SHIP-Tool Handbook

Serge Autexier

German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
serge.autexier@dfki.de

Abstract. The SHIP-Tool provides an implementation, execution and simulation platform for workflow processes integrating and monitoring existing devices and services to assist the user. (i) The real world is represented in a description logic and kept consistent with the actual world by propagating updates in both directions; (ii) Expected system behavior of external processes or users is described as monitors using a linear temporal logic over world representations; (iii) Processes can be implemented that interact with the environment via updates and that can react on the success or failure of behaviors observed by monitors. This note provides a brief overview over the different specification and implementation languages of the SHIP-Tool and its user interface.

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 workflow 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 a first 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 workflow 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. 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. Processes 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 workflows 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.

Overall, the system allows the integration of existing components, the modeling and monitoring of complex workflows, as well as the verification of designed systems. It is part of a general SHIP IDE that supports the specification of such workflow processes in a formal environment, as well as a mechanism to verify properties about the overall system.

SHIP-Tool Formalisms. An expressive but still decidable logic formalism to represent environments and for which the problem of applying updates has been studied is Description Logic [1], which is used in the SHIP-Tool and introduced in Section 2. The monitors shall observe a specific behavior over the sequence of ontology updates. A technique known from run-time verification is LTL monitoring, where the run-time behavior of existing black-box system is monitored using an LTL description over the trace of its outputs [3]. For the SHIP-Tool we use a first-order temporal logic over ABox-assertions for monitoring which is introduced in Section 3.

Using Description Logic to represent the environment results in processes being dynamic description logic (DDL) programs [4] over environment representations in description logic ontologies. As a difference to existing DDL formalisms, we consider concurrent processes over the same environment representation with an interleaving semantics. 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. All this will be presented in detail in Section 4. The SHIP-Tool and its user interface providing the environment to run the processes and monitors is then presented in Section 5.

2 The SHIP-DL Ontology Language

The SHIP-Tool provides its own description language SHIP-DL to represent the knowledge of an application domain in a concise way. The language supports modularization and renaming.

Like other description languages, a domain specified in SHIP-DL consists of three parts: *concepts*, which represents a class of objects sharing some common characteristics, and *roles*, representing a binary relationship between two objects, and *individuals*, which are concrete objects in the domain and can belong to several concepts. For a general and extensive introduction to description logics we refer to [5, 2].

```
%TBOX
Door = Closed <+> Open
Time = Day <+> Night
Situation = Normal <+> Emergency

%RBOX
situation:Normal
time:Time
time:Day
bathroomdoor:Open
lowerLeftDoor:Closed
```

Listing 1.1. House ontology describing a small house

It is common to distinguish the following three parts of a knowledge base: (i) a *TBox* (terminological box), which describes how concepts are related to each other, (ii) a *RBox* (role box), which contains information about relations that can hold between individuals of the domain, and (iii) an *ABox* (assertion box), which describes a concrete world by defining individuals together with their relations. For each part, SHIP-DL provides language constructs, which will be explained in the sequel.

To get a feeling of how a knowledge base specification looks like in SHIP-DL, consider the knowledge base shown in Listing 1.1, which describes very basic objects of a house, namely lights and doors. The purpose of this ontology is to provide sufficient information to define a simple room controller for a house that closes the doors of the house at the night, unless there is an emergency that was communicated by a safety device, e.g., a fire alarm. Therefore, we can abstract away most details of the house and define just two basic concepts, namely a concept Door which contains all doors of the house. To be able to distinguish between open and closed doors, we introduce two subclasses Open and Closed, filtering those objects that are closed respectively open. To be able to distinguish the time of the day we introduce the concept Day or Night. The assertion `c = a <+>b` defines `c` to be the disjoint union of the concepts `a` and `b`, meaning that individuals cannot be instances of `a` and `b` at the same time.

In general, a SHIP-DL specification follows the grammar given in Fig. 1 and consists of three parts: (1) a header, where other SHIP-DL specifications can be referenced and be imported, (2) A TBox-part, where concepts and their relations are defined, and (3) a RBox-part, where relations between individuals and the concrete objects are defined, (4) an ABox part where concrete objects and their relations are defined.

