developed at Radboud University Nijmegen. iTask uses a textual domain specific embedded language to express workflow models. The language is embedded in the functional programming language Clean. This approach enables to use all expressive power of the Clean language in workflow models. However, to understand an iTask workflow, one needs to be trained in functional programming.

This thesis presents a *Graphical iTask notation*, in order to make iTask modeling more accessible for a larger group of users. We demonstrate how common iTask modeling constructs can be mapped to a graphical notation. As a proof of concept, we implement a graphical editor, which is integrated into the iTask system.

Keywords: workflow modeling, domain specific language, functional programming, Clean

Contents

Chapter 1

Introduction

1.1 Workflow modeling

A business proces can be defined as "a collection of interrelated work tasks, initiated in response to an event, that achieves a specific result for the customer of the process."[43]. In this definition, we can see a number of key criteria. First, a business process produces a specific result, which can be identified as such. Thus, "Logistics" is not a process, but "deliver order" is. Second, the result should be useful to a customer - this may be either an internal or external person or entity. Third, all tasks that constitute a process should be related and all contribute to the end result.

Workflows involve the automation of business processes. The Workflow Management Coalition defines workflow as "The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules."[23]. Workflows are case-oriented: each task is enacted within the context of a particular case. A case represents a single execution of a workflow, in onder to produce a specifc result. Cases are domain-dependent. For instance, a case may represent a trip booking in a travel agency, an insurance claim in an insurance company, or an order in an online store.

During information systems development, the structure of the business processes is used as a basis for system design. In tailor-made systems, this causes parts of the process structure to become embedded - "hard coded" - in the implementation. However, business processes are subject to changes. Adapting an existing tailor-made system to reflect changes in business processes often takes a significant investment in time and money. In contrast, *workflow management systems* (WFMSs) separate the process structure from the rest of the system. A WFMS typically consists of two parts. First, a formalism to specify a *workflow model* and second, a *workflow engine*. The workflow model is usually expressed in a graphical modeling language. The workflow engine is a generic software component which takes the workflow model as input, and supports the end users in in the enactment of the modeled business processes.

Besides business process management, workflow modeling techniques are also being used in the area of scientific computations, for managing data sets on grid computing systems[3]. This thesis focuses on workflow modeling in business processes.

1.2 The iTask system

The iTask system, is a prototype WMFS, developed at Radboud University Nijmegen[33]. The core of iTask is a combinator library, written in the general-purpose functional programming language Clean. Combinator programming is a way to create an embedded domain specific language (EDSL) within a functional programming language. In iTask, workflow models are specified as Clean functions, using a set of monadic task combinators to create and compose tasks. These combinators express both control flow and data flow.

The advantage of the embedded DSL-approach is that the expressive power of the host language can be used. Thus, workflows modeled in iTask can use all the power of the Clean language like recursion, functional abstraction, lazy evaluation and a strong type system[35]. Further, the end-user can extend the functionality of iTask by defining new combinators or higher-order tasks - in contrast to many contemporary WFMSs, which cannot be extended with new workflow patterns by end users.

1.3 Problem statement

All the flexibility of iTask comes with a price: in order to understand iTask workflow models, one needs to be familiar with functional programming.

Workflow models are usually constructed by workflow engineers in collaboration with domain experts. Domain experts often do not have a background in (functional) programming. Having to learn functional programming first in order to understand iTask models would be a too steep learning curve.

Our assumption is that for non-programmers, like most domain experts, a graphical notation is easier to use than a textual notation. Furthermore, domain experts may already be familiar with contemporary graphical workflow notations. A graphical notation may ease the communication between workflow engineers and domain experts in the development process of iTask workflow models. Therefore, our goal is to investigate the feasibility of a graphical notation for iTask.

In order to substantiate this assumption, an empirical study which compares the comprehensibility of the graphical and textual iTask notation would be necessary. Such an empirical study is outside the scope of this thesis.

1.4 Research questions

Our main research question is

> In which way can workflow models in iTask be expressed in a graphical notation?

The intent is to make iTask workflow modeling more accessible for domain experts without programming experience. The following sub questions will guide our research:

1. Which contemporary graphical languages are used for modeling business processes and workflows?
2. How can common graphical workflow modeling notations be mapped to iTask expressions?
3. To what extent can concepts that are unique to iTask, be expressed in a graphical notation?
4. How should a graphical editor be constructed, to enable integration in the iTask system?

1.5 Thesis structure

This thesis is structed as follows. In chapter 2, we investigate principles of workflow modeling. Chapter 3 describes related work: among others, we analyze graphical notations for expressing workflows. Next, we focus on the relation between graphical languages and iTask in chapter 4. We use the *workflow patterns* to examine this relation in a structured way. Chapter 5 introduces our Graphical iTask Notation. The subsequent chapters discuss the implementation of a proof-of-concept editor. Chapter 6 deals with the architecture and front-end implementation. The compilation of graphical workflows is described in chapter 7, error handling in chapter 8. Chapter 9 presents conclusions and future work.

Chapter 2

Workflow modeling

Workflow models are formal descriptions of business processes. A workflow model consists in principle of a set of tasks, their ordering and their data dependencies. Workflow models are expressed in workflow definition languages (WDLs). Workflow management systems use a wide variety of WDLs. There is still few consensus about the way business process should be specified.

2.1 Representation paradigms

The *control flow* of a workflow model defines the ordering of tasks. WDLs can use different paradigms to express control flow relations. We distinguish *graph-oriented* and *block-oriented* WDLs. Note that these terms do not imply any particular visual representation; they merely indicate the underlying control flow structure.

Graph-oriented WDLs use directed graphs as their underlying structure, consisting of a set of nodes and a set of directed edges. Tasks are represented as nodes, and the control flow is expressed by the directed edges. In addition, control flow primitives may introduce additional types of nodes. Graph-oriented WDLs usually impose few restrictions on the graph structure; edges are generally allowed to connect arbitrary pairs of nodes. Examples of graph-oriented WDLs are Event-driven Process Chains (EPCs), UML Activity Diagrams and Yet Another Workflow Language (YAWL). These languages are discussed in the next section.

In contrast, block-oriented WDLs can express control flow constructs only by use of nested blocks. For each construct, a separate block is used. Tasks are modeled as elementary blocks, while control-flow is expressed by nestable, composite blocks. The workflow models expressed in block-oriented WDLs are also known as *structured workflow models*[18]. Notable examples of block-oriented WDLs are ADEPT[38] and process algebras like Pi calculus[45].

The graph-oriented and block-oriented paradigms are each others opposites concerning structuring: while an unconstrained graph-oriented WDL can express arbitrary control flows, a block-oriented WDL can only express structured control flows.

Intermediate forms of structuring are also possible. For instance, BPEL4WS[59] is mostly block-oriented, although it includes a <bpel:link> construct which allows jumps to other places out of the block structure. Another example is XPDL (XML Process Definition Language)[48]. In XPDL, models can be assigned a *graph conformance class*. Its *non-blocked* class allows arbitrary unstructured models. The *loop-blocked* class inhibits arbitrary cycles, while the *full-blocked* class enforces one-to-one correspondence between splits and joins of the same type, effectively creating a structured model.

Within the workflow community, it has been debated whether structured models should be preferred over unstructured ones. Unstructured models provide more freedom to the modeler. On

the other hand, structured workflow models have attractive properties like the fact that they are guaranteed deadlock-free.

The structuring of workflows shows parallels with imperative programming. Unstructured workflow models resemble unstructured programming using goto-statements. Structured workflow models only allow control flow to have single entry points and single exit points, like structured imperative programming composed of nestable block-statements. Since the emergence of structured programming initiated by Dijkstra[6] writing unstructured programs is generally considered as bad practice, since it may lead to hard to understand "spaghetti code". Holl and Valentin[14] applied this argument to workflow modeling and advocates that structured workflows models are easier to understand. Gruhn and Laue[12] follow this approach, but discusses reasons to deviate from structuring.

Kopp et al.[21] argue that the choice for a particular WDL should depend on its intended use, and introduce a spectrum between intended for documentation and intended for execution. If workflow models are merely used for documentation purposes, one may use a unstructured, more flexible WDL; if the models should be executed by a WFMS one is better off by using a more structured WDL. Graph-oriented WDLs are placed more towards the documentation side; block-oriented WDLs towards the execution side of this spectrum.

2.2 Workflow patterns

Given the variety of WDLs, the question arises whether differences exist in expressive power between these languages. One could compare WDLs pair-wise and try to map concepts expressed in one notation to another and vice versa. However, if this approach was used to evaluate a larger set of languages, the number of comparisons one has to make would grow quadratic, which would soon be impractical.

A reference framework which describes concepts commonly found in process modeling notations would be more useful. Such a framework is not a WDL by itself, but merely a collection of modeling concepts. The framework could then be used to evaluate the expressive power of a particular notation. This kind of approach is used by van der Aalst, ter Hofstede et al. starting the *workflow patterns initiative*. They originally defined a set of *20* control-flow *workflow patterns* [54].

An individual pattern captures a business requirement. The workflow patterns have been used to evaluate the expressive power of particular notations. The patterns can also be used for other purposes, such as selecting WFMS tools to be used in particular domain. One could analyse the necessary workflow patterns for the particular domain, then select the tools based on their support for necessary patterns.

In the context of workflow modeling, the meaning of the term "expressive power" should not be restricted to its formal definition. Since many process modeling languages are Turing-complete, any process model could *in principle* be expressed in any Turing-complete modeling language. If a pattern is not supported in the particular language, one may resort to explicitly model all possible behaviors of the pattern. This does not imply that such a construct is straightforward, or easy to understand: that particular language would not be *suitable* to express the pattern. Therefore, a comparison of only formal expressiveness would not be useful in practice. Instead, workflow patterns are not a set of formal primitives, but are created based on occurrence in practical situations. In [17] the relation between expressive power and suitability is investigated.

Nowadays, the workflow patterns can be regarded as a standard in workflow modeling. Many contemporary WFMSs are inspired by the workflow patterns. The YAWL workflow language[51] was the first language whose design is based on the workflow patterns. Later, commercial vendors of WFMSs started to advertise their products as supporting "all major workflow patterns".

An individual pattern consists of a name, a description, one or more examples of situations where the pattern is used and implementation remarks how the pattern is implemented in existing modeling notations.

The workflow patterns are categorized in different *perspectives* to structurally describe different aspects of modeling. The *control* flow *perspective* (also known as the *process perspective*) focuses on the order in which tasks are executed. The *data flow* (or *information*) perspective describes the data involved in the execution of the tasks. The *resource perspective* (or organizational perspective) involves the allocation of tasks to resources within an organization.

2.3 Control flow perspective

The control-flow perspective focuses on the execution order of tasks. Tasks can be arranged in various ways: they may have to be executed sequentially, they can be executed in parallel, or the execution may be subject to a particular condition.

This section gives an overview of the original set of control flow patterns published by van der Aalst et al. [54]. This set of patterns has been used to structurally evaluate a wide range of WDLs like workflow nets[53], EPCs[26], UML Activity diagrams[58, 41] and product-specific notations[53, 60].

Since the publication of the original workflow patterns in 2002, van der Aalst et al. identified a set of 23 additional control flow patterns and evaluated a range of contemporary WDLs for support of these patterns[52]. The support for these new patterns in WDLs is still limited, with the exception of the graphical Business Process Modeling Notation (BPMN) and the textual XML Process Definition Language (XPDL). NewYAWL[40] is an extension of YAWL, of which the design is entirely based on these new patterns.

Our primary reason for using the workflow patterns is to identify common graphical notations in WDLs. Discussing control flow patterns which are only supported by few or even no WDLs would not contribute to this goal. Therefore, we decided to focus primarily on the widely supported original control flow patterns.

2.3.1 Basic control flow and synchronization patterns

Sequential routing The *sequence pattern* executes tasks one-after-another: when task A completes, tasks B starts.

Parallel routing Parallel routing is expressed by the *parallel split* and *synchronization* patterns: The workflow is split in multiple threads which can be executed concurrently. A parallel split is also known as an *and-split*. Multiple threads can be merged into a single thread of control, which will start executing as soon as all parallel threads have completed. This is known as an *and-join*. Alternative join conditions are described as separate patterns. The *multiple merge* pattern will instantiate the activity after the merge for each parallel thread again. The *discriminator* pattern waits until the first thread completes and ignores the completion of the other threads.

Conditional routing Conditional routing can take two forms, whether multiple choices are allowed or not.

1. In the *exclusive choice* and *simple merge* paterns, the decision of a single branch will exclude the execution of the others. Because no branches can be executed in parallel, there is no concurrency involved, so merging multiple split branches is simple: continue with the only executed branch. Splitting and merging multiple mutual-exclusive branches are known as respectively *xor-split* and *xor-join*.

2. The *multi-choice* and (*synchronizing merge* patterns allow the execution of one or more branches, which are all executed in parallel. Typically, a condition is placed on each branch, and the branch is executed if the condition evaluates to true. This kind of branching is also known as *or-split*. Similar to the synchronization in parallel routing, the branches that are

executed can be synchronized and be merged into a single thread, which is known as an *or-join*. In many WDLs, the or-split and or-join are separate operations. The or-join operation needs to know which threads were executed, in order to wait only for them to complete. However, this information is available only in the earlier or-split. Because of this information need, the or-join is said to have non-local semantics.

2.3.2 Structural patterns

Iteration Workflows may contain repetitive elements. A task or set of tasks may be repeated until a particular condition is met. Iteration can be expressed in several ways. Arbitrary cycles (*arbitrary cycles* pattern) are loops that have multiple entry points and/or multiple exit points. These unstructured loops can only be expressed in graph-oriented WDLs. Block-oriented WDLs do not support unstructured loops, since multiple entry points or exit points cannot be expressed in terms of a nested block structure. Structured loops are loops with a single entry point and a single exit point. These constructs repeat a task or set of tasks until a condition is met. Depending on the type of loop, the condition is evaluated before or after the execution of a single iteration.

Implicit termination Implicit termination is a property of a WFL which states that subprocesses will terminate if they have no work to do.

2.3.3 Multiple instances

Within a workflow, an individual task may have to be executed multiple times, each time with different data. Such a task is said to have multiple instances. For example, a conference paper may have to be reviewed by several reviewers. There is a single task definition"review paper", which is instantiated for each review performed by a particular reviewer. The number of instances may be determined in different ways: it may be constant (e.g. each paper is reviewed by 2 reviewers), determined at run-time (e.g. depending on the size of the conference), or also change at run-time (e.g. if the reviewers disagree, then additional reviewers will be involved). A workflow model may support the creation of multiple instances of a task and, after execution, the synchronization of their results.

The creation of multiple instances is described by the *multiple instances without synchronization* pattern. The synchronization is described by multiple patterns. The later the number of instances is determined, the more difficult it is to know if the synchronization may complete. Van der Aalst et al. define different synchronization patterns, depending on whether the number of instances is constant (*multiple instances with a priori design time knowledge* pattern), determined at run time before execution of the instances (*multiple instances with a priori run-time knowledge* pattern), or also changeable during execution (*multiple instances without a priori run-time knowledge* pattern).

2.3.4 State based patterns

In the *exclusive choice* pattern discussed earlier, the choice of a branch is based on an explicit condition. In the *deferred choice* pattern the chosen branch is not conditional. Instead, multiple branches are offered to the environment. As soon as one of the branches starts, the other branches are cancelled. The *interleaved parallel routing* pattern describes the execution of an unordered set of tasks, with the constraint that no two tasks can be executed at the same time. So, all tasks executions are interleaved. This pattern is used if multiple tasks need mutual exclusive access to a resource. The *milestone* pattern defines that a task is only available for execution after reaching a particular milestone, but before that milestone has expired.

2.3.5 Cancellation patterns

The *cancel activity* pattern defines the removal of a single activity (task) waiting for execution. The *cancel case* pattern describes the removal of an entire case (process instance) including all its running tasks.

2.4 Data flow perspective

A structured approach to investigate the data flow perspective has been performed by Russell et al. They identified a set of *data patterns*: recurring situations regarding data processing in workflows[39]. The data patterns cover four areas: data visibility, data interaction, data transfer and data-based routing.

2.4.1 Data visibility

Data visibility refers to the availability of data during the enactment of a workflow. Within the context of data patterns, data is stored in named variables. The data visibility is expressed in the scope of these variables.

Variables whose scope is limited to a single case, would be called *case attributes*. The scope can be further limited to either a single execution of a task (*task data* pattern), a composite block of tasks (*block data* pattern), a custom set of tasks (*scope data* pattern), multiple instances of the same task (*multiple instance data* pattern), or the entire case (*case data* pattern).

Alternatively, variables can be available to an entire workflow and shared to all tasks (*workflow data* pattern) or can imported from the external environment (*environment data* pattern).

2.4.2 Data interaction

Data interaction between tasks Within workflow processes, data created by tasks is often necessary for executing other tasks. Passing data from one task to another is an essential feature in WFMSs and described by the *Data interaction — task to task* pattern. The passing of data may coincide with the control flow, but it need not be: data-flow and control flow can be different.

Three approaches are used to describe data flows in graphical workflow models:

1. *Implicit data passing.* If both tasks have access to shared variables, they can pass data without describing the data flow explicitly. This requires tasks to know each others variable naming. The advantage of this approach is that the visual notation remains clean, since only the control flow is drawn explicitly. The drawback is that concurrency issues may arise: variables may be read and written by multiple concurrent tasks.

2. *Integration of data in the control flow.* Tasks can only pass data to each other via existing control flow relations. The data is passed at the moment the control is transferred. Although this approach reduces concurrency issues, one may be forced to route data via tasks that do not need the data, only because the data is needed later in the control flow.

3. *Explicit modeling of separate data flows.* If both the control flow and the data flow are modeled explicitly, data flow can be routed separately from control flow. Using this approach has several consequences. First, the visual notation becomes more cluttered, since both data flows and control flows need to be drawn. Second, the evaluation order is now influenced by both the control flow and the data flow. If an eager evaluation order is used at the task level, a task can only start when all of its incoming data is available.

Sub-workflows Workflow models should support composition: a workflow may be composed of several sub-workflows, each of them specify a specific part of a workflow model. In order to make composition possible, a sub-workflow should avoid to make assumptions about the environment in which the sub-workflow is used. In particular, a sub-workflow should not make assumptions of variable names from its environment.

The data patterns *block Task to sub-workflow decomposition* and *sub-workflow decomposition to block task* define three approaches for data-passing with sub-workflows. If data is passed implicitly, the sub-workflow and its callers share their address space, which can hinder the re-use of sub-workflows. Alternatively, data passing can be made explicit by means of parameters or data channels.

Multiple instances In a multiple-instance task, each of the individual instances may need individual data. The data patterns *data interaction — to multiple instance task* and *data interaction — from multiple instance task* discuss three approaches. Either each instance gets specific data, passed by reference or by value, or all instances have access to shared variables. In the latter case, the individual instances have to deal with concurrency.

Case, workflow and environment data The *case data* pattern describes to data which is available for all tasks which are part of a case. The *workflow data* pattern refers to data which is shared between cases. For instance, in an order processing workflow a single case refers to the handling of a single order. The total number of processed orders is data which is shared between cases. A set of *external data interaction* patterns describe data interaction between a WFMS and its operating environment. This requires a structured communication facility, such as remote procedure calls (RPC).

