CAptLang: A Language for Context-aware and Adaptable Business Processes

A. Bucchiarone, C. Antares Mezzina, and M. Pistore

Fondazione Bruno Kessler
via Sommarive 18, 38123 Trento, Italy
[bucchiarone,mezzina,pistore]@fbk.eu

ABSTRACT

Run-time adaptability is a key feature of dynamic business environments, where the processes need to be constantly refined and restructured to deal with context changes. In this paper, we present CAptLang, a language to model context-aware and adaptable business processes where the main feature is the possibility of leaving the handling of extraordinary or improbable situations to run time. We present CAptLang with its formal syntax and semantics. Moreover we show how its semantics have been used to guide the implementation of a Java-based business processes execution engine, component of the ASTRO-CAptEvo adaptation framework.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax,Semantics*

Keywords

Language, Business Processes, Adaptation, Context-awareness

1. INTRODUCTION

Modern business processes often operate in dynamic, open and non-deterministic environments. This means that the execution context is volatile, and the outcome of some activities is not completely controllable. In addition to this, the set of business policies is also changing dynamically. Dynamic context changes or undesirable outcome of some activities may often cause abnormal termination of the process and prevent the achievement of the business goals. The solution might be the run time modification (or adaptation) of the process instances so that they can properly react to such extraordinary situations.

One way to have a business process adaptable is to consider a priori all the cases in which the process can deviate from the normal behaviour (i.e., Exception Handling [6]). In this way, it is possible to completely characterize the reaction of the system at design time as the activities (or even sub-processes) implementing the recovery behaviour. This specification may be performed by extending standard languages (e.g., BPEL) with the adaptation-specific tools [8, 12], using a set of predefined adaptation rules [5, 11], using aspect-oriented approaches [4, 9], or modelling and managing business process variants [7]. However, in many situations, such approaches fail to completely solve the problem of adaptation. First, for complex processes the variety of variants that require specific recovery actions may be too large for the designers to consider, making the adaptation hard to define, implement, and revise. Still, unexpected situations and changes may occur at run time, requiring the adaptation to be performed even if the concrete case (and its handling) has not been anticipated at design time. To deal with such situations, the languages to manage adaptable business processes should be much more dynamic and flexible, so that the process can recover from critical deviations without defining them a priori. In order to address this problem, in this paper we present CAptLang, a language to model context-aware and adaptable business processes where the main feature is the possibility of leaving the handling of unusual or unexpected situations to runtime, instead of analyzing all cases at design time and embedding the corresponding adaptation activities in the business process model [2]. We show its own syntax and operational semantics. The latter have been used to guide the implementation of a java-based business processes execution engine, component of the ASTRO-CAptEvo adaptation framework [13].

The rest of the paper is structured as follows: Section 2 presents the motivating scenario used to present the potentiality of CAptLang; Section 3 is devoted to introduce the syntax and semantics of the language, while Section 4 demonstrates how the semantics has been used to guide the implementation of Java-based execution engine able to run business processes modelled using CAptLang. Section 5 concludes the paper suggesting some directions for future works.

2. MOTIVATING SCENARIO

The scenario is based on the operation of the sea port of Bremen, Germany [1], where nearly 2 million new vehicles are handled each year; the business goal is to deliver them from a manufacturer to a dealer. Each *car*, depending on its brand, model and specific order, needs to follow certain customizable procedures during the delivery process (as illustrated in Figure 1). Cars arrive by *ship* and are unloaded and unpacked at a certain terminal area. Each ship is able to approach and leave a *gate* interacting with the *landing manager*, which is in charge of coordinating and controlling the landing procedure for all the ships in the port. Once

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VaMoS'13, January 23-25, 2013, Pisa, Italy.
Copyright 2013 ACM 978-1-4503-1541-8/13/01 ...$15.00.

a car is unpacked to a terminal, a communication is established with the *storage manager* to book a place at storage facilities. The storage place is located in one of the available storage areas, each having its own parking procedure and suitable for a specific car type (e.g., covered/guarded areas for luxury cars). Once a stored car is ordered by a retailer, it continues its way towards the delivery. In particular, cars are treated at dedicated treatment areas (such as washing, painting, equipment, repairing) according to the details in the order. When a car is ready to be delivered, it is assigned to an available *delivery gate* and *truck*. The latter is responsible for loading a limited number of cars and deliver them to the dealer.

