Constructive DL update and reasoning for modeling and executing the orchestration of heterogeneous processes [⋆]

Serge Autexier and Dieter Hutter

German Research Center for Artificial Intelligence (DFKI), Bremen, Germany
{serge.autexier|dieter.hutter}@dfki.de

Abstract. Our digital world is characterized by various concurrent processes that interact with the physical world. Each of them follows its own rules and goals and only few of them are under a central control. While usually a human mediates between these processes, we developed the SHIP-tool to orchestrate these processes automatically. It uses Description Logic to represent a current state, which is constantly updated by messages sent by sensors and other processes. Temporal-logical formulas act as monitors that supervise the evolution of the system. The failure or success of monitors can initiate procedures given as programs on actions and specified in a Dynamic Logic. In this paper we describe in particular the aspect of storing and maintaining the state of the system in Description Logic ontologies. We formulate restrictions on ABoxes and their updates to always ensure a constructive specification of the current state while keeping the general rules of the system (i.e the TBox and RBOX) invariant.

1 Introduction

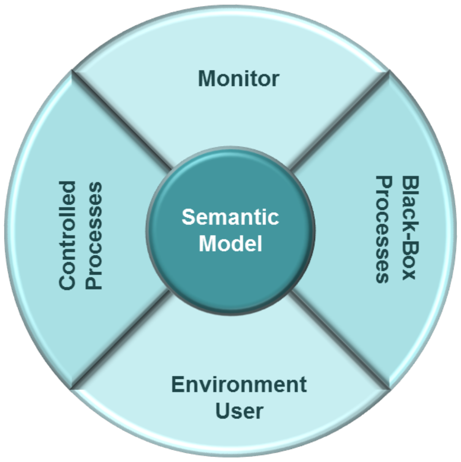
While independently developed computer systems obtain more and more the ability to communicate with each other on a technical level, there is an upcoming need to orchestrate and combine the processes performed by these systems on an operational or work-flow level. In the past, typically humans were responsible to supervise the processes and to translate and exchange important information between them. The SHIP-Tool is an attempt to automate this task and is based on an abstract (logic) symbolic representation of an actual situation, the ways the processes can change a situation, and the expectations of future developments in a situation.

It provides an implementation, execution and simulation platform for work-flow processes allowing the user to integrate and monitor existing devices, services or even entire systems. This is achieved by providing a logical formalism to describe the real world in an abstract way. Changes in the real world are communicated to the tool in terms of updates changing its abstract representation.

[⋆] This work is supported by the German Federal Ministry of Education and Research (BMBF) under grant 01 IW 10002 (SHIP)

Vice versa, the tool can initiate changes in the real world by executing actions or processes.

Complex behaviors can be defined guided by changes in the abstract world. The behavior of the world is monitored. Process descriptions can be provided to react on successful or failed developments of the world. This enables us, for instance, to monitor the behavior of a black-box device or some users and to react to any non-expected behavior. At the other end of the spectrum it allows us to define complex activities and work-flows in terms of processes acting on existing devices and services and communicating with the user. The interweavement of process descriptions with monitoring as well as the interleaved execution of parallel processes, both based on the same world representation, provides a powerful formalism to implement processes to assist and monitor activities in the environment in interaction with users and based on existing services and devices.



Using Description Logic to represent the environment allows for processes being dynamic description logic (DDL) programs [7] over environment representations in description logic ontologies. Monitors can be invoked from dynamic logic processes: if the behavior is entirely observed, the monitor invocation is successful (it may well never terminate if it is an invariant that always holds, or expecting eventually a property to hold, but which never holds). If the behavior is violated, the monitor invocation process fails and can be dealt with on the dynamic logic level like any other failing process.

2 The Basic Concepts

Modeling the World. The SHIP-Tool uses description logic [9, 4] to represent the knowledge of an application domain in an ontology. The syntax SHIP-DL is an extension of standard DL to supports modularization and renaming and syntactic sugar to ease the declaration of abstract datatypes.