2.4.3 Data transfer

Data transfer patterns *data transfer patterns* describe data transfer styles used in workflow modeling languages. These patterns are conceptually equivalent to parameter passing mechanisms found in imperative programming languages: call-by-value (*data transfer by Value — incoming* pattern, *data transfer by value — outgoing* pattern and *data transfer by value — outgoing* pattern), call-by-copy-restore (*data transfer — copy in/ copy out* pattern), and call-by-reference (*data transfer by reference — unlocked* pattern and *data transfer by reference — with lock* pattern, respectively without and with synchronization).

Data transformation patterns The application of functions on passed data is captured by *data transformation patterns*. Functions can be applied to a task's input parameters (*data transformation — input* pattern) or its output parameters (*data transformation — output* pattern).

2.4.4 Data-based routing

Four patterns are defined for the specification of pre- and postconditions on tasks, based on either the existence or the value of task data: *task precondition — data existence*, *task precondition — data value*, *task postcondition — data existence*, *task postcondition — data value*. These patterns are supported by a WDL if action can be taken if the condition does not hold, or — in case of the existence patterns — the either the WDL ensures the data is always present.

Triggers allow action to be taken if an external event takes place (*event-based task trigger*) pattern) or an condition based on workflow data evaluates to true (*data-based task trigger*).

The *data-based Routing* pattern allow decisions in the control flow to be taken based on data values.

Chapter 3

Related work

Our goal is to make iTask modeling more accessible for non-technical users, by designing a graphical notation for iTask. This choice is based on the assumption that for non-technical users, a graphical notation is easier to comprehend than a textual one. We do not have evidence to support this assumption. Therefore, we first study available literature to either support or refute this assumption.

The comprehensibility of a notation may also depend on the experience a user has with similar notations. For this reason, we investigate notational conventions commonly found in contemporary WDLs.

Since the iTask WDL is embedded in the functional language Clean, it is possible — and even common practice — to use typical functional programming features in iTask models, such as recursion and higher-order functions. These features are commonly not available in graphical workflow languages. Graphical notations are also being used in the area of functional programming. These notations may provide us insight in how functional programming concepts can be expressed graphically. However, we focus primarily on graphical WDLs, since our graphical notation is intended for workflow modeling, and not for functional programming in general.

3.1 Graphical versus textual modeling

Our assumption is that for non-programmers, a graphical WDL is easier to comprehend than a textual notation. This may sound plausible, but we have no evidence to support this claim: there are no empirical studies which compare the comprehensibility of textual and graphical WDLs. There exists empirical evidence on the comparison of visual and textual programming languages, but there is no unambiguous conclusion. A survey by Whitley[57] lists studies for and against visual notations. An experiment by Green and Petre [10] shows that for experienced users, graphics were slower than text, a result later confirmed by Moher et al.[28]. Based on a series of experiments, Petre concludes that "overall, graphics was significantly slower than text" [30], but notices differences in strategy between novices and expert users. An experiment by Kiper et al.[20] distinguishes between programmers and non-programmers with a technical background. In the experiment, both groups read graphical and textual decision diagrams. They conclude that "graphics may be better for technical, non-programmers than they are for programmers because of the great amount of experience that programmers have with textual notations in programming languages".

3.2 Graphical workflow modeling languages

In this section, we give an overview of the notations used in a number of graphical WDLs, namely Workflow nets, YAWL, Event-driven Process Chains and UML Activity Diagrams. We chose to

analyse these WDL for several reasons. First, the chosen WDLs are well studied in the literature. There are workflow pattern analysis available on each of these languages, which enables us to compare these languages in a structured way. Second, these WDLs are standards and not tied to a particular WFMS. YAWL is an exception, but is specifically added because of its extensive support for workflow patterns.

3.2.1 Workflow nets

Standard Petri nets allow to express the basic control flow of workflow models and have a formal foundation. Petri nets have a static structure which cannot change during execution. However, the execution state of a workflow, by means of tokens can be shown in the same kind of diagram. Workflow concepts can be mapped straight-forward onto Petri nets: a transition indicates a *task*, a place is a *condition* and a token indicates a *case*.

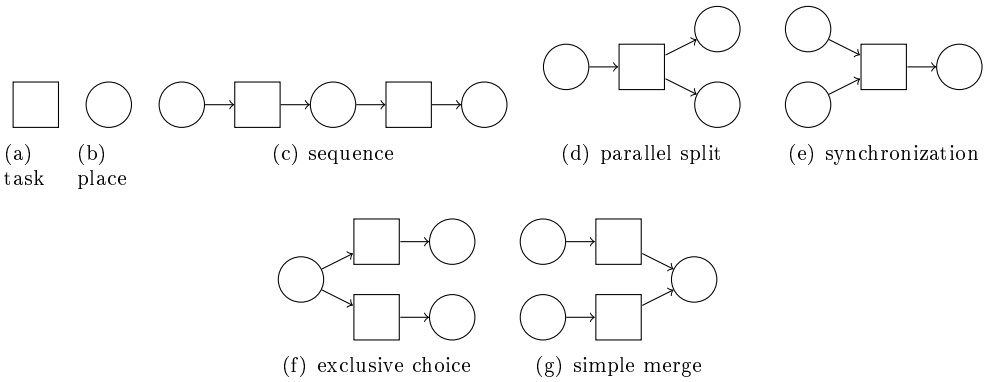

Figure 3.1: basic control flow expressed in Petri nets

Sequential routing Sequential execution is denoted as a concatenation of transitions, with places between them (fig. 3.1(c)).

Parallel routing Parallel routing is modeled by an AND-split: a transition with a single incoming arc and multiple outgoing arcs (fig. 3.1(d)). Although the AND-split is modeled as a transition, it does not represent work in the real world, but is merely added for control purposes. If there is a token in the input place, the AND-split transition can fire: the token is removed from the input place, and tokens in the output places are created. The tasks (transitions) connected to these places can be executed in parallel. Synchronization is modeled by an AND-join: a transition with multiple incoming arcs, and a single outgoing arc (fig. 3.1(e)). Such a transition enforces synchronization, because the transition can only fire if tokens are present in all input places.

Conditional routing Conditional routing is modeled by an OR-split: a place with a single incoming arc and multiple outgoing arcs (fig. 3.1(f)). Each arc may have a condition attached, which depends on the case attributes. If a transition attached to the one of the outgoing arcs fires, the token is removed from the place, preventing the transitions attached to the other arcs from firing — effectively creating a XOR. Merging exclusive branches is modeled by an OR-join: a place with multiple incoming arcs and one outgoing arc (fig. 3.1(g)).

Iteration Iteration is modeled by cycles in the Petri net model. Arbitrary cycles are supported, these are loops with one or more entry points, and one or more exit points.

Workflow nets[46, 55] are based on Petri nets. Workflow nets must have a single entry point, i.e. a source place with no incoming transitions. They have a single exit point, i.e. a sink place with

no outgoing transitions. If the sink-place is connected to the source place, the resulting net would be strongly connected.

In the standard petri net notation, it is not clear which tasks represent real work, and which tasks are merely used for routing. Therefore, workflow nets employs a shorthand notation for parallel and conditional routing, shown in figure 3.2(a)...(d).

Further, workflow nets add three extensions to Petri nets: a *color extension* to distinguish different tokens (cases), a *time extension* to model temporal behavior and a *hierarchy extension*. The hierarchy extension allows to make abstractions by means of subprocesses. A *subprocess* is represented by a single symbol (figure 3.2(e)) that represents a separate workflow net defined elsewhere.

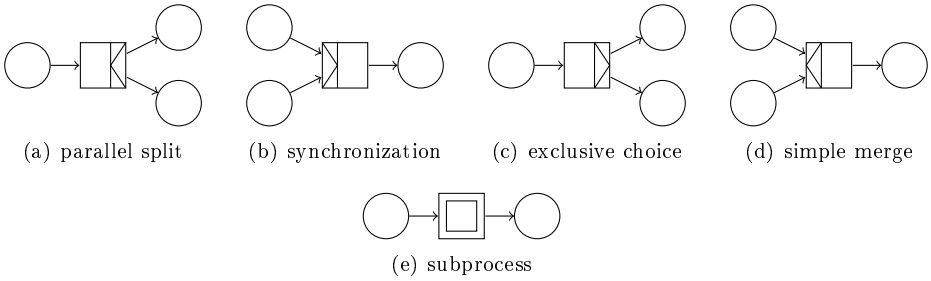

Figure 3.2: Workflow net extensions to petri-nets

3.2.2 YAWL: Yet Another Workflow Language

Many workflow management systems have a limited support for workflow paterns. *YAWL*, which stands for *Yet Another Workflow Language*, is a WDL explicitly designed with support for control flow workflow patterns[51]. It does *not* explicitly address data flow.

Because many workflow patterns can be expressed in high-level petri-nets, YAWL is based on high-level petri-nets. However, high-level petri nets cannot properly express situations in which a case involves multiple instances, synchronization of optional tasks and cancellation of running tasks. We will now give an overview of the YAWL notation. For the complete semantics, we refer to [51].

A YAWL workflow specification consists of an *Extended Workflow Net* (*EWF-Net*). Each EWF-net has exactly one *entry condition* where the net starts and one *output condition* where it ends. Further, EWF-nets may use conditions in other places, which act similar to *places* in workflow nets.

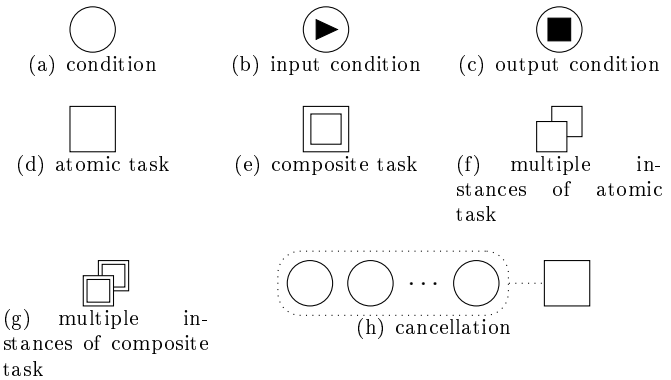

Figure 3.3: Extended Workflow Net notation used in YAWL

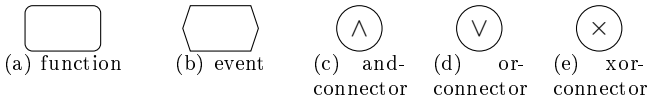

Figure 3.4: EPC notation

Figure 3.5: EPC sequential routing

EWF-nets consist of tasks, conditions and connector symbols. A simple, *atomic* task is indicated by a box symbol. EWF-nets can be nested; a double-box symbol indicates a composite task, which is a separate EWF-net lower in the hierarchy. *Sequence* is modeled by arcs between tasks; in contradiction to petri nets there is no need to model places in between. However, it is allowed, which will act as conditions. Both atomic and composite tasks can have multiple instances, which is indicated by overlapping boxes. A number of parameters can be specified. The number of instances can be bounded. Further, a threshold can be specified, which causes a multiple-instance task to complete if the number of completed instances exceeds the threshold keyword *dynamic* is specified, the number of instances may be increased during execution. These parameters are denoted as [*min*, *max*, *threshold*, *static*|*dynamic*].

Parallelism and decisions are taken directly from the workflow nets. This includes their connector symbols (AND-split, XOR-split, OR-split, AND-join, XOR-join, OR-join). A new construct is introduced for modeling cancellation. If the task attached to the dashed line is executed, all tasks in the are dashed rounded rectangle are cancelled.

Timing and events are not explicitly part of the language, but modeled as separate task types. Timers (tasks that wait for a particular time before completing) are modeled as tasks marked with a "T", and events (which wait for an external event) are modeled as tasks marked with an "E".

3.2.3 Event-driven process chains

The event-driven process chain (EPC)[16] is a language to model business processes. Initially, EPCs were used to *document* business processes. Since this does not require a strictly formal approach, no formal semantics for EPCs exist. Formalizations of EPCs have been researched, by means of a mathematical foundation[19] or mapping EPCs directly onto Petri nets[47]. Today, the use of EPCs is widespread in workflow modeling and in Enterprise Resource Planning (ERP). EPCs focus primarily on control flow.

An event-driven process chain diagram uses a graph-based notation, which is shown in figure 3.4. A *function* is a single task or activity, and is indicated by a round rectangle. An *event* is the activation of a particular state. Events can be regarded as the pre- and postconditions of functions: before a function can be executed, all its preceding events have to be activated. After execution of a function, all it following events will be activated. Directed edges are used to indicate the relationship between functions and events. Functions and events must have an indegree and outdegree of exactly 1. Further, functions cannot be connected directly to other functions, but always have events between them. To enable more complex routing, *logical connectors* are used. These connectors are denoted as small circles. Three kinds of connectors are used: AND, XOR, and OR. Connectors can be used as split connectors (outdegree > 1), join connectors (indegree > 1) or both.

Sequential routing Sequencing is expressed by an successive series of events and functions, as shown in figure 3.5

Parallel routing Parallel routing is expressed using the AND-connector, which is both used for splitting the control flow in multiple parallel branches, as well as synchronizing the parallel

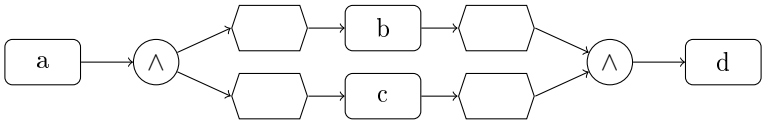

Figure 3.6: Example of parallel routing in EPC

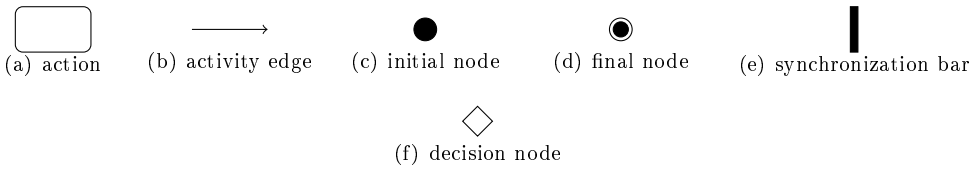

Figure 3.7: UML activity diagram notation

branches into a single branch. Figure 3.6 shows an example of two functions b and c executed in parallel.

Conditional routing An exclusive choice between two branches is expressed by the XOR-connector. A XOR-connector with multiple outgoing branches is a XOR-split. As soon as the preceding function is completed, one of the succeeding events is activated. Merging multiple exclusive branches is done by a XOR-join, which will cause the succeeding function to be executed as soon as one of its preceding events is activated.

Non-exclusive choice is expressed by the OR-connector. An OR-connector with multiple outgoing branches is an OR-split. When the preceding function completes, one *or more* succeeding events can be activated. Merging multiple exclusive branches is done by an OR-join, which will cause the succeeding function to be executed as soon as *all events of activated branches after the preceding OR* split have completed. Since this requires knowledge of the earlier OR-split, the semantics of the OR-join are non-local. Moreover, it is not required for OR-joins to be preceded by OR-splits, which leads to confusion about the formal semantics[49, 19].

Iteration The EPC is a graph-oriented WDL, which allows to express cycles in the graph structure without imposing additional constraints. Therefore, arbitrary loops with multiple entry points and multiple exit points can be expressed.

Standard EPCs lack support for composite task and multiple instances. In [26], Mendling et al. introduce an extension of EPCs called *yEPCs*. This extension added support for hierarchical functions, multiple instances and cancellation of functions, using the YAWL notation.

3.2.4 UML Activity diagrams

The Unified Modeling Language (UML)[11] is a general-purpose modeling language primarily used in object-oriented software engineering. The UML is a visual language, defined by the Object Management Group (OMG) and has seen a number of revisions. As of august, 2010 the actual version is version 2.3. A number of diagram types are used to model various aspects of object-oriented software. *Activity diagrams* are being used to describe control-flow aspects of workflows[7, 8]. The semantics of UML 2.x activity diagrams is based on token flow, which is conceptually similar to Petri nets.

In UML activity diagrams, the tasks that comprise a workflow are called *activities*. These are indicated by rounded-edged rectangles. Each workflow starts with an *initial node* and ends with a final node.

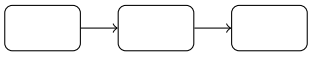

Figure 3.8: Example of sequence in UML activity diagram

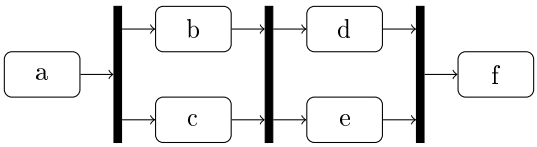

Figure 3.9: Example of parallel routing in UML activity diagram

Sequential routing In contrast to Workflow nets, UML activity diagrams do not explicit model states or events. Sequence can be expressed by merely an successive series of actions, connected by *activity edges*.

Parallel routing Parallellism and synchronization are modeled by thick bars called *synchronization bars*. Parallellism is indicated by a *fork node*: a synchronization bar with multiple outgoing edges. This is comparable to the *and-split* in workflow nets. Synchronization is indicated by a *join node*, which is a synchronization bar with multiple incoming edges. The join node is comparable to the *and-join* in workflow nets. Fork and join nodes can be combined, which is denoted as a synchronization bar with multiple incoming and multiple outgoing edges. Figure 3.9 shows an example involving parallel routing. After execution of activity a, activity b and c are run in parallel. After they both completed, activities d and e are run in parallel. After completion of d and e, activity f starts.

Conditional routing Conditional routing is modeled by *decision nodes*, which have two or more outgoing edges. Edges may be annotated by *guards*, which are boolean expressions between square brackets in the Object Constraint Language (OCL)[29]. The nodes attached to an edge may only be executed if the guard expression is true. After a decision node, only a single outgoing branch is selected, which is comparable to the *xor-split* in workflow nets. Merging multiple alternate edges is done by *merge nodes*, which have multiple incoming edges. Decision and merge nodes are denoted by diamond symbols. Analog to synchronization bars, they can be combined to a single decision/merge node. Figure 3.10 shows an example of conditional routing.

Iteration The UML activity diagram is a graph-oriented language. Arbitrary cycles can be expressed; a conditional loop can be expressed by a cycle and an XOR-split.

3.3 Graphical notations in functional languages

3.3.1 Vital

Vital[13] is a document-centered environment for Haskell. The goal of Vital is making a functional language accessible to broad range of end users. Vital presents Haskell modules as documents,

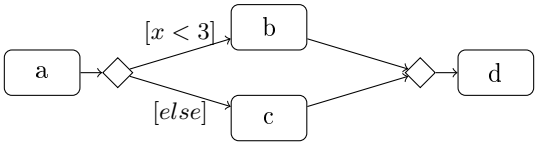

Figure 3.10: Example of conditional routing in UML activity diagram

which are displayed in a textual or graphical editor. By interacting with these editors, the underlying expressions in the Haskell program can be modified.

Our research shows similarities to Vital, in the sense that our research also intends to introduce a new, graphical notation for existing expressions in a functional language. However, Vital is intended as a general approach for visualizing functional data structures, while our research concentrates on visualizing expressions within a specific domain, namely workflow modeling.

3.3.2 Nets in Motion

Nets in Motion (NiMo)[4] is a visual environment for programming in data-flow style. it is a language in itself and is not based on an existing textual notation. However, NiMo incorporates typical functional language concepts like higher-order functions, partial application and laziness. Although the NiMo language may provide us insight how functional language constructs like higher-order functions can be expressed graphically, the use is limited for our goals. The workflow paradigm heavily relies on control flow, while NiMo is entirely dataflow based. However, NiMo may be suitable to visualize complete (non-workflow related) Clean expressions. An investigation of these possibilities is outside the scope of this thesis.