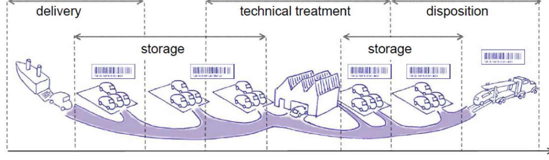

Figure 1: Process Chain of the Car Logistics Scenario [1].

The goal is to develop an application (the Car Logistics System, CLS) to support the management and operation of the port as described above, where numerous actors (i.e., cars, ships, gates, managers, trucks, treatment areas, etc.) need to cooperate in a synergic manner respecting their own procedures and business policies. The system needs to deal with the dynamicity of the scenario, both in terms of the variability of the actors' procedures (customizable processes), and of the exogenous context changes affecting its operation. *Customization* means that different brands and models of cars should be treated in a similar but customizable way. Moreover new car models, having specific requirements and procedures, must be easily integrated in the application. Similarly, the application needs to flexibly deal with changes in the procedures of external entities such as ships and trucks. Finally, the application needs to promptly implement changes in international regulations and laws.

The CLS operation is modelled through a set of *entities* (e.g., ships, cars, gates, managers, etc..), each specifying its behaviour through a *business process*. Unlike traditional business processes (e.g., BPEL), where they are static descriptions of the expected run-time operation, we define *dynamic business processes* that are refined at run time according to the operational environment of the system (i.e., *context model*). In addition to the classical workflow language constructs (e.g., input, output, data manipulation activities, complex control flow constructs), we add the possibility to relate the process execution to the system context by annotating activities with *preconditions*, *postconditions* and *compensations*.

Preconditions constrain the activity execution to specific context configurations, and are used to catch violations in the expected behaviour and trigger run time adaptation. *Postconditions* indicates states of the system context that should be reached when a process activity is executed and are used to automatically reason on the consequences of process adaptations. Moreover, we introduced the possibility of specifying *abstract activities* within processes. An abstract activity is defined at design time in terms of the *goal* it needs to achieve, expressed as context configurations to be reached, and is automatically refined at run time into an executable process considering the current context configuration, the goal to reach and using a precise adaptation mechanisms (see [3]).

3. CAptLang: FORMAL SYNTAX AND SEMANTICS

Types, identifiers, operations and variables. We assume the existence of the following denumerable infinite mutually-disjoint sets: the set $\mathcal{L}$ of model types, the set of instance identifiers $\mathcal{I}$, the set of operation $\mathcal{O}$ and the set of variables $\mathcal{V}$. We let (together with their decorated versions): h, l, k to range over $\mathcal{L}$; t, s to range over $\mathcal{I}$; o to range over $\mathcal{O}$ and x, y to range over $\mathcal{V}$. Moreover, we assume the existence of a set of assumptions g on (some properties of) the context, written in a suitable logic $\mathcal{G}$ (such as the one used in [10]) and by abusing of the notation we will also indicate $\mathcal{G}$ as the set of context assumption and we let g and its decorated versions to range over $\mathcal{G}$. For the sake of simplicity we let this logic $\mathcal{G}$ undefined, since we just want to focus on the semantics of the language. To this end, we assume the existence of a predicate $C \models g$ indicating the fact that a certain context C satisfies the assumption g (or that g *holds* in C), and $C \not\models g$ when C does not satisfy g. Expressions e are left unspecified but we require that contain, at least, variables and simple values, e.g. integers and strings, and primitive operators on these values.