Given a set N_C of concept names, a set N_R of role names and a set of N_I of individuals, concepts are defined inductively from N_C and roles from N_R. We introduce the description logic SROIQ, which will be used for queries, but only its SRIQ fragment will be used to define the ontology. For SROIQ, concepts are formed using the constructors given in Table 1, where `T` and `F` are predefined top and bottom atomic concepts, `C` and `D` are either concept names from N_C or

Name	`Syntax`	Semantics under interpretation $\mathcal{I}$
Top	`T`	$\Delta^{\mathcal{I}}$
Bottom	`F`	$\emptyset$
Atomic concept	$c \in N_C$	$c^{\mathcal{I}}$
Intersection	$\mathtt{C} \sqcap \mathtt{D}$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$\mathtt{C} \sqcup \mathtt{D}$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg\mathtt{C}$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Value Restriction	$\forall\mathtt{R.C}$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R \Rightarrow b \in C^{\mathcal{I}}\}$
Existential quant.	$\exists\mathtt{R.C}$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R \Rightarrow b \in C^{\mathcal{I}}\}$
Number Restriction	$=$ `n R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\rvert = n\}$
	$\geq$ `n R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\rvert \geq n\}$
	$\leq$ `n R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\rvert \leq n\}$
Nominal concept	$\{i_1, \ldots, i_n\}$	$\{i_1^{\mathcal{I}}, \ldots, i_n^{\mathcal{I}}\}$

Table 1. SROIQ Syntax and Semantics

complex concept expressions formed accordingly, `R` is either a role name from N_R or the inverse of a role name $\mathtt{R}^{-1}$, and i_j are individual names from N_I.

The terminological part of an ontology consists of definitions and inclusion axioms of concept and role names. Concept names `c` can be declared as subconcepts of concepts by concept inclusion axioms $\mathtt{c} \sqsubseteq \mathtt{D}$ or defined by concept equality `c = D`. Disjointness of concepts is declared by `Disjoint`. A concept name `c` directly depends on a concept name `d` if `c` is defined by $\mathtt{c} = \mathtt{D}$ and `d` occurs in `D`. A concept name `c` depends on a concept name `d`, if `c` directly depends on `d` or there exists a concept name `e` which depends on `d` and `c` directly depends on `e`. We denote the dependency relation between concept names by $>_c$.

Roles are declared by indicating their domain and range. Similar to concepts, subroles can be declared by limited complex role inclusion axioms $\mathtt{r_0}.\ldots.\mathtt{r_n} \sqsubseteq \mathtt{r}$ where $r \in N_R$ and '.' denotes role composition. Furthermore, role names can be defined as the composition of roles $\mathtt{r} = \mathtt{r_0}.\ldots.\mathtt{r_n}$ or as the reflexive, transitive closure of another role $\mathtt{r} = \mathtt{r_0}*$. This is not expressible in description logic and it is translated to $\mathtt{r_0} \sqsubseteq \mathtt{r}$, `Trans(r), Ref(r)` when translating to DL. However, it is important as a meta-property to ensure that nothing else than the transitive closure of $\mathtt{r_0}$ is in r when enforcing constructiveness of ontologies and it will be used there. SHIP supports the standard role properties `Sym`, `Asym`, `Trans`, `Ref`, `Irref`, `Func`, and `FuncInv`). A role name directly depends on a role name $\mathtt{r}'$ if `r` is defined by $\mathtt{r} = \mathtt{R}$ and $\mathtt{r}'$ occurs in `R`. A role name `r` depends on a role name $\mathtt{r}'$, if `r` directly depends on $\mathtt{r}'$ or there exists a role name $\mathtt{r}''$ which depends on $\mathtt{r}'$ and `r` directly depends on $\mathtt{r}'$. We denote the dependency relation between role names by $>_r$.

For SHIP, a TBox consists of the concept inclusions, definitions and disjointness assertions from Table 2 and those obtained to encode role declarations $\mathtt{r{:}C{\times}D}$. An RBox consists of role inclusions and definitions as show in Table 2 and property assertions. Furthermore, we require the induced dependency rela-

