A Formalism and Method for Representing and Reasoning with Process Models Authored by Subject Matter Experts

José Manuel Gómez-Pérez, Michael Erdmann, Mark Greaves, and Oscar Corcho

Abstract—Enabling Subject Matter Experts (SMEs) to formulate knowledge without the intervention of Knowledge Engineers (KEs) requires providing SMEs with methods and tools that abstract the underlying knowledge representation and allow them to focus on modeling activities. Bridging the gap between SME-authored models and their representation is challenging, especially in the case of complex knowledge types like processes, where aspects like frame management, data, and control flow need to be addressed. In this paper, we describe how SME-authored process models can be provided with an operational semantics and grounded in a knowledge representation language like F-logic to support process-related reasoning. The main results of this work include a formalism for process representation and a mechanism for automatically translating process diagrams into executable code following such formalism. From all the process models authored by SMEs during evaluation 82 percent were well formed, all of which executed correctly. Additionally, the two optimizations applied to the code generation mechanism produced a performance improvement at reasoning time of 25 and 30 percent with respect to the base case, respectively.

1 Introduction

Building knowledge-based systems is an activity that has been traditionally carried out by a combination of software and knowledge engineers (KEs) and of subject matter experts (SMEs), also known as domain experts. Software engineers (SEs) are focused on architectural and user interface issues related to the development of software. KEs are focused on knowledge acquisition and representation tasks, with the aim of building the required knowledge bases. For these tasks, KEs usually work in collaboration with SMEs, who normally act as repositories of domain knowledge. The combination of KEs and SMEs, although feasible, has two main drawbacks, first characterized as the *knowledge acquisition bottleneck* [13]: 1) it is costly and 2) it can be error prone, especially in complex domains.

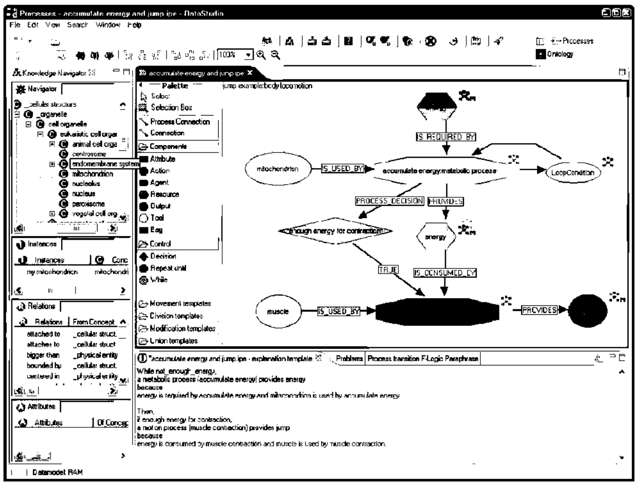
A large amount of work in knowledge-based systems in the past three decades, like [11], [4], and [27], has concentrated on providing frameworks and tools that support the collaboration of KEs and SMEs with the goal of alleviating the knowledge acquisition bottleneck. However, despite progress shown by existing knowledge acquisition tools like KRAKEN [25], SHAKEN [3] and, more recently, AURA [10] and the Halo extension of the Semantic MediaWiki [18], it is still a complex problem to enable SMEs to capture the knowledge from a domain by themselves, especially for some knowledge types.

Among the different types of knowledge that can be used in knowledge-based systems, e.g., factual, rule or causal knowledge, we focus on *process knowledge*, which is widely used across domains while posing important challenges for knowledge acquisition. A process can be defined as *a naturally occurring or designed sequence of changes of properties of a system or object*. Examples of processes include the replication of DNA, the mitosis of the cell or a combustion reaction. Processes encapsulate such things as preconditions and postconditions, results, contents, actors or causes and relate to the sequence of operations and involved events taking up time, space, expertise or other resources, which lead to the production of an outcome. For example, consider the case of a complex chemical reaction comprising several steps, with different inputs and outputs, where it is necessary to reason about what would happen at a certain stage if a previous one was suppressed or modified.

Enabling SMEs to formulate process knowledge without the intervention of KEs is a complex problem that needs to be addressed from a multidimensional perspective to 1) provide the required knowledge artifacts that support the acquisition of process knowledge and 2) develop usable tools that allow SMEs to exploit such artifacts. To this purpose, we have produced the following models, methods, and tools:

1. A *process metamodel*, which provides the terminology necessary to express process entities and the relations between them.

- *J.M. Gómez-Pérez is with iSOCO S.A., Av. del Partenón, 16-18, 1° 7ª, Campo de las Naciones, 28042 Madrid, Spain. E-mail: jmgomez@isoco.com.*
- *M. Erdmann is with Ontoprise GmbH, An der RaumFabrik 33a, 76227 Karlsruhe, Germany. E-mail: erdmann@ontoprise.de.*
- *M. Greaves is with Vulcan Inc., 505 Fifth Ave S, Suite 900, Seattle, WA 98104. E-mail: MarkG@vulcan.com.*
- *O. Corcho is with Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Despacho 2104, 28660 Boadilla del Monte, Madrid, Spain. E-mail: ocorcho@fi.upm.es.*



While not_enough_energy,
a metabolic process (accumulate energy) provides energy
because
energy is required by accumulate energy and mitochondrion is used by accumulate energy

Then,
if enough energy for contraction,
a motor process (muscle contraction) provides jump
because
energy is consumed by muscle contraction and muscle is used by muscle contraction.

Fig. 1. A *jump* process.

2. A *library of Problem Solving Methods* (PSM) [21], which provides high-level reusable abstractions for process representation.
3. A *graphical process modeling and reasoning tool*, which uses the process metamodel and the PSM library for process modeling by SMEs without intervention of KEs.
4. A *formalism for process knowledge representation* and *a method for the automatic synthesis of executable process models* from SME-authored process diagrams.

The results of our work on the first three topics were presented in [15]. Fig. 1 shows a screenshot of our process editor, which provides SMEs with a modeling and reasoning environment for process knowledge. The main entities of the process metamodel are resources, actions, decisions, and relations (see Fig. 2 for their graphical representations) and allow describing domain entities, e.g., energy or mitochondrion, in terms of their role in a process (resource and tool, respectively). The process metamodel is used by SMEs to model their own processes, through drag and drop operations on the drawing canvas, in combination with reusable, previously existing process models and templates from the PSM library.

We hereby focus on representing and reasoning with process knowledge. The following multiple choice questions, selected from Advanced Placement competence level exams in Biology, illustrate some of the main types of process reasoning that we address in this work:

Reasoning about process entities: The intended goal of this type of reasoning is to retrieve information about the role played by the elements participating in a given process. The following question asks for the role of the agent responsible for initiating the synthesis of DNA. Answering this question requires that the DNA synthesis process is formally described in terms of a process vocabulary like our process metamodel.

```
The primer that initiates the synthesis of a new DNA
strand is usually:
a. RNA
b. DNA
c. an Okazaki fragment
d. a structural protein
```

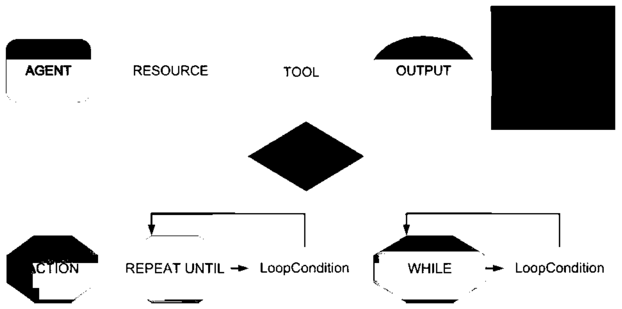


Fig. 2. Resources, conditional fork, and actions.

The question can be answered correctly, and therefore option 1 (RNA) be identified as the agent responsible for the DNA replication process, only if RNA is explicitly modeled as the agent responsible for such process, following the vocabulary provided by the metamodel. In this case, RNA would have been modeled as an *agent* that *performs* the DNA replication process.

