From Use Case Diagrams to Executable Context-aware Ambients

Francois Siewe
Software Technology Research Laboratory
De Montfort University
Leicester, United Kingdom
fsiewe@dmu.ac.uk

Ahmed Al-alshuhai
Software Technology Research Laboratory
De Montfort University
Leicester, United Kingdom
p07143453@myemail.dmu.ac.uk

Abstract—**This paper proposes an approach to translating a use case diagram into an executable context-aware ambients. The requirements of a context-aware system is captured and represented in an extension of UML use case diagrams called *context-aware use case diagrams*. Then an algorithm is proposed that translates a context-aware use case diagram into a process in the Calculus of Context-aware Ambients (CCA). This process can then be analyzed using the CCA simulator. The proposed approach is evaluated using a real-word example of a context-aware collision avoidance system.**

Keywords-Use case diagram; use context diagram; context-aware use case diagram; calculus of context-aware ambients; CCA

I. INTRODUCTION

Context-aware computing envisions a new generation of smart applications that have the ability to perpetually sense the user's context and use these data to make adaptation decision in response to changes in the user's context so as to provide timely and personalised services anytime and anywhere. Thanks to the advances in information and communications technology, the emergence of small sensing devices (e.g. GPS, accelerometer, and gyroscope) and miniaturized wireless communication technologies (e.g. blue-tooth, WiFi, and RFID) embedded in small handheld or wearable computing devices such as smartphones is making this paradigm steadily becoming a reality.

Unlike the traditional distribution systems where the network topology is fixed and wired, context-aware computing systems (CASs) are mostly based on wireless communication due to the mobility of the network nodes; hence the network topology is not fixed but changes dynamically in an unpredictable manner as nodes join and leave the network. These factors make the design and development of context-aware computing systems much more challenging as the system requirements change depending on the context of use.

The notion of context-aware use case diagram has been proposed [1] as an abstract, graphical notation for describing the requirements context-aware systems. It is a powerful tool for requirement capturing and analysis at the early stage of the system development life-cycle. More importantly, it seamlessly integrates both the functional requirements and the context-awareness requirements, showing the dependencies between the two types of requirements. However, these use case diagrams can be interpreted manually but are not machine executable. Therefore the analysis of these diagrams may be time consuming and physically demanding, especially for large scale systems. Meanwhile, a machine executable version of these diagrams will ease and speed up requirements analysis a great deal, and enable various scenarios to be tested and validated timely.

The Calculus of Context-aware Ambients (CCA) [2] is a process calculus for modelling context-aware and mobile systems. The main features of the calculus include concurrency, mobility and context-awareness. More importantly, CCA processes are fully executable and can be analysed using the SPIN model-checker [3].

This paper proposes an approach to translate a context-aware use case diagram into a CCA process. This process can then be analysed using the CCA tools such as ccaPL the interpreter and ccaSPIN a model-checking tool based on SPIN. The contribution of this work is threefold:

- An algorithm is proposed to translate a context-aware use case diagram into a CCA process (Sect. IV).
- It is demonstrated how ccaPL can be used to analyse system requirements through simulation (Sect. V).
- The proposed approach is evaluated using a real-word example of a context-aware collision avoidance system (Sect. V).

II. OVERVIEW OF CONTEXT-AWARE USE CASE DIAGRAMS

A context-aware use case diagram (CA-UCD) is built from a set of use cases, use contexts, actors, context sources (CSs) and their relationships. Use cases are used to capture the functional requirements of applications. A use case describes the desired behaviour of an application or part of an application (i.e. what an application or part of an application can do), without telling how that behaviour is to be implemented. A use case has a name and is graphically rendered as an ellipse as depicted in Fig. 1. Use contexts are used to capture the relevant CIs that affect the behaviour of the application under development, without having to specify how the measurement of those CIs is actually implemented. They