Header: SHIP-DL provides a structuring mechanism to modularize the development of ontologies and to share parts between different developments. Referenced ontologies can be imported multiple times and their signature be renamed. Suppose for example a generic ontology for lists, as shown in Listing 1.2. It provides

⟨*ontdecl*⟩ ::= ⟨*import*⟩* %TBOX ⟨*tboxformula*⟩* %RBOX ⟨*rboxformula*⟩* ⟨*aboxformula*⟩*
⟨*import*⟩ ::= 'import' (⟨*name*⟩ | ⟨*owlURI*⟩) ⟨*renaming*⟩?
⟨*renaming*⟩ ::= 'with' 'concepts' ⟨*nmap*⟩ 'roles' ⟨*nmap*⟩ 'individuals' ⟨*nmap*⟩
⟨*nmap*⟩ ::= ⟨*name*⟩ 'as' ⟨*name*⟩ (',' ⟨*nmap*⟩)?
⟨*tboxformula*⟩ ::= ⟨*Disjoint*⟩ | ⟨*conceptADT*⟩
| ⟨*concept*⟩ ('<' | '=') ⟨*concept*⟩ (('|' | '<+>') ⟨*concept*⟩)*
⟨*Disjoint*⟩ ::= 'Disjoint' '(' ⟨*concept*⟩ (',' ⟨*concept*⟩)* ')'
⟨*conceptADT*⟩ ::= ⟨*tboxconstructor*⟩ ('|' ⟨*tboxconstructor*⟩)*
⟨*tboxconstructor*⟩ ::= ⟨*concept*⟩ '(' ⟨*constructorargs*⟩? ')'
⟨*constructorargs*⟩ ::= ⟨*role*⟩ ':' ⟨*concept*⟩ (',' ⟨*role*⟩ ':' ⟨*concept*⟩)*
⟨*rboxformula*⟩ ::= ⟨*roledecl*⟩ | ⟨*roleinclusion*⟩ | ⟨*roleeq*⟩ | ⟨*roleprop*⟩
⟨*roledecl*⟩ ::= ⟨*role*⟩ ':' ⟨*concept*⟩ ('*' | '->' | '<-' | '<->') ⟨*concept*⟩
⟨*roleinclusion*⟩ ::= ⟨*role*⟩ ('.' ⟨*role*⟩)* '<' ⟨*role*⟩
⟨*roleeq*⟩ ::= ⟨*role*⟩ '=' role ('.' ⟨*role*⟩)*
⟨*roleprop*⟩ ::= ('Sym' | 'Trans' | 'Ref' | 'Irref' | 'Func' | 'Asym' | 'Invfunc') '(' ⟨*role*⟩ ')'
⟨*concept*⟩ ::= ⟨*conceptand*⟩ ('+' concept)?
⟨*conceptand*⟩ ::= ⟨*conceptprim*⟩ ('&' ⟨*conceptand*⟩)?
⟨*role*⟩ ::= ⟨*roleand*⟩ ('+' role)?
⟨*roleand*⟩ ::= ⟨*roleprim*⟩ ('&' ⟨*roleand*⟩)?
⟨*roleprim*⟩ ::= 'inv' ⟨*role*⟩ | '{' '(' ⟨*string*⟩ ',' ⟨*string*⟩) '}' | '~' ⟨*role*⟩ | ⟨*string*⟩
⟨*conceptprim*⟩ ::= '~' ⟨*concept*⟩
| ('all' | 'ex') ⟨*role*⟩ '.' ⟨*concept*⟩
| '{' ⟨*string*⟩ '}' | '(' ⟨*concept*⟩ ')' | ⟨*constorvar*⟩
| ('>=' | '<=' | '=') ⟨*number*⟩ ⟨*role*⟩ '.' ⟨*concept*⟩
⟨*constorvar*⟩ ::= 'T' | ⟨*string*⟩
⟨*aboxformula*⟩ ::= ⟨*string*⟩ ':' ⟨*concept*⟩ | '('⟨*string*⟩ ',' ⟨*string*⟩ ')' ':' ⟨*role*⟩
| ⟨*string*⟩ ('!=' | '==') ⟨*string*⟩

Fig. 1: SHIP-Tool Grammar for Ontologies

a generic representation for lists by introducing an individual representing the empty list `nil`, as well as the well-known tail relation `tl` that is defined on non-empty lists. On demand, generic lists can be imported and renamed, as shown in Listing 1.3. Copying the content of Listing 1.2 to the ontology and performing the renaming, we obtain the ontology shown in Listing 1.4.

Name	Syntax	Semantics
Top	`T`	$\Delta^{\mathcal{I}}$
Bottom	`~T`	$\emptyset$
Intersection	`C & D`	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	`C + D`	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Disjoint Union	`C` $\langle+\rangle$ D	$(C \cup D) \backslash (C \cap D)$
Negation	`~C`	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Value Restriction	`all R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R \Rightarrow b \in C^{\mathcal{I}}\}$
Existential quant.	`ex 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`	$\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\mid = n\}$
	`>= n R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\mid \geq n\}$
	`<= n R.C`	$\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\mid \leq n\}$
Nominal concept	$\{i_1, \ldots, i_n\}$	$\{i_1^{\mathcal{I}}, \ldots, i_n^{\mathcal{I}}\}$