Reasoning on intermediate results: The question below focuses on the outcome of an elongation subprocess in the context of a larger process, i.e., DNA synthesis. To evaluate the first alternative answer, option 1, the query associated with the question will need to retrieve all Okazaki fragments in the knowledge base resulting from the execution of the elongation subprocess. Thus, we will require a knowledge representation language with introspective capabilities to address reasoning on process steps and intermediate results. Approaches based on process description and execution languages like BPEL[1] do not completely address this issue and systems based on these languages tend to behave as a black box.

```
The elongation of the leading strand during DNA
synthesis:
a. produces Okazaki fragments
b. depends on DNA polymerase
c. does not require a template strand
```

Reasoning on process stages: This kind of questions requires reasoning on the effect that the different process steps have on the overall process, and their relation with the entities involved. In this case, the example question aims at situating the occurrences of particular cell organelles in the appropriate mitosis stage.

```
Which part of the animal cell is required only in the
first stage of mitosis and what is the name of such
stage?
a. chromatin and prophase
b. chromatid and prometaphase
c. centromere and anaphase
d. plasma membrane and telophase
```

An interesting application of reasoning about process stages is the identification of similar processes, potentially from different domains, through analogies detected in their structure and in the roles of the entities involved. This feature can be especially useful to analyze processes by contrasting them against well-known process specifications (or patterns), and projecting the properties of the latter against the former.

Reasoning on process preconditions: In addition to reasoning on process inputs, outputs, and intermediate results, a

1. http://www.oasis-open.org/committees/wsbpel.

fourth type of process reasoning addresses questions related to the preconditions that must hold to accomplish a process, as illustrated by the example question below.

```
At least, what amount of energy does an athlete need to
consume in order to jump more than 8m long?
a. 100 cal
b. 50 cal
c. 250 cal
d. 1 cal
```

In the remainder of this paper, we present an approach to bridge the gap between the formulation of process knowledge by SMEs and the formal representation of the resulting process models. We present our process representation formalism and mechanism for automatically translating process diagrams into executable code and apply two main optimizations for reasoning with process knowledge. In Section 2, we summarize previous related work and describe the formalism for representing and reasoning with processes. Section 3 analyzes the requirements, especially in terms of the process frame, of the underlying language for process representation and proposes F-logic as such language. In Section 4, we present the code generation mechanism, focusing on how process diagrams are automatically translated into different types of F-logic rules and how they relate with our process metamodel. Section 5 describes the optimizations applied to the generated process code. In Section 6, we present the results of the evaluation of our approach from a performance perspective. Finally, Section 7 concludes the paper and presents future and related lines of work.

2 A Formalism for Representing and Reasoning with Processes

Process modeling by SMEs needs to be supported by a formalism that endows the resulting models with the corresponding operational semantics, thus bridging the gap between the knowledge level and the symbolic and operational levels [24] and enabling reasoning. Our formalism is based on the concept of process frame and addresses data and control flow management as an incarnation of the frame problem [26].

2.1 The Process Frame

One of the main challenges for our process knowledge representation and reasoning (KRR, from now on) formalism deals with expressing the portion of the knowledge base that a process action can have effect on, and how such effects propagate throughout the process. We define the frame of a process as the relevant portion of the knowledge base for each process step. For example, in Fig. 1, the process frame of *muscle contraction* contains the updated amount of energy resulting from the previous step, *accumulate energy*. The process frame is dynamic; it changes throughout execution and propagates among successive actions, which modify it as they occur, in a similar way to previous work in Logic Programming [30], [7], [6].

The process frame needs to be kept as terse and small as possible. This contributes to reduce the number of concepts and instances to reason with and to minimize the need of complex and potentially costly evaluation mechanisms for reasoning. As we will show, an effective management of the process frame based on these principles is essential to implement process data and control flow.

In our process KRR formalism, we address frame management as a case of ontology modularization [16] in a dynamic setting. Ontology modules containing the process frame are created during process execution, where the actions to be executed at each process step perform their computations and whose updates are then used in subsequent actions. This modularization structure addresses a threefold goal:

1. Building the *initial (preexecution) process frame* by encapsulating the relevant parts of the knowledge base required for the execution of a process to be triggered.
2. Building the *process frame at each process step* by encapsulating the relevant portion of the knowledge base, including the updates produced by the execution of previous actions, which are required for the current computation.
3. Enabling the *computational flow between actions*, calculating their *prestates* and *poststates* and supporting process data and control flow.

2.2 Formalization

Our formalism is action centric, built on the basis of actions as the backbone entities of processes. A process consists of a partially ordered set of actions, connected as a directed graph, with preconditions and postconditions whose evaluation both determines the flow of data between the steps comprised by the process and controls the order in which such actions are executed. Such ordering allows a number of behavioral patterns, including forks, joins, and parallelism.

We define the *prestates* and *poststates* of an action as the content of the process frame immediately before and after its execution, respectively. The *prestate* contains all the knowledge base entities that serve as inputs to the action, while the *poststate* contains the outcomes of its execution, obtained by the computations performed on the contents of the *prestate*. The process frame comprises the *prestates* of all the actions ready to be executed. After execution of an action, its *poststate* fits into the new process frame as part of the *prestate* of the subsequent action(s). Thus, the process frame corresponds exactly to the aggregation of the *prestates* of the actions ready to be executed.

Fig. 1 shows a process diagram that represents a particular type of *body locomotion* i.e., a *jump* process. In this case, the *prestate* of action *accumulate energy* in the process model comprises concepts *energy* and *mitochondrion*, as well as their respective instances and the relations between them. Likewise, the *poststate* of the action comprises the resulting instances of concept *energy* updated by the execution of the action.

Actions can be classified in terms of their *prestates* and *poststates* as *input*, *output*, or *intermediate actions*. In the process diagram of Fig. 1, *accumulate energy* is an input action, whereas *muscle contraction* is an output action. In a process model, the *prestate* of an input action does not contain the output of any other action. Likewise, the updates to the process frame contained in the *poststate* of an output action contribute to none of the *prestates* of other actions in its model, because such action has no further successors. The remaining

TABLE 1
Process Rules per Type of Process Action

	Input actions	Intermediate actions	Output actions
Setup rules	X	-	-
Transition rules	X	X	X
Precedence rules	-	X	X

actions are intermediate, with both predecessors and successors.

We propose to formally describe process models as a directed (weakly) connected graph G [14] denoted by *(N, E)*, where E is the set of edges and N the set of nodes. According to our process metamodel, N consists of the disjoint subsets

$$N_{Resource}, N_{Action}, N_{Decision} \text{ so that } (N_{Resource} \cap N_{Action}) = \emptyset \wedge (N_{Resource} \cap N_{Decision}) = \emptyset \wedge (N_{Action} \cap N_{Decision}) = \emptyset \wedge \forall n \in N : n \in (N_{Resource} \cup N_{Action} \cup N_{Decision}).$$

Each node $n \in N$ has a set of incoming and outgoing edges $E_{in}(n)$ and $E_{out}(n)$, respectively. The source node of an edge e is described by $N_{Source}(e)$, while its target node is $N_{Target}(e)$. Thus, the actions connected to each particular action can be classified as predecessor and successor actions as follows:

$$\forall n, m \in N_{Action} : m \in Predecessor(n) \longleftrightarrow \exists r \in N_{Resource} \wedge \exists e_1 \in E_{in}(n) \wedge \exists e_2 \in E_{out}(m)/r = N_{Target}(e_2) \wedge r = N_{Source}(e_1). \forall n, m \in N_{Action} : m \in Successor(n) \longleftrightarrow \exists r \in N_{Resource} \wedge \exists e_1 \in E_{in}(m) \wedge \exists e_2 \in E_{out}(n)/r = N_{Target}(e_2) \wedge r = N_{Source}(e_1).$$

According to the previous classification of actions as input, output, and intermediate actions, we can say that $\forall n \in N_{Action} : n \in (N_{Input} \cup N_{Intermediate} \cup N_{Output})$. As we will show in Section 4, this determines the type of inference required to evaluate the *prestates* and *poststates* of actions (Table 1). This classification is formally expressed as:

- $\forall n \in N_{Action} : n \in N_{Input} \longleftrightarrow Predecessor(n) = \emptyset$
- $\forall n \in N_{Action} : n \in N_{Output} \longleftrightarrow Successor(n) = \emptyset$
- $\forall n \in N_{Action} : n \in N_{Intermediate} \longleftrightarrow Predecessor(n) \neq \emptyset \cap Successor(n) \neq \emptyset$

In case a process is formed by a single, atomic action, the first two definitions would hold, but not the third. Accordingly, intermediate actions will only occur in processes with at least three actions. So, two constraints need to be defined for $N_{Decision}$, the third subset of nodes in a process graph:

- $\forall n \in N_{Decision} : |E_{in}(n)| = 1 \wedge |E_{out}(n)| \leq 2$, i.e., all decisions are preceded by one action and have a maximum of two successors, associated with the *true* and *false* branches, respectively.
- $$\forall n \in N_{Decision}, \forall e_{in} \in E_{in}(n), \forall e_{out} \in E_{out}(n) : N_{Source}(e_{in}) \subset N_{Action} \wedge N_{Target}(e_{in}) \subset N_{Action},$$
  i.e., all the nodes connected to a *decision* node are actions

Next, we provide a formal definition of the *prestates* and *poststates* of an action, where we elaborate on the previous informal definition. Respectively, we define the *prestates* and *poststates* of an action as

- $\forall r \in N_{Resource}, a \in N_{Action} : r \in Pre(a) \longleftrightarrow \exists e \in E_{in}(a) \wedge \exists c \in C/r = N_{Source}(e) \wedge r \text{ is\_a } c.$
- $\forall r \in N_{Resource}, a \in N_{Action} : r \in Post(a) \longleftrightarrow \exists e \in E_{out}(a) \wedge \exists c \in C/r = N_{Target}(e) \wedge r \text{ is\_a } c.$

As described in [15], the components of a process diagram are first modeled by choosing a role of the process metamodel and then mapping them manually against concrete domain entities. Class c in the previous definition illustrates this, where C is the set of classes in the knowledge base.

For a given action to be triggered, its *prestate* must be completely instantiated, i.e., all the process resources contained in the *prestate* must be instantiated in the knowledge base. As we will show in following sections, the implementation of our formalism encapsulates the frame of such action and the process in which it occurs, thus enabling actions and resources to appear simultaneously in several processes without risk of inconsistencies during reasoning. In our example process, instances of concepts *energy* and *mitochondrion*, respectively, represented as a resource and a tool in terms of the process metamodel comprise the *prestate* of the action *accumulate energy*. Analogously, once updated by the execution of such action, *energy* becomes its *poststate* and, subsequently, the *prestate* of *muscle contraction*.

The formalism provides the means to represent and reason with processes as directed graphs consisting of actions, their inputs and outputs (associated to their *prestates* and *poststates*), and directed arcs corresponding to the process relations between resources and actions, which run from inputs to actions and from those actions to their outputs. According to this representation, we define data and control flow as

- *Data flow* is the path followed from process inputs to process outputs across the directed graph representing the process.
- *Control flow* is the mechanism that evaluates and enacts the actual flow of data upon process execution.

We have kept the process metamodel deliberately simple with respect to control flow primitives to facilitate modeling by SMEs from target domains like Biology and Chemistry, with limited requirements in this directions. However, the formalism supports complex forms of control flow, which can be leveraged to model processes in other domains, like business or software, simply by extending the metamodel. In our approach, control flow is, therefore, based on two main constructs, forks and loops, as building blocks for more complex forms of controlling the execution of a process. Forks correspond to conditional arcs, informing process decisions from the metamodel, between two actions, which explicitly represent a precedence relation enabled only upon satisfaction of a certain condition. In the example of Fig. 1, process decision *enough energy to jump?*, represents a condition whose satisfaction is required to enable the *true* branch of the fork, i.e., the subsequent action *muscle contraction.*

Loops, e.g., *accumulate energy*, can be explicitly implemented as iterative actions from the metamodel, which are repeatedly executed while (or until) a given condition holds. Iterative actions follow a structured loop pattern[2] with an associated pretest or posttest, which are evaluated either at the beginning or at the end of the loop to determine whether it should continue iterating or not. Additionally, the formalism supports arbitrary cycles in process models without requiring specific loop operators or restrictions.

3 Language Support for the Process KRR Formalism

The formalism introduced in the previous section needs to be grounded in a computer language, which allows expressing process entities and their interactions, the problem-solving behavior associated to processes, and the management of process data and control flow. The underlying language needs to provide high expressivity and efficient and safe reasoning capabilities to support SMEs to model and reason with process knowledge. Furthermore, modularization capabilities are required that support the effective manipulation of the process frame, which is fundamental for the realization of our approach.

3.1 Requirements

Several approaches have been proposed from different areas and perspectives to represent process knowledge, including

1. knowledge acquisition and representation languages, e.g., OWL [22], OCML,[3] and F-Logic [17];
2. process-specific representation and reasoning languages, e.g., PSL [5] and SPARK-L [23];
3. Semantic Web service ontologies, e.g., WSMO[4] and OWL-S[5]; and
4. process specification and execution languages, e.g., BPEL and XPDL.[6]

However, while individually each of these approaches is expressive in terms of workflow constructions and reasoning capabilities, SMEs still require further support to effectively model processes, as shown in [31] and [15].

Knowledge representation languages, e.g., OWL and OCML, are well suited to semantically describe lower-level (compared to processes) declarative knowledge entities like concepts, instances, and rules, as well as problem-solving behavior. However, the complexity of process representation hinders the straightforward adoption of these languages as exclusive means to effectively represent processes at the required level of abstraction for SMEs and to support their execution.

On the other hand, process-specific description and execution languages, e.g., BPEL and XPDL, can be effectively used to express processes as workflows and to enact them. Though their formal semantics is usually based on Petri nets, these approaches use to focus on the operational aspects of processes and therefore fail at reasoning with processes at the knowledge level. For example, reasoning about the consequences of removing a certain stage from the overall mitosis process in Biology or analyzing the influence of an additional compound in a chemical reaction is not possible through such approach exclusively. On the other hand, languages like PSL and SPARK-L provide higher expressivity and allow defining constraints. However, like general purpose knowledge representation languages, they can be complex for use by SMEs.

Other approaches, like Episodic Logic (EL) and their implementations as in the EPILOG [29] system, allow for explicit situational variables denoting episodes, events, and states of affairs linked to arbitrary formulae that describe them. However, though the expressiveness and inference capabilities of EL are certainly high, the extensive use of second order reasoning tends to compromise performance.

Thus, while fundamental for the construction of knowledge-based and workflow systems, further solutions are required that can be used to address the problem for process knowledge. We therefore propose an integrated approach that addresses the above-mentioned issues for process KRR in a way that

1. supports the *symbolic representation of process entities*, according to the process metamodel,
2. supports *process execution*, following the specification of the operational semantics described by the process KRR formalism, which defines the possible interactions between process entities,
3. enables a *single entry point* for reasoning across the whole system independently from the knowledge types involved,
4. allows *domain-level reasoning within processes*, e.g., by means of rule-based inference, and
5. keeps *introspective properties* for reasoning with meta-level information about processes, like, e.g., subprocesses or intermediate process results.

3.2 F-Logic as a Process Representation and Reasoning Language

Our approach builds on previously existing KRR languages and focuses on the knowledge representation aspects required to describe and reason with process knowledge. We aim at emphasizing the relevance of the knowledge level over the operational level for use by SMEs through the appropriate user interfaces and at facilitating the combination of the process knowledge type with others, like rules and factual knowledge.