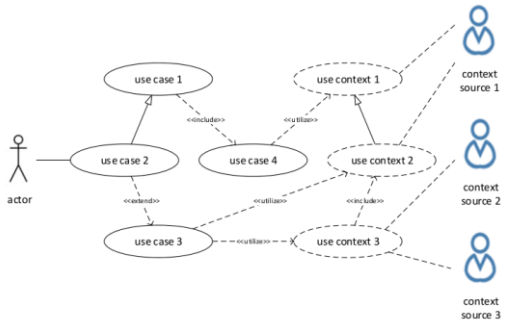


Figure 1. Context-aware use case diagram

also provide the developers a way to come to a common understanding with the application's end user and domain experts as to what CIs the application must be aware of. They are a description of a set of sequence of actions, including variants that an application performs to acquire, to infer or to aggregate CIs from CSs. A use context has a name and is graphically rendered as a dashed ellipse.

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases [4]. Actors can be human or they can be automated systems. An actor is connected to a use case by an association (graphically rendered as a solid line) which indicates that the actor and the use case communicate with one another, possibly by exchanging messages. An actor is represented graphically as a stick figure like in Fig. 1. Context sources are to use contexts what actors are to use cases. Use contexts communicate with context sources to gather raw context data from which CIs are calculated. Typically, context sources are sensors; physical sensors (e.g. a temperature sensor or a light sensor) and virtual sensors (e.g. a weather web service or a calendar) alike. Graphically they are rendered as shown in Fig. 1. Context sources may be connected to use contexts only by a context association represented by a dashed line.

There are three kinds of relationships between use cases. A generalization relationship between use cases means that the child use case can inherit the behaviour and the meaning of the parent use case; the child may add to or override the behaviour its parent; and the child may be substituted any place the parent occurs [4]. The generalization relationship is represented graphically as a solid directed line with a large open arrowhead. For example in Fig. 1, 'use case 1' is a generalization of `use case 2'. Conversely, 'use case 2' is a specialization of 'use case 1'.

An include relationship between use cases means that the base use case explicitly incorporates the behaviour of another use case; while an extend relationship between use cases means the base use case implicitly incorporates the behaviour of another use case. Graphically, both relationships are rendered as a dependency, stereotyped as <<include>> and <<extend>> respectively. In Fig. 1, 'use case 1' includes 'use case 4' while 'use case 2' extends 'use case 3'.

TABLE I. SYNTAX OF CCA

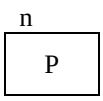
P, Q	::=	$0 \mid$ 'P|Q' $\mid (\nu\ n)\ P \mid !P \mid n[P] \mid \kappa?M.P \mid$ **if** $\kappa_1?M_1.P_1 \ldots \kappa_m?M_m.P_m$ **fi**
M	::=	**in** n $\mid$ **out** $\mid \alpha$ **recv**$(y_1, \ldots, y_m) \mid \alpha$ **send**$(z_1, \ldots, z_m)$
α	::=	$\uparrow \mid n\uparrow \mid \downarrow \mid n\downarrow \mid :: \mid n:: \mid \varepsilon$
κ	::=	True $\mid \bullet \mid n{=}m \mid \neg\kappa \mid$ '$\kappa_1\|\kappa_2$' $\mid \kappa_1 \wedge \kappa_2 \mid \oplus\kappa \mid \Diamond\kappa$

These three kinds of relationships also apply to use contexts. An include relationship is used to avoid describing the same CI several times, by putting the common CI in a use context of its own. An extend relationship is used to model the part of a use context the user may see as optional CI. In this way, optional CIs are separated from mandatory ones. The *utilize* relationship is the only relationship between a use case and a use context. A utilize relationship between a use case and use context means that the behaviours specified by the use case depend upon the CIs described by the use context. For example, 'use case 3' utilizes 'use context 2' and 'use context 3'. A utilise relationship is graphically rendered as a dependency, stereotyped as <<utilize>>, like in Fig. 1. A utilize relationship always points from a use case towards a use context.

