Domain-Specific Languages for Enterprise Systems

Jesper Andersen[2], Patrick Bahr[1], Fritz Henglein[1], and Tom Hvitved[1]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark
{bahr,henglein,hvitved}@diku.dk
[2] Configit A/S, Kristianiagade 7, 2100 Copenhagen, Denmark
ja@configit.com

Abstract. The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. is a novel software architecture for enterprise resource planning (ERP) systems. POETS employs a pragmatic separation between (i) transactional data, that is, what has happened; (ii) reports, that is, what can be derived from the transactional data; and (iii) contracts, that is, which transactions are expected in the future. Moreover, POETS applies domain-specific languages (DSLs) for specifying reports and contracts, in order to enable succinct declarative specifications as well as rapid adaptability and customisation. In this paper we present an implementation of a generalised and extended variant of the POETS architecture. The extensions amount to a customisable data model based on nominal subtyping; support for run-time changes to the data model, reports and contracts, while retaining full auditability; and support for referable data that may evolve over time, also while retaining full auditability as well as referential integrity. Besides the revised architecture, we present the DSLs used to specify data definitions, reports, and contracts respectively. Finally, we illustrate a use case scenario, which we implemented in a trial for a small business.

1 Introduction

Enterprise Resource Planning (ERP) systems are comprehensive software systems used to integrate and manage business activities in enterprises. Such activities include—but are not limited to—financial management (accounting), production planning, supply chain management and customer relationship management. ERP systems emerged as a remedy to heterogeneous systems, in which data and functionality are spread out—and duplicated—amongst dedicated subsystems. Instead, an ERP system it built around a central database, which stores all information in one place.

Traditional ERP systems such as Microsoft Dynamics NAV[1], Microsoft Dynamics AX[2], and SAP[3] are three-tier architectures with a client, an application

[1] http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx.
[2] http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx.
[3] http://www.sap.com.

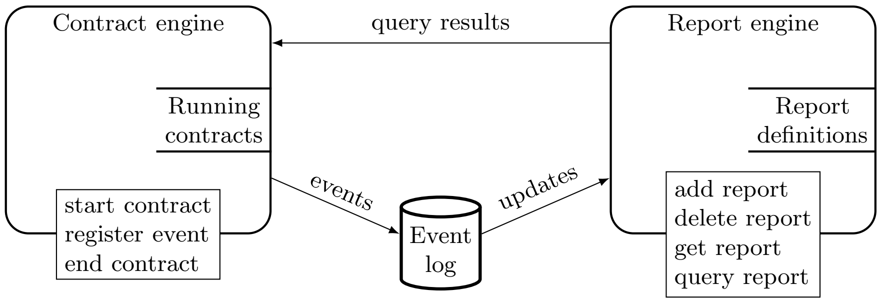

Fig. 1. Bird's-eye view of the POETS architecture (diagram copied from [6])

server, and a centralised relational database system. The central database stores information in tables, and the application server provides the business logic, typically coded in a general purpose, imperative programming language.

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. [6] is a qualitatively different approach to ERP systems. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between transactional data, which is persisted in an *event log*, and *contracts*, which are explicit representations of business processes, stored in a separate module. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises a declarative domain-specific language (DSL) [1]. The use of a DSL not only enables compositional construction of formalised business processes, it minimises the semantic gap between requirements and a running system, and it facilitates treating processes as data for analysis. Henglein et al. take it as a goal of POETS that "[...] the formalized requirements *are* the system" [6, page 382].

The bird's-eye view of the POETS architecture is presented in Figure 1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent "things that take place" such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items in an inventory. The append-only restriction serves two purposes. First, it is a legal requirement in ERP systems that transactions, which are relevant for auditing, are retained. Second, the report engine utilises monotonicity of the event log for optimisation, as shown by Nissen and Larsen [19].

Besides the radically different software architecture, POETS distinguishes itself from existing ERP systems by abandoning the double-entry bookkeeping (DEB) accounting principle [28] in favour of the Resources, Events, and Agents (REA) accounting model of McCarthy [13].

1.1 Outline and Contributions

The motivation for our work is to assess the POETS architecture in terms of a prototype implementation. During the implementation process we have added features for dynamically managing values and entities to the original architecture. Moreover, in the process we found that the architecture need not be tied to the REA ontology—indeed to ERP systems—but can be viewed as a discrete event modelling framework. Its adequacy for other domains remains future research, however.

Our contributions are as follows:

- We present a generalised and extended POETS architecture (Section 2) that has been fully implemented.
- We present domain-specific languages for data modelling (Section 2.1), report specification (Section 2.4), and contract specification (Section 2.5).
- We illustrate small use case that we have implemented in our system as part of a trial for a small business (Section 3).

The POETS server system has been implemented in Haskell. Its client code has been developed in Java, primarily for Android. The choice of Haskell, specifically the Glasgow Haskell Compiler (GHC), is due to: the conciseness, affinity and support of functional programming for enterprise software [14] and declarative DSL implementation; its expressive type system, which supports statically typed solutions to the Expression Problem [3,2]; and competitive run-time performance due to advanced compiler optimisations in GHC. The use of Java on the client side (not further discussed in this paper) arises from POETS, conceived to be cloud-based and mobile from the outset, targeting low-cost mobile devices and a practical desire to reuse code as much as possible across smartphones, tablets, portables and desktops.

The source code of this implementation is available from the repository at https://bitbucket.org/jespera/poets/. In addition, the repository also includes the full source code for the use case presented in Section 3.

2 Revised POETS Architecture

Our generalised and extended architecture is presented in Figure 2. Compared to the original architecture in Figure 1, the revised architecture sees the addition of three new components: a *data model*, an *entity store*, and a *rule engine*. The rule engine is currently not implemented, and we will therefore not return to this module until Section 4.2.

As in the original POETS architecture, the event log is at the heart of the system. However, in the revised architecture the event log plays an even greater role, as it is the *only* persistent state of the system. This means that the states of all other modules are also persisted in the event log, hence the flow of information from all other modules to the event log in Figure 2. For example, whenever a contract is started or a new report is added to the system, then an event reflecting this operation is persisted in the event log. This, in turn, means that the state of each module can—in principle—be derived from the event log. However, for performance reasons each module—including the event log—maintains its own state in memory.