More specifically, we build on F-logic as the underlying language for process KRR. F-logic provides high expressivity and inference capabilities and combines the advantages of frame-based languages and the expressivity, compact syntax, and well-defined semantics of logic programming languages. The original features of F-logic include signatures, object identity, complex objects, methods, classes, inheritance, encapsulation, and deductive rules, which make it suitable to represent and reason seamlessly with both the static and dynamic aspects of processes. F-logic

2. http://www.workflowpatterns.com.
3. http://technologies.kmi.open.ac.uk/ocml.
4. http://www.wsmo.org.
5. http://www.w3.org/Submission/OWL-S.
6. http://www.wfmc.org/standards/XPDL.htm.

```
jumpExample:PROCESS@ProcessModule.
jumpExample[SUBPROCESS->accumulateEnergy]@ProcessModule.
jumpExample[SUBPROCESS->muscleContraction]@ProcessModule.
accumulateEnergy[PRECEEDS->muscleContraction]
        @ProcessModule(jumpExample).
accumulateEnergy:WHILE@ProcessModule(jumpExample).
muscleContraction:ATOMIC@ProcessModule(jumpExample).
```

Fig. 3. F-logic axioms in process *jump example*.

supports negation and well-founded model semantics [32], allowing safe execution of nonstratified rules.

There are several implementations of reasoners for the F-logic language, e.g., FLORA-2,[7] FLORID,[8] and OntoBroker,[9] which provide different means for efficient reasoning in F-logic, ranging between bottom-up evaluation to magic set [2] and dynamic filtering [17]. In all cases, F-logic reasoning is based on a forward chaining approach, which facilitates deduction and thus fits nicely with the propagation of reasoning results throughout process steps, from process inputs to outputs, as defined in our formalism.

In our implementation, we have adopted the OntoBroker F-logic language and reasoning engine, which is based on dynamic filtering and supports computationally complex cases like nonstratified negation by means of well-founded evaluation. It also incorporates a mechanism for answer explanation that allows inspecting rule execution, supporting the analysis of the reasoning trace. Though OntoBroker's performance and scalability have been proved in the OpenRuleBench [19] suite of benchmarks, well-founded semantics is in general costly and needs to be avoided whenever possible. As we will show in Sections 4 and 5, we achieve this by maximizing the amount of stratified F-logic process code synthesized for each process model, whose evaluation does not require well-founded semantics even in the presence of negation.

OntoBroker also allows defining parameterized modules through the @/2 operator to manage the process frame and encapsulate the contents of action prestates and poststates. For example, `p(a)@m.` states that `p(a)` is true in module `m` but, following the open world assumption, no statement is made with respect to the rest of the knowledge base. Such modularization capabilities provide means to create a well-defined interface to the process frame at each process step. Thus, it is possible to constrain the scope of a particular action to the portion of the knowledge base encapsulated by the corresponding module and to approach the creation of the *prestate* and the propagation of the updates produced by such action from its *poststate* into the following process steps.

4 Synthesis of Process Code

At modeling time, we automatically synthesize, from process diagrams, executable code in the form of F-logic axioms and rules (see [1] for a detailed description of OntoBroker's F-logic syntax). The synthesized F-logic axioms describe the process, its actions, and its data flow in terms of the process metamodel, and encapsulate them in the corresponding module as shown in Fig. 3. Paraphrasing the code, *jump example* is a process with two subprocesses (*accumulate energy* and *muscle contraction*), which, in the context of this particular process, are modeled, respectively, as a *while* iterative action and an *atomic* action, where the later is preceded by the former.

Associated with each action in a process model, a set of F-logic rules are generated. These *process rules* have the fundamental role of coping with the frame problem to define how resources are consumed within actions and to address management of data and control flow. In a broad sense, which we will refine in the following sections, the rules resulting from this synthesis process can be classified in terms of their inference behavior on the *prestates* and *poststates* of actions as follows:

- *Setup rules*, which build the *prestate* of input actions from the overall knowledge base, therefore addressing the creation of the initial process frame (see first goal in Section 2.1).
- *Transition rules*, which describe the actual execution of an action, i.e., the necessary inference to be applied to the *prestate* of actions to produce new computation results and to enable the propagation of such results into the *poststate*. Transition rules are, thus, the main enablers of the construction of the process frame at each process step (second goal).
- *Precedence rules*, which build the *prestate* of intermediate and output actions from the *poststate* of their predecessor actions. Precedence rules, thus, enable the data flow between actions (third goal).

Table 1 shows the type of process rules used in each type of process action. While setup rules are only used in input actions, transition rules apply to all action types, which emphasizes the relevance of frame management for data and control flow. Transition rules are, therefore, fundamental for reasoning with process knowledge. They describe how the inputs of an action are transformed into its outputs and update the process frame with the results of such reasoning. Precedence rules are in charge of establishing the data flow throughout a process by connecting actions through their inputs and outputs. Finally, setup rules create the initial frame of a process, enabling its execution.

Fig. 4 illustrates the correlation between process rules and actions, as described in the process metamodel. In the

7. http://flora.sourceforge.net.
8. http://www.informatik.uni-freiburg.de/ dbis/florid.
9. http://www.ontoprise.de/en/home/products/ontobroker.

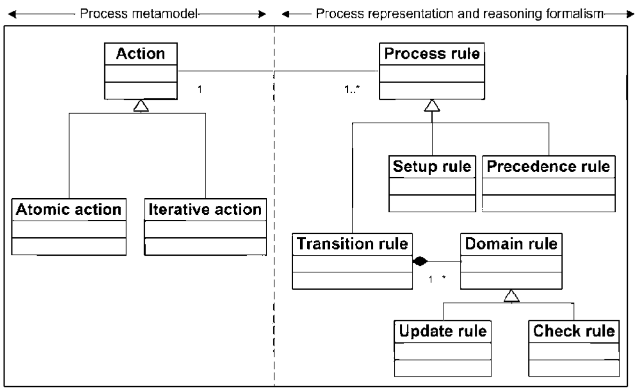


Fig. 4. Correlation between process actions and rules.

```
FORALL m, e, v
  m:mitochondrion@preState(accumulateEnergy) AND
  m:TOOL@preState(accumulateEnergy) AND
  e: energy@preState(accumulateEnergy) AND
  e:RESOURCE@preState(accumulateEnergy) AND
  e[hasEnergyValue -> v]@preState(accumulateEnergy)
<-
  m:mitochondrion AND
  e:energy AND
  e[hasEnergyValue -> v].

FORALL m
  m:muscle @preState(muscleContraction) AND
  m:TOOL@preState(muscleContraction)
<-
  m:muscle.
```

Fig. 5. Setup rules in *jump example* process.

next sections, we describe these process rules and detail their role in the formalism.

4.1 Synthesis of Setup Rules for the Construction of the Initial Frame

The code synthesis mechanism generates setup rules that encapsulate the inputs of input actions into their *prestate* module to build the process frame corresponding to the input actions of a process. Fig. 5 shows the setup rules corresponding to the input actions of the example process in Fig. 1.

The first rule corresponds to the input action *accumulate energy*, whose inputs are instances of concepts *energy* and *mitochondrion*. According to the process diagram, these concepts have been represented by the SME as *resources* and *tools* from the process metamodel, respectively. The associated setup rule asserts this information in the *prestate* of the action, declaring that all *mitochondrion* are *tools* and *energy* is a *resource*.

For performance reasons (see the optimizations described in Section 5 and empirically demonstrated in Section 6), it is important that concept and attribute names appear as grounded terms instead of as free variables. Therefore, all concept attributes appear explicitly in the synthesized code. Consequently, *hasEnergyValue* explicitly appears both in the head and the body of the rule, ensuring that the specific values of the attribute are also preserved in the context of the *prestate* of the action.

The second rule corresponds to the input action *muscle contraction* and situates all instances of concept *muscle* in the *prestate* of the action. The metamodel entity used by the SME to model such input is *tool*. This information is also asserted in the *prestate*. Additionally, in this case, concept *muscle* has been modeled by the SME without any attributes in the underlying ontology, and therefore, the synthesized code focuses only on its *is-a* relations.