III. OVERVIEW OF CCA

This section presents the syntax and the informal semantics of CCA. Table I depicts the syntax of CCA, based on three syntactic categories: processes (denoted by P or Q), capabilities (denoted by M) and context-expressions (denoted by κ). We assume a countably infinite set of names, elements of which are written in lower-case letters, e.g. n, x and y. Keywords are highlighted in bold.

Processes: The process **0**, aka *inactivity process*, does nothing and terminates immediately. The process P|Q denotes the concurrent execution of the processes P and Q. The process (ν n) P creates a new name n and the scope of that name is limited to the process P. The replication !P denotes a process which can always create a new copy of P, i.e. !P is equivalent to P|!P. Replication, first introduced in the π-calculus [5], can be used to implement both iteration and recursion. The process n[P] denotes an ambient named n whose behaviours are described by the process P. The pair of square brackets '[' and ']'outlines the boundary of that ambient. An ambient is represented graphically as:



A context expression specifies a condition upon the state of the environment. A *context-guarded prefix* κ?M.P is a process that waits until the environment satisfies the context expression κ, then performs the capability M and continues like the process P. The dot symbol '.' denotes the sequential composition of processes. We let M.P denote the process **true**?M.P, **true** is a context expression satisfied by all context. The *selection* '**if** $\kappa_1?M_1.P_1 \ldots \kappa_m?M_m.P_m$ **fi**' waits until at least

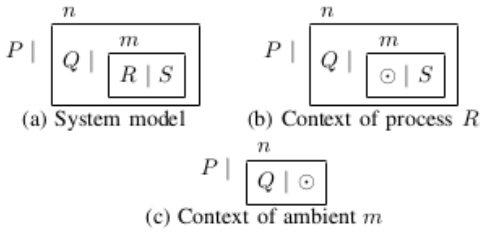


Figure 2. Graphical illustration of the context of a process

one of the context-expressions $(\kappa_i)_{1\le i\le m}$ holds; then proceeds non-deterministically like one of the processes $\kappa_j?M_j.P_j$ for which κ_j holds.

Capabilities: Ambients exchange messages using the output capability α **send**(z_1, …, z_m) to send a list of names z_1, …, z_m to a location α, and the input capability α **recv**(y_1, …, y_m) to receive a list of names from a location α into the variables y_1, …, y_m. The location α can be '↑' to mean any parent, 'n↑' to mean a specific parent n, '↓' to mean any child ambient, 'n↓' to mean a specific child n, '::' to mean any sibling, 'n::' to mean a specific sibling n, or ε (empty string) to mean the executing ambient itself. The mobility capabilities **in** and **out** are defined as follows. An ambient that performs the capability '**in** n' moves into the sibling ambient n. The capability **out** moves the ambient that performs it out of that ambient's parent.

Context model: In CCA, a context is modelled as a process with a hole in it. The hole (denoted by Θ) in a context represents the position of the process that context is the context of. For example, suppose a system is modelled by the process 'P | n[Q | m[R | S]]'. The context of the process R in that system is 'P | n[Q | m[Θ | S]]', and that of the ambient named m is 'P | n[Q | Θ]' as depicted graphically in Fig. 2. A context-expression (CE, for short) is a formula representing some property of a context model.

Context expressions: The CE **true** always holds. A CE n=m holds if the names n and m are lexically identical. The CE ● holds solely for the hole context, i.e. the position of the process evaluating that context expression. Propositional operators such as negation (¬) and conjunction (∧) expand their usual semantics to context expressions. A CE $\kappa_1|\kappa_2$ holds for a context if that context is a parallel composition of two contexts such that κ_1 holds for one and κ_2 holds for the other. A CE n[κ] holds for a context if that context is an ambient named n such that κ holds inside that ambient. A CE ⊕κ holds for a context if that context has a child context for which κ holds. A CE ◊κ holds for a context if there exists somewhere in that context a sub-context for which κ holds. The operator ◊ is called *somewhere modality*, while ⊕ is aka *spatial next modality*.