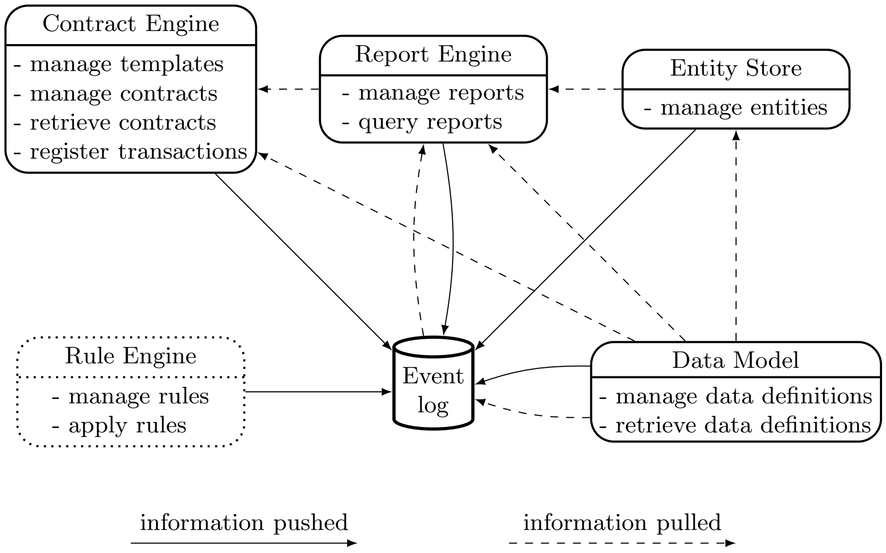

Fig. 2. Bird's-eye view of the generalised and extended POETS architecture

Function	Input	Output
addDataDefs	ontology specification	
getRecordDef	record name	type definition
getSubTypes	record name	list of record names

Fig. 3. Data model interface

We describe each module of the revised architecture in the following subsections. Since we will focus on the revised architecture in the remainder of the text, we will refer to said architecture simply as POETS.

2.1 Data Model

The data model is a core component of the extended architecture, and the interface it provides is summarised in Figure 3. The data model defines the *types* of data that are used throughout the system, and it includes predefined types such as events. Custom types such as invoices can be added to the data model at run-time via *addDataDefs*. For simplicity we currently only allow addition of types, not updates and deletions, which can be supported by suitable namespace management.

Types define the structure of the data in a running POETS instance manifested as *values*. A value—such as a concrete invoice—is an instance of the data specified by a type. Values are not only communicated between the system and its environment but they are also stored in the event log, which is simply a list of values of a certain type.

Types. Structural data such as payments and invoices are represented as *records*, that is, typed finite mappings from field labels to values. Record types define the structure of such records by listing the constituent field labels and their associated types. In order to form a hierarchical ontology of record types, we use a nominal subtyping system [22]. That is, each record type has a unique name, and one type is a subtype of another if and only if stated so explicitly or by transitivity. For instance, a customer can be defined as a subtype of a person, which means that a customer contains all the data of a person, similar to inheritance in object oriented programming.

The choice of nominal types over structural types [22] is justified by the domain: the nominal type associated with a record may have a semantic impact. For instance, the type of customers and premium customers may be structurally equal, but a value of one type is considered different from the other, and clients of the system may for example choose to render them differently. Moreover, the purpose of the rule engine, which we return to in Section 4.2, is to define rules for values of a particular semantic domain, such as invoices. Hence it is wrong to apply these rules to data that happens to have the same structure as invoices. Although we use nominal types to classify data, the DSLs support full record polymorphism [20] in order to minimise code duplication. That is, it is possible for instance to use the same piece of code with customers and premium customers, even if they are not related in the subtyping hierarchy.

The grammar for types is as follows:

$$\begin{array}{rll} T ::= & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \mathbf{String} \mid \mathbf{Timestamp} \mid \mathbf{Duration} & \text{(type constants)} \\ \mid & \mathit{RecordName} & \text{(record type)} \\ \mid & [T] & \text{(list type)} \\ \mid & \langle \mathit{RecordName} \rangle & \text{(entity type)} \end{array}$$

Type constants are standard types Booleans, integers, reals, and strings, and less standard types timestamps (absolute time) and durations (relative time). Record types are named types, and the record typing environment—which we will describe shortly—defines the structure of records. For record types we assume a set $\mathit{RecordName} = \{\mathsf{Customer}, \mathsf{Address}, \mathsf{Invoice}, \dots\}$ of record names ranged over by r. Concrete record types are typeset in sans-serif and begin with a capital letter. Likewise, we assume a set *FieldName* of all field names ranged over by f. Concrete field names are typeset in sans-serif beginning with a lower-case letter.

List types $[\tau]$ represent lists of values, where each element has type τ, and it is the only collection type currently supported. Entity types $\langle r \rangle$ represent entity values that have associated data of type r. For instance, if the record type Customer describes the data of a customer, then a value of type $\langle \mathsf{Customer} \rangle$ is a (unique) customer entity, whose associated Customer data may evolve over time. The type system ensures that a value of an entity type will have associated data of the given type, similar to referential integrity in database systems [4]. We will return to how entities are created and modified in Section 2.3.

All data are type checked before they enter the system, both in order to check that record values conform with the record typing environment, but also to check that entity values have valid associated data. In particular, events are type checked before they are persisted in the event log. We will explain what this means in detail in Section 2.2 and 2.3. The typing judgement has the form $\mathcal{R}, \mathcal{E} \vdash v : \tau$, where $\mathcal{R}$ is a record typing environment, which contains record type definitions, $\mathcal{E}$ is an entity typing environment, which maps each defined entity to its declared type, v is a value, and τ is a type. Both $\mathcal{R}$ and $\mathcal{E}$ are given by the data model and the entity store, respectively. The POETS system has a type checker that checks whether a value v has type τ in the context of $\mathcal{R}$ and $\mathcal{E}$.

Ontology Language. In order to specify record types, we use a variant of Attempto Controlled English [5] due to Jønsson Thomsen [10], referred to as the *ontology language*. The approach is to define data types in near-English text, in order to minimise the gap between requirements and specification. A simple example in the ontology language is given below:

Person is abstract.
Person has a String called name.

Customer is a Person.
Customer has an Address.

Address has a String called road.
Address has an Int called no.

Predefined Ontology. Unlike the original POETS architecture [6], our generalised architecture is not fixed to an enterprise resource planning (ERP) domain. However, we require a set of predefined record types.