4.2 Synthesis of Transition Rules between Action Prestates and Poststates

Transition rules are a specific type of rules produced by the code generation mechanism that describe the conditions required to accomplish an action and to transit from its *prestate* to its *poststate*. For example, Fig. 6 shows the transition rule associated with atomic action *muscle contraction* (see English transcription below):

```
FORALL m, e, j
  j: jump@postState(muscleContraction) AND
  j: OUTPUT@postState(muscleContraction) AND
  muscleContraction[PROVIDES -> j]
    @postState(muscleContraction)
<-
  m:muscle @preState(muscleContraction) AND
  m:TOOL@preState(muscleContraction) AND
  m[IS_USED_BY -> muscleContraction]
    @preState(muscleContraction) AND
  e:energy@ preState(muscleContraction) AND
  e:RESOURCE@preState(muscleContraction) AND
  e[IS_CONSUMED_BY -> muscleContraction]
  @preState(muscleContraction).
```

Fig. 6. Transition rule in *jump example* process.

```
When a muscle is used as a tool in a muscle contraction
and energy is a resource consumed in such muscle
contraction, then the muscle contraction provides a
jump as a result.
```

The code shows a simplified version of a transition rule, where variable *j* appears only in the head of the rule and should be existentially quantified. In terms of our formalism, this means that the action (*muscle contraction*) produces a resource of a given class (*jump*) that was not one of its inputs but, quite on the contrary, has been produced from those inputs, by transforming the process frame contained in the action *prestate*, and added to its *poststate*. Since F-logic does not allow existential quantifiers in the head of rules, the code generation mechanism uses Skolem functions to address this issue, following the F-logic proof theory in [17].

Transition rules describe, in the context of the action, how inputs are taken from the *prestate*, transformed, and left in the *poststate* as outputs. They also describe, in terms of the process metamodel, the semantics of the relations of inputs and outputs with the action itself, as well as the changes in the process roles associated with them throughout the process. Thus, transition rules are also instrumental for the inference occurring in a process and, in addition, their evaluation determines its control flow.

Transition rules can host domain-specific reasoning in the form of domain rules. Transition rules alone allow representing and reasoning with process-specific aspects like the phases of the process, *accumulate energy* and *muscle contraction* in the example, how resources like muscles are used, and how the accumulated energy is finally consumed. On the other hand, domain rules allow reasoning on aspects like the available amount of energy to determine whether such energy is sufficient to jump or not. By importing domain rules into actions SMEs are able to use rule inference capabilities within process models. We consider two main types of rules for domain-level reasoning on the process frame:

- *Update rules*, whose inference modifies the process frame, transforming the *prestate* of an action into its *poststate*.
- *Check rules*, which implement predicates used to evaluate the process frame to compute control flow conditions.

For example, the rule in Fig. 7 obtains the estimated length of a jump using a certain amount of energy. The code generation mechanism automatically rewrites it to use it as an *update* rule in the context of action *muscle*

```
FORALL aJump, length, anEnergy, v
  aJump:jump AND
  aJump [hasLength -> length]
<-
  anEnergy:energy AND
  aJump:jump AND
  anEnergy:energy[hasEnergyValue -> v] AND
  multiply(length, 2, v).
```

Fig. 7. Domain-level rule for jump length calculation.

```
FORALL length, anEnergy, v
  aJump(out(hasLength,length):jump
    @update(muscleContraction)
  aJump(out(hasLength,length)[hasLength -> length]
    @update(muscleContraction)
<-
  anEnergy:energy@preState(muscleContraction) AND
  anEnergy:energy[hasEnergyValue -> v]
    @preState(muscleContraction) AND
  multiply(length,2,v).
```

Fig. 8. *Update* rule for jump length calculation.

contraction (Fig. 8). Domain rules are applied within the scope of the action, where they are imported. Thus, rule bodies access facts from the *prestate* of the action. Similarly, rule heads build the next process frame by asserting new facts in the *poststate.*

In the example, occurrences of concept *jump* appear only in the head of the transition rule, i.e., *jump* is an output of action *muscle contraction* but not an input according to the process model (see Fig. 1). So, skolemization is needed for *muscle contraction*. In the case of *update* rules, existential quantification is applied to the domain rule and not to the transition rule that imports it. This enables the invocation of the former from the body of the latter and simplifies the code generation process. The result of the inference of the *update* rule is encapsulated as an OntoBroker module, e.g., *update(muscle contraction)*, ensuring rule stratification and preventing undesired inference cycles.

We also use domain rules as *check* rules for the evaluation of control flow conditions, like loop termination and forks. In our example, action *muscle contraction* is triggered only if there is sufficient energy for the muscle to contract. Following the same principles as in the case of *update* rules, the code of the rules implementing this kind of predicates is rewritten. This enables control flow evaluation by allowing invocation of such predicates from within the body of transition rules against the *prestate* of the action. The result of such evaluation is encapsulated in a specific OntoBroker module for the particular action at hand (Fig. 9).

Fig. 10 shows the resulting transition rule, including both *update* (highlighted in light gray) and check (highlighted in a darker gray) rules imported into the transition rule of action *muscle contraction*. Note that the evaluation of the false branch of the predicate *enough_energy_for_contraction* requires negating the invocation of the check rule. Rule stratification prevents cycles introduced by negation, which would require well-founded semantics evaluation and reduce performance.

4.3 Synthesis of Precedence Rules for Data Flow Management

The third type of process rules, *precedence* rules, focuses on data flow. Precedence rules describe which actions can be connected with each other by means of their outputs and

```
FORALL anEnergy, v
  enough_energy_for_contraction(anEnergy)
    @check_enough_energy_for_contraction(accumulateEnergy)
<-
  anEnergy:energy @preState(muscleContraction) AND
  anEnergy:energy[hasEnergyValue -> v]
    @preState(muscleContraction) AND
  greater(v, 5).
```

Fig. 9. *Check* rule for energy level evaluation.

```
FORALL j, m, e, length
  j:jump@postState(muscleContraction) AND
  j:OUTPUT@postState(muscleContraction) AND
  muscle contraction[PROVIDES -> j]
    @postState(muscleContraction) AND
   j[hasLength-> length]@postState(muscleContraction)
<=
  m:muscle@preState(muscleContraction) AND
  m:TOOL@preState(muscleContraction) AND
  m[IS_USED_BY -> muscleContraction]
    @preState(muscleContraction) AND
  e:energy@ preState(muscleContraction) AND
  e:RESOURCE@preState(muscleContraction) AND
  e[IS_CONSUMED_BY -> muscleContraction]
    @preState(muscleContraction) AND
   enough_energy_for_contraction(e)
     @check_enough_energy_for_contraction(accumulateEnergy)
   AND j:jump@update(muscleContraction) AND
   j[hasLength -> length]@update(muscleContraction).
```

Fig. 10. Extended transition rule (incl. check and update).

```
FORALL e, v
  e:energy@preState(muscleContraction) AND
  e[hasEnergyValue -> v]@preState(muscleContraction)
<-
  e:energy@postState(accumulateEnergy) AND
  e[hasEnergyValue -> v]@postState(accumulateEnergy).
```

Fig. 11. Precedence rule in *jump example* process.

inputs. Upon reasoning, these rules infer the *prestate* of an action whenever all the *poststates* of its preceding actions are enabled. The so inferred precedence relation between actions is transitive. Precedence rules contribute to shift the process frame from the context of just executed actions to subsequent actions, building the *prestates* of the latter from the *poststates* of the former.

The code generation mechanism analyzes the directed graph associated with the process diagram and detects the actions that are connected with the current one, i.e., those that produce as output one or several process resources that are part of the input of the current action. In the example, *accumulate energy* and *muscle contraction* are connected by *energy*, an output of the former and an input of the latter. For each pair of connected actions, a precedence rule is produced that implements such connection (Fig. 11).

4.4 Code Synthesis for Iterative Actions

In the previous sections, we have described setup, transition, and precedence rules. In particular, we have shown how transition and precedence rules can be produced from SME-authored process diagrams to implement data and control flow, based on the management of the process frame. In the case of iterative actions, as defined in the process metamodel, the number and complexity of the transition rules to be produced increases and additional issues need to be considered to endow iterative actions with the required operational semantics.