The following section demonstrates how a context-aware use case diagram can be translated into a CCA process.

IV. Translating Use Case Diagrams into CCA Processes

Algorithm 1 shows how a context-aware use case diagram can be translated into a CCA process. It calls

Algorithm 1: Translating a context-aware use case diagram into a CCA process

Input: A context-aware use case diagram D
Output: A CCA process
P_1 = Algorithm 2(D);
P_2 = Algorithm 3(D)
return $P_1 \mid P_2$

two other algorithms: Algorithm 2 which translates each actor and each use case into an ambient; and Algorithm 3 which translates each context source and each use context into an ambient. The final process is the parallel composition of all the ambients so created. Note that associations and dependency relationships are modelled as interactions (i.e. communications) between these ambients.

An actor is modelled as ambient that may interact with any use case it is connected to by sending a message REQUEST_USE_CASE to activate a use case (see (2)) and receiving notifications as depicted in (1). The notation compose(P_1, …, P_n) represents one of the four different ways an actor may invoke the use cases it is connected to:

- None: compose(P_1, …, P_n) = **0**
- Sequentially: compose(P_1, …, P_n) = P_1. … .P_n
- Concurrently: compose(P_1, …, P_n) = P_1 | … | P_n
- Randomly: compose(P_1, …, P_n) = **if true**?$M_1.P_1$ … **true**?$M_n.P_n$ **fi**

Any combination of these patterns of actor's behaviours may be considered during simulation and analysis, depending on the application in hand.

Consequently, a use case is modelled as an ambient that receives a request (from one of its actors, or from another use case it extends, or from another use case it is included into) and acquires all the CI it needs by interacting with the use contexts it utilizes and then invokes all the use cases it includes and a subset (possibly empty) of the all the use cases that extend it (see (3) and (4)). The function F_U in (3) is an abstract representation of the intended behaviours of a use case U; parameterised with that use case interactions with others use cases and use contexts. The concrete specification of this function is application dependent.

A context source is modelled as an ambient that passes fresh sensed raw context values onto use contexts requesting them (see (5)). Freshness is modelled by random selection of a value from a representative sample of possible context values. Of course the determination of such sample is application dependent; and hence left to the system designer.

A use context is modelled as an ambient that receives a request from a use case or from another use context that it extends, or from another use context that includes it; then reads all the raw context values it needs from context sources

Algorithm 2: Translating actors and use cases into ambients

Input: A context-aware use case diagram
Output: A CCA process
foreach *actor* A **do**
- Let $U_1, \ldots, U_n$ be the use cases this actor is connected to by an *association*.
- Create an ambient of the form:

$$A[\ compose(P_1, \ldots, P_n) \mid !::\texttt{recv}(y_1, \ldots, y_\ell).\mathbf{0}\ ] \quad (1)$$

where each process P_i models a request to perform the use case U_i, $1 \leq i \leq n$, and has the form:

$$P_i = U_i::\texttt{send}(A, \texttt{REQUEST\_USE\_CASE}) \quad (2)$$

end
foreach *use case* U **do**
- Let $C_1, \ldots, C_k$ be the use contexts that U *utilizes*.
- Let $I_1, \ldots, I_\ell$ be the use cases that U *includes*.
- Let $E_1, \ldots, E_m$ be the use cases that *extend* U.
- Create an ambient of the form:

$$U[\ !::\texttt{recv}(sender, request).\ F_U(< P_1, \ldots, P_k >, < Q_1, \ldots, Q_\ell >, < R_1, \ldots, R_m >)\ ] \quad (3)$$