The predefined ontology defines five root concepts in the data model, that is, record types maximal with respect to the subtyping relation. Each of these five root concepts Data, Event, Transaction, Report, and Contract are abstract and only Event and Contract define record fields. Custom data definitions added via *addDataDefs* are only permitted as subtypes of Data, Transaction, Report, and Contract. In contrast to that, Event has a predefined and fixed hierarchy.

Data types represent elements in the domain of the system such as customers, items, and resources.

Transaction types represent events that are associated with a contract, such as payments, deliveries, and issuing of invoices.

Report types are result types of report functions, that is, the data of reports, such as inventory status, income statement, and list of customers. The Report structure does not define *how* reports are computed, only *what kind* of result is computed. We will describe the report engine in Section 2.4.

Contract types represent the different kinds of contracts, such as sales, purchases, and manufacturing procedures. Similar to Report, the structure does not define what the contract dictates, only what is required to instantiate the contract. The purpose of Contract is hence dual to the purpose of Report: the former determines an input type, and the latter determines an output type. We will return to contracts in Section 2.5.

Event types form a fixed hierarchy and represent events that are logged in the system. Events are conceptually separated into *internal* events and *external* events, which we describe further in the following section.

2.2 Event Log

The event log is the only persistent state of the system, and it describes the complete state of a running POETS instance. The event log is an append-only list of records of the type Event. Each event reflects an atomic interaction with the running system. This approach is also applied at the "meta level" of POETS: in order to allow agile evolution of a running POETS instance, changes to the data model, reports, and contracts are reflected in the event log as well.

The monotonic nature of the event log—data is never overwritten or deleted from the system—means that the state of the system can be reconstructed at any previous point in time. In particular, transactions are never deleted, which is a legal requirement for ERP systems. The only component of the architecture that reads directly from the event log is the report engine (compare Figure 2), hence the only way to access data in the log is via a report.

All events are equipped with an internal timestamp (internalTimeStamp), the time at which the event is registered in the system. Therefore, the event log is always monotonically decreasing with respect to internal timestamps, as the newest event is at the head of the list. Conceptually, events are divided into *external* and *internal* events.

External events are events that are associated with a contract, and only the contract engine writes external events to the event log. The event type TransactionEvent models external events, and it consists of three parts: (i) a contract identifier (contractId), (ii) a timestamp (timeStamp), and (iii) a transaction (transaction). The identifier associates the external event with a contract, and the timestamp represents the time at which the external event takes place. Note that the timestamp need not coincide with the internal timestamp. For instance, a payment in a sales contract may be registered in the system the day after it takes place. There is hence no a priori guarantee that external events have decreasing timestamps in the event log—only external events that pertain to the same contract are required to have decreasing timestamps. The last component, transaction, represents the actual action that takes place, such as a payment from one person or company to another. The transaction is a record of type Transaction, for which the system makes no assumptions.

Internal events reflect changes in the state of the system at a meta level. This is the case for example when a contract is instantiated or when a new record definition is added. Internal events are represented by the remaining subtypes of the Event record type. Figure 4 provides an overview of all non-abstract record types that represent internal events.

A common pattern for internal events is to have three event types to represent creation, update, and deletion of respective components. For instance, when a report is added to the report engine, a CreateReport event is persisted to the log, and when it is updated or deleted, UpdateReport and DeleteReport events

Event	**Description**
AddDataDefs	A set of data definitions is added to the system. The field defs contains the ontology language specification.
CreateEntity	An entity is created. The field data contains the data associated with the entity, the field recordType contains the string representation of the declared type, and the field ent contains the newly created entity value.
UpdateEntity	The data associated with an entity is updated.
DeleteEntity	An entity is deleted.
CreateReport	A report is created. The field code contains the specification of the report, and the fields description and tags are meta data.
UpdateReport	A report is updated.
DeleteReport	A report is deleted.
CreateContractDef	A contract template is created. The field code contains the specification of the contract template, and the fields recordType and description are meta data.
UpdateContractDef	A contract template is updated.
DeleteContractDef	A contract template is deleted.
CreateContract	A contract is instantiated. The field contractId contains the newly created identifier of the contract and the field contract contains the name of the contract template to instantiate, as well as data needed to instantiate the contract template.
UpdateContract	A contract is updated.
ConcludeContract	A contract is concluded.

Fig. 4. Internal events

are persisted accordingly. This means that previous versions of the report specification can be retrieved, and more generally that the system can be restarted simply by replaying the events that are persisted in the log on an initially empty system. Another benefit to the approach is that the report engine, for instance, does not need to provide built-in functionality to retrieve, say, the list of all reports added within the last month—such a list can instead be computed as a report itself!

Since we allow the data model of the system to evolve over time, we must be careful to ensure that the event log, and thus all data in it, remains well-typed at any point in time. Let $\mathcal{R}_t$, $\mathcal{E}_t$, and l_t denote the record typing environment, entity typing environment, and event log, respectively at time t. Since an entity might be deleted over time, and thus is removed from the entity typing environment, the event log may not be well-typed with respect to the current entity typing environment. To this end, we type the event log with respect to the *accumulated entity typing environment* $\widehat{\mathcal{E}_t} = \bigcup_{t' \leq t} \mathcal{E}_{t'}$ at time t. That is, $\widehat{\mathcal{E}_t}(e) = r$ iff there is some time $t' \leq t$ with $\mathcal{E}_{t'}(e) = r$. The *stable type* invariant, which we will discuss in Section 2.3, guarantees that $\widehat{\mathcal{E}_t}$ is indeed well-defined.

Entity Store

Function	**Input**	**Output**
createEntity	record name, record	entity
updateEntity	entity, record	
deleteEntity	entity	

Fig. 5. Entity store interface

For changes to the record typing environment, we require the following invariants for any points in time t, t' and the event log l_t at time t:

$$\text{if } t' \leq t \text{ then } \mathcal{R}_{t'} \subseteq \mathcal{R}_t, \text{ and} \qquad \text{(monotonicity)}$$

$$\mathcal{R}_t, \widehat{\mathcal{E}_t} \vdash l_t : [\mathsf{Event}]\,. \qquad \text{(log typing)}$$

Note that the *log typing* invariant follows from the *monotonicity* invariant and the type checking $\mathcal{R}_t, \mathcal{E}_t \vdash e : \mathsf{Event}$ for each new incoming event, provided that for each record name r occurring in the event log, no additional record fields are added to r, and r is not made an abstract record type. We will refer to the two invariants above collectively as *record typing invariants*. They will become crucial in the following section.