3.4 Summary

There is no unambigous evidence whether graphical workflow modeling is easier to comprehend than textual workflow modeling. However, having experience with a particular notation may make similar notations easier to comprehend. Hence, we investigated graphical WDLs. The WDLs which we analysed all used similar notations. Tasks are generally described by rectangles and control flow is modeled by directed edges. Branching and merging is indicated by connectors. In most cases, different split and merge connectors are used. A notable difference are whether workflow states are implicit (in UML) or explicit (in workflow nets, YAWL, EPCs).

Chapter 4

iTask

4.1 Introduction

iTask is a workflow management system, written in the general-purpose functional programming language Clean. In contrast to many conventional workflow systems, iTask uses a *domain-specific embedded language (DSEL)* approach: the formalism used to express workflow models is a combinator library. The advantage of this DSEL approach is that the expressive power of the host language can be used. Thus, workflows modeled in the iTask language can use all the power of the Clean language like recursion, higher order functions, lazy evaluation and a strong type system[35]. Further, the functionality of iTask can be extended by defining new combinators — in contrast to many contemporary WFMSs, in which user-defined constructs are not supported.

4.2 Architecture

iTask workflow models are specified at a very high level of abstraction. From this high-level model, the iTask WFMS is able to generate a user interface and a structured data storage.

An iTask workflow model is specified in a Clean implementation module (an `.icl` file). The workflow model and the iTask base library are compiled to an executable, which contains the iTask server application (figure 4.1).

During workflow enactment, iTask uses a multi-user, client/server web architecture. The compiled iTask server application is launched either from a standard web server via CGI, or can run standalone via its built-in web server. The server uses a data store on disk to save the current workflow state.

End users use a standard web browser to access the WFMS. Upon accessing the web server, the end-user's web browser downloads a Javascript client application and executes it locally. The client application is based on the ExtJS library[42] and communicates with the web server using JSON data over HTTP, as shown in figure 4.2. A discussion of the iTask implementation can be found in [25].

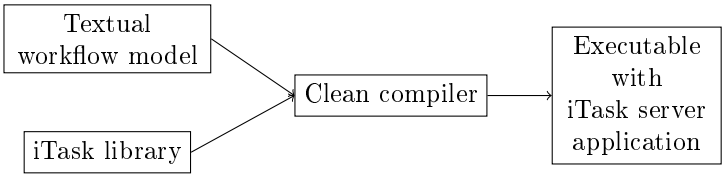

Figure 4.1: Compilation of an iTask workflow

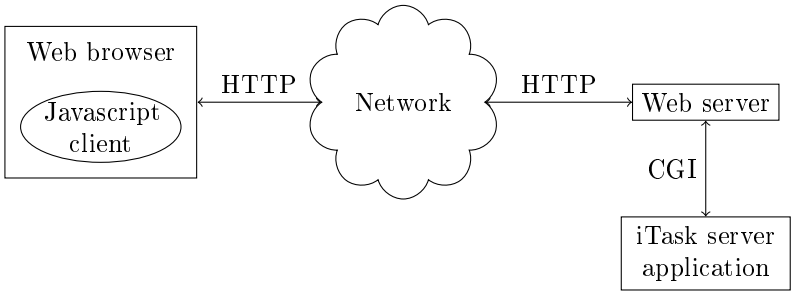

Figure 4.2: iTask runtime environment

4.3 The iTask workflow definition language

iTask has a strong integration of control flow and data flow. In the iTask WDL, the basic control flow "building block" is a *task*. However, a task also determines data flow: each task yields a result upon completion. iTask workflows are strongly typed. Each task has a type `Task a | iTask a`, in which `a` is the type of the value the task yields upon completion. For instance, a task that return a boolean has the type `Task Bool`. Even a task which returns no useful value has a type, namely `Task Void`, where `Void` is defined as a nullary constructor.

The type class restriction `iTask a` ensures that instances of generic functions are available for generation of user interfaces and serialization of type `a`. These functions can be derived automatically by the Clean compiler. To use a type t in a workflow specification, an application programmer only has to declare **derive class** `iTask` t. A discussion of the generic implementation can be found in [25].

The `Task` data type is a monad. Application programmers do not deal with the task states directly, but only use its monadic interface.

```
return :: a → Task a | iTask a
(>>=) infixl 1 :: !(Task a) !(a → Task b) → Task b | iTask a & iTask b
```

Besides the monadic unit and bind operators, the iTask library contains a set of additional *task combinators*, which allow the user to create complex workflows by combining individual tasks. We discuss these combinators in the next sections.

Representation paradigm In section 2.1, we analyzed the difference in control flow between the graph-oriented and block-oriented representation paradigm. The iTask WDL is primarily a block-oriented WDL. The only way to combine basic tasks to make complex tasks, is to pass the basic tasks as arguments to task combinators. The obtained expression can be used as an argument for another combinator, and so on. This results in a well-nested and non-overlapping composition. However, it is possible to deviate from this block structuring by means of *process combinators*, a class of task combinators. A process is the iTask term for a separate execution of a task specification; in other workflow terminology this would be called a *case*. Using the process combinator `spawnProcess`, a task can be started as a separate process, which runs separately from the original process. Other processes can obtain a tasks' result using the `waitForProcess` combinator. Using these two combinators, it is possible to make arbitrary, non-structured jumps between tasks. However, this style of modeling is advised against, because it may lead to deadlocks and makes the model more difficult to comprehend.

4.4 Workflow pattern analysis

The previous chapter presented an overview of the workflow patterns: a set of patterns describing frequently occurring situations in workflow models. Now, we shall discuss the current iTask WDL within the context of the workflow patterns. There are several reasons to do so.

First, the workflow patterns may serve as a benchmark to assess the expressive power of iTask, in comparison to existing tools. In [34], Plasmeijer et al. state that iTask "covers all known work flow patterns that are found in contemporary commercial work flows tools, and is thus suited to describe real-world applications.". However, it does not explicitly identify the support of all individual workflow patterns in iTask. By discussing iTask on a per-pattern basis, we can substantiate this coverage claim, or possibly uncover workflow patterns which are not supported by iTask.

Second, we use the workflow patterns as a framework to develop a new graphical notation for iTask. The new notation should at least be able to express common workflow patterns.

4.4.1 Basic control flow and synchronization patterns

Workflow pattern 1: sequence Sequencing in iTask is expressed by the `>>|` operator. Given that `a` and `b` are tasks, the expression `a >>| b` yields a new composed task. After task `a` completes, task `b` is started. The result of task a is discarded.

```
(>>|) infixl 1 :: !(Task a) (Task b) → Task b | iTask a & iTask b
```

In order to maintain the result of task `a`, the monadic bind combinator `>>=` should be used. It is similar to the `>>|` combinator, but expects a *function* as its second argument, which receives the result of task `a` and produces a task of type b. The function may yield *any* `Task b`, possibly dependent on the result of task `a`. This allows for dynamic construction of workflows at runtime.

```
(>>=) infixl 1 :: !(Task a) !(a → Task b) → Task b | iTask a & iTask b
```

While the binary `>>|` combinator only combines two tasks, a related combinator named `sequence` operates on lists of tasks. The sequence operator executes all tasks in the list sequentially, yielding a list of results. Its definition is:

```
sequence :: !String ![Task a] → Task [a]
```

Workflow patterns 2: parallel split and 3: synchronization In contradiction to many graphical workflow languages, iTask expresses parallel split and synchronization in a single combinator.

Parallelism is expressed by the AND-combinator, denoted as `-&&-`. This operator expresses both the splitting the flow into two concurrent branches, and synchronizing the results.

```
(-&&-) infixr 4 :: !(Task a) !(Task b) → Task (a,b) | iTask a & iTask b
```

Given that `a` and `b` are tasks, the expression `a -&&- b` will yield a composed task, in which `a` and `b` can be executed simultaneously. The results of both task `a` and `b` are kept, and the composed task does not complete before both tasks are finished.

While the AND-combinator allows the execution of two tasks, there is also a version which accepts any number of tasks. The `allTasks` combinator takes a list of tasks as input, and executes all tasks in parallel. The `allTasks` combinator is defined as:

```
allTasks :: ![Task a] → Task [a] | iTask a
```

Workflow patterns 4: exclusive choice and 5: simple merge An exclusive choice (XOR) based on a condition can be expressed straightforward, even without special support in the iTask library. Because the iTask WDL uses Clean as a host language, **if** and **case**-expressions, and functions with multiple alternatives can be used to choose between tasks.

For instance, to execute a task named `askPermission` only if a variable `price` is greater than 1000, and otherwise execute task `purchaseItem`, one could simply write:

```
if (price > 1000) askPermission purchaseItem
```

Workflow patterns 6: multi-choice and 7: synchronizing merge Multi-choice or OR-split, the concurrent execution of multiple branches based on criteria, is not available as a combinator in iTask. However, it can be defined easily. Assuming that a list of alternatives is available with a boolean condition for each alternative, `tasks :: [(Bool, Task)]`, we can execute only the tasks which match the conditions with the expression (`allTasks o map snd o filter fst`) tasks. The list of tasks is filtered first and subsequently executed. As soon as all selected tasks are executed, their results are combined. Due to the block-structuring of iTask, semantics problems related to OR-join do simply not exist.

Workflow pattern 8: multi-merge Expressing multi-merge in token-based workflow languages is straightforward: it involves the passing of all tokens from multiple incoming branches to a single outgoing branch — there is no synchronization of control flow.

Due to the iTask integration of control flow and data flow, all standard task combinators have built-in support for synchronization. It is not possible to specify directly that the entire subworkflow after a particular merge should be instantiated *for each task preceding the merge*.

For instance, given that `a`, `b` and `c` are tasks, we want to execute `a` and `b` in parallel, without synchronization, then execute task `c`. The expression `a -&&- b >>| c` will first execute task `a` and `b` in parallel. When both `a` and `b` have completed, `c` is executed only once — which is not correct for multi-merge. In this particular case, we can combine task `c` with both `a` and `b` sequentially: `(a >>| c) -&&- (b >>| c)`. This will yield the desired result: both `a` and `b` are executed in parallel, and `c` is executed twice, after both `a` and after `b`. Although this approach works, it becomes unwieldy if multiple paths lead to the sub-workflow `c` after a multi-merge.

iTask support an alternative approach, by means of process combinators. The *spawnProcess* task can start a task as an asynchronous sub-process.

```
spawnProcess :: !UserId !Bool !(Task a) → Task (ProcessReference a) | iTask a
```

The *spawnProcess* task takes three parameters: a *UserId* of the user who should execute the task, a boolean which indicates if the task will start immediately and of course the task to execute. The *ProcessReference* result can be used to manipulate the newly started process. If for each branch a process is created via spawnProcess, the branches run concurrently without any kind of synchronization:

```
spawnProcess getCurrentUser True (a >>| c) >>| (b >>| c)
```

However, the modeling issue remains: it is still necessary to adapt the workflow structure by moving the tasks after the multiple-instance task *within* the multiple-instance task. Therefore, abstracing the expression above to a new combinator would not be useful.

Workflow pattern 9: Discriminator The iTask library contains a XOR-combinator, denoted as `-||-`. However, its semantics are different from the XOR found in many conventional workflow languages. Instead, it corresponds to the discriminator workflow pattern, which waits for the first activity to complete. As soon as this happens, the other task is cancelled. The combinator yields the result of the first task which completes; the result of the other task is ignored.

```
(-||-) infixr 3 :: !(Task a) !(Task a) → Task a | iTask a
```

Just as with parallel composition, a list version of the XOR-combinator also exists, called `anyTask`. It takes a list of tasks as argument. The task which completes first is, all other tasks in the list are cancelled as soon as the first task completes. The definition is

```
anyTask :: ![Task a] → Task a | iTask a
```

4.4.2 Structural patterns

Workflow pattern 10: arbitrary cycles Due to the block structure of iTask, arbitrary cycles — i.e. loops with multiple points of entry or multiple points of exit — are not supported by the normal task combinators. It is possible, but advised against, to use the `spawnProcess` and `waitForProcess` combinators to make arbitrary jumps, as discussed above in section "block structuring".

However, structured loops can be expressed by the `forever` and `repeatTask` combinators. An infinite loop is expressed by `forever`. The `repeatTask` combinator executes a task until the predicate is valid (like a repeat-until loop in imperative programming languages).

```
forever :: !(Task a)→ Task a | iTask a
repeatTask :: !(a → Task a) !(a → Bool) a → Task a | iTask a
```

Workflow pattern 11: implicit termination Implicit termination is supported by iTask. A task terminates at the moment that it yields it value. Implicit termination is also supported for processes: they terminate at the moment that their top-level task completes.

4.4.3 Multiple instances

As discussed in 2.3, the handling of multiple instances of a task involves two kinds of requirements. First, one needs to be able to create multiple instances of the same task. Second, the results of the individual tasks may need to be synchronized.

The typical way to instantiate multiple tasks in iTask is to create a list of tasks, then execute all tasks in the list in parallel using the `allTasks` combinator. This combinator will wait until all tasks in the list have completed, then yield a list of results of the individual tasks in the list.

Workflow pattern 12: Multiple instances without synchronization The multiple instances without synchronization pattern involves the creation of multiple instances of the same task, then executing them without any kind of synchronization. The subworkflow succeeding the multiple-instance task should be executed *for each instance* of the multiple-instance task. The multiple instantiation of the succeeding subworkflow is difficult to express in iTask. The problem is very akin to the *multi-merge* pattern discussed above; a possible solution is to make the tasks that would succeed the multiple-instance task, part of the multiple-instance task itself.

Workflow pattern 13: multiple instances with a priori design time knowledge Because the iTask combinators have built-in support for synchronization, specifying synchronization of multiple instances is straightforward. First, one defines a list of task instances. If the same task `t` has to be executed a constant number `n` times, one would write `repeatn n t`. If each instance has different parameters, one would use an expression like a list comprehension to create the list. The list is passed to the `allTasks` combinator, which will execute all tasks in the list in parallel.

Workflow pattern 14: multiple instances with a priori run time knowledge iTask realizes this workflow pattern exactly the same as #13, with the only difference that n is not a constant, but an expression containing results obtained earlier in the workflow.

Workflow pattern 15: multiple instances without a priori runtime knowledge According to the workflow patterns web site [50], this workflow pattern is supported by only one WFMSs. However, it can be expressed in iTask as well by means of the `parallel` combinator. This combinator has a complex definition, for which we refer to the iTask source code. The `parallel` combinator keeps track of an internal state. Using a user-specified accumulator function, the state can be updated, new tasks can be added, or the entire parallel action can be stopped. The accumulator function is defined by:

```
((taskResult,Int) pState → (pState,PAction (Task taskResult) tag))
    | iTask taskResult & iTask pState
```

After completion of each task, the accumulator function gets the result of the completed task (`taskResult`), the number of completed tasks and the current state (`pState`). Using this information, the accumulator should produce a new state, and an action (`PAction`) how to continue:

```
:: PAction x t  = Stop          // stop the entire parallel/grouped execution
                | Continue      // continue execution without change
                | Extend .[x]   // dynamically extend list of tasks in parallel/group
                | Focus (Tag t) // focus child-tasks with given tag
```

The workflow pattern requires that all instances are synchronized before the next task starts. The `parallel` combinator satisfies this criterium, since completion of the individual instances is required for determining the next action (in `PAction`). Hence, multiple instances without a priori knowledge is fully supported by iTask.

4.4.4 State-based patterns

Workflow pattern 16: deferred choice The deferred choice describes a point in the workflow where a choice is offered between two branches. As soon as one of the branches starts executing, the other branch is excluded. On a first glance, this behavior is similar to the XOR-combinator `-||-` in iTask. There is an major distinction though. The deferred choice pattern prescribes that "... the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible" [54]. When using the XOR-combinator in iTask, the choice is made when one of the branches *completes,* which is too late to match the deferred choice pattern. If it would match, the deferred choice pattern would be equivalent to the discriminator pattern, which is not the case. Currently, iTask does not support another way to exclude a branch when another branch starts.

Workflow pattern 17: interleaved parallel routing If a shared resource has to be used by multiple tasks, but cannot be used by multiple tasks simultaneously, the tasks have to be executed interleaved. Support for the *interleaved parallel routing* pattern ensures that no two tasks can be executed simultaneously. Additionally, a partial ordering in tasks can be specified. Currently, iTask does have direct support for interleaved parallel routing.

Workflow pattern 18: Milestone The milestone pattern allows execution of a task only, if the workflow is in a given state. iTask directly supports this pattern if the concerning task is stand-alone. Suppose we have a sequence of tasks `a >>| b >>| c`, and another task `d` may only be executed *between* b and c. This can be expressed by `a >>| (b -|| d) >>| d`. The execution of `d` does not influence the original sequence, since the `-||` operator discards the result of its right task and waits for the result of its left. If the left task completes, the right can no longer be started. However, if task `d` is part of another sequence, it is not possible to synchronize task `d` with `b`.

4.4.5 Cancellation patterns

Workflow pattern 19: cancel task iTask does not have a direct primitive to cancel tasks, but it is possible to cancel an already started branch by using the XOR-combinator `-||-` to describe an alternative branch, which is executed if the task should be canceled. As soon as the alternative branch completes, the original branch will no longer yield a result, and iTask will abandon this branch. Cancellation of tasks by end users can easily be defined as a new *cancel* combinator.

```
cancel :: Task a → Task (Maybe a)
cancel task = (task >>= λx = return (Just x))
            -||- (showMessage "cancel task" >>| return Nothing)
```

A form with a "Cancel task" button is shown to the user. If the button is clicked,

Workflow pattern 20: cancel case A case — or process in iTask terms — is called a *process* in iTask. Running processes can be manipulated via process combinators. The *deleteCurrentProcess* combinator will abort the currently running process, and is defined as

```
deleteCurrentProcess :: Task Bool
```

4.4.6 Conclusion

The above workflow pattern analysis demonstrates that iTask supports the vast majority of the original control flow patterns, even patterns which are hardly supported in contemporary graphical WDLs like the *multiple instances without a priori run time knowledge*. Due to its combinatory basis, synchronization is present in all standard combinators. Therefore, the patterns which are not supported by iTask are mainly the ones who assume a lack of synchronization, like the *multiple instances without synchronization* pattern.

Chapter 5

Gin: Graphical iTask notation

5.1 Design principles

Expressiveness Ideally, a graphical notation for iTask should be powerful enough to express all iTask workflows. However, the iTask WDL is embedded in the host language Clean, so any valid Clean expression can be used in an iTask workflow. Hence, being able to graphically denote all possible iTask workflows would imply having a complete graphical notation for Clean. Developing such an extensive graphical functional programming language is out of the scope of this project. The reason is that for non-programmers, such a notation would be as hard to learn as a regular functional language.

We deliberately choose to restrict the graphical language to a subset of constructs, namely those which are common in graphical WDLs. We use the workflow patterns as a guideline to determine these constructs. These correspond largely with the set of iTask task combinators, but some workflow concepts like conditionals are already part of Clean itself. This choice inevitably means that certain parts of iTask workflows cannot be expressed graphically. Therefore, the graphical notation should be hybrid by allowing the embedding of textual Clean expressions. Concepts which cannot be expressed grapically, can be embedded as textual expressions.