In the process metamodel, we define iterative actions as actions that are repeatedly executed until a termination condition holds. Iterative actions repeatedly trigger some inference that modifies the knowledge base and evaluates

```
FORALL e, v
  e:energy@postState(0,iteration,accumulateEnergy) AND
  e[hasEnergyValue - >v]
   @postState(0,iteration,accumulateEnergy)
<-
  e:energy@preState(accumulateEnergy) AND
  e[hasEnergyValue -> v]@preState(accumulateEnergy).
```

Fig. 12. Iterative action—base case rule.

such condition. In our code synthesis mechanism, the process rules corresponding to an iterative action decompose it into a series of atomic actions, one per iteration, which encapsulate the process frame in an OntoBroker module associated to that particular iteration.

As we will show in the following code samples, such modules are parameterized and represented by the triplet *(i, iteration, action name)*, where i indicates the iteration whose frame is encapsulated in such module, *iteration* is a fixed tag, which denotes that the module is used in the context of an iteration, and *name* stands for the iterative action to which the present iteration belongs. The module corresponding to an iteration is related to the module of the previous by incrementing its counter, i.e., $i_{n+1} = i_n + 1$, thus enabling transfer of the computed results between their respective *prestates* and *poststates*.

Iterative actions do not result into a single transition rule upon code synthesis. On the contrary, three different types of transition rules are required with respect to the case of atomic actions: *base case*, *iteration*, and *interface* rules, as graphically represented in Fig. 14.

Base case rules (Fig. 12) initiate the loop, preparing the *prestate* of the first iteration. The *prestate* of the original iterative action becomes the *prestate* of the first action resulting from its decomposition.

Iteration rules (Fig. 13) define the action that is iteratively executed, taking its *prestate* from the *poststate* of the previous iteration and storing its output into the *prestate* of the next. Additionally, iteration rules evaluate the termination condition of the iterative action. Both the predicate evaluating the termination condition and the action are implemented

```
FORALL e, i, v
  e:energy@postState(i,iteration,accumulateEnergy) AND
  e[hasEnergyValue -> v]
    @postState(i,iteration,accumulateEnergy)
<-
  e[hasEnergyValue->v]
   @update_hasEnergyValue(i,iteration,accumulateEnergy) AND
  keep_increasing(e)
   @check_keep_increasing(i,iteration,accumulateEnergy).

FORALL e, i, v, v0, i0
  e[hasEnergyValue -> v]
    @update_hasEnergyValue(i,iteration,accumulateEnergy)
<-
  e:energy@postState(i0,iteration,accumulateEnergy) AND
  e[hasEnergyValue -> v0]
    @postState(i0,iteration,accumulateEnergy) AND
  add(v0, 1, v) AND
  add(i0, 1, i).

FORALL e, v, i, i0
  keep_increasing(e)
    @check_keep_increasing(i,iteration,accumulateEnergy)
<-
  less(v, 5) AND
  e:energy@postState(i0,iteration,accumulateEnergy) AND
  e[hasEnergyValue -> v]
    @postState(i0,iteration,accumulateEnergy) AND
  add(i0, 1, i).
```

Fig. 13. Iterative action—iteration rules.

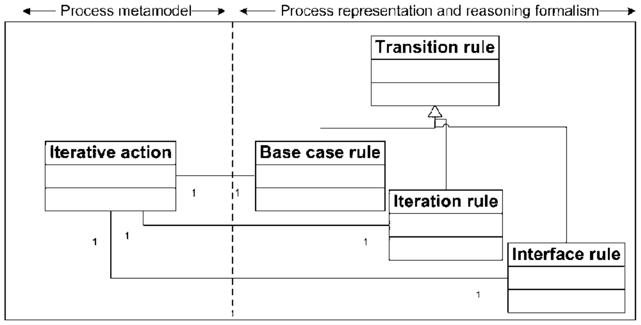


Fig. 14. Types of transition rules in iterative actions.

as domain rules. This type of rules implements the actual breakdown of the original iterative action into a sequence of actions for each iteration.

Fig. 13 shows the iteration rule itself plus the domain (*check* and *update*) rules implementing the action executed on the process frame at each iteration and the predicate evaluating the termination condition. In the example, the check rule implementing predicate *keep_increasing/1* evaluates whether or not the value of attribute *hasEnergyValue* of the instances of concept *energy* is less than 5. On the other hand, the update rule increases the value of *hasEnergyValue* at each iteration.

Since the *prestate* to be considered is not informed by the *poststate* of the preceding action but of the previous iteration instead, domain rules are rewritten slightly differently when used in iterative actions with respect to the case of atomic actions. In the example, inference results for update and check rules are, respectively, encapsulated in the Ontobroker modules *update_hasEnergyValue(i,iteration,accumulateEnergy)* and *check_keep_increasing(i,iteration,accumulate_energy)*.

Interface rules (Fig. 15) transfer the results from the *poststate* of the last iteration to the *poststate* of the overall iterative action to feed the *prestate* of subsequent actions. These rules validate that the *prestate* of the iterative action is fully and correctly instantiated and connect the output of the computation produced throughout the different iterations with its actual *poststate*. Interface rules also work at the level of the process metamodel, updating the roles

```
 FORALL e, i, v, m
 e:energy@postState(accumulateEnergy) AND
 e:RESOURCE@postState(accumulateEnergy) AND
 e[hasEnergyValue -> v]@postState(accumulateEnergy) AND
 accumulateEnergy[PROVIDES->e]@postState(accumulateEnergy)
<-
 m:mitochondrion@preState(accumulateEnergy) AND
 m:TOOL@preState(accumulateEnergy) AND
 e:energy@preState(accumulateEnergy) AND
 e:RESOURCE@preState(accumulateEnergy) AND
 e[IS_REQUIRED_BY -> accumulateEnergy]
   @preState(accumulateEnergy) AND
 m[IS_USED_BY -> accumulateEnergy]
   @preState(accumulateEnergy) AND
 e[hasEnergyValue -> v]
   @postState(i,iteration,accumulateEnergy) AND
 NOT EXISTS i0, v0 (
   e[hasEnergyValue -> v0]
     @postState(i0,iteration,accumulateEnergy) AND
   greater(i0, i)).
```

Fig. 15. Iterative action—interface rule.

associated with input and output process resources as well as their relations with the action.

Interface rules work in a way analogous to transition rules in atomic actions. They trigger the execution of the loop and detect its termination to propagate changes to subsequent actions. We use negation and existential quantification (highlighted in Fig. 15) to retrieve only the outcomes of the last iteration as the outcome of the whole action. Those outcomes are, thus, stored in the *poststate* of the overall iterative action. Rule stratification and a modular approach that encapsulates each iteration prevent undesired recursion and, therefore, well-founded semantics evaluation mode.

5 Optimization of the Synthesized Process Code

Our mechanism for the synthesis of code in the F-logic KRR language aims at minimizing the amount of process rules that need to be executed by OntoBroker in well-founded evaluation mode, preventing long reasoning times and memory consumption problems due to the overall complexity of the knowledge base. We accomplish such objective by a combination of the following two optimization methods.

5.1 Optimization 1: Prevention of Second Order

Reasoning requires F-logic predicate symbols to be transformed into Datalog. This is done through the Lloyd-Topor transformation [20] and can be achieved in two different ways. The first approach translates concept and attribute names into terms of a priori defined predicates like *subclassOf/2*, *instanceOf/2*, and *attributeType/3*, producing literals like *subclassOf(mammal, animal)*. Since concepts and attributes are always arguments, the resulting predicates will always be valid, even when concept and attribute names are variables, supporting second-order reasoning queries like *all the instances of class mammal*.

However, because the number of these predicates is small, all knowledge entities of the same type are joined together by the reasoner, e.g., the internal table corresponding to *subclassOf* will contain all the tuples of the classes from the knowledge base with an inheritance relation that connects them. The number of such tuples is potentially large. This hinders indexation and retrieval, because they cannot be uniquely identified by the name of the predicate (*subclassOf*), which is common to all of them. Consequently, reasoning can encounter performance problems in large knowledge bases in terms of time and memory consumed and of cycles over negation or aggregations, requiring the use of well-founded semantics.