Syntax. The syntax of CAptLang is depicted in Figure 2. *Basic activities* b can be an invocation $\mathtt{snd}_l\, o$ indicating an invocation of an operation o belonging to a model of type l, or an invocation receipt $\mathtt{rcv}\, o$. *Structured activities* a are made of basic activities or scoped activities $[g_1, b, g_2]$ indicating the fact that b should be executed only if precondition g_1 and postcondition g_2 are satisfied in the current context. A *business process* P can be the null process $\mathbf{0}$, a structured activity a, a compensable activity $a \div g$ indicating the fact that if the action a is executed then its compensation g should be saved and eventually used in case of *adaptation*, an abstract activity (g), a fail $\langle g \rangle$ indicating an adaptation need since the condition g is violated [1] , a value assignment $x := e$, a concatenation $P; Q$ of processes, a pick (or external choice) $\sum_{j \in J} \mathtt{rcv}\, o_j; P_j$ or a conditional (internal choice) $\mathtt{if}(e)\ \{P\}\ \{Q\}$. A *deployment* D, can be either a model instance (shortened as instance) $t_l\{\mu \vdash P\, ,\, Q\}_c$, an instantiating operation $\mathtt{start}\ l$, an invocation message $\langle t, o \rangle$ or a parallel composition $D_1 \parallel D_2$. An instance $t_l\{\mu \vdash P\, ,\, G\}_c$ represents the fact that the instance t of type l is executing the process P. μ is the local store of the process, used to evaluate all the expressions e of P. To this end we indicate $\emptyset$ as the empty store and $\circ$ the concatenating operator among stores recursively defined as follows:

$$(\mu \circ \mu')(x) = \begin{cases} \mu'(x) & \text{if } x \in \mathit{dom}(\mu') \\ \mu(x) & \text{otherwise} \end{cases}$$

where $\mathit{dom}(\mu)$ represents the *domain* of store μ. G is a list of compensations and c is the correlation set of the instance. We indicate with ϵ the empty list and :: the concatenation operator among lists. Hence when writing $g :: G$ we will point out that g is the head of the list and G the tail.

[1]Fails $\langle g \rangle$ are generated at runtime, hence it should be avoided their use at design time.

$$
\begin{array}{ll}
b ::= \mathtt{snd}_l\, o \mid \mathtt{rcv}\, o & \text{Basic Activities}\\
a ::= b \mid [g_1, b, g_2] & \text{Structured Activities}\\
P, Q ::= \mathbf{0} \mid a \mid a \div g \mid (g) \mid \langle g \rangle \mid x := e \mid P; Q \mid \sum_{j \in J} \mathtt{rcv}\, o_j; P_j \mid \mathtt{if}(e)\, \{P_1\}\, \{P_2\} & \text{Business Processes}\\
D, E ::= t_l\{\mu \vdash P\,,\, G\}_c \mid \mathtt{start}\ l \mid \langle t, o \rangle \mid D \parallel E & \text{Deployments}\\
M, N ::= l[\![P]\!] \mid M, N & \text{Models}\\
h, l, k \in \mathcal{L} \quad t, s \in \mathcal{I} \quad o \in \mathcal{O} \quad x, y \in \mathcal{V} \quad g, g_1, g_2 \in \mathcal{G} &
\end{array}
$$

Figure 2: Syntax of CAptLang

$$
\text{(P.OUT)}\ \mathtt{snd}_l\, o \xrightarrow{!\langle l, o \rangle} \mathbf{0} \qquad \text{(P.IN)}\ \mathtt{rcv}\, o \xrightarrow{?o} \mathbf{0} \qquad \text{(P.SC)}\ \frac{b \xrightarrow{\alpha} \mathbf{0}}{[g_1, b, g_2] \xrightarrow{\langle g_1, \alpha, g_2 \rangle} \mathbf{0}} \qquad \text{(P.CMP)}\ \frac{a \xrightarrow{\alpha} \mathbf{0}}{a \div g \xrightarrow{\alpha \div g} \mathbf{0}}
$$

$$
\text{(P.PRE)}\ \frac{\alpha \in \{(g), \langle g \rangle, x := e\}}{\alpha \xrightarrow{\alpha} \mathbf{0}} \qquad \text{(P.SEQ)}\ \frac{P \xrightarrow{\alpha} P'}{P; Q \xrightarrow{\alpha} P'; Q} \qquad \text{(P.PICK)}\ \sum_{j \in J} \mathtt{rcv}\, o_j; P_j \xrightarrow{?o_i} P_i \qquad \text{(P.NIL)}\ \mathbf{0}; P \xrightarrow{\tau} P
$$

$$
\text{(P.T)}\ \mathtt{if}(e)\, \{P\}\, \{Q\} \xrightarrow{e} P \qquad \text{(P.F)}\ \mathtt{if}(e)\, \{P\}\, \{Q\} \xrightarrow{\overline{e}} Q
$$

