SEMANTIC STUDIES OF A SYNCHRONOUS APPROACH TO ACTIVITY RECOGNITION

Ines SARRAY[1], Annie RESSOUCHE[1], Sabine MOISAN[1], Jean-Paul RIGAULT[1] and Daniel GAFFE[2]

[1]Universite´Coˆted'Azur, INRIA, SophiaAntipolis, France
[2]Universite´Coˆted'Azur, CNRS, LEAT, SophiaAntipolis, France

ABSTRACT

Many important and critical applications such as surveillance or healthcare require some form of (human) activity recognition. Activities are usually represented by a series of actions driven and triggered by events. Recognition systems have to be real time, reactive, correct, complete, and dependable. These stringent requirements justify the use of formal methods to describe, analyze, verify, and generate effective recognition systems. Due to the large number of possible application domains, the researchers aim at building a generic recognition system. They choose the synchronous approach because it has a well-founded semantics and it ensures determinism and safe parallel composition. They propose a new language to represent activities as synchronous automata and they supply it with two complementary formal semantics. First a behavioral semantics gives a reference definition of program behavior using rewriting rules. Second, an equational semantics describes the behavior in a constructive way and can be directly implemented. This paper focuses on the description of these two semantics and their relation.

KEYWORDS

Activity Recognition, Language, Synchronous Approach, Semantics

1. INTRODUCTION

Activity Recognition aims at recognizing sequences of human actions that follow the predefined model of an activity. Our research team mainly works on medical applications to help physicians detect abnormal behaviors or monitor patient activities such as serious games.

Recognition systems must satisfy stringent requirements: dependability, real time, cost effectiveness, security and safety, correctness, completeness... To enforce most of these properties, the chosen approach is to base the configuration of the system as well as its execution upon formal techniques. Therefore, these formal bases should permit static analysis, verification and validation, but also easy and direct implementation.

The aim is to build a generic recognition system for such activities. The authors chose to model these activities as synchronous finite automata. The synchronous paradigm ensures determinism and supports concurrency through parallel composition. In particular, critical races can be detected by static analysis. This model is also well-founded owing to formal semantics. Thus the recognition system benefits from the sound foundations of the synchronous approach and from the automata theory, allowing automatic proofs, static verification, powerful simulation, code generation, etc.

For the users to describe synchronous automata, languages such as Lustre, Esterel, Scade, and Signal [1] have been defined. These languages are for expert users. This paper proposes another language called ADeL (Activity Description Language). Building a complete generic recognition system involves many different aspects. The paper concentrates on the formal description of activities as synchronous automata and their mathematical semantics. However, the flavor of all these different aspects will begiven.

The paper is organized as follows. The next section is a short reminder of the synchronous model of reactive systems. An overview of the ADeL language is given in section 3. Section 4 is the core of the paper: it introduces the semantics and the mathematical concepts on which researchers rely to define and verify the behavior of programs and to compile them. Finally several related works are presented before concluding.

2. SYNCHRONOUS MODEL OF REACTIVE SYSTEMS

The Synchronous Paradigm relies on a discrete logical time composed of a sequence of logical instants, defined by the system reactions.

Reactive systems listen to input events coming from the external environment and react to them by generating output events towards the environment. Such systems can be complex. The synchronous model is a way to reduce the complexity of behavior description by considering their evolution along successive discrete instants. An instant starts when some input events are available. The output and internal events deriving from these inputs are computed until stability (fixed point) is achieved; the instant finishes by delivering the output events to the environment.

No inputs occurring “during” the instant are considered. Hence, instants are atomic, their sequence defines a logical time. In this model, instants take “no time” with respect to the logical time they define.

The synchronous paradigm is interesting because it ensures determinism and it supports concurrency through deterministic parallel composition. In particular, critical races are detected by static analysis. It supports a true notion of simultaneous events and provides not only a reaction to the presence of an event but also to its absence (to some extent). This model is also well-founded owing to formal semantics. Moreover, along the last decades, tool sets for simulation, verification, and code generation of synchronous automata have been developed.

The synchronous model has been applied to several different systems, from hardware design [2] to embedded real time systems [3]. In this work, the team proposes to apply it to another real time system, namely human activity recognition. Synchronous models can be represented as *Mealy machines*. The Mealy machines that they consider are 6-uples of the form: $<Q, q_{init}, I, O, \lambda, \tau>$, where Q is a finite set of states, $q_{init} \in Q$ is the initial state, I (resp. O)is a finite set of input (resp. output) events; λ: $(Q \times I) \rightarrow Q$ is the transition function and $\tau : (Q \times I) \rightarrow O$ is the output function. This is an explicit representation of Mealy machines as automata. Mealy himself introduced another representation as Boolean equation systems that calculate both the output event values and the next state from the input event values and the current state [4]. The authors call this representation "implicit" Mealy machines.

Synchronous languages such as Lustre, Esterel, Scade, and Signal[1] have been defined to describe synchronous automata. These languages are for expert users. This paper proposes another synchronous language that is easier to understand and to work with for non-computer scientists (e.g., doctors). To improve its acceptance and its ease of use by non-computer scientists, the authors are working in collaboration with ergonomists and doctors from Claude Pompidou hospital. This language is called ADeL (Activity Description Language) and is described in the next section.

3. ACTIVITY DESCRIPTION LANGUAGE (ADEL)

ADeL provides two different (and equivalent) formats: graphical and textual. It is a modular and hierarchical language, which means that an activity may contain one or more sub-activities. The description of an activity consists of several parts: first the user defines the participants in the activity, their types, their roles, as well as the initial state of the activity. Second, the user describes the expected behavior using a set of control operators detailed in table 1. These operators are the base of the ADeL language. They have a synchronous semantics and they deal with events coming from their environment.