Tbl. 1. Description Logic concept constructors

TBox: The TBox defines the available concepts together with relations that hold between different concepts. SHIP-DL provides the standard DL concept constructors that are summarized in Table 1. Concept equality is expressed by $=$ and concept inclusion by $<$. For convenience, the constructor

$$C(r_1 : C_1, \ldots, r_n : C_n)$$

```
%TBOX
List ::= EmptyList <+> NonEmptyList(hd:Elem, tl:List)
Disjoint(List, Elem)
nil : EmptyList
```

Listing 1.2. Generic specifications of lists

that mimics algebraic datatypes is provided. It introduces a new concept C, together with new functional roles $r_i : C \times C_i$. For example, `NonEmptyList(hd:Elem,tl:List)` introduces the standard accessors `hd:List*Elem` and `tl:List*List`.

RBox: The RBox contains the definitions of relations that may hold between two individuals. Roles are declared together with a type that indicates domain and range of the property. Role composition is supported using the symbol '`.`'. Similar to concepts, subroles can be declared using the symbol **`<`**.

SHIP supports the standard role properties **`Sym`** for symmetric roles ($(x,y) \in R \Rightarrow (y,x) \in R$), **`Trans`** for transitive roles ($(x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$), **`Ref`** for reflexive roles ($(x,x) \in R$), **`Irref`** for irreflexive roles ($(x,x) \notin R$), **`Func`** for functional roles ($(x,y) \in R \wedge (x,z) \in R \Rightarrow y = z$), **`inv`** for inverse roles (computes R^{-1}), and **`Invfunc`** (stating that R^{-1} is functional).

ABox: The ABox part consists of a list of concept assertions `a:C`, 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`.

Given an ontology in SHIP-DL, we automatically introduce the unique name assumption and close the world by setting $\Delta^{\mathcal{I}}$ to the union of all known individuals. In particular, this means that at least one individual must be defined to keep the resulting ontology consistent.

3 SHIP-TL Language

SHIP uses linear temporal logic (LTL) [6] to specify temporal properties about ontology updates that result from processes that interact with the world $\mathcal{V}$. In practice, these properties are usually used to express an expected system behavior of an external process, 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 an internal process, an external process or even a human user.

```
import Lists with
 concepts Elem as Requirement, List as Requirements, EmptyList
     as EmptyRequirements, NonEmptyList as NonEmptyRequirements
 roles hd as requirements_next, tl as requirements_rest
 individuals nil as emptyrequirements
```

Listing 1.3. Import and renaming of generic lists

⟨*monitor*⟩ ::= 'monitor' ⟨*string*⟩ ⟨*params*⟩ '=' ('"'⟨*formatstring*⟩'"')? ⟨*foltl*⟩

⟨*formatstring*⟩ ::= Any string without ¨. Formal Parameters to be substituted should be surrounded by %, e.g. %p% for the formal parameter p.

⟨*foltl*⟩ ::= 'true' | ⟨*aboxass*⟩ | ⟨*string*⟩ ⟨*params*⟩
| ⟨*foltl*⟩ ('U' | 'and' | 'or' | '=>') ⟨*foltl*⟩
| ('all' | 'ex') ⟨*vardecls*⟩ '.' ⟨*foltl*⟩ | '(' ⟨*foltl*⟩ ')'
| ('X' | 'F' | 'G') ⟨*foltl*⟩

⟨*aboxass*⟩ ::= 'not'? ⟨*aboxformula*⟩

⟨*vardecl*⟩ ::= ⟨*string*⟩ (',' ⟨*string*⟩)* ':' ⟨*concept*⟩

⟨*vardecls*⟩ ::= ⟨*vardecl*⟩ (',' ⟨*vardecl*⟩)*

Fig. 2: SHIP-TL language to describe/control ontology updates

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 (ii) the property evaluates to false independently of any continuation – in this case, the monitor terminates with failure; (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.

Fig. 2 shows the grammar for monitors, which are essentially first-order LTL formulas that can access the ontology $\mathcal{V}$. The temporal operators are `X` indicating the next $\mathcal{V}$, `F` indicating some point in the future, `G` stating that a property must globally hold, and `U` stating that a property must hold until another property becomes true. Consider for example Listing 1.5 that corresponds to the property that all doors that are registered in $\mathcal{V}$ must be closed unless an emergency situation occurs or the time is Day.