Figure 3: Process Execution

A correlation set is a set of mappings $\{l \hookrightarrow t\}$, from models to instances, indicating that, for a specific instance s, all the invocations of operations o of kind l should be addressed to the instance t. Correlations among instances are established at run-time. Since communications (operations invocation) among instances is asynchronous, a message of the form $\langle t, o \rangle$ models the invocation of the operation o to the instance t. A model $l[\![P]\!]$ represents the forge that will be used at runtime to create instances of type l. In other words models are static entities and instances represent their runtime instantiations.

We assume that all the model types are unique. We denote with $\mathcal{P}$ the set of processes and $\mathcal{D}$ the set of deployments. We let (together with their decorated version) P, Q to range over $\mathcal{P}$ and D, E to range over $\mathcal{D}$.

Operational Semantics. We now define the operational semantics of CAptLang which describes the behaviour of programs (written in the language) in a mathematically rigorous way. Operational semantics of CAptLang is expressed via a reduction relation $\mapsto$, which is a binary relation over deployments ($\mapsto \subset \mathcal{D} \times \mathcal{D}$), and a structural congruence relation $\equiv$, which is a binary relation over deployments ($\equiv \subset \mathcal{D} \times \mathcal{D}$). We define *deployment contexts* as "deployments with a hole $\bullet$", given by the following grammar:

$$\mathbf{D} ::= \bullet \mid D \parallel \mathbf{D} \mid \mathbf{D} \parallel D$$

A congruence on deployments is an equivalence relation $\mathcal{R}$ that is closed for deployment contexts:

$$D\, \mathcal{R}\, E \Longrightarrow \mathbf{D}[D]\, \mathcal{R}\, \mathbf{D}[E]$$

Structural congruence $\equiv$ is defined as the smallest congruence on deployments that satisfies following axioms:

$$D_1 \parallel D_2 \equiv D_2 \parallel D_1 \qquad (D_1 \parallel D_2) \parallel D_3 \equiv D_1 \parallel (D_2 \parallel D_3)$$

These rules deal with the commutativity and associativity of the parallel operator $\parallel$ on deployments.

We use a layered approach to devise the reduction relation of CAptLang. To this end we define three relation: the first one as a labelled transition system (LTS) on processes (that we call *process execution*), expressing all the possible behaviours of processes. We then constrain processes behaviour to a particular context, and this is expressed via another LTS called *context aware execution*. Finally the main relation (called *instance execution*) is expressed as a reduction relation and it is in charge of regulating the execution of instances.

The reduction relation $\mapsto$ is defined as the smallest binary relation on deployments satisfying the rules of Figure 5. Rules are given as judgements of the form $C, M \Vdash D \mapsto D'$ indicating that the reduction $D \mapsto D'$ is allowed under a certain context C and model M. This kind of judgements allows to abstract away from reductions details concerning the modelling or the evolution of the context C and the model M. Reduction $\mapsto$ exploits a LTS over pairs $\mathcal{P} \times G$, written $\xrightharpoonup{\alpha}$, which obeys to the rules in Figure 4. Relation $\rightharpoonup$ represents the execution of a process P under a context C. Hence rules are given in form of judgement of the form $C \Vdash P, G \xrightharpoonup{\alpha} P', G'$ indicating that the process P can do the action α under the context C. Relation $\rightharpoonup$ exploits a LTS over processes, written $\xrightarrow{\alpha}$, which obeys to the rules in Figure 3. All the rules in Figure 3 are quite straightforward as they just transform a prefix into a label. Said otherwise, relation $\rightarrow$ tell us what a process can *potentially* do.