Table1.ADeL operators. S, S_1 are events (received or emitted);p, p_1 and p_2 are instructions; condition is either an event or a Boolean combination of event presence/absence

nothing	does nothing and terminates instantaneously.
[**wait**] S	waits for event *S* and suspends the execution of the activity until *S* is present. Operator wait can be implicit or explicit.
p_1 **then** p_2	starts when p_1 starts; p_2 starts when p_1 ends; the sequence terminates when p_2 does.
p_1 **parallel** p_2	starts when p_1 or p_2 start; ends when both have terminated.
p_1 **during** p_2	p_1 starts only after p_2 start and must finish before $p_2$end.
while *condition* {*p*}	*p* is executed only if the *condition* is verified. When *p* ends, the loop restarts until the *condition* holds.
stop {*p*} **when** *S* [**alert** S_1]	executes *p* to termination as long as *S* is absent, otherwise when *S* is present, aborts *p*, sends an alert S_1, and terminates
if *condition* **then** p_1[**else** p_2]	executes p_1 if *condition* holds, otherwise executes p_2.

p **timeout** *S* {p_1}[**alert** S_1]	executes *p* ; stops if *S* occurs before *p* terminates and possibly sends alert S_1; otherwise executes $p_1$when *p* has terminated.
alert *S*	raises an alert.
Local (*events*){*p*}	declares internal events to communicate between sub parts of *p*.
Call (*activity*)	calls a sub-activity.

Some of these operators are "instantaneous" (**nothing, alert**) while others take at least one (synchronous) instant to process.

Compared to other synchronous languages where it is difficult or even impossible to treat the real clock time, ADeL can manipulate it thanks to the operator **"timeout"**. For example, deadlines are expressed as follows: P **timeout** *S* {P_1} (*S* is a timed signal). To compare with a classical approach, in Esterel, this operator should be written as:

```
abort{p} when S;
presentS then alert else P1;
```

This part of Esterel code seems easy for a programmer but it is not the case for non-computer scientists such as doctors. Indeed, it would be even more difficult to write this kind of code in a declarative synchronous language like Lustre. Moreover, it is more complex to use these languages to express the "**during**" operator.

The main issue of the synchronous paradigm is that the world is not synchronous in general. Thus it requires to transform asynchronous physical flows of events into a succession of discrete instants. The authors propose a synchronous transformer, called Synchronizer. The Synchronizer receives asynchronous events from the environment, filters them, decides which ones may be considered as "simultaneous", and groups them into a logical instant according to predefined policies. In general, no exact simultaneity decision algorithm exists but several empirical strategies may be used for determining instant boundaries, relying on event frequency, event occurrence, elapsed time, etc. To manage the real clock time, the Synchronizer considers the clock time as an event like others.

4. ADEL SEMANTICS AND COMPILATION

To provide the language with sound foundations, the authors turn to a formal semantic approach. First, logical rewriting rules are a classical and rather natural way to formally express the intuitive semantics. This form of behavioral semantics gives an abstract description of a program behavior and facilitates its analysis. However, it is not convenient as an implementation basis nor suitable for proofs (e.g., model-checking). Hence, an equational semantics, which maps an ADeL program to a Boolean equation system representing its finite state machine, was also defined. The ADeL compiler can easily translates this equation system into an efficient code. Using such a double semantics is somewhat traditional in the synchronous language area[5].

Since there are two different semantics, it is mandatory to establish their relationship. In fact the authors proved that the execution of a program based on the equational semantics also conforms to the behavioral semantics (see4.4).

4.1 Mathematical Context

One of the bases of ADeL semantics is the notion of an *environment*, which is a finite set of events. Environments record the status of events in each synchronous instant and the goal of the ADeL semantics is to compute the status of the output events for each reaction of a program. A 4-valued algebra ($\xi = \{\bot, 0, 1, \top\}$) is used to represent the status: $\bot$ means that the status of the event is not yet determined, 0 that the event is absent, 1 that the event is present, and $\top$ that the status of the event is over determined (error). Usually $\top$ Occurs because the event would have two incompatible status in the same instant (e.g., 0 and 1 in different parts of the program).

Previously, synchronous language semantics expressed the status of events using 3-valued algebras. Indeed, such semantics either cannot reflect how the information about event status grows when the algebra is a lattice ($0 \leq \bot \leq 1$), or fix the status of each event to 0 or 1 in each instant when the algebra is a complete partial order (c.p.o) ($\bot \leq 0$, $\bot \leq 1$)[6]. In this latter case, at the beginning of the computation of event status, all unknown status are set to $\bot$. Then, as soon as an event is present in a part of the program, its status grows to 1, otherwise it is set to 0. This way prevents any incremental compilation of activities. To compile a main activity without the knowledge of event status of a sub activity, these latter must be kept to $\bot$.An event never present in a program has 0 for status and if it is present in a sub program, its status cannot be changed to 1 when the already compiled sub program is included in the main one, because 1 and 0 are incomparable with respect to the c.p.o order and have no upper bound. It is why the authors consider a 4-valued algebra and a structure which allows us to rely on the semantics rules to compile programs in an incremental way. Moreover, this algebra provides us with a convenient means to compile activities.