where F_U is a process describing the behaviour of the use case U and its parameters are explained as follows: P_i is a process modelling the interactions between the use case U and the use context C_i, $1 \leq i \leq k$; Q_i is a process modelling the interactions between the use case U and the included use case I_i, $1 \leq i \leq \ell$; R_i is a process modelling the interactions between the use case U and the extending use case E_i, $1 \leq i \leq m$; they have the forms:

$$\begin{array}{lcl} P_i & = & C_i::\texttt{send}(U, \texttt{ACQUIRE\_CONTEXT}).C_i::\texttt{recv}(y_1, \ldots, y_{k_i}) \\ Q_i & = & I_i::\texttt{send}(U, \texttt{CALL\_USE\_CASE}).I_i::\texttt{recv}(y_1, \ldots, y_{\ell_i}) \\ R_i & = & E_i::\texttt{send}(U, \texttt{CALL\_USE\_CASE}).E_i::\texttt{recv}(y_1, \ldots, y_{m_i}) \end{array} \quad (4)$$

for some non negative integers k_i, ℓ_i, and m_i. The variables $sender$ and $request$ may occur free in F_U.
end
return *Parallel composition of all the ambients created.*

Algorithm 3: Translating context sources and use contexts into ambients

Input: A context-aware use case diagram
Output: A CCA process
foreach *context source* S **do**
- Let $V_1, \ldots, V_t$ be a sample of possible context values.
- Create an ambient of the form:

$$\begin{array}{l} S[\\ \quad !::\texttt{recv}(sender, request). \\ \quad \texttt{if} \\ \quad\quad (\texttt{true})?sender::\texttt{send}(V_1).\mathbf{0} \\ \quad\quad \ldots \\ \quad\quad (\texttt{true})?sender::\texttt{send}(V_t).\mathbf{0} \\ \quad \texttt{fi} \\ ] \end{array} \quad (5)$$

end
foreach *use context* C **do**
- Let $S_1, \ldots, S_k$ be the context sources connexted to C by a *context assocation*.
- Let $I_1, \ldots, I_\ell$ be the use contexts that C *includes*.
- Let $E_1, \ldots, E_m$ be the use contexts that *extend* C.
- Create an ambient of the form:

$$C[\ !::\texttt{recv}(sender, request).\ F_C(< P_1, \ldots, P_k >, < Q_1, \ldots, Q_\ell >, < R_1, \ldots, R_m >)\ ] \quad (6)$$

where F_C is a process describing the behaviour of the use context C and its parameters are explained as follows: P_i is a process modelling the interactions between the use context C and the context source S_i, $1 \leq i \leq k$; Q_i is a process modelling the interactions between the use context C and the included use context I_i, $1 \leq i \leq \ell$; R_i is a process modelling the interactions between the use context C and the extending use context E_i, $1 \leq i \leq m$; they have the forms:

$$\begin{array}{lcl} P_i & = & S_i::\texttt{send}(C, \texttt{READ\_RAW\_CONTEXT}).S_i::\texttt{recv}(y_1, \ldots, y_{k_i}) \\ Q_i & = & I_i::\texttt{send}(C, \texttt{CALL\_USE\_CONTEXT}).I_i::\texttt{recv}(y_1, \ldots, y_{\ell_i}) \\ R_i & = & E_i::\texttt{send}(C, \texttt{CALL\_USE\_CONTEXT}).E_i::\texttt{recv}(y_1, \ldots, y_{m_i}) \end{array} \quad (7)$$

for some non negative integers k_i, ℓ_i, and m_i. The variables $sender$ and $request$ may occur free in F_C.
end
return *Parallel composition of all the ambients created.*

and invokes all the use contexts it includes and a subset (possibly empty) of all the use contexts that extend it. The collected data are used to calculate the CI to be sent to the requester. Similarly to a use case, a use context is an abstraction of *what* CI an application needs and not *how* to calculate them. Hence, the actual calculation of the CI is application dependent and therefore cannot be specified in the general case. The function F_C represents such an abstraction for each use context C.

The CCA process generated by Algorithm 1 can be analysed and animated using CCA tools as shown in the following section.