Let us now comment of rules of Figure 4. For lack of space we just report rules dealing with compensations, other cases with no compensations uses similar rules with the difference that there is no need to add the compensation to G. A scope $[g_1, b, g_2]$ can be handled in three different cases: if both g_1 and g_2 hold (rule C.SCP) then the internal activity b is allowed to execute; if just g_1 holds (rule C.POST) then b is executed but the resulting process is annotated with the fact that postcondition g_2 is not satisfied. If precondition does not hold (rule C.PRE then b is not executed and the process is annotated with the fact that precondition g_1 does not hold. Rules C.ACT deals with (process) action different from a scope.

$$(\text{C.S}{\scriptstyle CP})\ \frac{P \xrightarrow{\langle g_1,\alpha,g_2\rangle \div g} P' \qquad C \models g_1 \qquad C \models g_2}{C \Vdash P, G \xrightarrow{\alpha} P', g :: G}$$

$$(\text{C.P}{\scriptstyle OST})\ \frac{P \xrightarrow{\langle g_1,\alpha,g_2\rangle \div g} P' \qquad C \models g_1 \qquad C \not\models g_2}{C \Vdash P, G \xrightarrow{\alpha} \langle g_2\rangle; P', g :: G}$$

$$(\text{C.P}{\scriptstyle RE})\ \frac{P \xrightarrow{\langle g_1,\alpha,g_2\rangle \div g} P' \qquad C \not\models g_1}{C \Vdash P, G \xrightarrow{\tau} \langle g_1\rangle; P, G}$$

$$(\text{C.A}{\scriptstyle CT})\ \frac{P \xrightarrow{\alpha \div g} P' \qquad \alpha \neq \langle _, _, _\rangle}{C \Vdash P \xrightarrow{\alpha} P', g :: G}$$

Figure 4: Context Aware Execution

$$(\text{I.S}{\scriptstyle ND})\ \frac{C \Vdash P, G \xrightarrow{!\langle l_1,o\rangle} P', G' \qquad \{l_1 \hookrightarrow s\} \in c}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_c \parallel \langle o, s\rangle}$$

$$(\text{I.N}{\scriptstyle EW})\ \frac{C \Vdash P, G \xrightarrow{!\langle l,o\rangle} P', G' \qquad \{l \hookrightarrow s\} \notin c \qquad l[\![Q]\!] \in M \qquad fresh(s) \qquad c' = c \cup \{l \hookrightarrow s\}}{C, M \Vdash t_h\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_{c'} \parallel \langle o, s\rangle \parallel s_l\{\emptyset \vdash Q\ ,\ \epsilon\}_{\{h \hookrightarrow t\}}}$$

$$(\text{I.R}{\scriptstyle CV})\ \frac{C \Vdash P, G \xrightarrow{?o} P', G'}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \parallel \langle o, t\rangle \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_c}$$

$$(\text{I.I}{\scriptstyle NT})\ \frac{C \Vdash P, G \xrightarrow{\tau} P', G'}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_c}$$

$$(\text{I.S}{\scriptstyle TORE})\ \frac{C \Vdash P, G \xrightarrow{x:=e} P', G'}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \circ [x := e] \vdash P'\ ,\ G'\}_c}$$

$$(\text{I.S}{\scriptstyle TART})\ \frac{l[\![Q]\!] \in M \qquad fresh(t)}{C, M \Vdash \texttt{start}\ l \mapsto t_l\{\emptyset \vdash Q\ ,\ \epsilon\}_\emptyset}$$

$$(\text{I.I}{\scriptstyle F}\text{-T})\ \frac{C \Vdash P, G \xrightarrow{e} P', G' \qquad \mu \models e}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_c}$$

$$(\text{I.I}{\scriptstyle F}\text{-F})\ \frac{C \Vdash P, G \xrightarrow{\overline{e}} P', G' \qquad \mu \not\models e}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash P'\ ,\ G'\}_c}$$

$$(\text{I.N}{\scriptstyle EED})\ \frac{C \Vdash P, G \xrightarrow{\alpha} P', G' \qquad \alpha \in \{(g), \langle g\rangle\} \qquad adapt(C, P', \alpha, G') = (Q, G_1)}{C, M \Vdash t_l\{\mu \vdash P\ ,\ G\}_c \mapsto t_l\{\mu \vdash Q\ ,\ G_1\}_c}$$

$$(\text{I.C}{\scriptstyle TX})\ \frac{C, M \Vdash D \mapsto D'}{C, M \Vdash \mathbf{D}[D] \mapsto \mathbf{D}[D']}$$

$$(\text{I.E}{\scriptstyle QV})\ \frac{D \equiv E \qquad C, M \Vdash E \mapsto E' \qquad E' \equiv D'}{C, M \Vdash D \mapsto D'}$$