As a consequence, the authors supply ξ with a bilattice structure [7]. Bilattices are mathematical structures having two distinct orders denoted $\leq_B$ (Boolean order) and $\leq_K$ (knowledge order) and a $\neg$ operation, such that both $(\xi, \leq_B)$ and $(\xi, \leq_K)$ are lattices for their respective orders. In ξ, $\leq_B$ represents an extension of the usual Boolean order and $\leq_K$ expresses the level of information about the presence of an event. These two orders are defined as follows: $\bot \leq_K 0 \leq_K \top$; $\bot \leq_K 1 \leq_K \top$; $0 \leq_B \bot \leq_B 1$; $0 \leq_B \top \leq_B 1$. These two orders play complementary roles in the ADeL semantics: the Boolean order is a means to calculate the event status while the knowledge order reflects the growth of information about event status when computing the equation system. As a consequence, four operations in ξ have been introduced: $\boxdot$ and $\boxplus$ are respectively the “meet” and “join” operations of $(\xi, \leq_B)$ and $\sqcup$ and $\sqcap$ play the same roles for $(\xi, \leq_K)$.

Finally, the $\neg$ operator is used to reverse the notion of truth from a Boolean point of view, but its role with respect to $\leq_K$ has to be transparent: actually, no more nor less information about x and $\neg x$ is known, then the authors give the following definitions for this $\neg$ operator: $\neg 1 = 0$, $\neg 0 = 1$, $\neg \top = \top$, and $\neg \bot = \bot$.

Another important feature of our approach is the ability to encode ξ elements into pairs of Boolean ones. There exist several possible encoding functions and the researchers choose one which is compatible with the $\leq_K$ order:

$$e : \xi \mapsto \mathbb{B} \times \mathbb{B} : x \in \xi,\ e(x) = (x_h, x_l) : \begin{cases} \bot & \mapsto & (ff, ff) \\ 0 & \mapsto & (ff, tt) \\ 1 & \mapsto & (tt, ff) \\ \top & \mapsto & (tt, tt) \end{cases}$$

Here $\mathbb{B}$ is the usual Boolean algebra $\{ff, tt\}$.

This encoding function extends to the ξ operators. The structure ($\mathbb{B}$, $\leq$) is a complete lattice for the $ff \leq tt$ order. Then, the structure: $\mathbb{B} \odot \mathbb{B} = (\mathbb{B} \times \mathbb{B}, \leq_B, \leq_K, \neg)$ defined as follows:

$$\begin{cases} (x_1, x_2) \leq_B (y_1, y_2) & \text{iff} \quad x_1 \leq y_1 \textit{ and } y_2 \leq x_2 \\ (x_1, x_2) \leq_K (y_1, y_2) & \text{iff} \quad x_1 \leq y_1 \textit{ and } x_2 \leq y_2 \\ \neg(x_1, x_2) = (x_2, x_1) & \end{cases}$$

is a bilattice and the following theorem holds:

Theorem 1 *(ξ, $\leq_B$, $\leq_K$, $\neg$) and* $\mathbb{B} \odot \mathbb{B}$ *are isomorphic.*

To justify this theorem, the authors show that the encoding *e* previously defined is an isomorphism between (ξ, $\leq_B$, $\leq_K$, $\neg$) and $\mathbb{B} \odot \mathbb{B}$ Indeed, the four binary operations and the negation one of the (ξ, $\leq_B$, $\leq_K$, $\neg$) bilattice are preserved in $\mathbb{B} \odot \mathbb{B}$ The proof is detailed in [8].

As a result of the theorem, the encoding *e* previously defined for ξ elements can be extended to the operators of the bilattice (ξ, $\leq_B$, $\leq_K$, $\neg$)[1]:

$$\begin{bmatrix} e(x \sqcup y) & = & (x_h + y_h, x_l + y_l) \\ e(x \sqcap y) & = & (x_h . y_h, x_l . y_l) \\ e(x \boxplus y) & = & (x_h + y_h, x_l . y_l) \\ e(x \boxdot y) & = & (x_h . y_h, x_l + y_l) \end{bmatrix}$$

Thus, one can efficiently convert ξ-equation systems into the Boolean universe.

4.1.0.1 Extension to Environments

Owing to the ξ algebra, it is now possible to formally introduce the notion of *environments*. Environments are finite sets of events where each event has a single status.

[1]In the following equations, + and .denote the join and meet operations of the lattice (B, $\leq$))

More formally, consider a finite set of events S = {S_0, S_1, ...S_n, ...}. A valuation $\mathcal{V} : \mathcal{S} \mapsto \xi$ is a function that maps an event $S \in \mathcal{S}$ to a status value in ξ. Each valuation $\mathcal{V}$ defines an environment : $E = \{S^x \mid S \in \mathcal{S}, x \in \xi, \mathcal{V}(S) = x\}$. The goal of the semantics is to refine the status of the events of a program in each instant from ⊥ to T according to the knowledge order ($\leq_K$).

Then, for each instruction *p*, built with ADeL operators, let us denote $\mathcal{S}(p)$ the finite set of its events and $\mathcal{E}(p)$ the set of all possible environments built from $\mathcal{S}(p)$.Operations in (ξ, $\leq_B$, $\leq_K$,¬) can be extended to environments[2]:

$$\begin{aligned} \neg E &= \{S^x | S^{\neg x} \in E\} \\ E \sqcup E' &= \{S^z | \exists S^x \in E, \exists S^y \in E', z = x \sqcup y\} \\ &\quad \cup \{S^x | S^x \in E, \nexists y \in \xi, S^y \in E'\} \\ &\quad \cup \{S^y | S^y \in E', \nexists x \in \xi, S^x \in E\} \end{aligned}$$

The order relation $(\preceq)$ on environments is defined as follows:

$$E \preceq E' \text{ iff } \forall S^x \in E, \exists S^y \in E' | S^x \leq_K S^y$$

Thus $E \preceq E'$ means that each element of E is less than an element of E' according to the lattice knowledge order of ξ. As a consequence, the $\preceq$ relation is a total order on $\mathcal{E}(p)$ and $\sqcup$ and $\sqcap$ operations are monotonic according to $\preceq$. Moreover, ($\mathcal{E}(p)$, $\preceq$) is a complete lattice, its greatest element is $\{S^\top \mid S \in \mathcal{S}(p)\}$ and its least element is $\{S^\perp \mid S \in \mathcal{S}(p)\}$. According to Tarski's theorem, each monotonic increasing function F has a least fixed point, computed by iteration of F from the least element [9]. This ensures that the behavioral semantics has solutions.