2.3 Entity Store

The entity store provides very simple functionality, namely creation, deletion and updating of entities, respectively. To this end, the entity store maintains an entity environment ϵ_t that maps each defined entity e to its value $\epsilon_t(e)$. In addition, the entity store also maintains a compact representation of the history of entity environments $\epsilon_0, \ldots, \epsilon_t$. The interface of the entity store is summarised in Figure 5.

In order to type check entities, the entity store also maintains an *entity typing environment* $\mathcal{E}_t$, that is, a finite partial mapping from entities to record names. Intuitively, an entity typing environment maps an entity to the record type that it has been declared to have upon creation.

The entity store checks a number of invariants that ensure the integrity of the system. Specifically, the entity store ensures the following invariants, where we use the notation $\mathcal{E}_t$, $\mathcal{R}_t$ and ϵ_t, for the entity typing environment, the record typing environment, and the entity environment, respectively at time t:

$$\text{if } \mathcal{E}_t(e) = r \text{ and } \mathcal{E}_{t'}(e) = r', \text{ then } r = r', \qquad \text{(stable type)}$$

$$\text{if } \mathcal{E}_t(e) \text{ is defined, then so is } \epsilon_t(e), \text{ and} \qquad \text{(well-definedness)}$$

$$\text{if } \epsilon_t(e) = v, \text{ then } \mathcal{E}_t(e) = r \text{ and } \mathcal{R}_{t'}, \mathcal{E}_{t'} \vdash v : r \text{ for some } t' \leq t. \qquad \text{(well-typing)}$$

We refer to the three invariants above collectively as the *entity integrity invariants*. The *stable type* invariant states that each entity can have at most one declared type throughout its lifetime. The *well-definedness* invariant guarantees that every entity that is given a type also has an associated record value. Finally, the *well-typing* invariant guarantees that the record value associated with an entity *was* well-typed at some earlier point in time t'.

The creation of a new entity via *createEntity* at time $t+1$ requires a declared type r and an initial record value v, and it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : r$. If the value type checks, a *fresh* entity value $e \notin \bigcup_{t' \leq t} \mathrm{dom}(\epsilon_{t'})$ is created, and the entity environment and the entity typing environment are updated accordingly:

$$\epsilon_{t+1}(x) = \begin{cases} v & \text{if } x = e, \\ \epsilon_t(x) & \text{otherwise,} \end{cases} \qquad \mathcal{E}_{t+1}(x) = \begin{cases} r & \text{if } x = e, \\ \mathcal{E}_t(x) & \text{otherwise.} \end{cases}$$

Moreover, a CreateEntity event is persisted to the event log containing e, r, and v for the relevant fields.

Similarly, if the data associated with an entity e is updated to the value v at time $t+1$, then it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : \mathcal{E}_t(e)$, and the entity store is updated like above. Note that the entity typing environment is unchanged, that is, $\mathcal{E}_{t+1} = \mathcal{E}_t$. A corresponding UpdateEntity event is persisted to the event log containing e and v for the relevant fields.

Finally, if an entity e is deleted at time $t+1$, then it is removed from both the entity store and the entity typing environment:

$$\epsilon_{t+1}(x) = \epsilon_t(x) \text{ iff } x \in \mathrm{dom}(\epsilon_t) \setminus \{e\}$$
$$\mathcal{E}_{t+1}(x) = \mathcal{E}_t(x) \text{ iff } x \in \mathrm{dom}(\mathcal{E}_t) \setminus \{e\} \, .$$

A corresponding DeleteEntity event is persisted to the event log containing e for the relevant field.

Note that, by default, $\epsilon_{t+1} = \epsilon_t$ and $\mathcal{E}_{t+1} = \mathcal{E}_t$, unless one of the situations above apply. It is straightforward to show that the *entity integrity invariants* are maintained by the operations described above (the proof follows by induction on the timestamp t). Internally, that is, for the report engine compare Figure 2, the entity store provides a lookup function $\mathrm{lookup}_t : \mathit{Ent} \times [0, t] \rightharpoonup_{\mathrm{fin}} \mathit{Record}$, where $\mathrm{lookup}_t(e, t')$ provides the latest value associated with the entity e at time t', where t is the current time. Note that this includes the case in which e has been deleted at or before time t'. In that case, the value associated with e just before the deletion is returned. Formally, lookup_t is defined in terms of the entity environments as follows:

$$\mathrm{lookup}_t(e, t_1) = v \text{ iff } \exists t_2 \leq t_1 \colon \epsilon_{t_2}(e) = v \text{ and } \forall t_2 < t_3 \leq t_1 \colon e \notin \mathrm{dom}(\epsilon_{t_3}).$$

In particular, we have that if $e \in \mathrm{dom}(\epsilon_{t_1})$, then $\mathrm{lookup}_t(e, t_1) = \epsilon_{t_1}(e)$.

From this definition and the invariants of the system, we can derive the following fundamental safety property for the entity store:

Proposition 1. *Given timestamps $t \leq t_1 \leq t_2$ and entity e, the following holds:*

If $\mathcal{R}_t, \widehat{\mathcal{E}_t} \vdash e : \langle r \rangle$, *then* $\mathrm{lookup}_{t_2}(e, t_1) = v$ *for some* v *and* $\mathcal{R}_{t_2}, \widehat{\mathcal{E}}_{t_2} \vdash v : r$.

Report Engine

Function	**Input**	**Output**
addReport	name, type, description, tags, report definition	
updateReport	name, type, description, tags, report definition	
deleteReport	name	
queryReport	name, list of values	value

Fig. 6. Report engine interface

That is, if an entity value previously entered the system, and hence type checked, then all future dereferencing will not get stuck, and the obtained value will be well-typed with respect to the accumulated entity typing environment.

2.4 Report Engine

The purpose of the report engine is to provide user-definable views, called *reports*, of the system's event log.[4] Conceptually, a report is compiled from the event log by a *report function*, a function of type $[\mathsf{Event}] \to \mathsf{Report}$. The *report language* provides a means to specify such a report function in a declarative manner. The interface of the report engine is summarised in Figure 6.

The Report Language. The report language is—much like the query fragment of *SQL*—a functional language *without side effects.* It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log, the report language must provide operations to relate, filter, join, and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous—containing data of different kinds—the report language offers a comprehensive type system that allows us to safely operate in this setting.

The report language is based on the simply typed lambda calculus extended with polymorphic (non-recursive) let expressions as well as type case expressions. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}\,,$$

where x ranges over variables, and c over constants which include integers, Booleans, tuples and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we assume a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. However, the full language provides syntactic sugar to express operations on lists more intuitively in the form of list comprehensions [26].