4 SHIP Processes

SHIP provides a language to define complex processes that interact with $\mathcal{V}$. Starting from atomic processes called *actions*, complex system behaviors can be described by combining other processes as well as by interacting with monitors.

```
%TBOX
Requirements ::= EmptyRequirements
  <+> NonEmptyRequirements(requirements_next:Requirement,
                           requirements_rest:Requirements)
Disjoint(Requirements, Requirement)
emptyrequirements : EmptyRequirements
```

Listing 1.4. Renamed generic lists

```
monitor MonitorDoors(sit,time) =
  all d:Door . (d:Closed U (sit:Emergency or time:Day))
```

Listing 1.5. Temporal specification

Actions $\alpha(\boldsymbol{p}) = \langle \mathsf{pre}, \mathsf{eff} \rangle$ represent parametrized atomic interactions with the environment and consist of a finite set of A-Box assertions pre, the preconditions, and a set of possibly annotated A-Box assertions eff, the effects. Given a virtual world $\mathcal{V}$, an action is executable if all its preconditions are satisfied in $\mathcal{V}$, i.e., $\mathcal{V} \models \mathsf{pre}$. In this case, $\mathcal{V}$ is modified by applying the effects of the action to $\mathcal{V}$. If the preconditions can be refuted in $\mathcal{V}$, i.e., $\mathcal{V} \models \neg\mathsf{pre}$, the action fails, otherwise the action stutters, i.e., waits until one of the above eventualities occurs. The atomic action which does not modify the environment and always succeeds without stuttering is abbreviated as **`skip`**.

Using free variables in the effects, new individuals can be added to $\mathcal{V}$, where a fresh name is created at runtime. By annotating a variable with the keyword `delete`, individuals can also be removed from the ontology. This is important to keep $\mathcal{V}$ small and to allow for efficient reasoning. Listing 1.6 shows an action `CloseDoor` that takes a single individual d as parameter. The precondition of the `CloseDoor` checks whether d is an individual belonging to the class Open, which also means that d must be a door due to the subclass relation $\mathsf{Open} \sqsubseteq$ Door. If so, the effects of `CloseDoor` are added to $\mathcal{V}$.

```
action CloseDoor(d) = {
  pre = d:Open
  effect = d:Closed
}
```

Listing 1.6. A simple atomic action

Furthermore, actions can have conditional effects expressed by `if` statements. For instance, consider the following refined `CloseDoor`-action, which if the door to close is an emergency-exit door sets the door alarm to `off`:

```
action CloseDoor(d) = {
  pre = d:Open
  effect = d:Closed
  if (d:EmergencyDoor and da:DoorAlarmOn and (d,da):DoorAlarm)
     da:DoorAlarmOff
}
```

Processes: Starting from atomic actions, more complex processes can be defined using the combinators shown in Table 2. As usual, an interpretation consists of a non-empty set $\Delta^{\mathcal{I}}$, called the domain of the interpretation, and an interpretation function, which assigns to every concept A a set $A^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$ and to every atomic role a binary relation $R^{\mathcal{I}} \subset \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

The complete operational semantics for processes is given in Appendix A.

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`	branches according to the specified cases
	`case` c_1 `=>` p_1	which are checked in order.
	...	
	`case` c_k `=>` p_k	_ 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 interaction	`init m`	starts the monitor, continues when the monitor succeeds or fails when the monitor fails
alternative	`p +> q`	starts executing p, when p fails, q is executed, but the modifications of p are kept
parallel	`p | q`	executes p and q in parallel, terminates when both p and q terminate

Tbl. 2. Process combinators

```
process DoorControl(sit,time) = {
  if (sit:Emergency) skip
  else { if (time:Night)
   (
   (init MonitorDoors(sit,time))
   +>
   { if (sit:Emergency) skip
     else { CloseAllDoors; DoorControl(sit,time) }}
   )
  else skip } }
```

Listing 1.7. An example door control process

```
indirect effect R = {
 init = (WC,P):At
 causes = (X,P):At
 cond = (WC,X):Carries, X:Person, WC:WheelChair
}
```

Listing 1.8. Indirect effects

Indirect Effects: Sometimes, actions or updates that are provided by an external process 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 general, an arbitrary update – in a concise way is known as the *ramification problem*. SHIP provides indirect effect rules to describe indirect effects that are caused by *arbitrary* updates.

An example of an indirect effect rule is shown in Listing 1.8. It works as follows: Given an ontology update that transforms $\mathcal{V}$ to $\mathcal{V}'$, init is evaluated on $\mathcal{V}'$. If cond is satisfied by $\mathcal{V}$, then the indirect effects described by causes are added to the ontology.