	φ	$\mathcal{I} \models \varphi$ if ...
Concept inclusion	$c \sqsubseteq \mathtt{C}$, $\mathtt{c} \in \mathrm{N_c}$, C in SRIQ	$c^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
Concept definition	c = C, $\mathtt{c} \in \mathrm{N_c}$, C in SRIQ	$c^{\mathcal{I}} = C^{\mathcal{I}}$
Concept disjointness	Disjoint(c,d), $\mathtt{c}, \mathtt{d} \in \mathrm{N_c}$	$c^{\mathcal{I}} \cap d^{\mathcal{I}} = \emptyset$
Role declaration	$\mathtt{r} : \mathtt{C} \times \mathtt{D}$, $\mathtt{r} \in \mathrm{N_r}$, C, D in SRIQ	$R^{\mathcal{I}} \subseteq C^{\mathcal{I}} \times D^{\mathcal{I}}$
Role inclusion	$\mathtt{r}_0 . \ldots . \mathtt{r}_n \sqsubseteq \mathtt{r}$, $\mathtt{r} \in \mathrm{N_R}$, $n \geq 1$	${r_0}^{\mathcal{I}} \circ \cdots \circ {r_n}^{\mathcal{I}} \subseteq r^{\mathcal{I}}$
Role definition	$\mathtt{r} = \mathtt{r}_0 . \ldots . r_n$, $\mathtt{r} \in N_R$, $n \geq 1$	${r_0}^{\mathcal{I}} \circ \cdots \circ {r_n}^{\mathcal{I}} = r^{\mathcal{I}}$ where $R^{\mathcal{I}} \circ R'^{\mathcal{I}} = \{(x,z) \mid \exists y.(x,y) \in R^{\mathcal{I}}, (y,z) \in R'^{\mathcal{I}}\}$.
	$\mathtt{r} = \mathtt{r}_0*$, $\mathtt{r} \in N_R$	${r_0}^{\mathcal{I}^*} \subseteq r^{\mathcal{I}}$
Role properties	Fun(r), Invfun(r), Ref(r), Irref(r), Trans(r), Sym(r), Asym(r), $r \in N_r$	as usual

Table 2. TBox- and RBox-assertions and their satisfiability in an interpretation $\mathcal{I}$

tion on concept and role names to be irreflexive (i.e. we consider acyclic TBoxes and RBoxes modulo the transitivity of roles). The TBox and RBox form the *terminological* part of a SHIP-ontology.

In addition we use the following syntactic sugar to ease the definition of ontologies in a style inspired by abstract datatypes:

- $\mathtt{c} ::= \mathtt{C}_1(\mathtt{r}_{11} : \mathtt{D}_{11}, \ldots, \mathtt{r}_{1\mathtt{n}_1} : \mathtt{D}_{1\mathtt{n}_1}) | \ldots | \mathtt{C}_\mathtt{m}(\mathtt{r}_{\mathtt{m}1} : \mathtt{D}_{\mathtt{m}1}, \ldots, \mathtt{r}_{\mathtt{mn}_\mathtt{m}} : \mathtt{D}_{\mathtt{mn}_\mathtt{m}})$, $c \in N_C$ is expanded to
  - $\mathtt{c} = \mathtt{C}_1 \sqcup \ldots \sqcup \mathtt{C}_\mathtt{n}$ and
  - $\mathtt{Disjoint}(\mathtt{C}_\mathtt{i}, \mathtt{C}_\mathtt{j})$ $\forall i \neq j$.
  - $\mathtt{C}_\mathtt{i} \sqsubseteq (\exists \mathtt{r}_{\mathtt{i}1}.\mathtt{D}_{\mathtt{i}1}) \sqcap \ldots \sqcap (\exists \mathtt{r}_{\mathtt{in}_\mathtt{i}}.\mathtt{D}_{\mathtt{in}_\mathtt{i}})$ $\forall 1 \leq i \leq m$, and
  - $\mathtt{r}_{\mathtt{ij}} : \mathtt{C}_\mathtt{i} \times \mathtt{D}_{\mathtt{ij}}$, $\mathtt{Fun}(r_{ij})$ $\forall 1 \leq i \leq m, 1 \leq j \leq n_i$.

As an example consider the DL declarations defining Lists over elements Elem.

```
List ::= EmptyList   | NonEmptyList(hd:Elem,tl:List)
```

which is expanded to

```
List = EmptyList ⊔ NonEmptyList
Disjoint(EmptyList, NonEmptyList)
NonEmptyList ⊑ (∃hd . Elem) ⊓ (∃tl . List)
hd:NonEmptyList × Elem
Fun(hd)
tl:NonEmptyList × List
Fun(tl)
```

The ABox part consists of a list of concept assertions a:C, C in SRIQ, declaring a to be a member of the concept C, and role assertions (a,b):R, stating that relation R holds between the individuals a and b. For every SHIP-DL-ontology we assume the unique name assumption. Complex ontologies can be composed by importing and renaming existing ontologies. For instance from the List-ontology before we build an ontology for Routes specifying routes along Positions by