[4] The term "report" often conflates the data computed and their visual rendering; here "report" denotes only the computed data.

The extended list comprehensions of the report language also allows the programmer to filter according to run-time type information, which builds on type case expressions of the form **type** $x = e$ **of** $\{r \to e_1; \_ \to e_2\}$ in the core language. In such a type case expression, an expression e of some record type r_e gets evaluated to record value v which is then bound to a variable x. The record type r that the record value v is matched against can be any subtype of r_e. Further evaluation of the type case expression depends on the type r_v of the record value v. This type can be any subtype of r_e. If r_v is a subtype of r, then the evaluation proceeds with e_1, otherwise with e_2. Binding e to a variable x allows us to use the stricter type r in the expression e_1.

Another important component of the report language consists of the dereferencing operators ! and @, which give access to the lookup operator provided by the entity store. Given an expression e of an entity type $\langle r \rangle$, both dereferencing operators provide a value v of type r. That is, both ! and @ are unary operators of type $\langle r \rangle \to r$ for any record type r. In the case of the operator !, the resulting record value v is the latest value associated with the entity to which e evaluates. More concretely, given an entity value v, the expression $v!$ evaluates to the record value $\text{lookup}_t(v, t)$, where t is the current time ("now").

On the other hand, the *contextual* dereference operator @ yields the value of an entity at the time of the event it is extracted from. Concretely, every entity v that enters the event log is annotated with the timestamp of the event it occurs in. That is, each entity value embedded in an event e in the event log, occurs in an annotated form (v, s), where s is the value of e's internalTimeStamp field. Given such an annotated entity value (v, s), the expression $(v,s)@$ evaluates to $\text{lookup}_t(v, s)$ and given a bare entity value v the expression $v@$ evaluates to $\text{lookup}_t(v, t)$.

Note that in each case for either of the two dereference operators, Proposition 1 guarantees that the lookup operation yields a record value of the right type. That is, they are total functions of type $\langle r \rangle \to r$ that never get stuck.

Lifecycle of Reports. Like entities, the set of reports registered in a running POETS instance—and thus available for querying—can be changed via the external interface to the report engine. To this end, the report engine interface provides the operations *addReport*, *updateReport*, and *deleteReport*. The former two take a *report specification* that contains the name of the report, the definition of the report function that generates the report data and the type of the report function. Optionally, it may also contain further meta information in the form of a description text and a list of tags.

The remaining operation provided by the report engine—*queryReport*—constitutes the core functionality of the reporting system. Given a name of a registered report and a list of arguments, this operation supplies the given arguments to the corresponding report function and returns the result.

Contract Engine

Function	**Input**	**Output**
createTemplate	name, type, description, specification	
updateTemplate	name, type, description, specification	
deleteTemplate	name	
createContract	meta data	contract ID
updateContract	contract ID, meta data	
concludeContract	contract ID	
getContract	contract ID	contract state
registerTransaction	contract ID, timestamp, transaction	

Fig. 7. Contract engine interface

2.5 Contract Engine

The role of the contract engine is to determine which transactions—that is, external events, compare Section 2.2—are expected by the system. Transactions model events that take place according to an *agreement*, for instance a delivery of goods in a sale, a payment in a lease agreement, or a movement of items from one inventory to another in a production plan. Such agreements are referred to as *contracts*, although they need not be legally binding contracts. The purpose of a contract is to provide a detailed description of *what* is expected, by *whom*, and *when*. A sales contract, for example, may stipulate that first the company sends an invoice, then the customer pays within a certain deadline, and finally the company delivers goods within another deadline.

The interface of the contract engine is shown in Figure 7.

Contract Templates. In order to specify contracts such as the aforementioned sales contract, we use an extended variant of the contract specification language (CSL) of Hvitved et al. [9], which we will refer to as the POETS contract specification language (PCSL) in the following. For reusability, contracts are always specified as *contract templates* rather than as concrete contracts. A contract template consists of four parts: (i) a template name, (ii) a template type, which is a subtype of the Contract record type, (iii) a textual description, and (iv) a PCSL specification. We describe PCSL in Section 2.5.

The template name is a unique identifier, and the template type determines the parameters that are available in the contract template.

Example 1. We may define the following type for sales contracts in the ontology language (assuming that the record types Customer, Company, and Goods have been defined):

Sale is a Contract.
Sale has a Customer entity.
Sale has a Company entity.

Sale has a list of Goods.
Sale has an Int called amount.

With this definition, contract templates of type Sale are parametrised over the fields customer, company, goods, and amount of types ⟨Customer⟩, ⟨Company⟩, [Goods], and **Int**, respectively.

The contract engine provides an interface to add contract templates (*createTemplate*), update contract templates (*updateTemplate*), and remove contract templates (*deleteTemplate*) from the system at run-time. The structure of contract templates is reflected in the external event types CreateContractDef, UpdateContractDef, and DeleteContractDef, compare Section 2.2. A list of (non-deleted) contract templates can hence be computed by an appropriate report.

Contract Instances. A contract template is instantiated via *createContract* by supplying a record value v of a subtype of Contract. Besides custom fields, which depend on the type at hand, such a record always contains the fields templateName and startDate inherited from the Contract record type. The field templateName contains the name of the template to instantiate, and the field startDate determines the start date of the contract. The fields of v are substituted into the contract template in order to obtain a *contract instance*, and the type of v must therefore match the template type. For instance, if v has type Sale then the field templateName must contain the name of a contract template that has type Sale. We refer to the record v as *contract meta data.*

When a contract c is instantiated by supplying contract meta data v, a *fresh* contract identifier i is created, and a CreateContract event is persisted in the event log with with contract $= v$ and contractId $= i$. Hereafter, transactions t can be registered with the contract via *registerTransaction*, which will update the contract to a *residual contract* c', written $c \xrightarrow{t} c'$, and a TransactionEvent with transaction $= t$ and contractId $= i$ is written to the event log. The state of the contract can be acquired from the contract engine at any given point in time via *getContract*, which enables run-time analyses of contracts, for instance in order to generate a list of expected transactions.

Registration of a transaction $c \xrightarrow{t} c'$ is only permitted if the transaction is expected in the current state c. That is, there need not be a residual state for all transactions. After zero or more successful transactions, $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, the contract may be concluded via *concludeContract*, provided that the residual contract c_n does not contain any outstanding obligations. This results in a ConcludeContract event to be persisted in the event log.