5 SHIP-Tool User Interface

The SHIP-Tool is implemented in Scala and includes the Pellet Reasoner [7] as a Description Logic reasoner. The SHIP-Tool is distributed as a single Jar file `SHIPTool.jar`, which can be started using Java 1.6 or later in the usual manner

```
java -jar SHIPTool.jar
```

This starts the GUI which screenshot is shown on the right hand side of Fig. 4. The GUI is divided in a menu bar at the top-level, the status frame of

⟨*shipdecl*⟩ ::= (⟨*action*⟩ | ⟨*process*⟩ | ⟨*monitor*⟩ | ⟨*indeffect*⟩)*
⟨*action*⟩ ::= 'action' ⟨*string*⟩ ⟨*params*⟩ '=' '{' ⟨*pre*⟩ ⟨*effect*⟩ ⟨*condeffect*⟩* '}'
⟨*params*⟩ ::= ('(' ⟨*string*⟩ (',' ⟨*string*⟩)* ')')?
⟨*pre*⟩ ::= 'pre' '=' ⟨*aboxasserts*⟩?
⟨*aboxasserts*⟩ ::= ⟨*aboxass*⟩ (',' ⟨*aboxass*⟩)*
⟨*effect*⟩ ::= 'effect' '=' ⟨*aboxdeletes*⟩?
⟨*condeffect*⟩ ::= 'if' '(' ⟨*aboxasserts*⟩ ')' ⟨*aboxdeletes*⟩
⟨*aboxdeletes*⟩ ::= ⟨*aboxordelete*⟩ (',' ⟨*aboxordelete*⟩)*
⟨*aboxordelete*⟩ ::= ⟨*aboxass*⟩ | 'delete' '(' ⟨*string*⟩ (',' ⟨*string*⟩)* ')'
⟨*process*⟩ ::= 'process' ⟨*string*⟩ ⟨*params*⟩ '=' '{' ⟨*proc*⟩ '}'
⟨*proc*⟩ ::= ⟨*proc*⟩ (';' | '+>' | '|') ⟨*proc*⟩
| ⟨*proc*⟩ '*' | '{' ⟨*proc*⟩ '}' | '(' ⟨*proc*⟩ ')'
| 'while' '('⟨*aboxassertsand*⟩')' ⟨*proc*⟩
| 'if' '('⟨*aboxassertsand*⟩')' 'then'? ⟨*proc*⟩ 'else' ⟨*proc*⟩
| 'switch' ('case' (⟨*aboxassertsand*⟩ | '_') '=>' ⟨*proc*⟩)+
| 'forall' ⟨*aboxassertsand*⟩ '=>' ⟨*proc*⟩
| 'init' ⟨*foltl*⟩ | ⟨*string*⟩ ⟨*params*⟩ | 'skip'
⟨*aboxassertsand*⟩ ::= aboxass ('and' ⟨*aboxass*⟩)*
⟨*indeffect*⟩ ::= 'indirect' 'effect' ⟨*string*⟩ '=' '{' ⟨*init*⟩ ⟨*causes*⟩ ⟨*cond*⟩ '}'
⟨*init*⟩ ::= 'init' '=' ⟨*aboxasserts*⟩
⟨*causes*⟩ ::= 'causes' '=' ⟨*aboxdeletes*⟩?
⟨*cond*⟩ ::= 'cond' '=' ⟨*aboxasserts*⟩?

Fig. 3: SHIP-Tool Grammar of Actions, Processes, Monitors and Indirect Effects rules

the currently running processes on the upper left side, a status frame of the currently active monitors on the upper right side, a large frame for logging and pretty printing below the status windows, and finally an input frame for command line input at the bottom.

Process Status Frame. The process status frame show the list of active processes and their current state. The processes are in the left column and displayed in their original form. The status of the processes are in the right column showing: (i) `Done` if the process has terminated successfully, (ii) `Fail` if the process terminated in a failure state, (iii) `Error` if the process terminated with an error such as invocation of an undeclared action or function, and (iv) `Stutter` if the process *stutters*, e.g. an action is waiting to be applicable, a `switch` waiting that one of its case clauses is applicable, or a monitor which is not yet terminated.

Monitor Status Frame. This frame displays the currently active monitors. The left column shows a description of the monitor which is either the initial LTL formula or, if available, the textual description of the monitor given upon monitor invocation or at monitor declaration, where formal parameters indicated by surrounding %A% have been replaced by their actual parameter. The status of the monitor is the monitor progress, which is the formula computed by formula