Familiarity Users who have experience with workflow modeling in contemporary graphical WDLs, should be able to use their prior knowledge in the construction of graphical iTask models. Therefore, we employ notations commonly found in existing graphical WDLs.

iTask semantics The semantics of the graphical notation must be expressible in terms of textual iTask expressions. Each graphical construct must have a mapping to an iTask expression. The other way round does not hold: since the graphical notation covers a subset of the iTask WDL, not every iTask expression will be graphically representable.

5.2 Graphical mapping

In this section, we present the syntax of *Gin*: a Graphical iTask notation. We describe the semantics in an semi-formal way, by means of a mapping from textual iTask expressions to Gin diagrams.

We use the following notational conventions. Variables which are not part of the literal syntax, but merely serve as a placeholder, are written in *italics*. The map from a Clean expression e to its corresponding Gin representation is denoted as $[\![e]\!]$.

The Gin language is based on a directed graph, consisting of nodes and directed edges. There are two types of nodes: *tasks* and *connectors*. Edges can optionally have a textual *pattern*.

Tasks Functions f:: `Task` α are mapped to *tasks* in Gin and have a graphical representation. Both application and definition of these task functions is supported. Lists of tasks [`Task` α] also have a graphical representation, these are used to express multiple instances. Expressions of other data types cannot be represented graphically and are denoted textually.

As seen in chapter 2, it is common in graphical WDLs to denote the application of a task by a rectangle containing its name. We adopt this notational convention: task functions are denoted as rounded rectangles; the function name is written in the rectangle. Optionally, an task can have a custom icon, shown at the left of the function name. Such an icon provides an additional visual cue of the meaning of the task.

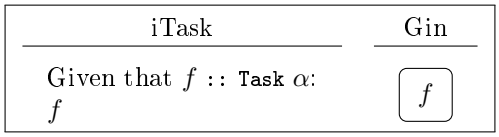

Figure 5.1: Application of a task

Tasks with arguments Functions $f :: \alpha_1 \ldots \alpha_n \rightarrow$ `Task` α are mapped to parameterized tasks. Actual parameters are depicted as name-value pairs below each other. Values are textual Clean expressions (see figure 5.2). In case of higher-order tasks, the higher order argument is denoted as a directed graph.

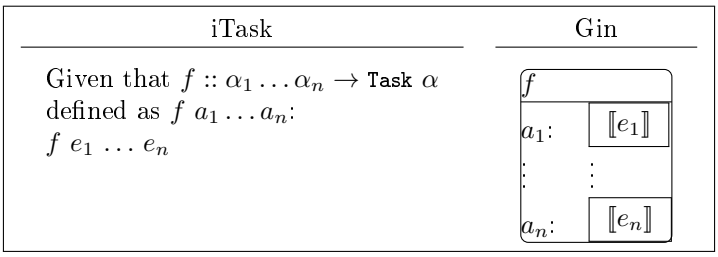

Figure 5.2: Application of a task with arguments

Task definition Function definition $f :: \alpha_1 \ldots \alpha_n \rightarrow$ `Task` α, defined as $f\ a_1 \ldots a_n = e$ can be mapped to Gin, as shown in figure 5.3. Each formal parameter $a_1 \ldots a_n$ must be a single variable name. The variables $a_1 \ldots a_n$ are in scope of e. Pattern matching in function definitions and functions having multiple alternatives are not supported.

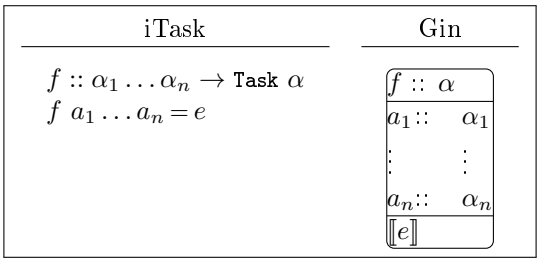

Figure 5.3: Task definition

Recursion From the definitions of tasks and task definition, it follows that recursive tasks can be expressed in Gin. When defining a task f, the task f may occur in the definition, as shown in figure 5.4.

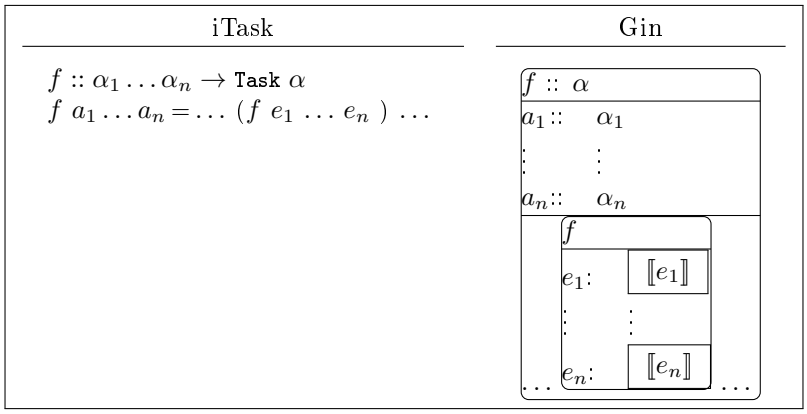

Figure 5.4: Recursive tasks

Monadic return The monadic `return` combinator lifts a value to the task domain. The return is indicated by an ellipse containing an expression (figure 5.5).

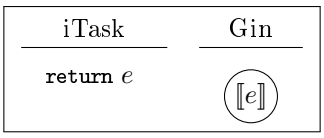

Figure 5.5: Monadic return

Sequential routing Sequencing of tasks is indicated by an arrow between two tasks: upon completion of the first task, the second task will start (figure 5.6). The results of the first task are discarded.

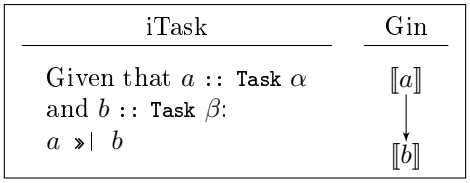

Figure 5.6: Sequential routing, discard result

The monadic bind operator `»=` preserves the result of the first task. Expressions of the form a `»=` $\lambda\ p \rightarrow b$, where a ::Task α, b :: Task β are depicted by an arrow between two tasks having a label attached with pattern p.

The lambda abstraction λp is only in scope within task b. However, b can be a composite task consisting of a sequential, parallel, or conditional block, or any (nested) combination thereof. In order to keep Gin visually concise, the scopes of these lambda abstractions are not visualized.

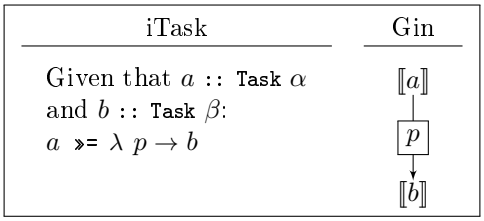

Figure 5.7: Sequential routing, monadic bind

If the second argument of the monadic bind is not a lambda abstraction, but a function defined elsewhere, a direct mapping is not possible. However, there is a simple solution. In Lambda calculus, one can apply η-abstraction, that is, replacing a lambda term m by $\lambda x.m\,x$, in which x is a free variable in m. This principle can be used as a transformation rule in expressing existing iTask workflows in Gin: if the second argument of the monadic bind operator does not have the form $\lambda x \to y$, one should apply η-abstraction on the second argument, obtaining the form $\lambda x \to y$, which is supported.

For instance, the following workflow cannot be expressed directly:

```
taskA :: Task Int
fTaskB :: Int → Task Void

workflow = taskA >>= fTaskB
```

(We assume that `x` is free in `fTaskB`). After applying η-abstraction, we obtain

```
workflow = taskA >>= λx → fTaskB x
```

which can be expressed graphically as in figure 5.8.

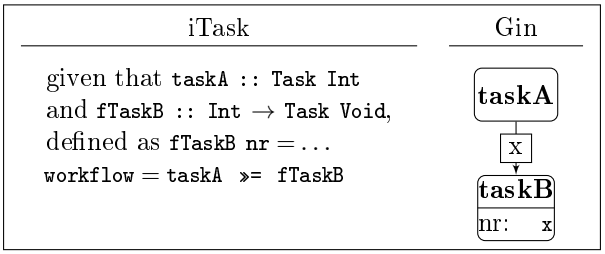

Figure 5.8: Example of monadic bind

Parallel routing Graphical WDLs commonly denote parallel routing by a parallel split connector, a set of branches, and a parallel merge connector. We adopt this approach. However, the iTask WDL has two combinators for parallel composition, namely a binary operator and a combinator which operates on lists of tasks:

```
(-&&-) infixr 4 :: !(Task a) !(Task b)  → Task (a,b) | iTask a & iTask b
allTasks        :: ![Task a]            → Task [a]   | iTask b
```

Conceptually, the behavior of these combinators is similar. They differ only differ by their arguments (two parameters or a list) and the way the results of the individual tasks are merged (in a tuple or a list, respectively).

We introduce a single *parallel split connector* and two *parallel merge connectors*, one for tuples and one for lists (figure 5.9). The mapping is shown in figure 5.10 and figure 5.11.

Note that this mapping in figure 5.11 can only be used if the argument of allTasks is a list of explicit enumerated tasks. If the list is constructed by means of a list comprehension, a different graphical notation is used, described below in the *multiple instances* section.

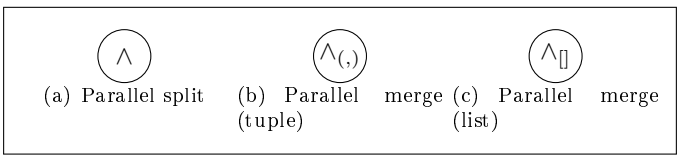

Figure 5.9: Parallel split and merge connectors

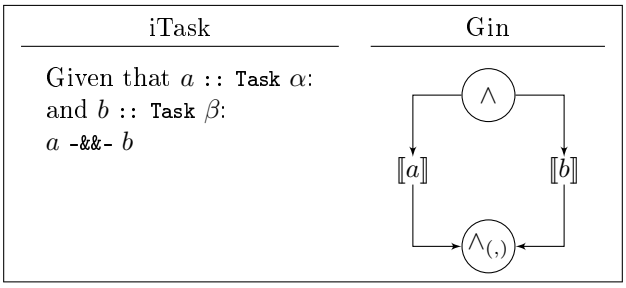

Figure 5.10: Parallel routing (result in tuple)

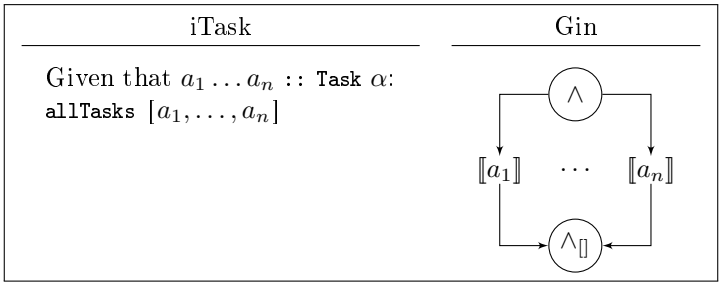

Figure 5.11: Parallel routing (result in list)

Conditional routing The Gin language supports exclusive choice by means of **case**-distinction.

A case distinction is denoted by a *case split connector*, one or more branches, and a *case merge connector*. In the case split connector, any Clean expression can be entered. Each case alternative is denoted as a separate branch. The incoming edge of each branch may contain a Clean pattern. If the pattern is omitted, the alternative is considered to be the default alternative. A case distinction can have at most one default alternative (figure 5.12).

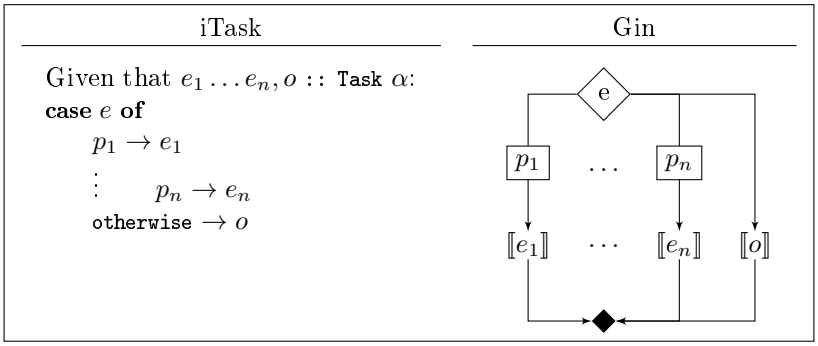

Figure 5.12: Case distinction

if-expressions are treated as a special case of *case* expressions. The expression **if** p a b, where a :: Task α, b :: Task α is rewritten to **case** p **of** True $\rightarrow a$; False $\rightarrow b$ which is visualized in figure 5.13.

Iteration An infinite loop is expressed by the forever higher-order task. The *task* parameter contains the task to be executed (figure 5.14).

A conditional loop is expressed by the repeat-until task. The execution of task a is repeated until its result satisfies p (figure 5.15).

Multiple instances Multiple instances in iTask are expressed by combinators which operate on lists of tasks. Lists in Clean can be composed in many ways: either by explicit enumeration, using

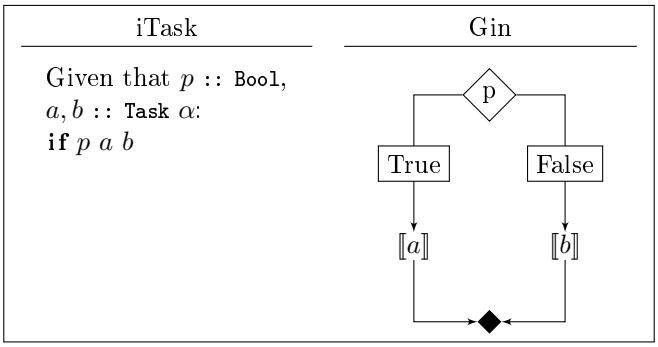

Figure 5.13: If expression

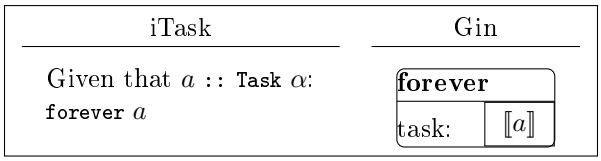

Figure 5.14: Infinite loop

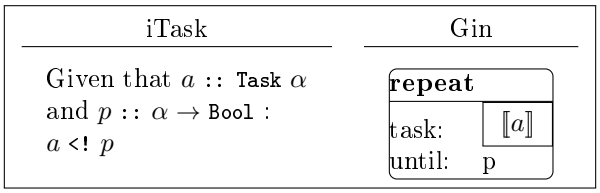

Figure 5.15: Conditional loop

list comprehensions, dot-dot expressions or functions and operators which yield lists, like the (++) operator for list concatenation.

Gin supports a visualization of two types of lists: explicitly enumerated lists and simple list comprehensions, with a single generator and filter. Other types of lists composition are not graphically supported and can only be entered by means of textual Clean expressions.

Lists are indicated by double-edged rectangles. Enumerated lists show their elements below each other, like in figure 5.16. Simple list comprehensions are denoted as shown in figure 5.17.

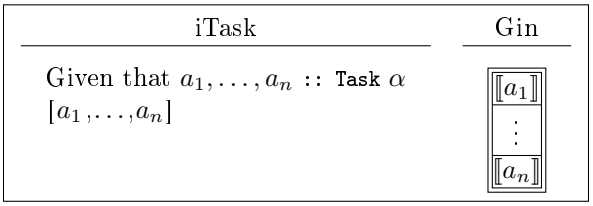

Figure 5.16: Enumerated list

List combinators for sequential composition (`sequence`), parallel composition (`allTasks`), and discriminator (`anyTask`) are expressed like normal higher order tasks, see figure 5.18.

```
sequence :: !String ![Task a] → Task [a] | iTask a
allTasks :: ![Task a] → Task [a] | iTask a
anyTask :: ![Task a] → Task a | iTask a
```

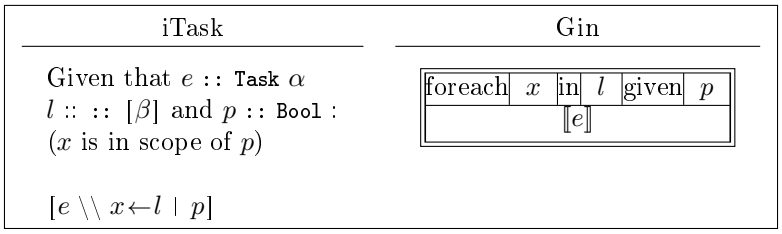

Figure 5.17: List comprehension

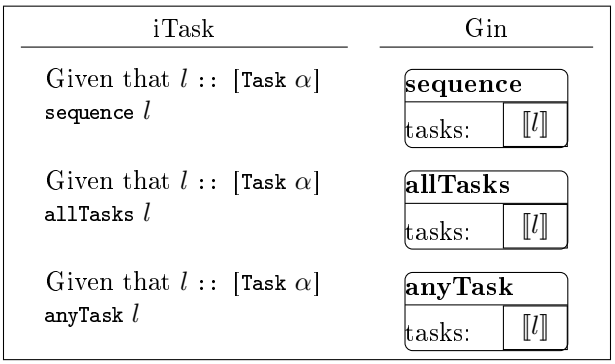

Figure 5.18: Examples of list combinators

5.3 Meta model

Models are abstractions of reality. For instance, a workflow model is an abstract description of a real-world business process. A modeling notation itself can also be described in an abstract way by means of a model. Such a "model of a model" is called a meta model.

Above, we have explained a number of concrete expressions in Gin. The Gin meta model, shown in figure 5.19 is an abstract representation of the Gin concepts, their attributes and mutual relations. We use the UML class diagram notation to denote the meta model.

Overview iTask makes a distinction between basic tasks, task combinators and combined tasks. The basic tasks and task combinators which are part of the iTask library are atomic units. The implementation of these tasks is considered a black box; we only know their type declaration, available in a definition module (.dcl file). This *declaration* and its *formal parameters* are represented in the meta model. In order to display tasks in a diagram, we add three properties: (1) the name of each formal parameter, (2) an optional icon indicating the meaning of the task, and (3) the task shape. Split and merge connectors, used in parallel and conditional branching are also modeled as a declaration.

Combined tasks modeled by the user have an accessible *definition*. This definition includes a declaration as well, which is indicated by the relation in the meta model. A workflow consists of a single definition at top-level. The definition body consists of an *expression*.

Gin is a hybrid notation, which allows graphical as well as textual notation. Every expression can be denoted as a textual *Clean expression*, like in the iTask WDL. Expressions of type :: Task α can be denoted as a graph (*graph expression*). Lists of tasks, i.e. expressions of type :: [Taskα], have a graphical representation as well and are therefore explictly modeled.

A graph expression consists of a set of nodes and a set of edges. A *node* is the instantiation of a declaration; the appearance of a node is determined by the shape attribute of its declaration. A directed *edge* connects two nodes. An edge can optionally be labeled with a pattern.