⟨*monitor*⟩ ::= ‘`monitor`’ ⟨*string*⟩ ⟨*params*⟩ ‘`=`’ ⟨*foltl*⟩

⟨*foltl*⟩ ::= ⟨*aboxass*⟩ | ‘`not`’ ⟨*aboxass*⟩ | ⟨*string*⟩ ⟨*params*⟩ | ⟨*foltl*⟩ ‘`and`’ ⟨*foltl*⟩
| ⟨*foltl*⟩ ‘`or`’ ⟨*foltl*⟩ | ⟨*foltl*⟩ ‘`=>`’ ⟨*foltl*⟩ | (‘`all`’ | ‘`ex`’) ⟨*aboxass*⟩ ‘`.`’ ⟨*foltl*⟩
| ‘`(`’ ⟨*foltl*⟩ ‘`)`’ | (‘`X`’|‘`F`’|‘`G`’) ⟨*foltl*⟩ | ⟨*foltl*⟩ ‘`U`’ ⟨*foltl*⟩

Fig. 1. SHIP-TL language to monitor ontology updates

```
import basic.Lists with
 concepts Elem as Position, List as Route, EmptyList as
     EmptyRoute, NonEmptyList as NonEmptyRoute
 roles hd as route_next, tl as route_rest
 individuals nil as emptyroute
```

where `nil:EmptyList` is declared in the `List`-ontology as an empty list.

Throughout the rest of the paper we will denote by $O \vdash \varphi$ that φ is satisfied by the ontology O, where φ can be a TBox, RBox or ABox-assertion.

Monitors. SHIP uses linear temporal logic (LTL) [11] to observe temporal properties about ontology updates. In practice, these properties are used to express an expected behavior of the environment, or to detect specific situations over time that require some actions. We call programs that monitor such properties simply *monitors*. In SHIP, monitors can be started at any time and are evaluated w.r.t. a finite set of ontology updates that occurred between the start of the monitor up to the current time or its termination. Thereby, it does not matter whether the update was performed by a SHIP-process or the environment.

When monitoring a property, different situations can arise: (i) the property is satisfied after a finite number of steps – in this case, the monitor evaluates to true and terminates successfully; or (ii) the property evaluates to false independently of any continuation – in this case, the monitor terminates with failure; or (iii) the behavior that was observed so far allows for different continuations that might both lead to a failure as well as success – in this case, the monitor is still running.

The language is a first-order temporal logic over description logic ABox-assertions and similar to that used in [2]. First-order quantification is over individuals, which are flexible constants from a logic point of view, and interpreted over the current world. Hence, quantification is over the finite set of individuals in the current ontology at the moment when the quantifiers are expanded. This can be arbitrarily often as in `F(all x . x:C)` and not be determined beforehand. We use the formulas as part of the process language and to reason about the SHIP processes, though. Fig. 1 shows the grammar for monitors. The temporal operators are `X` indicating the next world, `F` indicating some point in the future, `G` stating that a property must hold globally, and `U` stating that a property must hold until another property becomes true. As ABox-assertions in the monitor language we allow for concept assertions `a` : `C`, where `C` is from the *full* SROIQ and not only from SRIQ as in the ABox.

As an example for a monitor formula we consider a use case where different autonomous wheelchairs operate in a building in which they can ride along routes consisting of positions. The SHIP-Tool has been used to add a control layer on top of the wheelchairs to avoid conflicting situations by scheduling the routes of the wheelchairs. One simple behavior one wants to monitor is when a new schedule must be computed. The part of the ontology specifying `Routes` has been introduced before. For the purpose of illustrating this monitor, we use the following TBOX/RBOX-assertions:

```
IDObject ::= WheelChair(whc_route:Route, whc_carries:OptPerson)
           | OptPerson
OptPerson = Person | Nobody
WCEmptyroute = WheelChair ⊓ ∃whc_route . EmptyRoute
WCNonEmptyroute = WheelChair ⊓ ∃whc_route . NonEmptyRoute
```

It defines `WheelChairs` to have a route and carrying a `Person` or `Nobody`. Subconcepts are wheelchairs having an empty resp. a non-empty route. To know if a new schedule must be computed, we must successfully observe the behavior:

```
(∃wc:WCEmptyroute.F(wc:WCNonEmptyroute)) or
(∃wc:WCNonEmptyroute.
          F(wc:WCEmptyroute and F(wc:WCNonEmptyroute)))
```

This expresses that either in the current situation we have a wheelchair with an empty route and that wheelchair eventually gets a non-empty route. Or there is a wheelchair which currently has a non-empty route, eventually gets an empty route and then again eventually a non-empty route. If one of these behaviors is observed, then we must compute a new schedule. The actual monitoring is performed by formula progression (see, e.g., [5]) after each update. Although formula progression in principle has some disadvantages over the translation into automata, we use formula progression as at each state of the progression the expectations on the future are immediately available in a formula and can be used for inspection, for instance to determine suitable next actions. Note that we are at this stage not interested in proving properties about SHIP-processes, but rather use it at run-time to observe behavior of the environment. Hence, even trivially true formulas like `X(X(X True))` is not superfluous, but a valid means to observe four clock-steps.