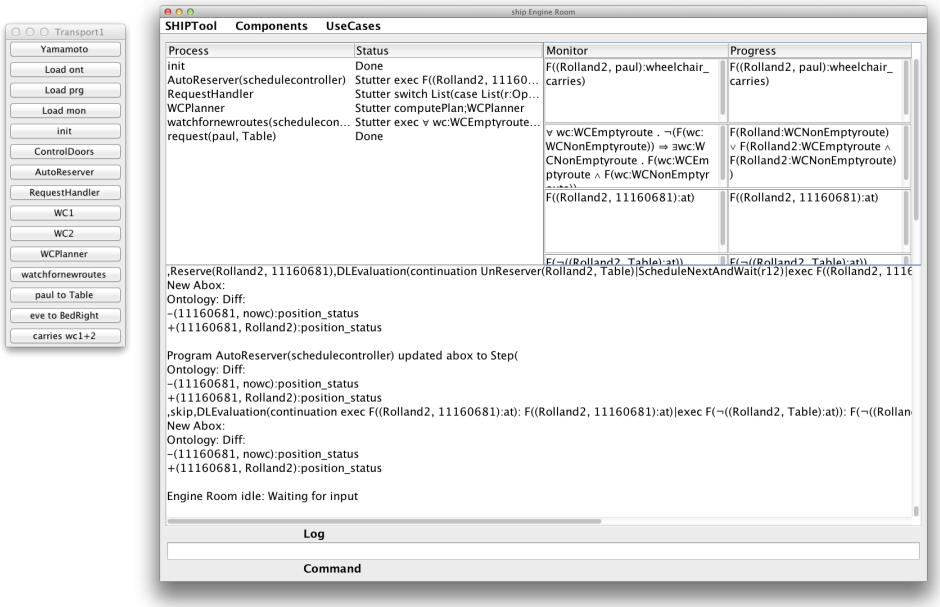

Fig. 4: SHIP-Tool Graphical User Interface

progress so far. If that formula is `True`, then the monitor has successfully observed the behaviour it was supposed to observe; in that case the background of the formula is green. If the formula is `False` then the monitor failed indicating that the sequence of ontology updates violated the expected behavior; in that case the background of the window is red and diagnostic information is displayed, like the last version of the formula which was not `False`. Otherwise the monitor is still running and the background is yellow. Each terminated monitor is removed from that window after the next update.

Command Line Input. The command line input frame controls the SHIP-Tool. It allows the user to load ontologies and declarations of actions, processes, indirect effect rules and monitors from files. The file names are relative to the current working directory. Actions, processes, indirect effect rules and monitors can also be directly be specified in the command line. Furthermore, it allows one to execute processes and to initialize monitors. Descriptions of the current ontology or declared actions, processes and monitors can be pretty printed in log frame and the current ontology can be saved in OWL 2 in a file. Administratives command are to exit the SHIP-Tool, to reset of the ontology, the processes or monitors.

The current ontology can be queried by a list of ABox assertions containing pattern variables: the query formulas are analysed for their individuals and each

individual name not contained in the current ontology is treated as a pattern variable.

The main interaction between the user and the running processes and monitors is to update the ontology manually. The syntax for updates is the same as in the effect-part of actions. All names of individuals are treated as constant, which allows to introduce new individuals on the fly.

The list of available commands and their syntax is given in Tables 5 (p. 14) and 6 (p. 15).

Menu Bar. The menu bar consist of three menus: the first allows to reset or exit the SHIP-Tool, which corresponds to the commands `reset` and `stop`. The components menu provides the `startclient` and `stopclient` entries for the current components. The use-case menu is generated dynamically from *Demonstration files* which allow to collect standard commands to run demos. The files must be in the `examples` directory of the current working directory and have the suffix `.dem`. The syntax of demonstration files is a list of

```
<Action Name> '->' <cmd>(',' <cmd>)*
```

The commands are a comma-separated list of commands as used in the command line input and must be without newlines. The name is used as button description in a window automatically generated from each demonstration file. An example demonstration window is shown on the left hand side of Fig. 4. The buttons can be pressed in any ordered and more than once. Pressing a button executes the corresponding sequence of commands from the demonstration file. The name of the window is generated from the name of the demonstration file. The demonstration file for the example demonstration windows in Fig. 4 is:

```
Yamamoto          -> startclient Yamamoto
Load ont          -> load examples/transport6.ont
Load prg          -> load examples/transport5.prg
Load mon          -> load examples/transport5.mon
init              -> run init
ControlDoors      -> run controldoors
AutoReserver      -> run AutoReserver(schedulecontroller)
RequestHandler    -> run RequestHandler
WC1               -> run WheelChairProcess(Rolland)
WC2               -> run WheelChairProcess(Rolland2)
WCPlanner         -> run WCPlanner
watchfornewroutes -> run watchfornewroutes(schedulecontroller)
paul to Table     -> run request(Paul, Table)
eve to BedRight   -> run request(Eve, BedRight)
carries wc1+2     -> update (Rolland, Eve):wheelchair_carries, \
                     (Paul, BedRight):at, (Rolland2,Paul):wheelchair_carries
```