4.2 Behavioral Semantics

Behavioral semantics is a classical and formal way to describe behaviors in an axiomatic way. This semantics formalizes each reaction of a program by computing the output environment from the input one. To this aim, it defines a set of rewriting rules of the form:

$$p \xrightarrow[E]{E',term} p'$$

where *p* and *p'* are two instructions of ADeL, p' is the derivative of *p*, i.e. the new instruction that will react to the next input environment. *E* is the input environment, E' is the resulting output environment, and *term* is a Boolean flag which describes the termination of *p*, and which turns to true when *p* terminates. The rewriting rules of the whole program apply from the root instruction, structurally following the syntactic tree of the program.

[2]Only the operations needed to define both semantics are introduced. However, the five operators of ξ can be similarly extended.

Due to lack of space, the behavioral semantics of all the operators cannot be described. Only the rules for two operators are presented: **parallel** which is specific to synchronous languages and **timeout** that takes into consideration the synchronous time. Nevertheless, a complete description is detailed in[10].

4.2.0.1 Operator parallel.

Operator **parallel** has two argument instructions that are executed and computed concurrently, possibly broadcasting events between them. Thus the evolution of both instructions can have an impact on both environments. The operator ends when the two instructions terminate, i.e. when $term_{p}1$ and $term_{p}2$ become true, and the resulting output environment is the unification of the respective resulting environments computed for p_1 and p_2.

$$\frac{p_1 \xrightarrow[E \sqcup E_2]{E_1, term_{p_1}} p_1' \quad , \quad p_2 \xrightarrow[E \sqcup E_1]{E_2, term_{p_2}} p_2'}{p_1 \| p_2 \xrightarrow[E]{E_1 \sqcup E_2,\ term_{p_1}\ and\ term_{p_2}} p_1' \| p_2'} \tag{1}$$

4.2.0.2 Operator timeout.

The behavior of: p **timeout** $S\{p_1\}$ **alert** S_1 depends both on the behavior of its instruction p and on the status of S. If S is not present and p terminates, p_1 starts and the behavior of the operator turns out to be the behavior of p_1 (rule2).

$$\frac{p \xrightarrow[E]{E_1, 1} \textbf{nothing}, S \notin E,\ p_1 \xrightarrow[E1]{E_2, term_{p_1}} p_1'}{\textbf{timeout}(p,\ S\ \{p_1\}) \xrightarrow[E]{E_2, term_{p_1}} p_1'} \tag{2}$$

If S is present (i.e., timeout elapsed), the computation of the operator stops the execution of p and finishes by generating **nothing** as final result and changing $term_p$ to true. The final environment is the output environment E, where the status of the event S_1 becomes true (rule3).

$$\frac{p \xrightarrow[E]{E_1, term_p} p' \quad , \quad S \in E}{\textbf{timeout}(p,\ S,\ p_1) \xrightarrow[E]{E \cup \{S_1^1\}, 1} \textbf{nothing}} \tag{3}$$

The behavioral semantics is a "macro" step semantics that gives the meaning of a reaction for each ADeL instruction. Nevertheless, a reaction is the least fixed point of a "micro" step semantics [6] that computes the output environment from the input one. As mentioned in subsection 2, for each instruction p, each monotonic increasing function from $\mathcal{E}(p)$ to $\mathcal{E}(p)$has a least fixed point which defines the semantics of the program of which p is the root instruction.

More precisely, $p \xrightarrow[E]{E', term} p'$ represents a sequence of micro steps such that:

$$p \xrightarrow[E]{E_1, term_1} p_1, p_1 \xrightarrow[E_1]{E_2, term_2} p_2, \ldots, p_n \xrightarrow[E_n]{E_{n+1}, term_{n+1}} p'$$

and where, at each step, $E_{i+1} = F(E_i)$ (F represents the application of one of the semantic rules to calculate the output environment from the input one). Since the F functions rely on the $\sqcup$ operator on environments, they are monotonic and increasing with respect to the $\preceq$ order. Then $\forall i,\ E_{i+1} \preceq F(E_i)$ and E' is the least fixed point of the F^n function application.

The behavioral semantics is a logical one based on rewriting rules. However, it cannot be really usable to build compilers because it requires the non-trivial computation of fixed points. Nevertheless, this semantics is the reference for the ADeL language and any other semantics must conform to it.

To get an efficient means to compile programs, the authors introduced another semantics based on *constructive* Boolean logic. Hence, this second semantics is also constructive: one can deduce the status of events by propagating the status of input events instead of computing fixed points[6].

4.3 Equational Semantics

Equational semantics allows us to make an incremental compilation of the ADeL programs by translating each root instruction of programs into a ξ-equation system. An equation system is definedasthe4-tuple $<I,O,R,D>$ where I are the input events, O are the output events, $R$are the registers, i.e specific variables acting as memories to record values useful to compute the next instant, and D is the definition of the equation system to calculate the status of each event.