Figure 5: Instance execution

Finally we can comment on rules of $\mapsto$ depicted in Figure 5. By looking at rule I.SND we can see that communication, or operations invocation, is asynchronous. When a process P of a certain instance l invokes an operation (using the label $!\langle l_1, o\rangle$) then it is checked whether the instance contains a correlation for the model type l_1. If it is the case, then a message of the form $\langle o, s\rangle$ is released. Let us note that according to the correlation $\{l_1 \hookrightarrow s\}$ the message is sent to the instance s. Hence we can say that the communicating partners of an instance are determined at *run-time*. If an instance invokes an operation of a certain type l for which it has no correlation, then a new instance of model type l has to be created (if it exists in the model M). This is done by rule I.NEW. Let us note that the correlation set of the invoking instance is enriched with the new correlation $\{l \hookrightarrow s\}$, and that to the new instance is given as correlation set the correlation $\{h \hookrightarrow s\}$ where s and h are respectively the identifier and type of the invoking instance. Moreover, a message $\langle o, s\rangle$ is created, with s the identifier of the new created instance of type l. Naturally the new instance is created with an empty $\emptyset$ local store and an empty ϵ compensations list. Let us note the use of the function $fresh(s)$ assures that the identifier of the new created instance is unique. Rule I.RCV just allows an instance to consume a message addressed to it. Rule I.INT deals with internal non-visible actions of instances. Rule I.STORE deals with expressions produced by an instance. A deployment of the form `start` l has the effect of creating a new instance of type l. This is done by rule I.START. Rules I.IF-T and I.IF-F deal with conditional processes. To this end we override the definition of $\models$ to work also on stores μ and expressions e. Rule I.NEED deals with adaptation need. When a process of an instance produces a need of the form $\langle g\rangle$, in case of a condition violated, or (g), in case of a refinement need, then the adaptation function *adapt* is asked to find a solution to the *adaptation problem*. The adaptation problem is a tuple of the form (C, P', α, G) indicating the context in which the need has to be handled, the process that raised the need, the need itself and eventually the list of compensation that the process saved. All these information are used by the adaptive function which returns the adapted process and the new list of compensations. Then the adapted process, along with new the compensation list, take the place of the old process and list that raised the need. The adaptation function implements several adaptation mechanisms and strategies as described in [3]. Rules I.CTX and I.EQV deal respectively with the closure of $\mapsto$ under deployment contexts and structural congruence.

4. PROCESS ENGINE IMPLEMENTATION

The goal of this section is to present the *Process Engine* component of the ASTRO-CAptEvo framework [13]. It has been implemented using the operational semantics defined in section 3 and provides a set of functionalities that we are going to explain in detail. As we can see in Figure 6, it is part of the *Execution Layer* and is in charge of (i) *exe-*

cuting the context-aware and adaptable business processes modelled using CAptLang, (ii) *detecting* execution problems and triggering adaptation by passing necessary information to the adaptation layer, and (iii) *adapting* the process instance according to the solution produced by the *adaptation layer*.

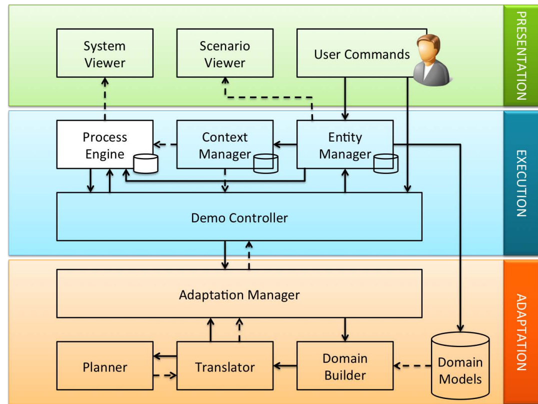

Figure 6: ASTRO-CAptEvo Architecture.