The lifecycle described above does not take into account that contracts may have to be updated at run-time, for example if it is agreed to extend the payment deadline in a sales contract. To this end, running contracts are allowed to be updated, simply by supplying new contract meta data (*updateContract*). The difference in the new meta data compared to the old meta data may not only be a change of, say, items to be sold, but it may also be a change in the field templateName. The latter makes it is possible to replace the old contract by a qualitatively different contract, since the new contract template may describe a different workflow. There is, however, an important restriction: a contract can only be updated if any previous transactions registered with the contract also conform with the new contract. That is, if the contract has evolved like $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, and an update to a new contract c' is requested, then only if $c' \xrightarrow{t_1} c'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c'_n$, for some $c'_1, \ldots, c'_n$, is the update permitted. A successful update results in an UpdateContract event to be written to the event log with the new meta data.

For simplicity, we only allow the updates described above. Another possibility is to allow updates where the current contract c is replaced directly by a new contract c'. This effect can be attained by prefixing c' with $[t_1, \ldots, t_n]$ as contract actions.

As for contract templates, a list of (non-concluded) contract instances can be computed by a report that inspects CreateContract, UpdateContract, and ConcludeContract events respectively.

The Contract Language. The fourth component of contract templates—the PCSL specification—is the actual normative content of contract templates. PCSL extends Hvitved's CSL [9] mainly at the level of expressions E, by adding support for the value types in POETS, as well as lambda abstractions and function applications. At the level of clauses C, PCSL is similar to CSL, albeit with a slightly altered syntax. Typing of PCSL expressions is more challenging since we have added (record) polymorphism as well as subtyping.

We do not present PCSL formally here; instead, it is illustrated in the use case in Section 3 below.

3 Use Case: Legejunglen

We outline a use case that we implemented in a trial with a small business called *Legejunglen*, an indoor playground for children.

The user interface to the POETS system is provided by a client application for the Android operating system. The application is suitable for both phone and tablet devices. Although, for this trial we focused on the tablet user experience. The client application communicates with the POETS system running on a server via the APIs of individual subsystems as described in Section 2. The client provides a generic user interface guided by the ontology. There is functionality to visualise ontology elements as well as allowing user input of ontology elements. Additionally, a simple mechanism for compile-time specialised visualisations is provided. The generic visualisations handle ontology changes without any changes needed on the client. The central part of the user interface provides an overview of the state of currently instantiated contract templates as well as allowing users to interact with running contracts.

In the following, we present the final results of an iterative refinement process on modelling the *Legejunglen* business. We conclude with some reflections on using the DSLs for iterative model evolution.

The most important functionality for day-to-day use at *Legejunglen* is to (1) register bookings for customers and (2) to get an overview of the scheduled events for a single day. Apart from that, the system should provide standard accounting functionality.

The main workflow that we needed to implement is the booking system, that is, the system according to which a customer reserves a time at the playground. This workflow is encoded in a contract template Appointment. The data associated with this contract are defined in the following ontology definition:

Appointment is a Contract.
Appointment is abstract.
Appointment has a DateTime called arrivalDate.
Appointment has Food.
Appointment has a Location called placement.
Appointment has a Participants.
Appointment has an Int called numberOfTableSettings.
Appointment has a String called comments.
Appointment has an Adult entity called contactPerson.

The full ontology also contains declarations that define the auxiliary concepts Food, Location, Participants and Adult, which we have elided here. The fields that are associated with the Appointment record type have to be provided in order to instantiate the corresponding *Appointment* contract template. These fields are then directly accessible in the definition of the contract template.

Figure 8 details the definition of the contract template that describes the workflow for booking an appointment at *Legejunglen*. The full contract is defined at the very bottom by referring to the *confirm* clause. Note that we directly reference the arrivalDate, numberOfTableSettings and contactPerson field of the Appointment record. The three clauses of the contract template roughly correspond to three states an active *Appointment* contract may be in: first, in the *confirm* clause we wait for confirmation from the customer until one day before the expected arrival. After that we wait for the arrival of the customer at the expected time (plus a one hour delay). Finally, we expect the payment within one day.

Next we turn to the reporting functionality of POETS. For daily planning purposes, *Legejunglen* requires an overview of the booked appointments of any given day. This functionality is easily implemented in the reporting language. Firstly, we define the record type that contains the result of the desired report:

Schedule is a Report.
Schedule has a list of Appointment called appointments.

Secondly, we define the actual report function that searches the event log for the creation of *Appointment* contracts with a given arrivalDate. The report definition is given in Figure 9.

A more complex report specification is given in Figure 10. This report compiles an overview of all appointments made during a month as well as the sum of all payments that were registered by the system during that time. This report

```
name: appointment
type: Appointment
description: "Contract for handling a appointment."

// A reference to the designated entity that represents the company
val me = reports.me ()

clause confirm(expectedArrival : Duration, numberOfTableSettings : Int)
          ⟨me : ⟨Me⟩, contact : ⟨Adult⟩⟩ =
  when ContactConfirms
      due within expectedArrival ⟨−⟩ 1D
      remaining newDeadline
  then
      arrival(newDeadline)⟨me, contact⟩
  else arrival(expectedArrival)⟨me, contact⟩

clause arrival(expectedArrival : Duration)⟨me : ⟨Me⟩, contact : ⟨Adult⟩⟩ =
  ⟨me⟩ GuestsArrive
      due within expectedArrival ⟨+⟩ 1H
  then payment(me)⟨contact⟩

clause payment(me : ⟨Me⟩)⟨contact : ⟨Adult⟩⟩ =
  ⟨contact⟩ Payment(sender s, receiver r)
          where r ≡ me ∧ s ≡ contact
          due within 1D

contract = confirm(subtractDate arrivalDate contractStartDate,
                numberOfTableSettings)⟨me, contactPerson⟩
```

Fig. 8. Contract template for booking an appointment

```
name: DailySchedule
description:
   Returns a list of appointments for which the expected
      arrival is the same as the given date.
tags: legejunglen

report : Date → Schedule
report expectedArrival =
   Schedule { appointments = [arra |
         putC : PutContract ← events,
         arra : Appointment = putC.contract,
         expectedArrival ≡ arra.arrivalDate.date ] }
```

Fig. 9. Report definition for compiling a daily schedule

specification uses an explicit *fold* in order to accumulate the payment and appointment information that are spread throughout the event log.