The equational semantics computes the equation system of a program instruction. The authors defines it first for the operators of ADeL and then they extend these definitions to programs. The equational semantics is a function $\mathcal{S}_e$ which calculates an output environment from an input one. Let p be an ADeL instruction and E an input environment. Let us denote $\mathcal{D}(p)$ its equation system and $\langle p \rangle_E$ the resulting output environment, computed by $\mathcal{S}_e$ which is expressed as follows: $\mathcal{S}_e(p, E) = \langle p \rangle_E$ iff $E \vdash \mathcal{D}(p) \hookrightarrow \langle p \rangle_E$. From the event valuation of E, the equation system $\mathcal{D}(p)$ gives the event valuation of $\langle p \rangle_E$. Thanks to theorem 1, ξ-equation systems can be represented as Boolean ones to calculate output events value, and turn back into the 4-valued world to build the resulting environment. So, $\hookrightarrow$ means that $\langle p \rangle_E$ is deduced by applying well known Boolean algebra properties on Boolean equation systems. Then, for an ADeL program and a global input environment E (i.e an environment where output and local events have $\perp$ as status and the registers have 0 as value), the global output environment E' computed by the equational semantics is $\mathcal{S}_e(p, E)$, p being the root instruction of the program. Thus, the equation system of an instruction is deduced from semantic rules expressed for each operators of the language. To define these rules, three specific events are defined for each operator: START to start the instruction, KILL to kill the instruction, and FINISH to send the termination information to the enclosing instruction.

The operator equation systems are defined by operator semantic rules to compute the status of the FINISH, output, and local events, according to the status of START, KILL, input and local events.

As example, here follows the equational semantics of the two operators already considered in section 4.2.

4.3.0.1 Operator parallel

Operator **parallel** unifies (operation $\sqcup$) the output environments of its two operands. The output environment is computed according to the following rule:

$$\langle p_1\rangle_E \sqcup \langle p_2\rangle_E \vdash \mathcal{D}_{p_1\|p_2} \hookrightarrow \langle p_1\|p_2\rangle_E.$$

The rule to define $\mathcal{D}_{p_1\|p_2}$ (see Fig. 1) introduces two registers R_1 and $R_2$to memorize the respective statuses of the FINISH events of the two parallel arguments, since this operator ends when both of its two operands have finished their execution [3]. Note that the operands do not in general terminate in the same instant.

4.3.0.2 Operator timeout.

The output environment of p **timeout** $S\{p_1\}$ **alert** S_1 is calculated as follows:

$$\langle p\rangle_E \vdash \mathcal{D}_{timeout(p,S,p_1,S_1)} \hookrightarrow \langle timeout(p,S,p_1,S_1)\rangle_E.$$

The $\mathcal{D}_{timeout(p,S,p_1,S_1)}$ equation system (see Fig. 2) contains also two registers to record the way this instruction terminates: either with the normal termination of its argument (p) or when the timeout event becomes true. To express the rule for timeout operator, the same rules to denote events as in the previous operator are used.

[3]In operator parallel equation system, the specific signals of the considered operator(here parallel) are denotedSTART, KILL and FINISH while the specific signals of the arguments $p1$ and $p2$ are indexed with the argument respective name.

$$\mathcal{D}_{p_1 \| p_2} = \left[\begin{array}{lcll}
R_1^{+} & = & R_1 \boxdot \neg\mathrm{FINISH}_{p_2} \boxdot \neg\mathrm{KILL} \boxplus \neg R_2 \boxdot & \\
 & & \mathrm{FINISH}_{p_1} \boxdot \neg\mathrm{FINISH}_{p_2} \boxdot \neg\mathrm{KILL} & (1) \\
R_2^{+} & = & R_2 \boxdot \neg\mathrm{FINISH}_{p_1} \boxdot \neg\mathrm{KILL} \boxplus \neg R_1 \boxdot & \\
 & & \neg\mathrm{FINISH}_{p_1} \boxdot \mathrm{FINISH}_{p_2} \boxdot \neg\mathrm{KILL} & (2) \\
\mathrm{START}_{p_1} & = & \mathrm{START} & (3) \\
\mathrm{START}_{p_2} & = & \mathrm{START} & (4) \\
\mathrm{KILL}_{p_1} & = & \mathrm{KILL} & (5) \\
\mathrm{KILL}_{p_2} & = & \mathrm{KILL} & (6) \\
\mathrm{FINISH} & = & R_1 \boxdot \neg R_2 \boxdot \mathrm{FINISH}_{p_2} \boxplus R_2 \boxdot \neg R_1 \boxdot \mathrm{FINISH}_{p_1} \boxplus & \\
 & & \neg R_1 \boxdot \neg R_2 \boxdot \mathrm{FINISH}_{p_1} \boxdot \mathrm{FINISH}_{p_2} & (7)
\end{array} \right]$$

Figure 1.Equational semantics of parallel operator.