The second method translates statements that represent instance-of and attribute-value relations into a form that introduces specific predicates for individual concepts and attributes. This corresponds to the standard interpretation of first order logic for these primitives. For example, *Peter is a person* is mapped to *person(Peter)*. This more efficient representation maps concepts and attributes to predicates and can be enabled only if all concept and attribute names in the knowledge base are ground, i.e., if there is no need for second-order reasoning. We follow this approach in the F-logic code automatically produced for processes, preventing attributes and concepts from appearing as free variables in the code and leveraging such optimization.

```
p(X) :- q(X), not r(X).    (1)
r(X) :- t(X).              (2)
q(a).
q(b).
t(a).
```

Fig. 16. Example of stratified logic program.

5.2 Optimization 2: Maximizing Stratified Code

In the presence of negation, inference rules implementing predicates used in negated subgoals of other rules must be completely evaluated before the evaluation of the rules depending on such predicates occurs. This usually requires costly evaluation modes like well-founded evaluation. However, if the corresponding Horn logic program follows a stratified model we can assure that it also follows a well-founded model and this particular evaluation mode is no longer necessary.

Dependency graphs are used to evaluate whether or not a logic program is stratified. In dependency graphs: 1) each predicate is a node, 2) edges from predicate p to predicate q occur if both occur in a rule, where q is in the head and p appears in the body, and 3) edges with negation are marked. If no cycles with negative edges appear, then the program is stratified and well founded.

A predicate is in the same stratum as all the predicates connected with it through positive edges in the dependency graph. If there is a negative edge leading from a predicate p to a predicate q, the stratum of q is one higher than the stratum of p. For example, if rule (1) in Fig. 16 is executed first, producing p(a) and p(b), the application of the known facts to rule (2) would produce r(a) as a result. However, r(a) should have excluded p(a) from the solution. By calculating the strata in a logic program, the reasoning engine evaluates the different rules in the right order, starting with the lowest stratum. With not stratified programs, this is not possible and well-founded evaluation is required.

6 Evaluation

This work was evaluated by an independent team in the context of project Halo.[10] A total of six knowledge formulation (KF) SMEs participated, who formulated knowledge on the selected evaluation syllabi in the scientific domains of Chemistry and Biology and tested reasoning with it. These knowledge bases were later used by five Question Formulation (QF) SMEs, with the support of QF KEs, who formulated selected AP-level questions that were intended to be answered by the system. After receiving a limited amount of training, SMEs were isolated from developers, evaluators, and other SMEs and formulated the knowledge contained in the syllabi.

The scope of this evaluation went beyond usability aspects in a formative sense and aimed at providing empirical assessment of the coverage and performance of the system in a setting that is representative in terms of the profile of the recruited SMEs and their assigned tasks. In the

10. Vulcan Inc.'s Project Halo http://www.projecthalo.com.

TABLE 2
Processes Modeled by the Evaluation KF SMEs

	Number of processes modeled
SME2(Biology)	2
SME3(Biology)	6
SME5(Chemistry)	3
Total	11

particular case of process knowledge, the evaluation paid special attention to both direct SME feedback on process KF and reasoning performance, including the different optimizations applied to the mechanism for process code synthesis described herein. In this paper, we focus on the latter (reasoning performance evaluation), while the former, including complete usability and utility studies, was described in detail in [15].

The evaluation syllabi are summarized next:

- *Chemistry:* Sections 3.1 and 3.2 (pp. 75-83) on Stoichiometry, 4.1-4.4 (pp. 113-133) on aqueous reactions and solutions, and 16.1-16.11 (pp. 613-653) on chemical equilibrium, from [8].
- *Biology:* pp. 112-124, pp. 217-223, and pp. 239-245 on cell structure and cell processes, including mitosis and meiosis, and pp. 293-201 and pp. 304-311 on DNA structure and DNA structure processes, including DNA replication, repair, transcription, and translation, from [9].

Our SMEs modeled 806 concepts, 741 instances, 260 attributes, 273 relations, 610 rules, and 11 processes. The mechanism for process code synthesis generated an average of nine knowledge base modules per process, where action *prestates* and *poststates* and results from *check* and *update* process rules, where stored before the final results flowed into the main knowledge base module. Table 2 shows the distribution of the processes modeled by the SMEs across the domains.

As expected, we found in Biology (SME2 and SME3) the largest population of processes among the three domains. SME5 also produced a considerable number of process models in Chemistry, typically a poorer domain in terms of process knowledge. All process models were formulated by SMEs without intervention of KEs. SMEs only required initial training and sporadic support in the utilization of the tools.

The quality of the resulting knowledge bases was evaluated through the testing and debugging facilities of the system to check that SME-generated process models actually behaved as expected during execution. This tooling allows SMEs to create test sets containing process inputs and expected outputs, invoking the process, and comparing their results against expected results. As shown in [15], at modeling time the process editor continuously checks the compliance of process models with respect to the process formalism. This provides SMEs with modeling guidelines that make the process formulation task easier and at the same time prevents incorrect processes from being saved into the knowledge base and ensures the generation of stratified code. The testing and debugging tool showed 82 percent of the process models executed correctly, showing evidence of the high expressiveness and coverage of the approach. The remaining 18 percent process models were not executed either because they had design errors, and consequently were not saved in the knowledge base, or because the required preconditions for their execution did not hold.

TABLE 3
Performance of C1 and C2 versus Default C0

Query	C0	C1	s(C1)	C2	s(C2)
SME3-q0	31	0	0,00	16	0,52
SME3-q1	63	16	0,25	16	0,25
SME3-q2	31	16	0,52	16	0,52
SME3-q3	47	16	0,34	16	0,34
SME3-q4	15	0	0,00	0	0,00
SME3-q5	32	16	0,50	0	0,00
SME3-q6	203	219	1,08	234	1,15
SME3-q7	63	31	0,49	31	0,49
SME3-q8	47	31	0,66	16	0,34
SME3-q9	62	32	0,52	16	0,26
SME3-q10	203	218	1,07	203	1,00
Average	79,7	59,5	0,75	56,4	0,71
Median	47	16	0,34	16	0,34
Min	15	0	0,00	0	0,00
Max	203	219	1,08	234	1,15

As to execution performance, we studied the effects of the application of the optimizations described in Section 5 to the F-logic code resulting from the process models formulated by the SMEs. Since the Biology knowledge base produced by SME3 contained the largest sample of process knowledge produced in the evaluation, we focused on it to measure response times of a representative sample of ten queries with three different configurations of OntoBroker combining different uses of well-founded evaluation and second-order reasoning . From the sample types of process reasoning introduced in Section 1, q1 deals with reasoning about process entities, q7-q10 refer to intermediate results while collecting final results, q2 illustrates reasoning on process stages, and q0, q3, and q4 aim at retrieving additional process metadata.

Among such configurations, C0 represents the default, with the well-founded evaluation mode enabled. C1 and C2 apply the optimization methods described in Section 5. C1 aims at increasing performance with respect to C0 by enabling concept and attribute names ground while C2 extends C1 by additionally disabling well-founded evaluation.

The results of executing this query set with the three different configurations are shown in Table 3. Response times are measured in milliseconds (values equal to 0 correspond to queries with response times lower than 1 ms). Shaded columns s(C1) and s(C2) show the speedup obtained with respect to C0 (values inferior to 1 indicate performance increase) by applying configurations C1 and C2, respectively. The table shows an average performance improvement of 25 percent for C1 and almost 30 percent for C2. The main improvement factor is the consequence of concept and attribute names being ground both in C1 and C2. C2 adds little performance beyond C1 because the code generation mechanism produces most of the code in well-stratified form, hence reducing the need of well-found semantics. Furthermore, C1 still allows applying well-founded evaluation in the eventuality of nonstratified code. It can be concluded that configuration C1 shows an appropriate balance between safeness and performance for the generated code, with a significant speedup over the default configuration C0.

7 Conclusions

We have presented a process representation formalism and an optimized method for the automatic generation of high-performance, executable process models in a knowledge representation language (F-logic). Our method bridges the gap between processes modeled graphically at the knowledge level by SMEs and their formal representation, which follows the formalism and is grounded in the language. Evaluation results provide evidence that our approach effectively and efficiently serves this purpose, providing process models with an operational semantics so that process-related reasoning can be supported. The insight obtained has contributed to configure process work in subsequent stages of the Halo project. However, exciting research challenges about the process knowledge type still need to be addressed that will be subject of our research work in the coming years.