Although *Legejunglen* is a relatively simple business, a significant amount of the work done in the trial involved refining the workflows implicitly in use and formalising what reports were needed. The ability to specify a workflow using the contract language and then immediately try it out in the Android client, helped the modelling process tremendously. A basic contract template for keeping track of bookings was made quickly, which facilitated the process of iterative evaluation and refinement to precisely capture the way *Legejunglen* worked. Changes on the POETS side were quite easy to perform. Changes are typically isolated. That is, support for new workflows or reports does not require a change to the data model and only amounts to adding new contract templates respectively report specifications. This can be performed while the system is up and running, without any downtime. In addition, the subtyping discipline employed in POETS' data model is a key feature in enabling extending the ontology of at run time without compromising the integrity of its state or the semantics of its reports and contracts.

The effort for implementing changes in the data model and the workflow is quite modest. Minor changes in the requirements tended to require little changes in the ontology and contract specifications. Typically, this is also the case for changes in the report specifications. However, changes in report specifications turned out to be quite complicated in some instances. Reports have the ability to produce highly structured information from the flat-structured event log. Unfortunately, this ability is reflected in the complexity of the corresponding report specifications. Nonetheless, from the report specifications we have written, we can extract a small set of high-level patterns that cover most common use cases. Integrating these high-level patterns into the reporting language should greatly reduce the effort for writing reports and further increase readability.

Changes in the underlying modelling on the POETS side were rather easy to propagate to the Android client software. As mentioned, the client application provides a generic user interface to the POETS system that allows it to reflect any changes made in the modelling in the POETS system. However, this generic interface does not always provide the optimal user experience and therefore needs manual refinement to reflect changes in the modelling. Additionally, there have also been specific requirements to the client software, which had to be implemented.

4 Conclusion

We have presented an extended and generalised version of the POETS architecture [6], which we have fully implemented. It is based on declarative domain-specific languages for specifying the data model, reports, and contracts of a POETS instance, which offer enterprise domain concepts and encapsulate important invariants that facilitate safe run-time changes to data types, reports and contracts; full recoverability and auditability of any previous system state;

```
MonthlyOverview is a Report.
MonthlyOverview has a Real called total.
MonthlyOverview has a list of AppointmentInfo called appointments.

name: MonthlyOverview
description:
  Get information about payments received for given month.
tags: legejunglen

allContracts : [PutContract]
allContracts = [pc |
  cc : CreateContract ← events,
  pc = first cc [uc | uc : UpdateContract ← events, uc.contractId ≡ cc.contractId]]

allPayments : Date → [(Payment, PutContract)]
allPayments date =
  [(pay, putC) |
    putC ← allContracts,
    arra : Appointment = putC.contract,
    arra.arrivalDate.month ≡ date.month,
    arra.arrivalDate.year ≡ date.year,
    tr   : TransactionEvent ← transactionEvents,
    tr.contractId ≡ putC.contractId,
    pay : Payment = tr.transaction  ]

initialOverview = MonthlyOverview { total        = 0,
                                    appointments = [] }


addAppointment : (Payment, Appointment) → [AppointmentInfo] → [AppointmentInfo]
addAppointment payArr arrs = insertProj
  (λpa → pa.appointment.arrivalDate)
  (AppointmentInfo {appointment = payArr.2, payment = payArr.1})
  arrs

calc payPut overv =
  type x = payPut.2.contract of
    Appointment → overv {
      total         = overv.total + payPut.1.money.amount,
      appointments  = addAppointment (payPut.1, x) overv.appointments }
    _ → overv

report : Date → MonthlyOverview
report date = fold calc initialOverview (allPayments date)
```

Fig. 10. Report definition for compiling a monthly payment overview

and strict separation of logged raw data and efficiently computed user-specified derived data. In particular, in contrast to its predecessor, any historical system state is reestablishable for auditing since also master data, contract and report changes are logged, not only transactional data.

The use case presented illustrates the conciseness of POETS DSLs and support for rapid exploratory process and report design since the “specification is the implementation” approach made it easy to make an initial model of the business as well as evolve it to new requirements. While no significant conclusions for usability and fitness for use in complex commercial settings can be drawn without a suitable experimental design, we believe the preliminary results justify hypothesising that domain specialists should be able to read, understand and specify data models (types) and, with suitable training in *formalisation*, eventually contract and report specifications without having to worry about programming or system specifics.

4.1 Related work

This paper focuses on the radical *use* of declarative domain-specific languages in POETS motivated by the Resources, Event, Agents accounting model [13,6]. The syntactic and semantic aspects of its domain modelling language [25], its contract language [9] (evolved from [1]) and functional reporting[5] [19,18] are described elsewhere.

ERP systems relate broadly to and combine aspects of discrete event simulation, workflow modelling, choreography and orchestration, run-time monitoring, process specification languages (such as LTL), process models (such as Petri nets), and report languages (such as the query sublanguage of SQL and reactive functional programming frameworks), which makes a correspondingly extensive review of related work from a general ERP systems point of view a difficult and expansive task.

More narrowly, POETS can be considered an example of *language-oriented programming* [27] applied to the business modelling domain. Its contract language specifies detailed real-time and value constraints (e.g. having to pay the cumulatively correct amount by some deadline, not just some amount at some time) on contract partners, neither supporting nor fixing a particular business process. See [8, Chapter 1] and [7] for a survey of *contract* models and languages.

A hallmark of POETS is its enforcement of static invariants that guarantee auditability and type correctness even in the presence of run-time updates to data types, processes and reports. Recently the jABC approach [23,12] has added support for types, data-flow modelling and processes as first-class citizens. The resulting DyWA (Dynamic Web Application) approach [15,17,16] offers support for step-by-step run-time enhancement with data types and corresponding business processes until an application is ready for execution and for its subsequent evolution.

[5] Automatic incrementalisation is not implemented in the present version.

Automatic incrementalisation of report functions in POETS can be thought of as translating bulk-oriented queries that conceptually inspect the complete event log every time they are run to continuous queries on streams [24], based on formal differentiation techniques [21,11].

4.2 Future Work

Expressivity A possible extension of the data model is to introduce finite maps, which will enable a modelling of resources that is closer in structure to that of Henglein et al. [6]. Another possible extension is to allow types as values in the report language. There are instances where we currently use a string representation of record types rather than the record types themselves. This representation is, of course, suboptimal: we would like such runtime represenations of types machine checked and take subtyping into account.

Rules A rule engine is a part of our extended architecture (Figure 2), however it remains to be implemented. The purpose of the rule engine is to provide rules—written in a separate domain-specific language—that can constrain the values that are accepted by the system. For instance, a rule might specify that the items list of a Delivery transaction always be non-empty.