To execute process instances, the *Process Engine* respects the semantics rules presented in Figures 3, 4 and 5. At the same time it implements the semantics rules to detect situations that require process adaptation (see rule I.NEED of Figure 5 and rules C.PRE, C.POST of Figure 4). Once one of the previous situation is detected, then the adaptation problem (see rule I.NEED[2]) is passed to the *Adaptation Layer*. The*Process Engine* is also able to suspend process execution, executes adaptation processes received by the *Adaptation Layer* and resume instances after that an adaptation is applied. To see the entire framework in action, applied to the CLS scenario, one can download the complete demo or the video of a live demonstration at: `www.astroproject.org/captevo`.

5. CONCLUSION

CAptLang is a language to model context-aware and adaptable business processes. It is equipped with a well-defined operational semantics. The language has been introduced to model and run adaptable business processes taking into account context information, and its operational semantics has been used to guide the implementation of the *Process Engine* component of the ASTRO-CAptEvo framework. The CAptLang operational semantics has been defined to manage single and decentralized adaptation problems giving the responsibility at each entity to solve its associated adaptation goal and using its own adaptation techniques (i.e., adaptation layer). We plan to extend the semantics in order to coordinate also the various adaptation solutions, coming from the different entities, to prevent situations like conflicts, oscillations and race conditions and guaranteeing the reachability of the common goal. At the same time we want to extend the set of possible adaptation needs adding situations where a process need to be adapted due to its correlation with other processes in the system.

The execution of CAptLang business process instances results in a set of adapted process variants instantiated on the same process model but dynamically restructured to handle specific contexts. Process evolution exploits the information on process variants to identify the best performing recurring adaptations and adopt them as general solutions in the process model. However, process variants are strictly related to specific execution contexts and cannot be adopted as general solutions. In the future we want to extend the CAptLang operational semantics to support also context-aware evolution of business processes based on process instance execution and adaptation history.

[2]In the rule I.NEED the function *adapt* represents the Adaptation Layer.

6. REFERENCES

[1] F. Böse and J. Piotrowski. Autonomously controlled storage management in vehicle logistics applications of rfid and mobile computing systems. *International Journal of RT Technologies: Research an Application*, 1(1):57–76, 2009.

[2] A. Bucchiarone, A. Lluch-Lafuente, A. Marconi, and M. Pistore. A formalisation of adaptable pervasive flows. In *WS-FM*, pages 61–75, 2009.

[3] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. Dynamic Adaptation of Fragment-based and Context-aware Business Processes. In *ICWS 2012*, pages 33–41, 2012.

[4] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. In *Proc. WWW'07*, pages 309–344, 2007.

[5] M. Colombo, E. di Nitto, and M. Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *Proc. ICSOC'06*, pages 191–202, 2006.

[6] R. de Lemos and A. B. Romanovsky. Exception handling in the software lifecycle. *Comput. Syst. Sci. Eng.*, 16(2):119–133, 2001.

[7] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the Provop approach. *Journal of Software Maintenance*, 22(6-7):519–546, 2010.

[8] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. P. Buchmann. Extending BPEL for Run Time Adaptability. In *Proc. EDOC'05*, pages 15–26, 2005.

[9] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An Aspect-Oriented Framework for Service Adaptation. In *Proc. ICSOC'06*, pages 15–26, 2006.

[10] U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *AAAI/IAAI*, pages 447–454, 2002.

[11] I. Lanese, A. Bucchiarone, and F. Montesi. A Framework for Rule-based Dynamic Adaptation. In *Proc. TGC 2010*, pages 284–300, 2010.

[12] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger. Enabling Adaptation of Pervasive Flows: Built-in Contextual Adaptation. In *Proc. ICSOC/ServiceWave*, pages 445–454, 2009.

[13] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *SERVICES*, 2012.