The analysis of the AP questions used in this paper to illustrate reasoning with process knowledge and specific comments from SMEs during evaluation evidences a lacking support for QF to enable SMEs to formulate questions involving processes in natural and expressive ways. Current QF approaches based on controlled vocabularies, like [12], do not completely address the specific problem of querying process knowledge nor exploit all the expressive capabilities of process-specific representation frameworks like the one presented herein. The translation of process-related questions, expressed in natural or controlled language, into formal queries with minimal expressivity loss remains a problem that still needs to be addressed.

During the evaluation, it also became evident that the application of the methods presented in this paper to other domains, like business or software development, will benefit from extending the metamodel and the process code generation mechanism with new control flow primitives that support more complex process behavioral patterns, like parallelism or activity decomposition. This will allow exploiting the underlying process formalism in a richer way that can effectively support process reasoning in new domains.

Finally and most interestingly, we anticipate research challenges dealing with the acquisition and sharing of knowledge by online user communities, their representation and reasoning, raising problems like non monotonicity, inconsistencies between distributed but interacting knowledge bases, performance, and scalability. A particularly interesting research problem, which is especially relevant in distributed and collaborative environments for knowledge acquisition, deals with the detection of decay of higher level knowledge entities, especially processes, derived from eventual changes in the knowledge base that may render such knowledge entities useless [28]. Answer Explanation methods able to appropriately keep track of process reasoning mechanisms will be important both to detect process decay and to provide SMEs with interpretations of such reasoning at the required level of abstraction.

Acknowledgments

This work was funded by Vulcan Inc.'s project Halo.

References

[1] J. Angele, "How to Write F-Logic Programs," technical report, Ontoprise GmbH, 2005.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. Fifth ACM SIGACT-SIGMOD Symp. Principles of Database Systems (PODS),* 1986.

[3] K. Barker, J. Blythe, G. Borchardt, V. Chaudhri, P. Clark, P. Cohen, J. Fitzgerald, K. Forbus, Y. Gil, B. Katz, J. Kim, G. King, S. Mishra, K. Murray, C. Otstott, B. Porter, R. Schrag, T. Uribe, J. Usher, and P. Yeh, "A Knowledge Acquisition Tool for Course of Action Analysis," *Proc. 15th Innovative Applications of Artificial Intelligence Conf. (IAAI '03),* 2003.

[4] V.R. Benjamins and M. Aben, "Structure-Preserving Knowledge-Based System Development through Reusable Libraries: A Case Study in Diagnosis," *Int'l J. Human-Computer Studies,* vol. 47, no. 2 pp. 259-288, 1997.

[5] C. Bock and M. Grüninger, "PSL: A Semantic Domain for Flow Models," *Software and Systems Modeling J.,* vol. 4, pp. 209-231, 2005.

[6] A. Bonner and M. Kifer, "An Overview of Transaction Logic," *Theoretical Computer Science,* vol. 133, no. 2, pp. 205-265, Oct. 1994.

[7] S. Brandano, "The Event Calculus Assessed," *Proc. IEEE Eighth Int'l Symp. Temporal Representation Reasoning (TIME),* pp. 7-12, 2001.

[8] T. Brown, H. Lemay, B. Bursten, and J. Burdge, *Chemistry: The Central Science,* ninth ed. Prentice Hall, 2002.

[9] N. Campbell and J. Reece, *Biology,* sixth ed. Pearson Higher Education, 2001.

[10] V. Chaudhri, B. John, S. Mishra, J. Pacheco, B. Porter, and A. Spaulding, "Enabling Experts to Build Knowledge Bases from Science Textbooks," *Proc. Fourth Int'l Conf. Knowledge Capture (K-CAP),* 2007.

[11] W.J. Clancey, "Heuristic Classification," *Artificial Intelligence,* vol. 27, pp. 289-350, 1985.

[12] P. Clark, S. Chaw, K. Barker, V. Chaudhri, P. Harrison, J. Fan, B. John, B. Porter, A. Spaulding, J. Thompson, and P. Yeh, "Capturing and Answering Questions Posed to a Knowledge-Based System," *Proc. Fourth Int'l Conf. Knowledge Capture (KCAP),* 2007.

[13] E. Feigenbaum, "The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence,* 1977.

[14] A. Gibbons, *Algorithmic Graph Theory.* Cambridge Univ. Press, 1985.

[15] J.M. Gómez-Pérez, M. Erdmann, M. Greaves, O. Corcho, and R. Benjamins, "A Framework and Computer System for Knowledge-Level Acquisition, Representation, and Reasoning with Process Knowledge," *Int'l J. Human-Computer Studies,* vol. 68, pp. 641-668, Oct. 2010.

[16] B.C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler, "Just the Right Amount: Extracting Modules from Ontologies," *Proc. Sixth Int'l Conf. World Wide Web (WWW '07),* pp. 717-726, May 2007.

[17] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," *J. ACM,* vol. 42, pp. 741-843, 1995.

[18] M. Krötzsch, D. Vrandecic, M. Völkel, H. Haller, and R. Studer, "Semantic Wikipedia," *J. Web Semantics,* vol. 5, pp. 251-261, 2007.

[19] S. Liang, P. Fodor, H. Wan, and M. Kifer, "OpenRuleBench: An Analysis of the Performance of Rule Engines," *Proc. 18th Int'l Conf. World Wide Web (WWW '09),* pp. 601-610, 2009

[20] J.W. Lloyd and R.W. Topor, "Making Prolog More Expressive," *J. Logic Programming,* vol. 1, no. 3, pp. 225-240, 1984.

[21] J. McDermott, "Preliminary Steps Towards a Taxonomy of Problem-Solving Methods," *Automating Knowledge Acquisition for Expert Systems,* S. Marcus, ed., pp. 225-255, Kluwer, 1988.

[22] D.L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview," W3C Recommendation, Feb. 2004.

[23] D. Morley and K. Myers, "The SPARK Agent Framework," *Proc. Third Int'l Joint Conf. Autonomous Agents and Multi Agent Systems (AAMAS '04),* pp. 712-719, 2004.

[24] A. Newell, "The Knowledge Level," *Artificial Intelligence,* vol. 18, no. 1 pp. 87-127, 1982.

[25] K. Panton, P. Miraglia, N. Salay, R. Kahlert, D. Baxter, and R. Reagan, "Knowledge Formation and Dialogue Using the Kraken Toolset," *Proc. 14th Conf. Innovative Applications of Artificial Intelligence (IAAI '02),* pp. 900-905, 2002.

[26] Z.W. Pylyshyn, *The Robot's Dilemma: The Frame Problem in Artificial Intelligence.* Norwood, 1987.

[27] D. Rajpathak, E. Motta, Z. Zdrahal, and R.K Roy, "A Generic Library of Problem Solving Methods for Scheduling Applications," *IEEE Trans. Knowledge and Data Eng.,* vol. 18, no. 6, pp. 815-828, June 2006.

[28] D.D. Roure et al., "Towards the Preservation of Scientific Workflows," *Proc. Eighth Int'l Conf. Preservation of Digital Objects (iPRES '11),* 2011.

[29] L.K. Schubert and C.H. Hwang, "Episodic Logic Meets Little Red Riding Hood: A Comprehensive, Natural Representation For Language Understanding," *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language,* L. Iwanska and S.C. Shapiro, eds., pp. 111-174, MIT/AAAI Press, 2000.

[30] M. Thielscher, "Introduction to the Fluent Calculus," *Electronic Trans. Artificial Intelligence,* vol. 2, no. 34, pp. 179-192, 1998.

[31] A. Valente andTeam Omniscience, *Project Halo Analysis Report,* May 2004.

[32] A.V. Gelder, K.A. Ross, and J.S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *J. ACM,* vol. 38, no. 3, pp. 620-650, 1991.