Another example making use of the expressivity provided by SROIQ is when some light `l` in an area of the flat should be on depending on the current day time `time`, planned rides of the wheelchairs and where the occupants of the flat currently are:

```
monitor LightShouldBeOn (time,l) =
 time:Night and l:AbstractOff and
 ∃area:(Area ⊓ (∃associated_light⁻¹. { l })) .
 (// 1. Either some wheelchair has a non-empty route, and ...
  (∃wc:WCNonEmptyroute.    (
   // carries a person or there is a person in the area...
   (wc:WCCarriesPerson or ∃person:(Person ⊓ (∃elementIsInArea
        . { area })) . true)
   and // ... the wheelchair is already in or about to enter
   ((wc,area):elementIsInArea or (wc,area):wcnextarea)))
```

Name	Syntax	Semantics
simple condition	`if a then b else c`	branches according to a or waits until a can be decided
complex condition	`switch` `case c`$_1$ `=> p` . . . `case c`$_k$ `=> p`	branches according to the specified cases which are checked in order. _ can be used as default case, then, the condition never stutters.
iteration	`p*`	applies p until it fails, always succeeds
sequence	`p ; q`	applies p then q
monitor start	`init m`	starts the monitor, continues when the monitor succeeds or fails when the monitor fails
guarded execution	`p +> q`	executes p; if p fails, q is executed, but the modifications of p are kept
parallel	`p` || `q`	executes p and q in parallel (interleaved), terminates when both p and q terminate, fails when one of them fails
bounded parallel	`forall c => p`	executes p for all instances matching c in parallel, terminates when all terminate, fails when one of them fails

Table 3. Process combinators

```
or // the wheelchair is assigned to a person in the same area
 (∃person:(Person ⊓ (∃elementIsInArea . { area })) .
  ∃wc:(WheelChair ⊓ (∃wcAssignedToPerson . { person })) .
     (wc,area):elementIsInArea))
```

Actions and Processes. The SHIP processes are dynamic description logic processes based on actions and similar to those in the literature (e.g, [7]). Actions $\alpha(\boldsymbol{p}) = \langle \mathsf{pre}, \mathsf{eff} \rangle$ represent parametrized atomic interactions with the environment and consist of a finite set of ABox assertions pre, the preconditions, and a set eff of conditional effects φ/ψ, where φ and ψ are ABox assertions. An action is applicable on an ontology O if all its preconditions are satisfied in O, i.e., $O \vdash \mathsf{pre}$. In this case, O is updated to a new ontology O' by applying all conditional effects whose conditions also hold in O. If O' is inconsistent, then the action fails and we keep O as the current ontology. Otherwise the action succeeds and O' is the new current ontology. If the preconditions are not satisfied, the action stutters, i.e., waits until it gets applicable. `skip` is the action which is always applicable and does not modify the ontology.

Using free variables in the effects, new individuals can be added to O, where a fresh name is created at run-time. By annotating a variable with `delete`, individuals can also be removed from the ontology. This is necessary, for instance, to remove individuals whose counterpart in the real world no longer exists.

SHIP provides a language to define complex processes that interact with the environment. Starting from *actions*, complex system behaviors can be described by combining other processes as well as by interacting with monitors. Starting from atomic actions, more complex processes can be defined using the combinators shown in Table 3.

3 States and Their Updates

The states of a specified system are represented by ABoxes with respect to fixed TBox and RBox and the transition of states corresponds to ABox-updates. Thus ABox-updates do not change the underlying TBox/RBox semantics characterizing the overall system states. Hence the approach to ABox-updates from [8] is not suitable for our setting and we are in the spirit of the updates in [6]. However, we want to constrain our ontology in order to have checkable properties ensuring the completeness of ontologies to make the computation of ontologies resulting from updates efficient.

Following this line of reasoning we interpret the construction of complex concepts as a kind of specification for abstract datatypes. The use of an existential quantification in $\mathtt{IDObject} \sqsubseteq \exists \mathtt{at}\ .\ \mathtt{AbstPosition}$ corresponds to the definition of a mandatory attribute "at" indicating a position in the abstract datatype denoted by IDObject. Conjunction of concepts combines the attributes of the corresponding datatypes while disjunction of concepts resembles the notion of variants.