V. Analysis of Use Case Diagrams using CCA

There are three main tools for analysis CCA processes: (i) ccaPL: an interpreter that executes CCA processes, useful for simulation; (ii) ccaGraph: a tool that represents the execution traces of a process in the form of graphs (i.e. a communication graph, a mobility graph or place graph, and a combined graph that shows both types of information); and (iii) ccaSPIN: a model checking tool that generates from a process a semantically equivalent Promela program which is then

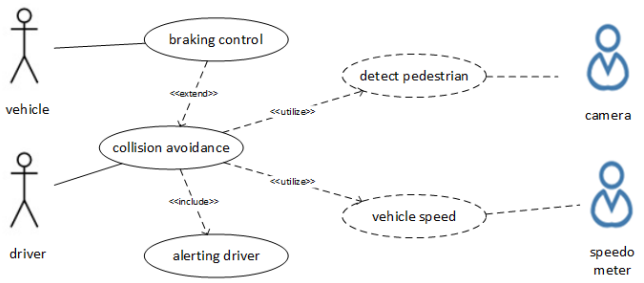


Figure 3. A context-aware use case diagram for a pedestrian collision avoidance system

analysed using the SPIN model-checker [3]. Due the space limit, only the first tool will be used in this paper to demonstrate how CCA can be used to analyse context-aware use case diagrams.

Consider the context-aware use case diagram of Fig. 3 for a pedestrian collision avoidance system that enables a vehicle to recognize and respond to potential pedestrian collision situations. The system uses a stereo camera to monitor the path in front of the vehicle and to detect the position and velocity of a pedestrian on the road. A speedometer informs the system of the vehicle current speed. Based on the vehicle speed and the pedestrian position and velocity, the collision avoidance system infers whether a collision may happen in which case the driver is alerted and optionally the braking control is activated. The breaking control applies torque to the wheels to decelerate the vehicle to a safe speed.

Algorithm 1 is applied to the context-aware use case diagram in Fig. 3 to generate the CCA process of Fig. 4, where the ambient coll_av represents the use case *collision avoidance*, the ambient detect_p corresponds to the use context *detect pedestrian* and the ambient speed models the use context *vehicle speed*. The camera senses the position and the velocity of a pedestrian. The possible values for the position are NONE (no pedestrian detected), CLOSE and FAR; while the values for the velocity are 0 (zero), SLOW, and FAST. As for the speed of the vehicle, the values are LOW, MEDIUM, and HIGH.

The process in Fig. 4 is randomly simulated in ccaPL and some simulation results are given below. The ambient coll_av acquires the vehicle speed and the pedestrian position and velocity from the respective ambients. The simulation shows that:

- **Scenario 1**: If no pedestrian is detected then the driver is not alerted and the braking control is not activated as depicted in Fig. 5. The simulation output is interpreted as follows. The symbol '-->' represents the reduction relation as defined in the formal semantics of CCA in [5]; it corresponds to one execution step. Each execution step is explained using a notation of the form {A ===(X)===> B} which means that during that execution step the ambient A sent the list of messages X to the ambient B.
- **Scenario 2**: If a pedestrian is detected (close and not moving) and the vehicle speed is high then the driver is alerted and the braking control is activated (see Fig. 6).
- **Scenario 3**: If a pedestrian is detected and is far away and the vehicle speed is low then the driver is alerted but the braking control is not activated (Fig. 7).

```
driver[
    coll_av::send(driver, REQUEST_USE_CASE)
   | ! ::recv(notification).0
] |
vehicle[
    ! ::recv(notification).0
] |
coll_av[
   ! ::recv(sender, request).
    detect_p::send(coll_av, CALL_USE_CONTEXT).
    detect_p::recv(pos, velo).
    speed::send(coll_av, CALL_USE_CONTEXT).
    speed::recv(val).if
     (not(pos=NONE) and val=HIGH)? alerting_driver::
       send(coll_av, CALL_USE_CASE).
       alerting_driver::recv(ack).
       braking_control::send(coll_av, CALL_USE_CASE).
       braking_control::recv(ack).0
     (pos=FAR and val=MEDIUM)? alerting_driver::
       send(coll_av, CALL_USE_CASE).
       alerting_driver::recv(ack).0
     (pos=CLOSE)? alerting_driver::
       send(coll_av, CALL_USE_CASE).
       alerting_driver::recv(ack).
       braking_control::send(coll_av, CALL_USE_CASE).
       braking_control::recv(ack).0
    fi.
    sender::send(DONE).0
] |
braking_control[
    ! ::recv(sender, request).vehicle::send(BREAK_ON).
       sender::send(DONE).0
] |
alerting_driver[
    ! ::recv(sender, request).driver::
       send(ALERT_PEDESTRIAN).
       sender::send(DONE).0
] |
detect_p[
    ! ::recv(sender, request).camera::
       send(detect_p, READ_RAW_CONTEXT).
     camera::recv(position, velocity).sender::
       send(position, velocity).0
] |
speed[
    ! ::recv(sender, request).speedometer::send(speed,
          READ_RAW_CONTEXT).
     speedometer::recv(val).sender::send(val).0
] |
camera[
    ! ::recv(sender, request).if
      (true)?sender::send(NONE, 0).0
      (true)?sender::send(CLOSE, 0).0
      (true)?sender::send(CLOSE, SLOW).0
      (true)?sender::send(CLOSE, FAST).0
      (true)?sender::send(FAR, 0).0
      (true)?sender::send(FAR, SLOW).0
      (true)?sender::send(FAR, FAST).0
    fi
] |
speedometer[
    ! ::recv(sender, request).if
       (true)?sender::send(LOW).0
       (true)?sender::send(MEDIUM).0
       (true)?sender::send(HIGH).0
    fi
]
```

Figure 4. CCA process corresponding to the use case diagram in Fig. 3

```
CCA Parser Version 4.03: Reading from file usecase.cca . . .
CCA Parser Version 4.03: CCA program parsed successfully.

Execution mode: interleaving

---> {driver ===(driver,REQUEST_USE_CASE)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> detect_p}
---> {detect_p ===(detect_p,READ_RAW_CONTEXT)===> camera}
---> {camera ===(NONE,0)===> detect_p}
---> {detect_p ===(NONE,0)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> speed}
---> {speed ===(speed,READ_RAW_CONTEXT)===> speedometer}
---> {speedometer ===(HIGH)===> speed}
---> {speed ===(HIGH)===> coll_av}
```

Figure 5. Simulation output of scenario 1

```
CCA Parser Version 4.03: Reading from file usecase.cca . . .
CCA Parser Version 4.03: CCA program parsed successfully.

Execution mode: interleaving

---> {driver ===(driver,REQUEST_USE_CASE)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> detect_p}
---> {detect_p ===(detect_p,READ_RAW_CONTEXT)===> camera}
---> {camera ===(CLOSE,0)===> detect_p}
---> {detect_p ===(CLOSE,0)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> speed}
---> {speed ===(speed,READ_RAW_CONTEXT)===> speedometer}
---> {speedometer ===(HIGH)===> speed}
---> {speed ===(HIGH)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CASE)===> alerting_driver}
---> {alerting_driver ===(ALERT_PEDESTRIAN)===> driver}
---> {alerting_driver ===(DONE)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CASE)===> braking_control}
---> {braking_control ===(BREAK_ON)===> vehicle}
---> {braking_control ===(DONE)===> coll_av}
---> {coll_av ===(DONE)===> driver}
```