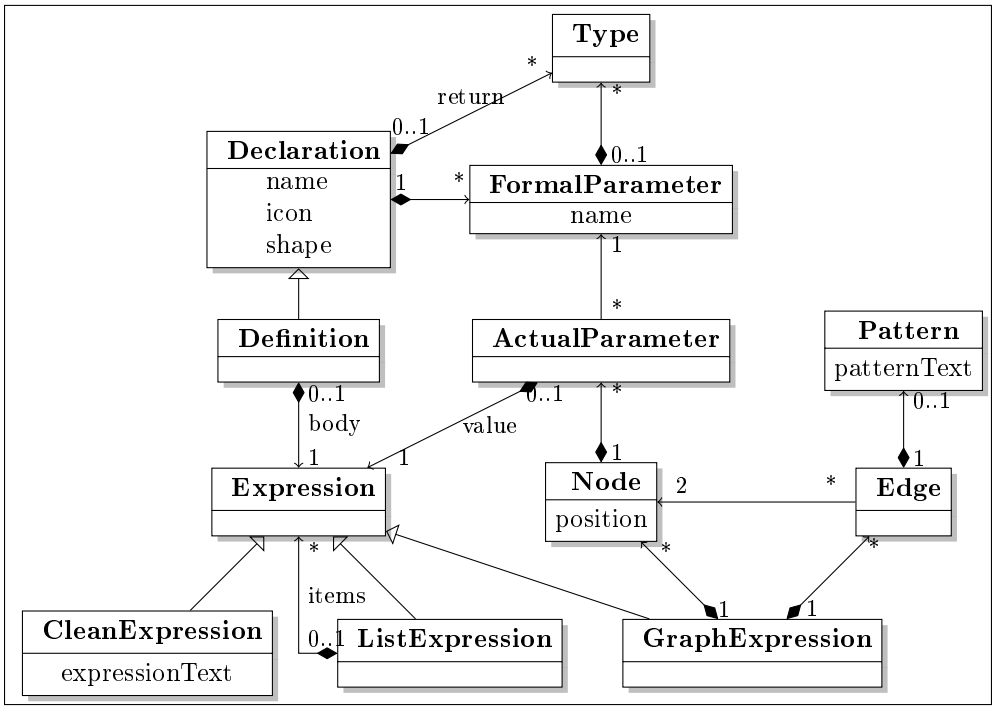

Figure 5.19: Gin meta model

Textual constraints Since function and parameter names, patterns and textual expressions in the Gin notation are one to one mapped to Clean, syntactic requirements in Clean are also applicable to Gin. These requirements follow from the Clean language report[36]:

1. The name of a *Declaration* must be be unique within a workflow model.
2. The name of a *Declaration* must be a valid `FunctionName`, as stated in section 2.1.1 of the Clean language report.
3. The name of a *FormalParameter* must be a valid `Variable`, as stated in section 2.1.1 of the Clean language report.
4. The text of a *CleanExpression* must be a valid `GraphExpr`, as stated in section 3.4 of the Clean language report.
5. The text of a *Pattern* must be a valid `Pattern`, as stated in appendix A.3.2 of the Clean language report.

Block structure In the previous chapter, we saw that iTask is essentially a block oriented language, but it allows graph structuring by means of process combinators. In order to keep the mapping between Gin and iTask comprehensible, we decide not to use the process combinators, but restrict ourselves to structured iTask modeling. As a consequence, we have to constrain Gin to only allow well-structured graphs, consisting of well-nested, non-overlapping blocks. The allowed graph structure is defined recursively:

1. A graph consisting of a single node is well-structured.
2. A sequential composition of two well-structured graphs yields a well-structured graph
3. A parallel composition of a split node, one or more branches, and a merge node yields a well-structured graph.

The consequence of this graph structure is that the graph has a unique start node (a *source*). This start node can either be a task or a split connector and has no incoming edges. The graph must have one unique end node (a *sink*), which can be either a task or a merge connector and has no outgoing edges.

More formally: Let $G = (N, E, s, t)$ be a Gin graph with a set of nodes N, a set of directed edges $E \in (N \times N)$, source $s \in N$, sink $t \in N$.

1. if N contains a single task node, then N is a well-structured graph.
2. Given that $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$ are well-structured workflow graphs, then $G = (N = N_1 \cup N_2, E = E_1 \cup E_2 \cup (t_1, s_2), s = s_1, t = t_2)$ is a well-structured graph (*sequential composition*).
3. Given that $n \geq 1$, $G_1 = (N_1, E_1, s_1, t_1) \ldots G_n = (N_n, E_n, s_n, t_n)$ are well-structured graphs, s is a split node and t is a merge node, then $G = (\bigcup_{i=1}^{n} N_i \cup \{s, t\}, (\bigcup_{i=1}^{n} E_i) \cup \{(s, s_1), \ldots, (s, s_n), (t_1, t), \ldots, (t_n, t)\}, s, t)$ is a well-structured graph (*parallel composition*).

5.4 Example: Bug reporting

The following example describes a bug reporting workflow. The example is adapted from [25]. Figure 5.20 shows the bug report expressed in Gin.

```
reportBug :: Task Void
reportBug
    = enterInitialReport
     >>= λ report →
    fileBugReport report
     >>= λ bugnr →
    case report.severity of
        Critical
            = selectAssessor report.application report.version
                 >>= λ assessor →
                assessor @:
                ("Bug report assessment",
                requestConfirmationAbout
                    "Is this bug really critical?" report)
                 >>= λconfirmed → selectDeveloper
                                      report.application report.version
                 >>= λdeveloper → if confirmed
                    developer @: (resolveCriticalBug bugnr)
                    developer @: (resolveBug bugnr)
        -
            = selectDeveloper report.application report.version
             >>= λdeveloper → developer @: resolveBug bugnr
```

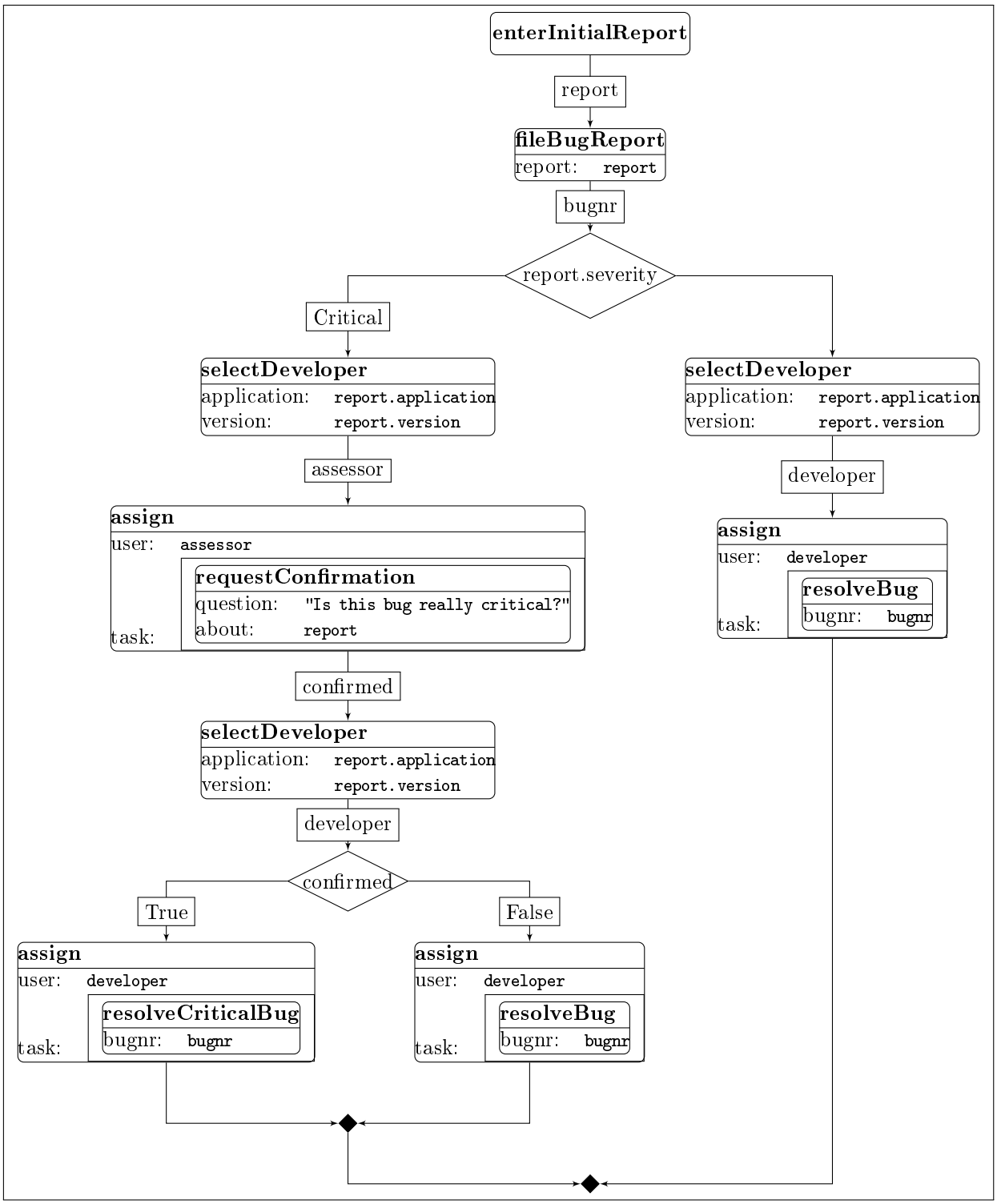

Figure 5.20: Example of bug reporting workflow

Chapter 6

Graphical editor

In order to demonstrate that workflows expressed in Gin are executable, we have implemented a proof of concept editor for Gin diagrams. A compilation process compiles these diagrams into executable tasks. This chapter describes the editor; chapter 7 deals with the compilation process.

6.1 Introduction

An *editor* is an interactive user interface component for entering and updating of a value. A single value can be a primitive value like an integer or boolean, but a value can also have an arbitrarily complex structure. An editor consists of an internal state, which keeps track of the value, and a front-end, which is shown to the user. An editor has two duties. First, the editor visualizes its internal value in the front-end. Second, the editor responds to user actions and updates its internal value. This principle is shown in figure 6.1 and is applicable to any editor.

A *workflow editor* is an editor designed for editing workflow models. We can think of the entire workflow model as a single value being edited.

6.2 Design principles

iTask integration Preferible, the editor should be integrated in the iTask system. By integrating, we mean that the end user can use the iTask web client not only to enact tasks, but also to design graphical workflow models.

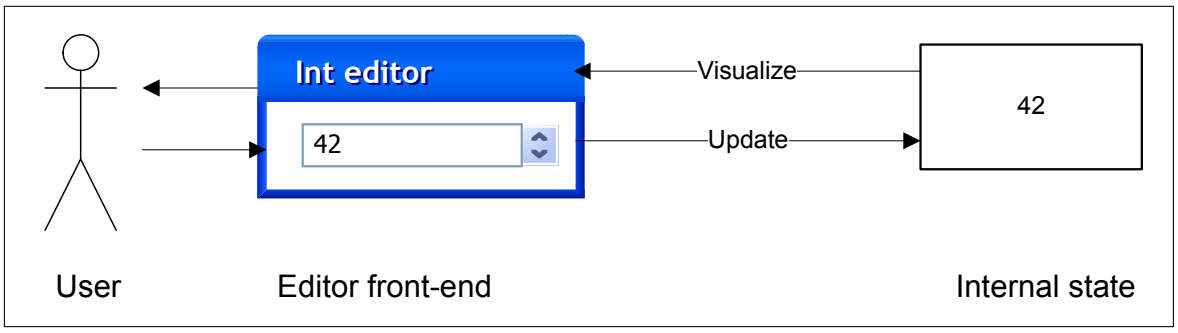

Figure 6.1: General principle of an editor

Editing style Graphical editors can be divided in *Structured editors* and *Free-hand editors*. A structured editor shows a correct diagram, and offers the user a set of operations to transform the diagram to another correct diagram. On the other hand, a free-hand editor leaves the user free in the placements of elements. The consequence is that the user can perform operations which lead to incorrect diagrams. The disadvantage of many structured editors is that they enforce a top-down modeling style. For instance, to model a sequential execution of task a and b, one first has to place a sequence-element, then place the tasks a and b *in* the sequence element. In contrast, a free-hand editor also allows a bottom-up modeling style, by placing all individual tasks first, and model the order later. We choose to use a free-hand editor, because it offers more freedom in the way of modeling.

Direct feedback A free-hand editor does not prevent the user from making modeling errors. Therefore, the editor has to support error handling: if the user is notified of errors in an early stage, fixing an incorrect model is easier. Error messages have to comprehensible and indicate the specific source of the error.

6.3 Approach

We may choose between several approaches for implementing the editor. We discuss three approaches with their pros and contras:

1. **Using iEditors** Jansen et al.[15] describe an iTask extension to integrate plugins named *iEditors* in the iTask system. Plug-ins are specified entirely in Clean, so they can use a high abstraction level. The plug-ins can be executed on the server side or in the client web browser, using a SAPL interpreter Java applet.

   The iEditors architecture allows the embedding of interactive graphical editor plugins. Therefore, it is in principle possible to implement a Gin editor using iEditors. However, there is a pragmatic reason not to do so. Since the publication of iEditors, the iTask system has undergone a major revision[25]. As of August, 2010 the new iTask v2 does not yet support the iEditors extension. Implementing the graphical editor by means of iEditor technology would require having to re-integrate the iEditors from iTask v1, which is — given time constraints — not a feasible option.

2. **Using a visual language toolkit** A visual language toolkit is a system which can generate a graphical editor and parser, based on a formal specification of a visual language. An example of a visual language toolkit is *VLDesk*[5], which uses eXtended Positional Grammars to specify a visual language. Other examples are DSL toolkits like the *Eclipse Graphical Modeling Framework* (GMF) [9] and the *Microsoft Visual Studio Visualization and Modeling SDK* (formerly: Visual Studio DSL tools)[27]. The advantage of using a visual language toolkit is that a graphical editor can be generated with few effort. Nevertheless, such an editor runs either standalone (VLDesk) or is tied to a particular IDE (Eclipse or Visual Studio, respectively). It is difficult to integrate such an editor in iTask.

3. **Building a custom editor** Building a custom editor provides the flexibility to integrate the editor in the iTask web client. The downside of building a custom editor is the programming effort required.

Out of these options, we chose to build a custom editor. We drop the iEditors approach purely for pragmatic reasons, given the time constraints of the project. A visual language toolkit may save us time in development, but can hardly be integrated in iTask. Besides, error handling is easier to implement in a custom editor. The syntax of Gin diagrams is dependent on Clean, by means of the type system and embedding of Clean expressions. Hence, we may need to integrate tools which perform syntax and type checking. This is easier to realize in a custom editor.

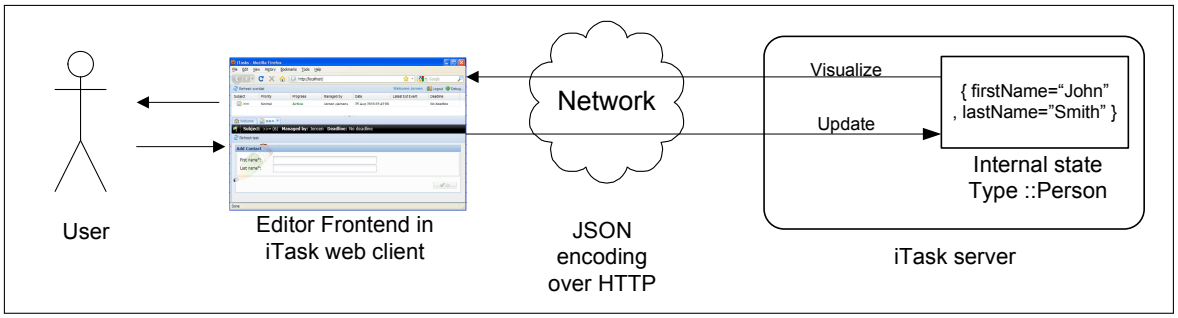

Figure 6.2: Architecture of editors in iTask

6.4 Editors in iTask

An important aspect of the *enactment* of workflows is the entering and editing of data. Therefore, editors are part of virtually any WFMS, including iTask.

In relation to the general editor principle shown in figure 6.1, the editors in iTask have the following characteristics:

1. The internal value of every editor has a type. In iTask, the value of an editor is represented by a Clean expression. Clean is strongly typed, so every expression has a type.
2. iTask is a client/server system. The storage of the internal value takes place on the server system, while the frontend runs on the client system. The client and server communicate via JSON data structures sent over an HTTP connection. The server sends a representation of the value to the client. As soon as the user has entered data, the update is sent to the server, which updates its internal value.
3. The iTask client is web based, so the visualization and user input takes places at the client web browser.

These characteristics are visualized in figure 6.2.

Generating editors In conventional systems, implementing an editor is manual work. iTask can automatically generate an editor for any data type. iTask "looks" at the data type and generates a form based editor based on the structure of this data type. For instance, a `String` data type is visualized as a text box, an algebraic type with multiple data constructors is shown as a set of radio buttons or a combo box, and a record is shown as a form.

iTask implements the generation of editors using generic programming techniques. Two generic functions are used: `gVisualize` visualizes a value, which is rendered by the client. and `gUpdate` maps an updated visualization from the client back to the internal value. For implementation details of these functions, we refer to [25].

```
generic gVisualize a :: (VisualizationValue a) (VisualizationValue a)
                        *VSt → ([Visualization], *VSt)
generic gUpdate a  :: a *USt → (a, *USt)
```

The generic implementation of these functions generates the form-based editor. In order to allow the automated generation of an editor for a user-defined type t, we have to ask the compiler to derive the generic instances of `gVisualize` and `gUpdate` for type t. These generic functions are part of the `iTask` class, so we derive the instances as follows:

```
derive class iTask t
```

Generated form-based editors are not always the most practical editors. For instance, entering a location is easier by clicking on a map than typing in its coordinates. In those cases, we do not **derive** an instance, but *specialize* the generic functions for a particular type.

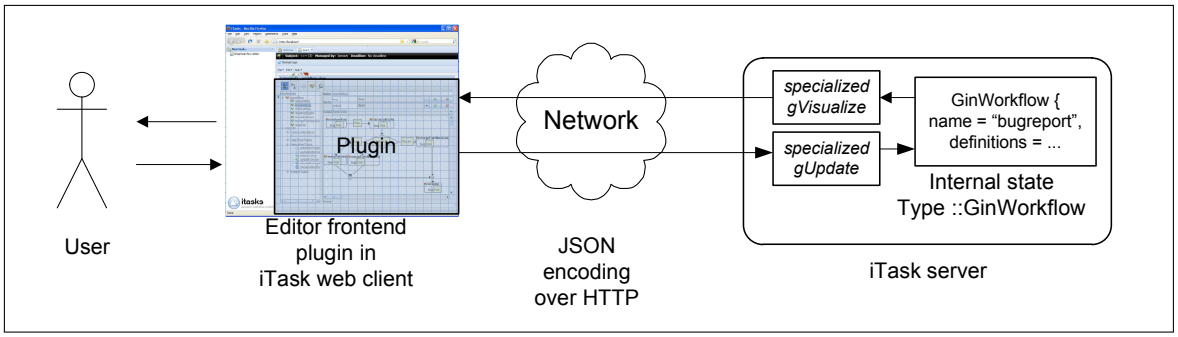

Figure 6.3: Integration of the Gin editor in iTask

Using editors in workflow models iTask workflow models can use the derived or manually specialized editors by means of a set of *interaction tasks*.