3.1 Constructive Ontologies

Similar to an initialization of all records in new instances of a data-structure we want to enforce a constructive definition of each individual of a complex concept in an ABox. For instance, having an individual d for the concept IDObject above, we know that d has a position and we demand that the ABox should also provide the knowledge which position d actually has. Furthermore, since we allow for disjunction of concepts $\mathtt{D} \sqsubseteq \mathtt{E} \sqcup \mathtt{F}$ we also want to know for each individual in D whether it belongs to E or F or both. That means that for any individual d of a (complex) concept the ABox always provides the full information about the composition and settings of the individual. In other words, the ABox provides the individuals necessary to name the values of the various attributes and there is no need to invent new values by introducing Skolem functions.

The same rigor of constructiveness is applied to the specification of roles. SHIP allows for the definition of composed roles, e.g. by defining $\mathtt{r} = \mathtt{r}_1 \cdot \mathtt{r}_2$. Knowing that two individuals a, b are in the relation r there must be some individual c such that $(\mathtt{a}, \mathtt{c}) : \mathtt{r}_1$ and $(\mathtt{c}, \mathtt{b}) : \mathtt{r}_2$ holds. We demand that also this witness is specified explicitly, i.e. the ABox contains some individual c and the necessary relations between c and the individuals a and b.

These properties can be formalized in the following definition of constructive ontologies, where $\overline{E}$ denotes the negation of concept E in negation normalform.

Let $O := \langle T, R, A \rangle$ be an ontology wrt. concept names N_C, role names N_R and individual names N_I. The signature of O is $\Sigma_O := (\Sigma_C, \Sigma_R, \Sigma_I) \subseteq (N_C, N_R, N_I)$ of those names of concepts, roles and individuals occurring in O. An ontology O is *constructive* iff the followings hold:

1. if $a \in \Sigma_I$, $O \vdash \mathtt{a} : \mathtt{D}$ and $O \vdash \mathtt{D} \sqsubseteq \exists \mathtt{r}.\mathtt{E}$ then there is some $b \in \Sigma_I$ with $O \vdash \mathtt{b} : \mathtt{E}$ and $O \vdash (\mathtt{a}, \mathtt{b}) : \mathtt{r}$

2. if $a \in \Sigma_I$, $O \vdash \mathtt{a} : \mathtt{D}$ and $O \vdash \mathtt{D} \sqsubseteq (\leq \mathtt{n}\ \mathtt{r}.\mathtt{E})$ then there exist distinct $b_1, \ldots, b_n \in \Sigma_I$ with $O \vdash \mathtt{b_i} : \mathtt{E}$ and $O \vdash (\mathtt{a}, \mathtt{b_i}) : \mathtt{r}$
3. if $a \in \Sigma_I$, $O \vdash \mathtt{a} : \mathtt{D}$ and $O \vdash \mathtt{D} \sqsubseteq \mathtt{E_1} \sqcup \mathtt{E_2}$ then $O \vdash \mathtt{a} : \mathtt{E_i}$ or $O \vdash \mathtt{a} : \overline{\mathtt{E_i}}$ for $i = 1, 2$
4. if $O \vdash (\mathtt{a}, \mathtt{b}) : \mathtt{r}$ with $a, b \in N_I$ and $O \vdash \mathtt{r} = \mathtt{r_1}.\mathtt{r_2}$ then there is some $c \in \Sigma_I$ such that $O \vdash (\mathtt{a}, \mathtt{c}) : \mathtt{r_1}$ and $O \vdash (\mathtt{c}, \mathtt{b}) : \mathtt{r_2}$.
5. if $O \vdash (\mathtt{a}, \mathtt{b}) : \mathtt{r}$ with $a, b \in \Sigma_I$ and $O \vdash \mathtt{r} = \mathtt{r}_1^*$ then there is a set $\{c_1, \ldots, c_k\} \subseteq \Sigma_I$ such that $O \vdash (\mathtt{c_i}, \mathtt{c_{i+1}}) : \mathtt{r_1}$ for all $i \in \{1, \ldots, k-1\}$ and $a = c_1$ and $b = c_k$.