$$\mathcal{D}_{timeout(p,S,p_1,S_1)} = \left[\begin{array}{lcll}
R_1^{+} & = & R_1 \boxdot \neg S \boxdot \neg\mathrm{FINISH}_{p} \boxdot \neg\mathrm{KILL} \boxplus & \\
 & & \neg R_1 \boxdot \neg R_2 \boxdot \mathrm{START} \boxdot \neg\mathrm{KILL} & (1) \\
R_2^{+} & = & R_1 \boxdot R_2 \boxdot S \boxdot \neg\mathrm{FINISH}_{p} \boxdot \neg\mathrm{KILL} \boxplus & \\
 & & R_1 \boxdot R_2 \boxdot \neg S \boxdot \mathrm{FINISH}_{p} \boxdot \neg\mathrm{KILL} \boxplus & \\
 & & R_1 \boxdot R_2 \boxdot \neg S \boxdot \neg\mathrm{FINISH}_{p} \boxdot \neg\mathrm{FINISH}_{p_1} \boxdot \neg\mathrm{KILL} \boxplus & \\
 & & R_1 \boxdot \neg R_2 \boxdot \neg S \boxdot \mathrm{FINISH}_{p} \boxdot \neg\mathrm{KILL} \boxplus & \\
 & & \neg R_1 \boxdot R_2 \boxdot \neg\mathrm{FINISH}_{p_1} & (2) \\
\mathrm{START}_{p} & = & \neg R_1 \boxdot \neg R_2 \boxdot \mathrm{START} & (3) \\
\mathrm{START}_{p_1} & = & R_1 \boxdot \neg R_2 \boxdot \neg S \boxdot \mathrm{FINISH}_{p} & (4) \\
\mathrm{KILL}_{p} & = & \mathrm{KILL} & (5) \\
\mathrm{KILL}_{p_1} & = & S \boxdot \neg\mathrm{KILL} \boxdot R_1 \boxplus \mathrm{KILL} & (6) \\
\mathrm{FINISH} & = & R_1 \boxdot \neg R_2 \boxdot S \boxplus \neg R_1 \boxdot R_2 \boxdot \mathrm{FINISH}_{p_1} & (7) \\
S_1 & = & R_1 \boxdot \neg R_2 \boxdot S \boxdot \neg\mathrm{FINISH}_{p} & (8)
\end{array} \right]$$

Figure 2.Equational semantics of timeout operator.

4.4 Relation between Behavioral and Equational Semantics

The behavioral semantics gives a meaning to each program: for each ADeL operator, it formally defines the computation of the output environment and of a Boolean termination flag. The equational semantics, by associating a ξ-equation system to each operator, provides a constructive way to perform the computation. It is important to establish the relation between the solutions obtained by both semantics. To this aim, the following theorem has been proved:

Theorem 2 *Let p be an ADeL instruction, O a set of output events and E an input environment. If* $\langle p \rangle_E$ *is the resulting environment computed by the equational semantics, then the following property holds:* $\exists p'$ *such that* $p \xrightarrow[E]{E', FINISH_P} p'$ *and* $\forall o \in O$, *o has the same status in* $\langle p \rangle_E$ *and* E'.

In short, the theorem means that if the equational semantics yields a solution, there exists also a behavioral solution with the same outputs. It is a proof by induction on the size of a program where the size of an instruction is roughly speaking the number of nodes in its syntax tree. The proof is detailed in[10].

4.5 Compilation and Validation

To compile an ADeL program, our system first transforms it into an equation system which represents the synchronous automaton as explained in section 2. Then it implements directly this equation system, transforming it into a Boolean equation system thanks to the encoding defined in section 2 and to theorem 1. The latter system provides an effective implementation of the initial ADeL program.

Since the equations may not be independent, a valid order (compatible with their inter-dependencies) is needed to be able to generate code for execution (recognition automata), simulation, and verification. Thus an efficient sorting algorithm has been defined [11], using a critical path scheduling approach, which computes all the valid partial orders instead of one unique total order. This facilitates merging several equation systems, hence, an incremental compilation can be performed: an already compiled and sorted code for a sub-activity can be included into a main one, without recompiling the latter.

The internal representation as Boolean equation systems also makes it possible to verify and validate ADeL programs, by generating a format suitable for a dedicated model checker such as our own BLIF CHECK[4]. The same internal representation also allows us to generate code for the off-the-shelf NuSMV model-checker[5].

4.5.0.1 Use Case.

To illustrate our purpose, a small use case in the domain of healthcare is detailed. The goal is to monitor the drug treatment of an Alzheimer person. The activity *medicine To Take* must check that the person is near a table, takes a glass, eats some drugs, and drinks. If the person does not drink before 2 minutes, a *danger* event is sent.

In the graphical format, users declare roles of actors in the declaration window of the graphical tool. Then, they declare sub-activities, and describe the steps of their activity along a ”timelined organigram” (see Fig. 3).

[4]http://www.unice.fr/dgaffe/recherche/outils blif.html
[5]http://nusmv.fbk.eu/

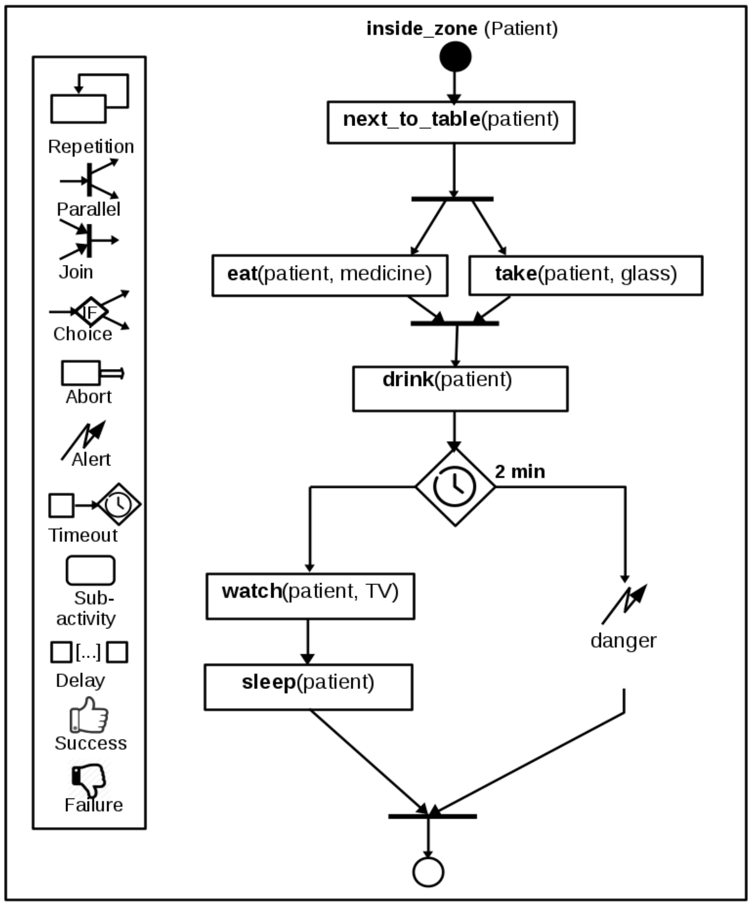

Figure 3. Graphical format of the activity description (organigram)