The `enterInformation` and `updateInformation` tasks model the entry of a new value and the editing of an existing value respectively:

```
enterInformation  :: question   → Task a | html question & iTask a
updateInformation :: question a → Task a | html question & iTask a
```

Upon enacting the workflow, these tasks will cause the iTask WFMS to generate an editor for the inferred type of the `enterInformation` or `updateInformation` expression.

6.5 Integrating the graphical workflow editor in iTask

If we want to integrate a graphical workflow editor in the iTask system, we have to satisfy the iTask editor architecture, as discussed above.

We stated earlier that we can think of a workflow model as a value being edited. Since every value in iTask has a type, we need to define a type which can represent an entire workflow model. We will call this type a `GinWorkflow`.

As soon as we have the `GinWorkflow` type, we can implement the workflow editor: this is an editor for values of type `GinWorkflow`.

A very fast way to implement the editor would be simply by stating

derive class `iTask GinWorkflow`

This would enable the generation of a form-based editor for values of type `GinWorkflow`. Although such a generated editor *can* be used to edit values of type `GinWorkflow`, it is by no means a graphical editor.

If we want to have a graphical editor, we should not use **derive**. Instead, we should specialize the `gVisualize` and `gUpdate` functions for the type `GinWorkflow`. We should implement a specialized `gVisualize{GinWorkflow}` function in such a way, that it generates a `GinWorkflow` representation which is rendered as a graphical front-end in the client web browser. The specialized `gUpdate{GinWorkflow}` function should handle the updates from the client-side editor and map them back to the internal `GinWorkflow` value.

The standard iTask client does not contain a graphical front-end. Therefore, we implement the front-end of the Gin editor using a plug-in, which integrates in the iTask client.

Figure 6.3 shows the integration of the Gin editor schematically. In the next sections, we elaborate on the individual parts: the `GinWorkflow` data type (section 6.6) and the client plugin (section 6.7).

6.6 Data type for graphical workflows

Based on the meta model introduced in the previous chapter, we define a data structure to represent Gin diagrams. Each of the concepts in the meta model is mapped to a record structure. The nesting of records is based on the composition relation. Inheritance relations are expressed by algebraic types with multiple constructors.

Because iTask workflow models are usually expressed in separate implementation modules, we adopt a module structure by introducing a type `GModule` which represents an implementation module. The `GImport` type represents importing a declaration module. The implementation module contains task definitions, while an import only contains declarations. The top level `GinWorkflow` type is a wrapper for a `GModule`.

Below, the significant parts of the `GinWorkflow` structure are listed. To be concise, we briefly mention the other defintions here. The `gTypeExpression` is an algebraic, recursive data type which represents a Clean type. `gShape` and `gIcon` define the shape and icon of a task declaration, respectively. `gPosition` indicates the coordinates of a node in a graph.

```
:: GinWorkflow = GinWorkflow GModule
:: GModule = { name        :: GIdentifier
             , types       :: [GTypeDefinition]
             , definitions :: [GDefinition]
             , imports     :: [GImport]
             }

:: GImport = { name         :: GIdentifier
             , types        :: [GTypeDefinition]
             , declarations :: [GDeclaration]
             }

:: GTypeDefinition = { name       :: GIdentifier
                     , expression :: GTypeExpression
                     }

:: GDefinition = { declaration :: GDeclaration
                 , body        :: GExpression
                 , locals      :: [GDefinition]
                 }

:: GDeclaration = { name         :: GIdentifier
                  , formalParams :: [GFormalParameter]
                  , returnType   :: GTypeExpression
                  , icon         :: GIcon
                  , shape        :: GShape
                  }

:: GFormalParameter = { name :: GIdentifier
                      , type :: GTypeExpression
                      }

:: GExpression = GGraphExpression GGraph
               | GListExpression [GExpression]
               | GListComprehensionExpression GListComprehension
               | GCleanExpression String

:: GListComprehension = { output :: GExpression
                        , guard :: Maybe GExpression
                        , selector :: GPattern
                        , input :: GExpression
                        }
```

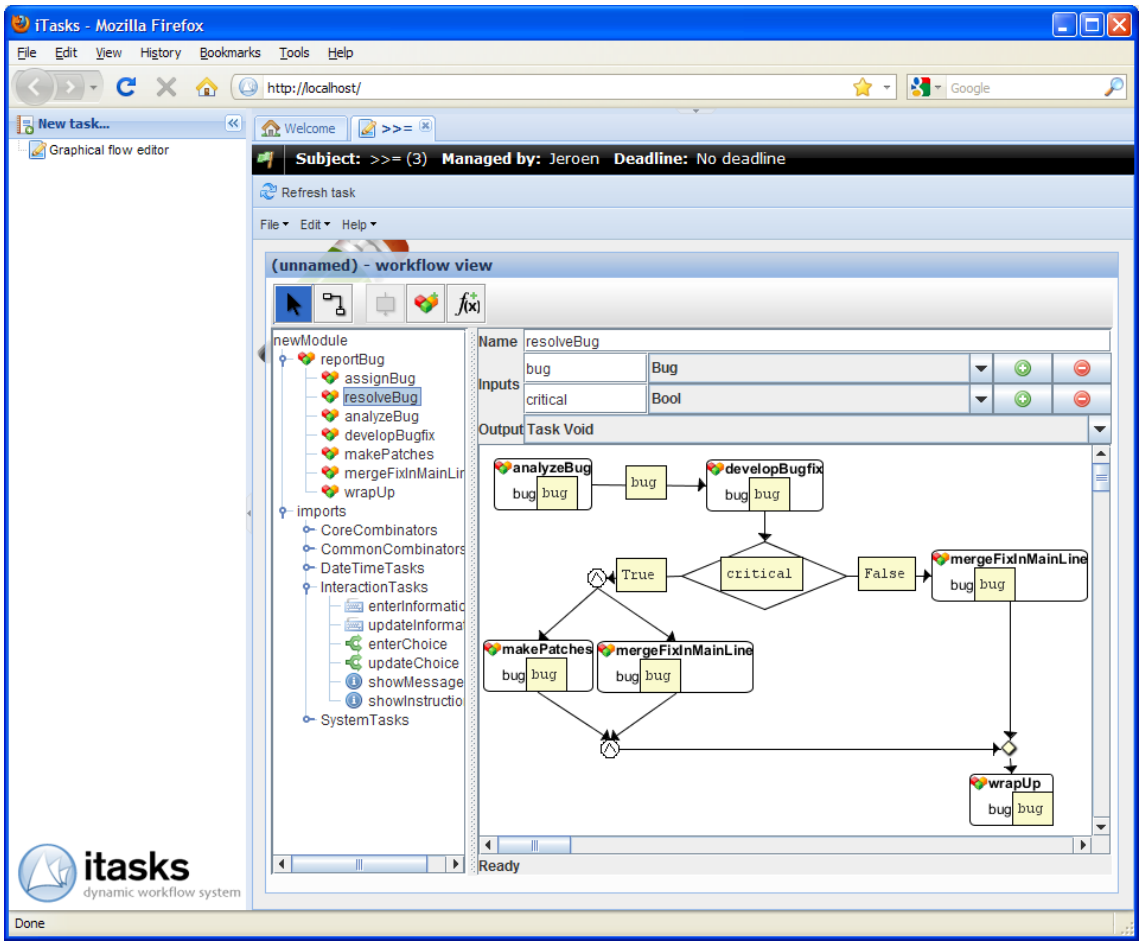

Figure 6.4: Screenshot of Gin frontend

```
:: GGraph = { edges :: [GEdge]
            , nodes :: [GNode]
            }

:: GNode = { actualParams :: [GExpression]
           , name         :: GIdentifier
           , position     :: GPosition
           }

:: GEdge = { fromNode :: Int             //index of node in GGraph.nodes
           , pattern  :: Maybe GPattern
           , toNode   :: Int             //index of node in GGraph.nodes
           }

:: GIdentifier :== String

:: GPattern :== String
```

Figure 6.5: declaration area of Gin frontend

Figure 6.6: toolbar of Gin frontend

6.7 Front-end

6.7.1 User interface

The front-end of the Gin editor (figure 6.4) shows a drawing canvas, a repository, a *declaration area* and a toolbar. The drawing canvas shows the Gin diagram being edited. The repository shows a list of all available nodes: both tasks and connectors. Tasks in the repository and user-defined subtasks are shown. A user-defined task can be opened by double clicking on its name.

Tasks can be added to the diagram by dragging their icons from the repository to the canvas. A recursive task is modeled in the same way: just drag the icon of the current task from the repository to the diagram. In order to pass a task as argument to a higher-order task, drag the task icon to the parameter value rectangle of the higher-order task. Tasks can be moved by drag and drop operations. The actual parameters of tasks can be edited in the diagram itself.

The *declaration area* (figure 6.5) allows editing the declaration of the current task. The task name, formal parameters names and types, and return type can be specified.

The toolbar (figure 6.6) allows to switch between selection mode (allows moving nodes by dragging them) and connector mode (allows drawing edges between nodes). Further, the toolbar contains buttons for adding a pattern to an edge, adding a new subworkflow and adding a new function.

6.7.2 Design choices

Several platforms are conceivable to implement a web-based front-end. The platform needs at least support for composition and interactive manipulation of graph structures or vector graphics. We considered the following options:

1. **SVG + Javascript**: SVG[61] is an XML-based standard to express vector graphics on the web. SVG can be used in interactive applications, by manipulating the XML DOM tree via Javascript. Alas, SVG has limited support for editing embedded text, and is not supported in all major web browsers, Microsoft's Internet Explorer the most notable exception.

2. **Adobe Flash**: Flash, with the Adobe Flex SDK, is a platform for the development of rich internet applications. Flash supports interaction and composition of vector graphics; application logic is developed in the ActionScript scripting language.

3. **Java applets**: Java applets are a proven technology for integrating interactive applications in web pages. Java AWT supports basic vector graphics, which capabilities can be extended with several third party libraries:

   - *JGraph*[1] is an open source library for manipulating graph structures like workflow diagrams. The editor for YAWL is JGraph based. JGraph is not compositional: it is not possible to embed graphs or other object in nodes, which complicates implementing Gin higher-order tasks using JGraph.

- *Piccolo2D*[31] is a 2D vector library for Java. It is based on a hierarchical "scene graph" based model, which is commonly found in 3D graphics. This is advantegeous, since the compositional behavior of the iTask workflow model can be mapped straightforward to a hierarchical structure of nodes — one can create nodes for task combinators, for tasks and task arguments.

We consider both Adobe Flash and Java applets to be equally viable options to implement the editor front-end. Given earlier experience with Java applets in both iTask as well as other projects, we choose to implement the editor front-end as a Java applet, using the Piccolo2D library for displaying graphics.

6.7.3 Implementation

The Gin front-end is implemented as a Java applet. We use the model-view-controller design pattern[22] to separate domain classes from visualization. The domain class structure is based on the Gin meta model. For each class from the domain model, a corresponding view class implements the view in terms of a Java Swing component or Piccolo2D graphical node.

When the applet is started, it receives a value of type `GinWorkflow`, encoded in JSON format. This value is visualized in the applet. As soon as the user makes a change, the new value is encoded as JSON and sent back to the server.

Chapter 7

Compiling workflow diagrams

Gin workflow diagrams are executable. In order to prove this, we implement a compilation process, which transforms Gin workflow diagrams into executable tasks.

7.1 Approach

We have seen in chapter 4 that in the current iTask implementation, workflow models are combined with the iTask libraries and compiled to a stand-alone server executable. This executable is launched. The usual procedure of adding a new workflow model is stopping the server process, recompiling the server executable and restarting the server process — which causes server downtime, unfortunately.

Preferibly, one would have a continuously running server process and the possibility of adding workflow models at runtime. Since an iTask workflow model is essentially a Clean expression of type `Task` α, adding a workflow model at runtime boils down to loading this expression at runtime. This is possible through the use of **dynamics**[32]. A recent extension of the Clean compiler allows overloaded types like `Task` α to be stored in a dynamic. The iTask implementation is able to load and execute a task stored in a dynamic.

Hence, we can add a new Gin workflow model to a running iTask system by creating a dynamic which contains an `Task` expression based on the Gin model. There are several ways to create such a dynamic:

1. **Composition of dynamics**. Two dynamics can be applied onto each other. If run-time unification of their types is possible, the result is a new dynamic. We can use this compositionality in the creation of a dynamic containing a `Task`. Each basic task and connector defined in Gin is mapped to an iTask expression, which is put in a dynamic. When compiling a Gin diagram, the corresponding dynamic for each task and connector in the diagram is looked up. These dynamics are applied onto each other, yielding a dynamic which represents the entire diagram.

   In a Gin diagram, task parameters may be specified as Clean expressions. These expressions need to be converted to dynamics as well in order to use them, which requires a parser and compiler. An option would be to use Esther[37], which provides a basic functional language and a parser for this language. However, Esther solves overloading in its own way, different from the overloading now supported in dynamics themselves. This would prohibit the use of overloaded iTask combinators in textual Gin task parameters.

2. **Using the Clean compiler**. If the dynamic linker is enabled, dynamics can be read from and written to disk by means of the functions `readDynamic` and `writeDynamic`:

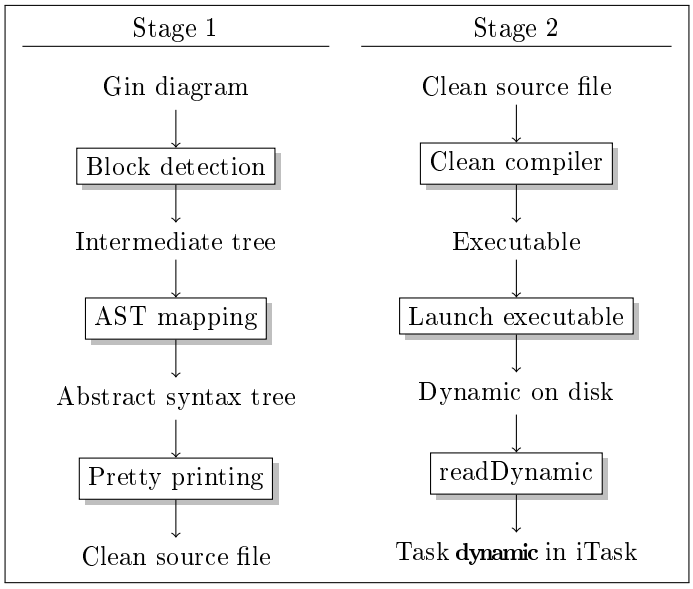

Figure 7.1: Compilation steps

```
writeDynamic :: String Dynamic *World → (Bool, *World)
readDynamic :: String *World → (Bool, Dynamic, *World)
```

This allows the sharing of dynamics between applications. We can exploit this facility: from a Gin diagram, we generate the source code of its corresponding expression of type `Task`. This expression is written to an `.icl` file, together with a `Start` function which writes the `Task` expression as a dynamic to disk. Next, we use the Clean compiler to compile a simple project containing the `.icl` file. This results in an executable. We run this executable, so the `Task` expression is written as a dynamic to disk. Now, iTask only has to load this dynamic from disk by means of the `readDynamic` function. The `Task` in the dynamic can then be executed.

The advantage of the latter approach is that the full expressive power of Clean can be used in textual parameters, and there are no further issues with overloading. Therefore, we choose to use the latter approach to implement the compilation process.

7.2 Architecture

We can divide the compilation process in two main stages: 1) the generation of Clean source code from a Gin diagram and 2) the compilation and loading of the generated code. A schematic overview of the compilation steps is shown in figure 7.1.

In the next sections, we elaborate on the individual steps. A Gin workflow model is stored in a value of the `GinWorklfow` data type introduced in the previous chapter (section 6.6). This data structure may contains graphs, which cannot be mapped directly to iTask expressions. Section 7.3 describes a block detection function to transform these graphs to intermediate tree structures. Next, the Gin data structure is mapped to an abstract tree, using the block detection algorithm to transform the graph structures. mapped to an abstract syntax tree (AST). The AST is introduced in section 7.4, the mapping to AST is described in section 7.5. The AST is pretty-printed (section 7.6) and written to a file. The Clean compiler is called (section 7.7) to compile the file. By means of writing and reading dynamics, the expression is loaded into iTask (section 7.8).

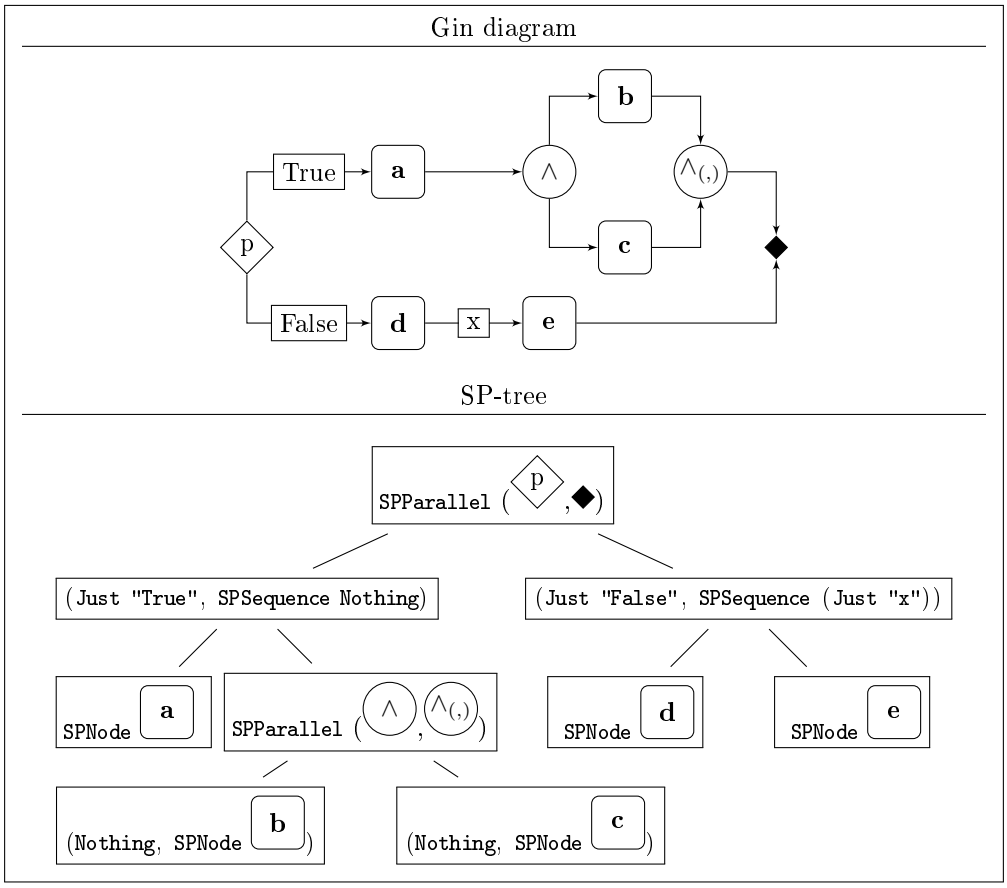

Figure 7.2: Example of a Gin diagram and its corresponding SP-tree

7.3 Block detection

A Gin workflow model is represented by a `GModule` data structure, which contain a set of task definitions. This data structure should be transformed to an abstract syntax tree. We have to realize that the structure may contain graphs (`GGraph` definition), with sets of nodes and edges, while the abstract syntax tree is only a tree. Ergo, we need a way to represent the graph structure as a tree.