3.2 Minimal Representation of States

Demanding constructive ontologies to specify a system (TBox + RBox) and its actual state (ABox), the concepts an individual belongs to are basically defined bottom up. Given an individual a of a fixed concept D the definition above requires the specification of additional information about a. In many cases this “additional” information is now sufficient to deduce that a is (also) of concept D making the original specification that $a : D$ redundant (even worse, specifying explicitly $\mathtt{a} : \overline{\mathtt{D}}$ would render the ontology inconsistent). Analogously to Nebel [10] we distinguish between *primitive* and *defined* concepts and roles. In a first step we modify the given TBox by making equalities between concepts explicitly. Whenever the TBox comprises a set of axioms of the form $c \sqsubseteq \mathtt{D_i}, \mathtt{i} = 1, \ldots, \mathtt{n}$ we check whether also $O \vdash \mathtt{D_1} \sqcap \ldots \sqcap \mathtt{D_n} \sqsubseteq \mathtt{c}$ holds. If it holds then we replace the set of axioms by an equation $c = \mathtt{D_1} \sqcap \ldots \sqcap \mathtt{D_n}$ if that does not violate the acyclicity condition of the TBox+RBox. Obviously, this change of axioms does not change the models of the ontology.

Based on this normalized TBox the acyclic dependency relation $>_c$ on concepts is used to define that a concept $c \in \Sigma_c$ is *defined* iff there is a concept $d \in \Sigma_c$ such that $c >_c d$ holds and otherwise *primitive*. Analogously, we use the acyclic relation $>_r$ on roles to define *defined* and *primitive* roles.

Now, given an ontology O we will enforce non-redundancy by imposing that all ABox-assertions are only primitive concept and role assertions. Furthermore, we use the criteria from the last section to check that O is constructive. As an example, consider the ontology for `Lists` from Section 2. The only primitive concept is `EmptyList` and the primitive roles are `hd` and `tl`. The ABox containing only the primitive assertions `nil:EmptyList, (a,nil):tl` is non-redundant, but not constructive. Adding the primitive assertion `(a,e):hd` makes it constructive.

3.3 Updates

In our setting, updates of the ABox correspond to state transitions of the specified system. The updates are initiated either by the processes (cf. Section 2) running in the system or by the environment interacting with the system. Regardless of the source of an update we demand that updates will always preserve the constructiveness of the ontology. To this end we enforce that they are specified in a non-redundant form by ABox-assertions exclusively over primitive concepts and roles. An update is a pair (α, δ), where α is a consistent set of primitive ABox-assertions to be added and δ primitive ABox-assertions to be removed. For a given ontology $O = \langle T, R, A\rangle$ and consistent primitive update (α, δ) the new ABox is determined from the old ABox by

1. If $(\mathtt{i} : \mathtt{C}) \in \alpha$ and $(\mathtt{i} : \mathtt{D}) \in \mathtt{A}$ and C and D are disjoint concepts, then $(\mathtt{i} : \mathtt{D})$ is removed;
2. If $((\mathtt{a}, \mathtt{b}) : \mathtt{R}) \in \alpha$, $((\mathtt{a}, \mathtt{c}) : \mathtt{R}) \in \mathtt{A}$ and R is functional, then $(\mathtt{a}, \mathtt{c}) : \mathtt{R}$ is removed.

Finally, all assertions from δ are removed. Formally, the update is defined by

$$\begin{aligned} A' := \alpha \cup (A \setminus (\delta \cup & \{(i : D) \in A | (i : C) \in \alpha, C \text{ and } D \text{ are disjoint}\} \\ & \cup\{((a, c) : R) \in A | ((a, b) : R) \in \alpha, R \text{ is functional}\}) \end{aligned}$$

The resulting ontology $\langle T, R, A'\rangle$ may well be inconsistent, for instance if number restrictions are violated. If so, the update is refused and we stick to the previous ontology O. If an action triggered the update, the action fails in the process semantics. If the environment triggered the update, respective repair processes must have been specified to synchronize the SHIP-Tool and the environment. If the ontology is consistent, it is not necessarily constructive. To this end we use the procedure from the last section to check if the resulting ontology $\langle T, R, A'\rangle$ is constructive. If not, the SHIP-Tool can provide detailed information which information is missing. This can be used to statically analyse the effects of actions of the defined processes whether they only contain primitive ABox-assertions and if they are complete enough to preserve constructiveness of the ontology.

3.4 Ramification: Indirect Effect Rules

Sometimes updates do also have indirect or implicit consequences that are not necessarily known to the process that performs the update. In such a situation, it might be necessary to integrate the indirect consequences immediately to the ontology in order to prevent some monitors to fail, as sending another update would introduce a new point in time on our time axis. The problem of how to describe the indirect effects of an action – or more generally, an arbitrary update – in a concise way is known as the *ramification problem.* In SHIP we use causal relationships as introduced in [3] called *indirect effect rules* to describe indirect effects that are caused by *arbitrary* updates. An example of an indirect effect rule is shown in Fig. 2. It works as follows: Given an ontology update from O to O', the `init`-assertions are checked on O'. If additionally `cond` is satisfied by O, then the indirect effects in `causes` are an additional update on O'.