More interestingly, the rule engine will enable values to be *inferred* according to the rules in the engine. For instance, a set of rules for calculating VAT will enable the field vatPercentage of an OrderLine to be inferred automatically in the context of a Sale record. That is, based on the information of a sale and the items that are being sold, the VAT percentage can be calculated automatically for each item type.

The interface to the rule engine will be very simple: a record value with zero or more *holes* is sent to the engine, and the engine will return either (i) an indication that the record cannot possibly fulfil the rules in the engine, or (ii) a (partial) substitution that assigns inferred values to (some of) the holes of the value as dictated by the rules. Hence when we, for example, instantiate the sale of a bicycle. then we first let the rule engine infer the VAT percentage before passing the contract meta data to the contract engine.

Forecasts A feature of the contract engine, or more specifically of the reduction semantics of contract instances, is the possibility to retrieve the state of a running contract at any given point in time. The state is essentially the AST of a contract clause, and it describes what is currently expected in the contract, as well as what is expected in the future.

Analysing the AST of a contract enables the possibility to do *forecasts*, for instance to calculate the expected outcome of a contract or the items needed for delivery within the next week. Forecasts are, in some sense, dual to reports. Reports derive data from transactions, that is, facts about what has previously happened. Forecasts, on the other hand, look into the future, in terms of calculations over running contracts. We have currently implemented a single forecast, namely a forecast that lists the set of immediately expected transactions for a given contract. A more ambitious approach is to devise (yet another) language for writing forecasts, that is, functions that operate on contract ASTs.

Practicality In order to make POETS useful in practice, many features are still missing. However, we see no inherent difficulties in adding them to POETS compared to traditional ERP architectures. To mention a few: (i) security, that is, authorisation, users, roles, etc.; (ii) module systems for the report language and contract language, that is, better support for code reuse; and (iii) checkpointing of a running system, that is, a dump of the memory of a running system, so the event log does not have to be replayed from scratch when the system is restarted.

Acknowledgements. Morten Ib Nielsen and Mikkel Jønsson Thomsen both contributed to our implementation and design of POETS, for which we are thankful. We thank the participants of the DIKU course "POETS Summer of Code" and Lejejunglen for testing, use and valuable input to POETS. This work has been made possible by a grant by the Danish National Advanced Technology Foundation (Højteknologifonden) for Project 3gERP.

References

1. Andersen, J., Elsborg, E., Henglein, F., Simonsen, J.G., Stefansen, C.: Compositional specification of commercial contracts. International Journal on Software Tools for Technology Transfer (STTT) 8(6), 485–516 (2006)
2. Bahr, P., Hvitved, T.: Compositional data types. In: Proc. 7th ACM SIGPLAN Workshop on Generic Programming (WGP), pp. 83–94. ACM (2011)
3. Bahr, P., Hvitved, T.: Parametric compositional data types. In: Proc. Mathematically Structured Functional Programming, MSFP (2012)
4. Bernstein, A.J., Kifer, M.: Databases and Transaction Processing: An Application-Oriented Approach, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
5. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto Controlled English for Knowledge Representation. In: Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S. (eds.) Reasoning Web. LNCS, vol. 5224, pp. 104–124. Springer, Heidelberg (2008)
6. Henglein, F., Larsen, K.F., Simonsen, J.G., Stefansen, C.: POETS: Process-oriented event-driven transaction systems. Journal of Logic and Algebraic Programming 78(5), 381–401 (2009)
7. Hvitved, T.: A survey of formal languages for contracts. In: Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS 2010), pp. 29–32 (2010)
8. Hvitved, T.: Contract Formalisation and Modular Implementation of Domain-Specific Languages. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU) (November 2011)
9. Hvitved, T., Klaedtke, F., Zălinescu, E.: A trace-based model for multiparty contracts. The Journal of Logic and Algebraic Programming 81(2), 72–98 (2012); Preliminary version presented at 4th Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2010) (2010)

10. Thomsen, M.J.: Using Controlled Natural Language for specifying ERP Requirements. Master's thesis, University of Copenhagen, Department of Computer Science (2010)
11. Liu, Y.A.: Efficiency by incrementalization: An introduction. Higher-Order and Symbolic Computation 13(4) (2000)
12. Margaria, T., Steffen, B.: Business process modelling in the jabc: the one-thing-approach. In: Handbook of Research on Business Process Modeling, pp. 1–26. IGI Global (2009)
13. McCarthy, W.E.: The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. The Accounting Review LVII(3), 554–578 (1982)
14. Murthy, C.: Advanced programming language design in enterprise software: A lambda-calculus theorist wanders into a datacenter. In: Proc. ACM Symp. on Principles of Programming Languages (POPL), ACM SIGPLAN Notices, vol. 42(1), pp. 263–264. ACM (2007)
15. Neubauer, J., Steffen, B.: Plug-and-play higher-order process integration. Computer 46(11), 56–62 (2013)
16. Neubauer, J., Steffen, B., Frohme, M., Margaria, T.: Prototype-driven development of web applications with dywa. In: These Proceedings (2014)
17. Neubauer, J., Steffen, B., Margaria, T.: Higher-order process modeling: Product-lining, variability modeling and beyond. Electronic Proceedings in Theoretical Computer Science (EPTCS) 129, 259–283 (2013)
18. Nissen, M.: Reporting technologies. In: 2nd 3gERP Workshop, Frederiksberg, Denmark (2008)
19. Nissen, M., Larsen, K.F.: FunSETL — Functional Reporting for ERP Systems. In: Chitil, O. (ed.) 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007, pp. 268–289 (2007)
20. Ohori, A.: A Polymorphic Record Calculus and Its Compilation. ACM Trans. Program. Lang. Syst. 17, 844–895 (1995)
21. Paige, R., Koenig, S.: Finite differencing of computable expressions. ACM TOPLAS 4(3), 402–454 (1982)
22. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
23. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-driven development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
24. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proc. SIGMOD Conference, vol. 21(2). ACM (1992)
25. Thomsen, M.J.: Using controlled natural language for specifying ERP requirements. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen (July 2010)
26. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2(04), 461–493 (1992)
27. Ward, M.P.: Language-oriented programming. Software-Concepts and Tools 15(4), 147–161 (1994)
28. Weygandt, J.J., Kieso, D.E., Kimmel, P.D.: Financial Accounting, with Annual Report. Wiley (2004)