In the textual format of ADeL, users first declare types of actors. For this use case, there are 2 types: a Zone, a Person, and Equipment. Then, they have to assign roles to actors: in our case, a patient(Person), a medicine, a glass, a TV(Equipment) are needed. The declaration is as follows:

```
Type Person, Equipment;
Activity medicineToTake :
Roles
patient:Person;
medicine:Equipment;
glass: Equipment;
TV:Equipment;
```

After that, users define the name of the activity, its expected events and sub-activities:

```
SubActivities
next_to_table(Person);
take(Person, Equipment);
eat(Person,Equipment);
watch(Person,Equipment);
drink(Person); sleep(Person);
```

Finally, they describe the activity by defining the initial state, and by combining the sub-activities using operators of the language.

```
InitialState:inside_Zone(Patient);
start
{
next_to_table(patient)
then
eat(patient,medicine) parallel take(patient, glass)
then
drink(patient) timeout 2.0 minutes
{    watch(patient,TV)then alert ( danger)
then
sleep(patient)
}
End
```

This code is not correct because the alert should be sent went the timeout is reached. To prove that the alert works correctly, the *medicine To Take* activity is compiled and the input code for the NuSMVmodel-checkerisgenerated.ThentheLTLtemporalpropertycanbechecked: if *danger* is true then *cond_timeout_2_minutes* must have been previously true ("*danger* $\Rightarrow$*O cond_timeout_2_minutes*")[6].

The property is false and a generated counter example allows to fix the problem. Hence, the correct body of the program is:

```
start
{
next_to_table(patient)
then
eat(patient,medicine) parallel take(patient, glass)
then
drink(patient) timeout 2.0 minutes
{
watch(patient,TV) then sleep(patient)
}alert(danger)
end
```

Now the property holds.

5. RELATED WORK

Synchronous languages such as Esterel [1] are meant to describe reactive systems in general and thus can be used to describe human activities. These languages and ADeL use a *logical* time which means that the recognition is performed only when something meaningful occurs. Although their syntax is rather simple, their large spectrum makes them difficult to master by

[6]*cond_timeout_2_minutes* is a Boolean variable true when the timeout is over

some end users. Being dedicated to activity description, a language like ADeL appears more ”natural” for its end users. All these synchronous languages have been given formal semantics. For instance, Esterel has several semantics, with different purposes. In particular, one of these semantics provides a direct implementation under the form of “circuits”. ADeL adopts a similar approach but it simplifies some operators whose semantics in Esterel is complex.

Message Sequence charts [12, 13], which are now introduced in UML, and Live Sequence Charts [14] are also specification languages for activities with a graphical layout that immediately gives an intuitive understanding of the intended system behavior. These languages may be given formal semantics liable to analysis. Message Sequence Charts (MSC) graphically represent the messages exchanged among the actors along time. It is possible to model a complex activity involving several different activities (i.e., MSCs) using High-level Message Sequence Charts (HMSCs). The HMSCs support also parallel composition. The MSC operators and the hierarchical composition of HMSCs are similar to our approach. However, [15] reveals “pathologies” in MSCs, due to defective MSC specifications. These pathologies mainly affect synchronization issues. For instance races may arise from discrepancies between the order of graphical description and system causalities. In our case, since the Synchronous Paradigm is meant to avoid these synchronization problems, race conditions are detected and the program is rejected at compile time. Another pathology comes from possible ambiguous choices between events. In the Synchronous approach, this kind of problems is avoided by producing deterministic systems, in particular mastering event simultaneity. MSCs address the pathology problems by using model checking and formal verification. In [13], the authors illustrate problems of the MSCs models verification for synchronous and asynchronous interpretations and suggest different techniques to fix these model checking problems in several kinds of MSCs representations. In our case, even though model checkers may be interfaced, it is not mandatory. Indeed, most of these pathologies are compile-time checked.

Live Sequence Charts (LSCs) [14] is another activity-based specification and modeling language. It is an extension of MSCs, more expressive and semantically richer. Similarly to ADeL, LSCs are used to specify the behavior of either sequential or parallel systems. They have a formal semantics and can be transformed to automataas ADeL. This allows analysis, verification, and testing using depth-first search methods. Model checking of LSCs is possible by translating them into temporal logic, but the size of the resulting formula, even for simple LSCs, makes it difficult. However, [16] proposes a more efficient translation, but only for a class of LSCs.

Many works in video understanding address the difficult task of extracting semantically significant objects and events from sequences of pixel-based images. A good survey of the corresponding techniques is presented in Lavee and all [17]. These techniques are based on well-founded mathematical methods such as hidden Markov models, (dynamic) Bayesian networks, finite state machines, Petri nets, constraint satisfaction, etc. The authors rely on tool based on such techniques to obtain reliable input events. These approaches allow a form of activity recognition (namely “composite events”) but ADeL addresses more complex activities with longer duration and involving variants, parallel behaviors and multiple actors. Moreover it loses the dependency on video sensors and proposes a more generic approach.