4 Implementation & Applications

The SHIP-Tool is implemented in SCALA[1] and uses the Pellet Reasoner [12] as a Description Logic reasoner. The SHIP-Tool is free software and available from

[1] www.scala-lang.org

```
indirect effect CarriedPersonMovesAsWell = {
 init = (wc,p):at
 causes = (x,p):at
 cond = (wc,x):whc_carries, x:Person,  wc:WheelChair}
```

Fig. 2. Indirect Effects

`http://www.dfki.de/cps/projects/ship/shiptool/shiptool.en.html`. The SHIP-Tool can be used both in direct interaction with the environment as well as for simulation and testing.

So far, we have used it mainly in three applications: First, for the orchestration of services in an instrumented living environment, the Bremen Ambient Assisted Living Lab (BAALL)[2]. In this application we have developed in the SHIP-Tool processes to manage transport requests from the inhabitants of the BAALL, to schedule the routes of the wheelchairs operating in the BAALL and to provide assistance by automatically opening and closing doors and switching on and off the lights along the routes of the wheelchairs. A second application is concerned with medical guideline conformant patient treatment, and is pursued in the course of the SIMPLE project.[3] Another application is the SmartTies system for the management of the documents arising during the development of safety-critical software.

In all these applications the possibility to include behavior monitoring via LTL over description logic axioms and dynamic description logic proceses provides a suitable mechanism to develop the processes. With respect to the updates, the first attempt to deal with updates was – roughly speaking – to simply include all effects of actions. If the new ontology is inconsistent, then we used axiom pinpointing to compute a minimal set of ABox-assertions already contained in the previous ABox which must be removed to restore consistency. That made updates computationally expensive and processes slow. With the approach presented in this paper, this could be resolved resulting in a sensible speedup. Moreover, the constraints imposed by the constructiveness criteria on the ontology and the updates resulted in cleaner ontologies as well as much more concise formulations of actions.

5 Conclusion

This paper presented the SHIP-Tool as a framework to manage the orchestration of heterogeneous programs. Each of these programs communicate with the SHIP-Tool via ABox-updates. The SHIP-Tool allows for the integration of black-box services by encoding their responses in terms of updates as well as of white-box programs defined in terms of Dynamic Description Logic and operating natively

[2] http://baall.informatik.uni-bremen.de/

[3] http://www.dfki.de/cps/projects/simple

on the concepts defined in the ontology specified in the SHIP-Tool. In the same way humans interact with the system by ABox-updates. Hence, ABox-updates become the central notion of computation and we introduced constructive ontologies to speed up the computation of such updates. The resulting restrictions on how ontologies in general and ABoxes in particular have to be specified also enforce a proper structuring of the provided concepts and roles. Monitors allow one to control the overall behavior of the orchestrated system and can initiate appropriate measures depending on the success or failure of the monitor. The semantics of the SHIP-Tool is based on combining concepts of dynamic logic, description logic and temporal logic. Embedding these concepts in a combined logic, which is actually under development, we will also be able to verify properties of the systems specified and executed inside the SHIP-Tool.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.
2. F. Baader, S. Ghilardi, and C. Lutz. LTL over description logic axioms. *ACM Transactions on Computational Logic*, 2012.
3. F. Baader, M. Lippmann, and H. Liu. Using causal relationships to deal with the ramification problem in action formalisms based on description logics. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2010.
4. F. Baader and W. Nutt. Basic description logics. In Baader et al. [1], pages 43–95.
5. A. K. Bauer and Y. Falcone. Decentralised ltl monitoring. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.
6. L. Chang, F. Lin, and Z. Shi. A Dynamic Description Logic for Representation and Reasoning About Actions. In Z. Zhang and J. Siekmann, editors, *Procedings of KSEM 2007*, volume 4798 of *LNAI*, pages 115–127. Springer, 2007.
7. L. Chang, Z. Shi, T. Gu, and L. Zhao. A family of dynamic description logics for representing and reasoning about actions. *J. Autom. Reasoning*, 49(1):1–52, 2012.
8. H. Liu, C. Lutz, M. Milicic, and F. Wolter. Updating description logic aboxes. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 46–56. AAAI Press, 2006.
9. D. Nardi and R. J. Brachman. An introduction to description logics. In Baader et al. [1], pages 1–40.
10. B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.
11. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
12. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.