Listing 1.9. Transport1.dem

References

1. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

Command	Description
'help'	displays help information for commands and input syntax for actions, processes and monitors
'init' ('"'<string>'"')? <foltl>	Starts the monitor formula <foltl>. If "<string>" is given, this is used as the description of the monitor; otherwise <foltl> is used as the description. All free individuals are interpreted as known individuals, not as variables.
'init' <string> <params>	Starts the designated declared monitor on the the designated individuals
'load' <filename>	loads the indicates file. Files with suffix '.ont' are treated as ontology files, with suffix '.prg' as action and processes description files and with suffix '.mon' as monitor declaration files. Loading an ontologies replaces the current ontology, while action, processes and monitors are added, possibly overwriting previous versions.
'ontology'	pretty prints the current ontology in the log window in SHIP DL syntax
'query' <dlquery>	Queries the current ontology; e.g. query F:Door returns all instances of concept Door; query (A,Kitchen):At
'reset' ('abox'|'proc'|'mon')*	deletes the current ontology, all processes or all monitors respectively or all if none is given.
'run' <proc>	executes the process <proc>. All free individuals are interpreted as known individuals, not as variables.
'saveont' <filename>	saves the current ontology as an OWL ontology in the designated file.
'show' <string>	pretty prints the declared action, process or monitor of the designated name
'startclient' <string>	starts the client interface <string>; current clients are Yamamoto and WCPlanner.
'stopclient' <string>	stops the client interface <string>

Fig. 5: SHIP-Tool commands

Command	Description
`'stop'`	stop the SHIP-Tool
`'trace'`	prints the history of actions and updates
`'update' <aboxdeletes>`	updates the current ontology by `<aboxdeletes>`. All free individuals are interpreted as known individuals, not as variables.
`<monitor>`	declares a monitor
`<action>`	declares an actions
`<process>`	declares a process
`<indeffect>`	declares an indirect effect

Fig. 6: SHIP-Tool commands (continued)

2. Franz Baader and Werner Nutt. Basic description logics. In Baader et al. [1], pages 43–95.
3. Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.
4. Liang Chang, Zhongzhi Shi, Tianlong Gu, and Lingzhong Zhao. A family of dynamic description logics for representing and reasoning about actions. *J. Autom. Reasoning*, 49(1):1–52, 2012.
5. Daniele Nardi and Ronald J. Brachman. An introduction to description logics. In Baader et al. [1], pages 1–40.
6. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
7. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

A Operational Semantics

Actions. Actions can succeed $\top$, fail $\bot$ or stutter. Their general form is $(pre, effect)$, where pre are the preconditions of the action and $effect$ are all effects of th eaction, including the conditional effects.

$$\frac{o \models_\sigma pre \qquad update(o, \sigma(effect)) \to o'}{(o, (pre, effect) \to (o', \top)} \qquad \frac{o \cup pre \text{ unsatisfiable}}{(o, (pre, effect)) \to (o, \bot)}$$

$$\frac{o \not\models pre \qquad o \cup pre \text{ not unsatisfiable}}{(o, (pre, effect)) \to (o, (pre, effect))}$$

Named Processes. Process declarations are of the form $f(x_1,\ldots,x_n) := p$ where p is a process body.

$$\frac{f(x_1,\ldots,x_n) := p \qquad \sigma := [a_1/x_1,\ldots,a_n/x_n] \qquad (o,\sigma(p)) \to (o',p')}{(o,f(a_1,\ldots,a_n)) \to (o',p')}$$

Sequential Composition.

$$\frac{(o,p_1) \to (o',\top)}{(o,p1;p2) \to (o',p2)} \qquad \frac{(o,p_1) \to (o,\bot)}{(o,p1;p2) \to (o,\bot)} \qquad \frac{(o,p_1) \to (o',p_1') \qquad p_1' \neq \top,\bot}{(o,p1;p2) \to (o',p_1';p2)}$$

Parallel Composition.

$$\frac{(o,p_1) \to (o',\top)}{(o,p_1|p_2) \to (o',p_2)} \qquad \frac{(o,p_1) \to (o,\bot)}{(o,p_1|p_2) \to (o,\bot)} \qquad \frac{(o,p_1) \to (o',p_1') \qquad p_1' \neq \top,\bot}{(o,p_1|p_2) \to (o',p_1'|p_2)}$$

(analogously for p_2)

Alternatives

$$\frac{(o,p_1) \to (o',\top)}{(o,p_1 \mathtt{+>} p_2) \to (o',\top)} \qquad \frac{(o,p_1) \to (o,\bot)}{(o,p_1 \mathtt{+>} p_2) \to (o,p_2)} \qquad \frac{(o,p_1) \to (o',p_1') \qquad p_1' \neq \top,\bot}{(o,p_1 \mathtt{+>} p_2) \to (o',p_1' \mathtt{+>} p_2)}$$

Simple Condition.