In [18], authors propose a natural and intuitive language to describe activity models using actors, sub activities, and a set of constraints. They also introduce a temporal constraint resolution techniques to recognize activities in real time. This approach is only dedicated to recognize activities using video interpretation, while the authors in this work aims to develop a generic approach that can be used in a large range of domains, by accepting basic events that can come not only from video interpretations but also from other sensors. On the other hand, as authors are working with video interpretation in real time, they can receive the same events (the same image frame) for a long lapse of time without any changements which makes the system awake and working for nothing. With the synchronous approach, the notion of logical time makes the system work only when it receives a significant event.

Researchers in [19] work in activity recognition in smart houses to provide Activities of Daily Living (ADL) and Instrumental Activities of Daily Living (IADL) assistance for their users. They have developed a generic conceptual activity model which allows the modeling of simple and composite activities. To this aim, they propose an hybrid approach which combines ontological formalisms, which describes the link between the activities and their entities, and temporal knowledge representations which specify the relationships between sub-activities that form the composite activity. Then, they encode their characteristics and forms. In our case, ontologies are not used, the ADeL language has only semantics which help to generate the needed activity model to recognize simple and complex activities. Actually, a basic activity can be represented as an event or a simple activity. Activity models for complex/composite activities can be created by composing the sub-activity models which constitute them.

6. CONCLUSION AND PERSPECTIVES

This paper presents a formalization of a synchronous approach to describe (human) activities and to generate a computer recognition system. The Synchronous Paradigm offers several advantages in terms of expression power, ease of implementation, verification through model checking, etc. The authors endowed their own activity description language (ADeL) with two complementary formal semantics, one to describe the abstract behavior of a program, the second to compile the program into an automaton described as an equation system. They proved a theorem which establishes a consistency relation between these two semantics.

The first tests show that the current code that ADeL generates, basically composed of Boolean equations, is easy to integrate in a recognition system, produces compact code, and is efficient at run time. There remains a fundamental issue, common to all synchronous approaches: at the sensor level, the events are asynchronous and they must be sampled to constitute input environments and to define the synchronous "instants". No exact solution is available; several strategies and heuristics have been already tested but large scale experiments are still necessary.
Based on formal foundations, work remains to be done to complete a full framework to generate generic recognition systems and automatic tools to interface with static and dynamic analysis tools, such as model checkers or performance monitors.

REFERENCES

[1] N. Halbwachs. Synchronous Programming of Reactive Systems. Kluwer Academic,1993.

[2] Ge´rard Berry. Mechanized reasoning and hardware design. chapter Esterel on Hardware, pages 87–104. Prentice-Hall, Inc., Upper Saddle River, NJ, USA,1992.

[3] Esterel Technologies. Scade suite. http://www.ansys.com/products/embedded- software/ansys-scade-suite.

[4] G. H. Mealy. A method for synthesizing sequential circuits. Bell Sys. Tech. Journal, 34:1045– 1080, September1955.

[5] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M.Tofte, editors, Proof, Language, and Interaction, Essays in Honor of Robin Milner.MIT Press, 2000.

[6] G. Berry. The Constructive Semantics of Pure Esterel. Draft Book, available at: http://www.esterel-technologies.com 1996.

[7] Matthew Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. Computational Intelligence, 4:265–316,1988.

[8] Daniel Gaffe´ and Annie Ressouche. Algebraic Framework for Synchronous Language Semantics. In Laviana Ferariu and Alina Patelli, editors, 2013 Symposium on Theoretical Aspects of Sofware Engineering, pages 51–58, Birmingham, UK, July 2013. IEEE Computer Society.

[9] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5(2):285–309,1955.

[10] Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, and Daniel Gaffe´. Synchronous Automata For Activity Recognition. Research report, Inria Sophia Antipolis, April 2017.

[11] Annie Ressouche and Daniel Gaffe´. Compilation modulaire d'un langage synchrone. Revue des sciences et technologies de l'information, se´rie The´orie et Science Informatique, 4(30):441–471, June2011.

[12] Thomas Gazagnaire, Blaise Genest ,Lo¨ıc He´loue¨t, P .S. Thiagarajan, Shaofa Yang, and Vasco T. Vasconcelos. Causal Message Sequence Charts, pages 166–180. Springer Berlin Heidelberg, Berlin, Heidelberg,2007.

[13] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts, pages 114–129. Berlin, Heidelberg,1999.

[14] L. Li, H. Gao, and T. Shan. An executable model and testing for Web software based on Live Sequence Charts. In 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), pages 1–6, June2016.

[15] Haitao Dan, Robert M. Hierons, and Steve Counsell. A framework for pathologies of Mes- sage Sequence Charts. Inf. Softw. Technol., 54(11):1283–1295, nov2012.

[16] Rahul Kumar, Eric G. Mercer, and Annette Bunker. Improving translation of Live Sequence Charts to temporal logic. Electron. Notes Theor. Comput. Sci., 250(1):137–152, September 2009.

[17] G. Lavee, E. Rivlin, and M. Rudzsky. Understanding video events: A survey of methods for automatic interpretation of semantic occurrences in video. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 39(5):489–504, Sept 2009.

[18] Van-ThinhVu, Francois Bremond, and Monique Thonnat. Automatic video interpretation: A novel algorithm for temporal scenario recognition. In Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI'03, pages 1295–1300, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[19] George Okeyo, Liming Chen, and Hui Wang. Combining ontological and temporal formalisms for composite activity modelling and recognition in smart homes. Future Generation Computer Systems, 39:29 – 43, 2014. Special Issue on Ubiquitous Computing and Future Communication Systems.