A well-structured Gin graph has a block-oriented structure: it may only consist of a single task node (`GNode`), a sequential composition of two well-structured Gin graphs or a parallel composition of (split connector, series of well-structured Gin graphs, merge connector). We use this property to parse the graph structure and procuce an intermediate tree structure, named *SP-tree*. This tree has leaves with unmodified task nodes from the Gin graph, and nodes for the sequential and parallel compositions. The sequential node contains the two graphs of the sequence and (if present) the edge pattern. The parallel node contains the split and merge connector and a list of branches with their patterns. This SP-tree is defined as follows[1]:

```
:: SPTree = SPNode GNode
   | SPSequence SPTree SPTree (Maybe GPattern)
   | SPParallel (GNode,GNode) [(Maybe GPattern,SPTree)]

:: GPattern :== String
```

Figure 7.2 shows an example of a Gin diagram and a depiction of its corresponding SP-tree.

[1] The actual implementation contains additional information for error handling, discussed in chapter 8

Transforming well-structured workflow graphs to trees has been studied before. In [2], Bae et al. describe an imperative block detection algorithm to reduce structured workflow graphs to a single node, yielding a tree structure. The algorithm repeatedly identifies the innermost block and reduces it.

We present a different approach here. We define an algorithm which first identifies unique source and sink nodes, then traverses the graph structure from source to sink. Our contribution is that we can present error messages if an incorrect structure is detected. The algorithm is implemented by the function `graphToSPTree`:

```
graphToSPTree :: GGraph → GParseResult SPTree
```

This function takes a value of type `GGraph` as input, and should produce an `SPTree` as output. However, it is possible that the input is not a structured graph. In that case, it cannot be represented as an `SPTree`. We use an error monad named `GParseResult` to abort parsing if an incorrect structure is detected. The definition of this monad is shown below[2]. The `parseMap` combinator is a monadic map function. The `parseError` function yields an error. The `orElse` combinator yields its first argument if it succeeds, otherwise its second.

```
:: GParseResult a = GSuccess a | GError String
instance Monad GParseResult where
    ret :: a → GParseResult a
    ret a = GSuccess a
    (>>>) infixr 5 :: (GParseResult a) (a → GParseResult b) → GParseResult b
    (>>>) (GSuccess a) k = k a
    (>>>) (GError e)   _ = GError e
    parseMap :: (a → GParseResult b) [a] → GParseResult [b]
    parseMap _ []     = ret []
    parseMap f [x:xs] = f x >>> λx‘ = parseMap f xs >>> λxs‘ =
                  ret [x‘:xs‘]
    parseError :: String → GParseResult a
    parseError e = GError e
    orElse :: (GParseResult a) (GParseResult a) → GParseResult a
    orElse (GSuccess a) _ = GSuccess a
    orElse (GError e)   b = b
```

Let $G = (N, E)$ be a Gin graph G with a set of nodes N and a set of directed edges $E \in (N \times N)$. We define the following helper functions:

- `pred :: GGraph GNode → [GNode]`
  returns the list of direct predecessors of a node n, i.e. $\{n' | (n', n) \in E\}$.
- `succ :: GGraph GNode → [GNode]`
  returns the list of direct successors of a node n, i.e. $\{n' | (n, n') \in E\}$.
- `branchtype :: GNode → BranchType`
  `BranchType :: BTTask | BTSplit | BTMerge`
  returns the branch type of a node: task, split connector or merge connector.
- `patternBefore :: GGraph GNode → Maybe String`
  returns the optional pattern on the (only) incoming edge of a node.
- `patternAfter :: GGraph GNode → Maybe String`
  returns the optional pattern on the (only) outgoing edge of a node.

[2] The monad operators are named `ret` and >>> to avoid naming conflicts with the `return` function from StdFunc and the iTask sequence combinator `>>=`

Identify unique source and sink nodes The first step is to find a unique source (start node) and a unique sink (end node). If one of these cannot be found, parsing is aborted with an error message:

```
graphToSPTree :: GGraph → GParseResult SPTree
graphToSPTree graph
# sources = [ n \\ n←graph.nodes | isEmpty (pred graph n) ]
# sinks   = [ n \\ n←graph.nodes | isEmpty (succ graph n) ]
= case sources of
  [source] = case sinks of
             [sink] = subgraphToTree graph source sink
             []     = parseError "No end node found"
             _      = parseError "End node is ambiguous"
  [] = parseError "No start node found"
  _  = parseError "Start node is ambiguous"
```

Transform a subgraph to an SP-Tree A subgraph, with a source and sink node, is transformed to an SP-tree. We try to detect a single task or a sequential composition of tasks. If the subgraph starts with a split node, we search for a parallel composition. First, the `findmerge` function finds the merge node corresponding to the split node; the `findbranches` function identifies the parallel branches. If none of these structures could be identified, the graph is not well-structured and parsing is aborted.

```
subgraphToTree :: GGraph GNode GNode → GParseResult SPTree
subgraphToTree graph source sink
= case branchtype source of
    //Detect single task node
    BTTask | source==sink = ret (SPNode source)

    //Detect sequential composition
    BTTask | length (succ graph source)==1 =
    subgraphToTree graph (hd (succ graph source)) sink >>> λtree =
                        ret (SPSeries source tree (patternAfter source)

    //Detect parallel composition
    BTSplit | length (succ graph source)==0 =
         parseError "Missing outgoing connections"
    BTSplit = parseMap (λbranch = findMerge branch 0) (succ graph source) >>> λtrees
       if (hd (succ graph trees)==sink)      //subgraph ends with parallel composition?
    (SPParallel (source,sink) trees))     //Yes: finished
    (case length (succ graph parsink) of  //No: find sequence
    1 = subgraphToTree graph (hd (succ graph parsink)) sink >>> λtree =
            //Sequence after merge node found
            ret (SPSeries (SPParallel (source,parsink) trees) tree (patternAfter graph parsink))
        _ = parseError "Merge connector cannot have multiple outgoing connections"

    BTTask = parseError "Task cannot have multiple outgoing connections"
    BTMerge = parseError "Merge unexpected"
```

Finding a matching merge node Given a subgraph which starts with a split node, the `findMerge` function tries to find a matching merge node, by counting the nesting level. Conceptually, this is a similar kind of issue as finding a matching parenthesis in a string containing an expression with nested parenthesis. However, in a parallel composition there are multiple paths leading to the same merge node. We traverse the first path and keep track of the nesting level. A split node increases the level, a merge node decreases the level, and a sequential composition keeps the level equal. As soon as we find a merge node and the level is zero, we're done.

```
findMerge :: GGraph GNode Int → GParseResult GNode
findMerge graph source level = findMerge‘ (branchtype source) where
    aftersource :: [GNode]
    aftersource = succ graph source
    findMerge‘ :: BranchType → GParseResult GNode
    findMerge‘ BTMerge   | level==0 = ret source
    findMerge‘ _         | length aftersource==0 =
        parseError "Could not find matching merge connector"
    findMerge‘ BTSplit = findMerge graph (hd aftersource) (inc level)
    findMerge‘ BTMerge = findMerge graph (hd aftersource) (dec level)
    findMerge‘ BTTask | length aftersource==1 = findMerge graph (hd aftersource) level
    findMerge‘ BTTask = parseError "A task cannot have multiple outgoing connections"
```

7.4 Abstract syntax tree

We define a simple AST as a set of record and algebraic data types which represent an implementation module. The AST contains only the elements needed for expressing Gin workflows, namely modules, type definitions, function definitions and expressions. Expressions can be literals, variables, pre- and infix applications, lambda abstraction, case expressions, tuples, lists and simple (one-generator) list comprehensions. Most of the definitions will be self-explanatory. The definition of types (`GTypeDefinition`) and formal parameters (`GFormalParameter`) is shared with the `GinWorkflow` data structure, as defined in section 6.6. The `Unparsed` data constructor of `AExpression` is used to embed unparsed literal strings of Clean code, which are entered by the user. `AExpression` is a higher order type and includes an `Extension` data constructor, which is explained in the next section.

```
:: AModule = { name        :: AIdentifier
             , definitions :: [ADefinition]
             , types       :: [GTypeDefinition]
             , imports     :: [AImport]
             }

:: AImport :== String

:: ADefinition = { name         :: AIdentifier
                 , formalParams :: [GFormalParameter]
                 , returnType   :: GTypeExpression
                 , body         :: AExpression Void
                 , locals       :: [ADefinition]
                 }

:: AExpression ex =
    Unparsed String
    | Lit String
    | Var AIdentifier
    | App [AExpression ex]
    | AppInfix AIdentifier AFix APrecedence (AExpression ex) (AExpression ex)
    | Lambda APattern (AExpression ex)
    | Case (AExpression ex) [ACaseAlt ex]
    | Tuple [AExpression ex]
    | List [AExpression ex]
    | ListComprehension (AListComprehension ex)
    | Extension ex

:: ACaseAlt ex = CaseAlt APattern (AExpression ex)

:: AListComprehension ex = { output    :: AExpression ex
                           , generator :: AGenerator ex
```

```
                        , guard     :: AExpression ex
                        }

:: AGenerator ex = Generator APattern (AExpression ex)

:: APattern :== String

:: AIdentifier :== String

:: AFix = Infixl | Infixr | Infix
:: APrecedence :== Int
```

7.5 Mapping to abstract syntax tree

Mapping modules The mapping from a top-level `GinWorkflow`, containing a `GModule` type to the abstract syntax tree is simple: the module name and type definitions are mapped one-to-one. From the imported modules, only the name is stored. Each `GDefinition` is mapped to an an `ADefinition` in the AST: name and type are mapped one-to-one. The GDefinitions' body has type `GExpression`, which mapping is described below.

Mapping expressions A Gin `GExpression` is mapped to an `AExpression` in the abstract syntax tree. A `GExpression` consists of a textual Clean expression, a static list, a list comprehension or a graph.

- **Textual Clean expressions** (`GCleanExpression`) are not parsed. Rather, these are literally embedded as `Unparsed` nodes in the AST.
- **Lists and list comprehensions** contain other `GExpressions`, which are mapped recursively. A `GListExpression` is mapped to a `List` in the AST, a `GListComprehensionExpression` to a `ListComprehension`.
- **Graphs** are mapped in two steps. First, the `GGraph` is transformed to an intermediate `SPTree`, as earlier described in section 7.3. The mapping of this `SPTree` is to an `AExpression` is presented below.

Mapping SP-trees The `SPTree` structure contains the same nodes as in an original `GGraph`, however more conveniently arranged as a tree. The mapping `spTreeToAExpression :: SPTree → AExpression Void` is defined as follows:

- **Sequential compositions without a pattern** are always mapped to infix application of the `»|` combinator:
  `SPSequence a b Nothing ⇒`
  `AppInfix "»|" Infixl 1 (spTreeToAExpression a) (spTreeToAExpression b)`
- **Sequential compositions with a pattern** are always mapped to infix application of the `»=` operator with a lambda abstraction:
  `SPSequence a b (Just p) ⇒`
  `AppInfix "»=" Infixl 1 (spTreeToAExpression a) (Lambda p (spTreeToAExpression b))`
- **Task nodes** are mapped according to a *node binding*
- **Parallel compositions** are mapped according to a *parallel binding*. The node binding and parallel bindings are explained below.

Bindings The mapping of Task nodes and parallel compositions to `AExpression`s is defined in a set of *bindings*. For each task node, and for each pair of split and merge connectors, a binding is defined.

The mapping of task nodes (`GNode`) is defined by a `NodeBinding` structure, which contains a `GGeclaration` of the node (section 6.6) and a parameter map, which defines how the node parameters are mapped to an expression:

```
:: NodeBinding = { declaration  :: GDeclaration
                 , parameterMap :: NBParameterMap
                 }
:: NBParameterMap = NBPrefixApp
                  | NBInfixApp AFix APrecedence
                  | NBCustom (AExpression ParameterPosition)
:: ParameterPosition :== Int
```

A node parameter map can be defined in three ways.

- A `NBPrefixApp` mapping results in de task name being applied to all its parameters in succession. Given a task node named t with parameters $a_1 \ldots a_n$, the `NBPrefixApp` mapping results in the expression `App [Var` t `:` $[a_1, \ldots, a_n]$`]`.
- A `NBInfixApp` mapping is used to map a node with two parameters to an infix task combinator. Given a task node named t with parameters a and b, the `NBInfixApp` $fix\ prio$ mapping results in the expression `AppInfix` $t\ fix\ prio\ a\ b$.
- In a `NBCustom` mapping, a custom `AExpression` can be specified. This allows a single task node to map to a complex iTask expression. The `Extension` constructor of an `AExpression` is used to indicate a parameter position. If this mapping is applied, the parameter position is replaced by the actual value of the parameter at the indicated position.

For parallel compositions, *pairs* of split and merge connectors map to a single `AExpression`. This mapping is defined by a `ParallelBinding` structure:

```
:: ParallelBinding = { split          :: GDeclaration
                     , merge          :: GDeclaration
                     , type           :: GTypeExpression
                     , fixedNrBranches :: Maybe Int
                     , parameterMap   :: AExpression PBParameter
                     }

:: PBParameter = PBSplitParameter ParameterPosition
               | PBMergeParameter ParameterPosition
               | PBBranch BranchPosition
               | PBBranchList
               | PBApply ([AExpression Void] [AExpression Void]
                          [(Maybe APattern, AExpression Void)] → AExpression Void)

:: ParameterPosition :== Int
:: BranchPosition :== Int
```

The `ParallelBinding` structure contains the declarations of the split and merge connector and type of the parallel expression. The `fixedNrBranches` parameter may limit a parallel composition to a fixed number of branches. The `parameterMap` allows to specify an `AExpression`. The `Extension` constructor of the `AExpression` is used to embed parameters found in the split connector, the merge connector, a specified branch number or a list of all branches. When the mapping is applied, these parameters are replaced by their actual values.

7.6 Pretty printer

The abstract syntax tree should be pretty printed to Clean source code.

A pretty printer has to apply the same amount of indentation to definitions on the same level. Preferibly, it can also break long lines to make the output human-readable as well. A concise way to implement such a pretty printer is by means of a pretty printing combinator library. Such a library contains combinators for horizontal and vertical composition and indentation. Regrettably, there is no such library available for Clean.

We decided to make a port of Leijens' PPrint library[24] to Clean. PPrint is a Haskell implementation of the pretty printing combinators described by Wadler[56]. The basic building block in PPrint is a document, indicated by the `Doc` type. The `<->` combinator concatenates documents horizontally, `<+>` is similar but adds a space in between, `</>` does vertical concatenation, and the `indent` function indents a document with a specified amount of spaces. `hsep` and `vsep` are the list-versions of `<->` and `</>`, respectively.

Using PPrint, the implementation of the pretty printer is straightforward. To illustrate the principle, the pretty-printing of a `ADefinition` is shown below. The pretty-printing of expressions is implemented in an analogous way. An expression `Unparsed` s is put in parenthesis and inserted literally in the output.

```
printADefinition :: ADefinition -> Doc
printADefinition def =: { ADefinition | name, formalParams, body, locals }
= printADefinitionType def
    </> text name
    <+> if (isEmpty formalParams) empty
        (hsep (map (λfp = text fp.GFormalParameter.name) formalParams) <-> space)
    <-> char '='
    <+> printAExpression body
    </> if (isEmpty locals) empty
        (text "where" </> indent 4 (vsep (map printADefinition locals)))
```

In addition to the abstract syntax tree, the pretty printer also adds a `Start` function to the generated document. This `Start` function writes the top-level `Task` definition def to a dynamic on disk:

```
Start :: *World -> *World
Start world
#(_, world) = writeDynamic "def" dynamic def world
= world
```

The entire document is written to an `.icl` file on disk.

7.7 Clean compiler

The next phase involves compiling the generated `.icl` file. We write a minimal project file (`.prj`) to disk, with the main module set to the generated `.icl` file, and with the environment paths containing the iTask libraries. Since the generated `.icl` makes use of writing dynamics, the dynamic linker is enabled in the project settings.

The project file is compiled by running the Clean IDE with the batch option:

```
CleanIDE.exe --batch-build myworkflow.prj
```

Because of the dynamic linker option, the output is a batch file which launches the dynamic linker.

7.8 Dynamics communication

After compilation, we launch the batch file generated in the compilation process. This will result in the task definition being written to a dynamic: a `.dyn` file on disk.

Assuming that the iTask system is also compiled with the dynamic linker option enabled, the task in the dynamic can be loaded and executed in another workflow, by means of a combinator named `readDynamicTask`:

```
readDynamicTask :: !String → Task (Bool,Task a) | iTask a
```

The `readDynamicTask` combinator has a simple implementation which uses the `readDynamic` function. It takes the filename of the dynamic and yields a task with a tuple: a Boolean indicating whether the loading succeeded and the loaded task itself. The example below shows how to use `readDynamicTask` to load and execute a `Task Void` from a dynamic:

```
runDynamicTask :: !String → Task Void
runDynamicTask s = readDynamicTask s >>= λ(success, t) =
                   if success t (showMessage "loading dynamic " +++ s
                   +++ "failed")
```

Chapter 8

Error handling

User interfaces should help users to reach their goals. During interaction, users may make mistakes. A user interface should

> "offer error prevention and simple error handling so that, ideally, users are prevented from making mistakes and, if they do, they are offered clear and informative instructions to enable them to recover"[44]

The Gin editor cannot, by design, prevent the user from making any modeling mistakes (see chapter 6, design principles). Hence, the editor should offer feedback if the user makes modeling errors. The feedback should be specific, clear and understanable for the user. This helps the user to find the cause of the error and recover from it.

This chapter discusses the architecture and implementation of error handling in the Gin editor.

8.1 Kinds of errors

In the compilation process of Gin diagrams, different kinds of errors may occur:

1. **Parse errors**. The first compilation stage involves transforming the Gin diagram to Clean source code. This fails if the diagram does not have a well-formed structure: either there is no unique source- or sink node, split and merge connectors are not matched correctly, or no path exists between a particular node and the source- and sink nodes.
2. **Compiler errors**. In the second compilation stage, the generated source code is compiled by the Clean compiler. Compilation may fail for several reasons. The editor does not prevent the user from making type errors, so the generated code may fail to typecheck. Furthermore, embedded textual Clean expressions may contain syntax errors. Since these expressions are put literally in the generated source code, an incorrect expression will cause the compilation to fail.

8.2 Error handling

The implementation of this error handling can be divided into a number of subproblems. In case of an error, the Gin applet in the users' web browser needs to highlight the location of the error and show an appropriate message. However, the whole compilation process is performed server side. We need to send the error location and message back to the client in a data structure. Besides, we have to translate the errors, originating in both the parsing and Clean compilation stage, back to the diagram.