$$\frac{o \models \varphi}{(o,\mathtt{if}\,\varphi\,\mathtt{then}\,p_1\,\mathtt{else}\,p_2) \to (o,p_1)} \qquad \frac{o \cup \varphi \text{ unsatisfiable}}{(o,\mathtt{if}\,\varphi\,\mathtt{then}\,p_1\,\mathtt{else}\,p_2) \to (o,p_2)}$$

$$\frac{o \not\models \varphi \qquad o \cup \varphi \text{ not unsatisfiable}}{(o,\mathtt{if}\,\varphi\,\mathtt{then}\,p_1\,\mathtt{else}\,p_2) \to (o,\mathtt{if}\,\varphi\,\mathtt{then}\,p_1\,\mathtt{else}\,p_2)}$$

Complex Condition.

$$\frac{\exists 1 \leq i \leq n.c_i = \_ \vee o \models_\sigma c_i \qquad \forall 1 \leq j < i.c_j \neq \_ \wedge \nexists \sigma'.o \models_{\sigma'} c_j}{(o,\mathtt{switch}\,\mathtt{case}\,c_1 \mathtt{=>} p_1 \ldots \mathtt{case}\,c_n \mathtt{=>} p_n) \to (o,\sigma(p_i))}$$

$$\frac{\forall 1 \leq i \leq n.c_i \neq \_ \wedge \nexists \sigma'.o \models_{\sigma'} c_i}{(o,\mathtt{switch}\,\mathtt{case}\,c_1 \mathtt{=>} p_1 \ldots \mathtt{case}\,c_n \mathtt{=>} p_n) \to (o,\mathtt{switch}\,\mathtt{case}\,c_1 \mathtt{=>} p_1 \ldots \mathtt{case}\,c_n \mathtt{=>} p_n)}$$

Monitors.

$$\frac{-}{(o,\text{init } True) \to (o,\top)} \qquad \frac{-}{(o,\text{init } False) \to (o,\bot)} \qquad \frac{\varphi \neq True, False}{(o,\text{init } \varphi) \to (o,\text{init } \varphi)}$$

A.1 Formula Progression

After each update (either received from the environment or by application of an action), all *active* monitors are evaluated one step on the new ontology o.

$$\frac{(o,F) \to H}{(o,G(F)) \to H \wedge G(F)} \qquad \frac{(o, \bigwedge_{c \in Individuals(o,C)} F(c)) \to H}{(o, \forall x : C.F(x)) \to H}$$

$$\frac{(o,G) \to True}{(o, F\ U\ G) \to True} \qquad \frac{(o,G) \to G' \neq True \qquad (o,F) \to True}{(o, F\ U\ G) \to F\ U\ G}$$

$$\frac{(o,G) \to G' \neq True \qquad (o,F) \to F' \neq True}{(o, F\ U\ G) \to False}$$

$$\frac{o \models P, P \text{ literal}}{(o,P) \to True} \qquad \frac{o \not\models P, P \text{ literal}}{(o,P) \to False} \qquad \frac{-}{(o, X(F)) \to F}$$

$$\frac{(o,H) \to True}{(o,F(H)) \to (o,True)} \quad \frac{(o,H) \to H' \neq True}{(o,F(H)) \to (o,F(H))}$$

$$\frac{M(x_1,\ldots,x_n) := F \qquad \sigma := [a_1/x_1,\ldots,a_n/x_n] \qquad (o,\sigma(F)) \to (o,F')}{(o, M(a_1,\ldots,a_n)) \to (o,F')}$$

$$\frac{\begin{array}{c} M(x_1,\ldots,x_n) := F \qquad \sigma := [a_1/x_1,\ldots,a_n/x_n] \\ H := NNF(\sigma(\text{not } F)) \qquad (o,H) \to (o,H') \end{array}}{(o, \text{not } M(a_1,\ldots,a_n)) \to (o,F')}$$

In the last rule $NNF(F)$ denotes the negation normalform of F. The active monitors of a program p are determined as follows:

$$\begin{aligned}
&\text{ActiveMon}((pre, effect)) := \emptyset \\
&\text{ActiveMon}(f(a_1,\ldots,a_n)) := \emptyset \\
&\text{ActiveMon}(p_1; p_2) := \text{ActiveMon}(p_1) \\
&\text{ActiveMon}(p_1 \mid p_2) := \text{ActiveMon}(p_1) \cup \text{ActiveMon}(p_2) \\
&\text{ActiveMon}(p_1 \mathrel{+\!\!>} p_2) := \text{ActiveMon}(p_1) \\
&\text{ActiveMon}(\texttt{if}\ \varphi\ \texttt{then}\ p_1\ \texttt{else}\ p_2) := \emptyset \\
&\text{ActiveMon}(\texttt{switch case}\ c_1 \texttt{=>}\ p_1 \ldots \texttt{case}\ c_n \texttt{=>}\ p_n) := \emptyset \\
&\text{ActiveMon}(\texttt{init}\ F) := \{\texttt{init}\ F\}
\end{aligned}$$