Figure 6. Simulation output of scenario 2

```
CCA Parser Version 4.03: Reading from file usecase.cca . . .
CCA Parser Version 4.03: CCA program parsed successfully.

Execution mode: interleaving

---> {driver ===(driver,REQUEST_USE_CASE)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> detect_p}
---> {detect_p ===(detect_p,READ_RAW_CONTEXT)===> camera}
---> {camera ===(FAR,SLOW)===> detect_p}
---> {detect_p ===(FAR,SLOW)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CONTEXT)===> speed}
---> {speed ===(speed,READ_RAW_CONTEXT)===> speedometer}
---> {speedometer ===(MEDIUM)===> speed}
---> {speed ===(MEDIUM)===> coll_av}
---> {coll_av ===(coll_av,CALL_USE_CASE)===> alerting_driver}
---> {alerting_driver ===(ALERT_PEDESTRIAN)===> driver}
---> {alerting_driver ===(DONE)===> coll_av}
---> {coll_av ===(DONE)===> driver}
```

Figure 7. Simulation output of scenario 3

VI. Related Work

UML is a diagram language which enables designers of information systems to illustrate high level system requirements, using use case diagrams, and to demonstrate low level system requirements, using activity diagrams [6]. Choi and Lee [7] proposed a model-driven approach that uses UML's use case diagrams to elicit the requirement of context-aware applications. In particular, the approach helps analysts and stakeholders pay more attentions to context related issues such as system platform, target users, intelligence, possible context-aware services and agreement with other stakeholders, and understanding contexts with decision tables and trees.

ContUML [8] is a UML-based language for model-driven development of context-aware applications. However, ContUML essentially extends the UML's class diagram with special classes for CIs and context-awareness mechanisms. Our context-aware use case diagrams are more abstract than class diagrams and so more suitable for requirement elicitation and analysis. It is understood that ContUML may be used for the realization of context-aware use case diagrams during system development. Almutairi et al. [9] extended the UML's use case diagram and activity diagram to capture the security requirement of context-aware application. In particular, they introduces a "requires" relationship between a use case and CIs to indicate the CIs the behaviours described by that use case depend upon. In our approach, use context diagrams are used to specify CIs and their corresponding CSs; separately from the use cases that will utilize those CIs. This separation of concerns between functional requirements and context-awareness requirements is helpful, especially when dealing with large scale or complex context-aware applications.

VII. Conclusion

This paper proposed an algorithm for translating a context-aware use case diagram into a CCA process in the aim of using the CCA tools to analyse the requirements of context-aware systems. It is demonstrated how the CCA interpreter can be used to execute and validate various scenarios of a use case diagram. The pragmatics of the approach is illustrated using a real-world example of a context-aware collision avoidance system. In future work, it will be demonstrated how the model-checking tool ccaSPIN can be used to analyze the requirements of context-aware systems.

References

[1] A. Al-Alshuhai and F. Siewe, "An extension of the use case diagram to model context-aware applications," in *SAI Intelligent Systems Conference*, 2015.

[2] F. Siewe, H. Zedan and A. Cau, "The Calculus of Context-aware Ambients," *Journal of Computer and System Sciences*, vol. 77, no. 4, pp. 597-620, July 2011.

[3] J. G. Holzmann, "The model checker spin," IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 1-17, May 1997.

[4] G. Booch, J.~Rumbaugh, and I.~Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[5] R. Milner, *Communication and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[6] A. Finkelstein and A. Savigni, "A framework for requirements engineering for context-awareness services," in *First International Workshop from Software Requirements to Architectures*, 2001.

[7] J. Choi and Y. Lee, "Use-case driven requirements analysis for context-aware systems," in *The Future Generation Information Technology Conference*. Springer, 2012.

[8] Q. Z. Sheng and B. Benatallah, "ContextUML: A UML-based modeling language for model-driven development of context-aware web services," in *International Conference on Mobile Business (ICMB05)*, 2005.

[9] A. Almutairi, A. Abu-Samaha, G. Bella, and F. Chen, "An enhanced use case diagram to model context aware system," in *SAI conference*, 2013.