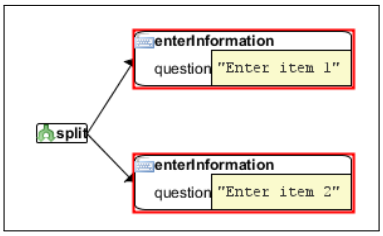

Figure 8.1: Gin editor with erroneous nodes highlighted

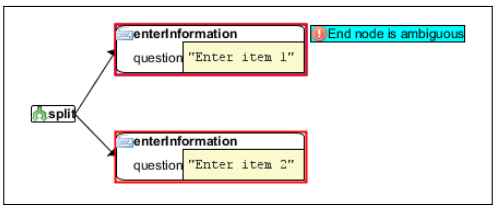

Figure 8.2: Gin editor showing error message pop-up window

8.3 Error paths

A Gin diagram is represented by the `GinWorkflow` data structure described in section 6.6. After each editing operation, the front-end applet sends a new `GinWorkflow` value to the server. The compilation of graphical workflows is implemented server side. If compilation errors occur, the error message and location of the error should be sent back to the client. In order to indicate the location of the error, a method is needed to uniquely identify a position within the `GinWorkflow` structure. Since this structure is essentially a tree structure composed of records and lists of records, we can use a path to uniquely identify elements within this tree. The path starts at top level with a GModule. A record field name is identified by its name, an item in a list of records can be identified by its zero-based index.

A path in the `GinWorkflow` type is defined by:

```
:: GPath = GRoot
         | GChildNode String GPath
         | GChildNodeNr String Int GPath
```

For example, the following path indicates the third node in the body of the first definition:

```
GChildNodeNr "nodes" 2
    (GChildNode "body" (GChildNodeNr "definitions" 0 GRoot))
```

The path is printed to a string in an XPath-like notation: elements are separated by slashes, list indexes are written in square brackets. The above example is printed as `"/definitions[0]/body/nodes[2]/"`

8.4 Visualization of errors

In case of errors, a list of (error path, message) is sent back from the server to the front-end applet. The front-end reads these messages and highlights the corresponding nodes by a red rectangle in the diagram. An example is shown in figure 8.1, which shows a screen shot of the editor. The diagram has no unique end node, so both end nodes are highlighted in red.

If the mouse is moved over the highlighted node, a pop-up window appears which displays the error message, as visible in figure 8.2.

8.5 Handling block detection errors

The block detection algorithm from section 7.3 uses the `GParseResult` error monad to report parse errors. This monad can report an error message, but does not know the path to the erroneous node if an error occurs.

In order to report both full path of an erroneous node, we change the definition of `GParseResult` to return both paths and error messages. Furthermore, we define a new monad named `GParseState`, which combines a state and error monad. This monad keeps track of the current path and aborts the current computation in case of an error.

```
:: GParseResult a = GSuccess a | GError [(GPath, String)]
:: GParseState a :== GPath → GParseResult a
```

For each monadic expression which traverses deeper in the hierarchy of the `GinWorkflow`, the path in the `GParseState` state monad has to be updated. For this reason, we define a set of combinators to update the path during traversal of child nodes, and in case of an error, to report an error message together with the current path. Similar combinators are defined for parsing lists of child nodes and reporting errors in child nodes.

```
//add GChildNode to path, then performs GParseState a
parseChild :: String (GParseState a) → GParseState a
//aborts computation, report current path and error message
parseError :: String → GParseState a
```

We alter the implementation of the block detection function `graphToSPTree` and its helper functions to use the above `GParseState` monad instead of the original `GParseResult` monad. At the traversal of each child node, we apply the `parseChild` combinator, or to map a list of child nodes, the `parseChildMap` combinator. To demonstrate this adjustment, we show the new implementation of `graphToSPTree` here. Its helper functions are adapted in an analogous way.

```
graphToSPTree :: GGraph → GParseState SPTree
graphToSPTree graph
# sources = [ n \\ n←graph.nodes | isEmpty (pred graph n) ]
# sinks   = [ n \\ n←graph.nodes | isEmpty (succ graph n) ]
= case sources of
  [source] = case sinks of
               [sink] = subgraphToTree graph source sink
               []     = parseError "No end node found"
               sinks  = parseErrorInChildren "nodes" sinks "End node is ambiguous"
  []       = parseError "No start node found"
  sources  = parseErrorInChildren "nodes" sources "Start node is ambiguous"
```

8.6 Handling compiler errors

The error messages outputted by the Clean compiler are not meaningful for the Gin user, since these messages relate to the generated source code, and not to the Gin diagram. We will illustrate this by means of an example. Figure 8.3 shows an erroneous Gin diagram. The generated code is shown below.

```
flow ::  Task Void
flow = enterInformation ("Enter amount") >>= λ amount = showMessage ("Amount is" + amount)
```

If we compile the file containing this workflow, we get the following error message:

```
Overloading error [flow.icl,11,\;11;53]:
    "+" no instance available of type {#Char}
Overloading error [flow.icl,10,flow]:
    internal overloading of "showMessage" could not be solved
```

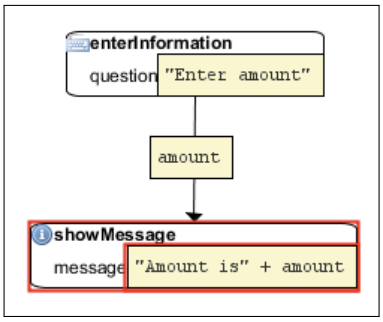

Figure 8.3: Workflow containing overloading error

We need a way to relate the line numbers in the error message back to nodes in the Gin diagram. The problem is that the node locations get lost in the compilation process, during the block detection phase. As soon as a node is put in the `SPTree` structure, we have lost its position in the diagram. Our solution is to keep track of the original node location throughout the compilation process. In order to realize this, we make several changes in the compilation process:

1. We alter the `SPTree` definition to also store the path to each original node in the diagram. The new definition of `SPTree` becomes:

   ```
   :: SPPathNode = SPPathNode GNode GPath
   :: SPTree = SPNode SPPathNode
             | SPSeries SPTree SPTree SPPattern GPath
             | SPParallel (SPPathNode,SPPathNode) [(SPPattern,SPTree)]
   ```

   The block detection function `graphToSPTree` function is updated to include the `GPath` of every node.

2. We add a `PathContext` constructor to the `AExpression` definition in the abstract syntax tree:

   ```
   :: AExpression ex =
       ...
       | PathContext GPath (AExpression ex)
   ```

   The mapping process to the abstract syntax tree is changed as well: For each mapping of an `SPTree`, the resulting `AExpression` is wrapped by the `PathContext` constructor, together with the path.

3. We alter the pretty printer to keep track of a map from line numbers to `GPaths`. Each time a `PathContext` has to be printed, the current line number and `GPath` is added to the map, Next, the expression in the `AExpression` parameter is printed.

4. If the compilation process fails, we parse the error messages from the compiler by means of a simple parser.

   We extract the line numbers from the error messages. Using the map constructed in the previous step, we can relate the line numbers back to the `GPaths` of nodes in the diagram. The `GPaths` and error messages are sent back to client.

Although this approach works *in principle*, it still has a problem. The compilation process may cause a complex Gin diagram to be translated to a very long expression on a single line. Therefore, a single line number may map to a larger set of nodes, many of which may actually be correct!

We tackled this problem by putting each subexpression in a separate **where** clause. We implement a transformation on abstract syntax trees which expands all subexpressions in the abstract syntax tree to separate local definitions in a **where** clause. This makes the code look more cluttered, but it is actually useful: now each subexpression is denoted on a separate line. Therefore, a single line number in a compiler error maps just to a single node, so we can uniquely identify incorrect

nodes in the diagram. The transformation is defined by a function named `expandModule`. We will not elaborate on the details of the algorithm, but illustrate its operation by means of an example. We show the code corresponding to figure 8.3 before and after the transformation.

Before expanding subexpressions, the generated code looks as follows. Note that the arguments of `enterInformation` and `showMessage` are on the same line. Hence, we cannot derive from the line number which node in the Gin diagram is incorrect.

```
flow :: Task Void
flow = enterInformation ("Enter amount") >>= λ amount = showMessage ("Amount is" + amount)
```

After expanding subexpressions, the same workflow looks like this:

```
flow :: Task Void
flow = v2 >>= λ amount = v4 amount
where
   v1 = ("Enter amount")
   v2 = enterInformation v1
   v3 amount = ("Amount is" + amount)
   v4 amount = showMessage (v3 amount)
```

The compiler reports an error in the line which starts with `v3`. This line maps to a unique node, namely the parameter of the showMessage argument. By means of the subexpression expansion, we are able to transform error messages back to individual nodes in the diagram.

Chapter 9

Conclusions

9.1 Conclusions

Workflow management systems (WFMSs) are systems for coordinating business processes, based on a workflow model. Workflow models are expressed in a workflow definition language (WDL). Most WDLs have a graphical nature. iTask is a WFMS which uses an embedded domain specific language as its WDL. This language is based on a combinator library, embedded in the Clean functional programming language. While this approach brings the expressive power of the Clean to workflow models, it simultaneously puts up a barrier for users not familiar with functional programming.

We assumed that fon non-technical users, a graphical WDL may be easier to use than a textual one. Although there is no unambiguous evidence to support this claim, this user group may be familiar with existing, mostly graphical, WDLs. Therefore, our research intended to investigate the expression of iTask workflows in a graphical notation.

First, we studied the field of existing graphical WDLs. A structured way of comparing WDLs is by means of workflow patterns. A workflow pattern describes a business requirement. Most workflow patterns are implemented by specific graphical elements. We analyzed four WDLs, namely workflow nets, YAWL, EPCs and UML activity diagrams. Broadly speaking, these WDLs use similar notations.

Using the same set of workflow patterns, we made an analysis of the iTask WDL. This served two purposes. First, it substantiated the claim in earlier work on iTask that it supports almost all original workflow patterns. Apart from patterns that assume a lack of synchronization, this claim turned out to be true. Second, it provided us a handle to structurally compare the graphical WDLs to iTask.

Starting from the iTask combinators, we defined a new notation named *Gin*: *Graphical iTask Notation*. Gin resembles the existing graphical WDLs where possible.

For common iTask constructs which do not have a direct equivalent in graphical WDLs, we introduced a new notation. However, we chose to limit these constructs to what is necessary for expressing workflows: Gin is not intended as a general purpose visual programming language.

In order to prove that workflows expressed in Gin are executable, we have implemented a proof of concept editor for Gin diagrams. The editor is integrated in the iTask system. This is realized by means of specialization of the generic functions which generate the iTask user interface. A compilation process transforms correct Gin diagrams into executable tasks.

9.2 Future work

The research on the graphical notation for iTask opens up a range of new questions, of which we hope they will be answered in the future.

Given that the Gin notation and implementation is now available, it would be useful to perform an empirical study on the usability of Gin. Only in this way, we can substantiate our assumption that a graphical WDL is easier to use for non-technical users than a textual one.

Gin currently supports only structured workflows. Given the debate on structured versus unstructured modeling in the workflow literature, it would be interesting to research if Gin can be extended to support unstructured workflows as well.

Graphical front-ends like Gin make good use cases for accessing the compiler via an API. An API for the Clean compiler would enable a much simpler implementation of the iTask compilation process: we could pass generated abstract syntax tree directly to the compiler, instead of having to print the ASTs, write source files which are read and parsed again by the compiler. Besides, compiler errors can be passed in type-safe way.

Finally, a graphical tool like Gin could be extended to assist in handling *running* workflows. Once a modeled workflow has started, it would be useful to monitor its current status in a diagram, and make changes in a graphical and type-safe way.

Bibliography

[1] Gaudenz Alder. Jgraph. `http://www.jgraph.com/jgraph5.html`.

[2] Hyerim Bae, Joonsoo Bae, Suk-Ho Kang, and Yeongho Kim. Automatic control of workflow processes using eca rules. *IEEE Trans. on Knowl. and Data Eng.*, 16(8):1010–1023, 2004.

[3] Adam Barker and Jano Van Hemert. Scientific workflow: a survey and research directions. In *PPAM'07: Proceedings of the 7th international conference on Parallel processing and applied mathematics*, pages 746–753, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] S. Clerici and C. Zoltan. Nets in motion. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5, Bristol, UK, 2004. Intellect.

[5] G. Costagliola, V. Deufemia, and G. Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4):431–487, October 2004.

[6] Edsger W. Dijkstra. *Chapter I: Notes on structured programming.* Academic Press Ltd., London, UK, UK, 1972.

[7] M. Dumas and A.H.M. ter Hofstede. Uml activity diagrams as a workflow specification language. In *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[8] M. Dumas, W.M.P. van der Aalst, and A. H. M. Ter Hofstede, editors. *Process Aware Information Systems - Bridging people and software through process technology.* Wiley- Interscience, 2005.

[9] Eclipse Foundation. Eclipse graphical modeling framework. `http://www.eclipse.org/gmf`.

[10] T.R.G. Green and M. Petre. When visual programs are harder to read than textual programs. In I G.C. van der Veer, M.J. Tauber, S. Bagnara, and M. Antalovits, editors, *Proceedings of the Sixth European Conference on Cognitive Ergonomics (ECCE 6)*, volume 6, pages 167–180, 1992.

[11] Object Management Group. The unified modeling language. `http://www.uml.org`.

[12] V. Gruhn and R. Laue. Good and bad excuses for unstructured business process models. In *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2007.

[13] K. Hanna. A document-centered environment for Haskell. In A. Butterfield, C. Grelck, and F. Huch, editors, *IFL 2005*, volume 4015, Heidelberg, 2006. LNCS.

[14] A. Holl and G. Valentin. Structured business process modeling (sbpm). *Information Systems Research in Scandinavia (IRIS 27) (CD-ROM)*, 2004.

[15] Jan Martin Jansen, Rinus Plasmeijer, and Pieter Koopman. iEditors : extending iTask with interactive plug-ins. In S.B. Scholz, editor, *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 170–186. University of Herfordshire, 2008.

[16] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische proceßmodellierung auf der grundlage "ereignisgesteurter proceßketten (epk)". In *Veröffentlichungen des Instituts für Wirtshartsinformatik (IWi), Universität des Saarlands*, volume Heft 89. Universität des Saarlands, January 1992.

[17] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Model ling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, 2003.

[18] Bartek Kiepuszewski, Arthur ter Hofstede, and Christoph Bussler. On structured workflow modelling. *Advanced Information Systems Engineering*, 1789:431–445, 2000.

[19] E. Kindler. On the semantics of epcs: A framework for resolving the vicious circle. In *Business Process Management*, volume 3080 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag, June 2004.

[20] James D. Kiper, Brent Auernheimer, and Charles K Ames. Visual depiction of decision statements: What is best for programmers and non-programmers? *Empirical Software Engineering*, 2(4):361–379, 1997.

[21] Oliver Kopp, Daniel Martin, Daniel Wutke, and Frank Leymann. The difference between graph-based and block-structured business process modelling languages. *Enterprise Modelling and Information Systems Architectures*, 4(1):3–13, 2009.

[22] 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, 1 1988.

[23] P. Lawrence. *The Workflow Handbook*. John Wiley and Sons, 1997.

[24] Daan Leijen. PPrint, a prettier printer. `http://research.microsoft.com/en-us/um/people/daan/pprint.html`, 2001.

[25] Bas Lijnse and Rinus Plasmeijer. itasks 2: itasks for end users. In Marco Morazan, editor, *Revised Papers of the 21st International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA. To appear*, 2010.

[26] J. Mendling, G. Neumann, and M. Nüttgens. Towards workflow pattern support of event-driven process chains (epc. In *Proc. of the 2nd Workshop XML4BPM 2005*, pages 23–38, 2005.

[27] Microsoft Corporation. Visual studio visualization and modeling sdk. `http://code.msdn.microsoft.com/vsvmsdk`.

[28] T. Moher, D.C. Mak, C. Blumenthal, and L.M. Leventhal. Comparing the comprehensibility of textual and graphical programs: the case of petri nets. In C. R. Cook, J. C. Schotz, and J. C. Spohner, editors, *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161, 1993.

[29] Object Management Group. Object constraint language. `http://www.omg.org/spec/OCL/2.2`.

[30] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[31] Piccolo2D Developers. Piccolo2d - a structured 2d graphics framework. `http://www.piccolo2d.org`.

[32] Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98)*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.

[33] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to iTasks: Defining interactive workflows for the web. In *Central European Functional Programming School, Revised Selected Lectures, CEFP 2007*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, June 23-30 2007. Springer.

[34] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: Executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 141–152, Freiburg, Germany, Oct 1-3 2007. ACM.

[35] Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten. Using a functional language as embedding modeling language for web-based workflow applications. Under submission to MODELS2009, 2009.

[36] Rinus Plasmeijer and Marko van Eekelen. Clean 2.1 language report. Technical report, Hilt - High Level Software Tools B.V., 2001.

[37] Rinus Plasmeijer and Arjen van Weelden. A functional shell that operates on typed and compiled applications. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 245–272, Tartu, Estonia, 14-21, August 2004. Springer.

[38] Manfred Reichert and Peter Dadam. Adeptflex: Supporting dynamic changes of workflow without loosing control. *Journal of Intelligent Information Systems*, 10:93–129, 1998.

[39] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[40] N. Russell and W.M.P. van der Aalst. newyawl: specifying a workflow reference language using couloured petri nets. In *Proceedings of the 8th International Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'07)*, 2007.

[41] N. Russell, W.M.P. van der Aalst, A. H. M. Ter Hofstede, and P. Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. In *APCCM '06: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*, pages 95–104, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[42] Sencha. Ext js. `http://www.sencha.com/products/js`.

[43] A. Sharp and P. McDermott. *Workflow Modeling: Tools for Process Improvement and Application Development*. Artech House Publishers, 2nd edition edition, 2008.

[44] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[45] W. M. P. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004.

[46] W.M.P. van der Aalst. The application of petri nets to workflow management. *The Journl of Circuits, Systems and Computers*, 81(1):21–66, 1998.

[47] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Inform. Software Technol*, 41(10):639–650, 1999.

[48] W.M.P. van der Aalst. Patterns and xpdl: A critical evaluation of the xml process definition language. QUT Technical report FIT-TR-2003-06, Queensland University of Technology, Brisbane, Australia, March 2003.

[49] W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of epcs: A vicious circle. *EPK 2002, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, pages 71–79, 2002.

[50] W.M.P. van der Aalst et al. Workflow patterns web site. http://www.workflowpatterns.com.

[51] W.M.P. van der Aalst and A. H. M. Ter Hofstede. Yawl: Yet another workflow language. QUT Technical report FIT-TR-2002-06, Queensland University of Technology, Brisbane, 2002.

[52] W.M.P. van der Aalst and M.Mulyar. Workflow control-flow patterns : A revised view. Technical Report BPM-06-22, BPMCenter.org, 2006.

[53] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20,. University of Aarhus, August 2002.

[54] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. QUT Technical report FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.

[55] W.M.P. van der Aalst and K. van Hee. *Workflow Management - Models, Methods and Systems*. The MIT Press, 2004.

[56] Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

[57] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8:109–142, 1996.

[58] P. Wohed, W.M.P. van der Aalst, M. Dumas, and A. H. M. Ter Hofstede. Pattern-based analysis of uml activity diagrams. BETA Working Paper Series WP 129, Eindhoven University of Technology, 2004.

[59] P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Pattern-based analysis of bpel4ws. QUT Technical report FIT-TR-2002-04, Queens- land University of Technology, Brisbane, 2002.

[60] Petia Wohed, Nick Russell, Arthur H. M. ter Hofstede, Birger Andersson, and Wil M. P. van der Aalst. Patterns-based evaluation of open source bpm systems: The cases of jbpm, openwfe, and enhydra shark. *Inf. Softw. Technol.*, 51(8):1187–1216, 2009.

[61] World Wide Web Consortium (W3C). Scalable vector graphics (svg) 1.1 specification. `http://www.w3.org/TR/2003/REC-SVG11-20030114`.