A Synchronous Approach to Activity Recognition

Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, Daniel Gaffé

A Synchronous Approach to Activity Recognition

Ines Sarray*, Annie Ressouche*, Sabine Moisan*, Jean-Paul Rigault*, and Daniel Gaffé**

*Université Côte d'Azur, INRIA, Sophia Antipolis, France (first.last@inria.fr)
**Université Côte d'Azur, CNRS, LEAT, Sophia Antipolis, France
(daniel.gaffe@unice.fr)

Activity Recognition aims at recognizing and understanding sequences of actions and movements of mobile objects (human beings, animals or artefacts), that follow the predefined model of an activity. We propose to describe activities as a series of actions, triggered and driven by environmental events.

Due to the large range of application domains (surveillance, safety, health care...), we propose a *generic* approach to design activity recognition systems that interact continously with their environment and react to its stimuli at run-time.

In our target applications, the data coming from sensors (video-cameras, etc.) are first processed to recognize and track objects and to detect low-level events. This low-level information is collected and transformed into higher level inputs to our activity recognition system.

Such recognition systems must satisfy stringent requirements: dependability, real time, cost effectiveness, security and safety, correctness, completeness... To enforce most of these properties our approach is to base the configuration of the system as well as its execution on formal techniques. We chose the *synchronous approach* which provides formal bases to perform static analysis, verification and validation, but also direct implementation.

Several synchronous languages such as Lustre, Esterel, Scade and Signal [2] have been defined to describe synchronous automata. These languages are for expert users. We propose a new user-oriented language, named ADeL (Activity Description Language) to express activities and to automatically generate recognition automata. This language is easier to understand and to use by non computer scientists (e.g., physicians) while relying on formal semantics.

1 The ADeL language and its semantics

ADeL is a (synchronous) language that allows non-computer scientists to describe activities and behaviors to be recognized. It is a modular and hierarchical language, which means that activities can be simple or composed of one or more sub-activities. ADeL has the notions of (typed) roles, events and sub-activities, flow of control... It supports parallelism, variants (choices), and repetitions and it relies on a set of formally specified control and temporal operators. Some of these operators are "instantaneous" while others take at least one (synchronous) instant to process.

We provide our language with both a graphical and textual format. We propose a graphical tool which displays several windows, mainly to declare the scene where the activity will take place (zones, roles and equipment) and to describe the activity (expected events and "story board"). However, it may be difficult to express complex activities in a purely graphical way, thus we also provide an equivalent textual form.

1.1 Semantics

To provide the language with sound foundations we turn to a formal semantic approach. First, a *behavioral semantics* formally describes the behavior of ADeL programs in a "natural way", using conditional rewriting rules, thus, it gives them a clear interpretation and facilitates their analysis. However, it is not a convenient implementation basis nor is it suitable for proofs (e.g., model-checking). Hence, we also define an *equational semantics* which transforms an ADeL program into a finite state machine represented as a Boolean equation system. The ADeL compiler can easily translate this system into efficient code.

These semantics rely on the notion of *environment* which is a finite set of events. Environments memorize the status of their events in each instant. The goal of both semantics is to compute the status of output events from input event status. We use a 4-valued algebra ($\xi = \{\bot, 0, 1, \top\}$) to represent the status of events, in order to get information about status of events (0 and 1 representing absent and present status; $\bot$ and $\top$ being associated with undefined and overdefined). We defined two orders in ξ: a knowledge order ($\leq_K$) to represent how the information about event status grows and a Boolean order ($\leq_B$) which naturally extends the usual Boolean order. Then, the structures (ξ, $\leq_K$) and (ξ, $\leq_B$) are lattices and their respective meet and join operations allow us to determine event status. Moreover, each element of ξ can be encoded as pairs of Booleans in a bijective way. A complete description is detailed in [5].

1.1.1 Behavioral Semantics

This semantics formalizes each reaction of a program by formally computing the output environment from the input one. It defines a set of rewriting rules of the form:

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

where p and p' are two ADeL instructions, 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. We compute an output environment from an input one by applying hierarchically these rules on the instruction root of a program.

1.1.2 Equational Semantics

The equational semantics allows us to make an incremental compilation of ADeL programs by translating each root instruction of programs into a ξ-equation system. A ξ-equation system is defined as the 4-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.

First, we defined the semantics for the operators of ADeL and then we extended 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, we denote $\mathcal{D}(p)$ its equation system and $\langle p \rangle_E$ the resulting output environment. We have $\mathcal{S}_e(p, E) = \langle p \rangle_E$ 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 $\langle p \rangle_E$. According to the bijective encoding of ξ elements into Boolean pairs, we can represent environments and ξ-equation systems as Boolean ones, then we can calculate output event values in the Boolean world and finally go back to the 4-valued world. The $\hookrightarrow$ notation means that $\langle p \rangle_E$ is deduced by applying well known Boolean algebra properties on Boolean equation systems.

1.1.3 Relation between Behavioral and Equational Semantics

Since we have two different semantics, it is important to establish the relation between the solutions obtained by both semantics. To this aim, we proved that the execution of a program based on the equational semantics also conforms to the behavioral semantics: if the equational semantics yields a solution, there exists also a behavioral solution with the same outputs:

Theorem 1 *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 $p \xrightarrow[E]{E', \text{FINISH}_p} p'$ and
$\forall o \in O$, o has the same status in $\langle p \rangle_E$ and E'

For more details see [5].

1.2 Compilation and Validation

To compile an ADeL program, we first transform it into an equation system which represents its synchronous automaton. Then we directly implement this equation system, transforming it into a Boolean equation system. The latter system provides an effective implementation of the initial ADeL program.

Since the equations may not be independent, we need to find a valid order (compatible with their inter-dependencies) to be able to generate code for execution, simulation, and verification. Thus we defined an efficient sorting algorithm [4], 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 and we can perform incremental compilation: we can include an already compiled and sorted code for a sub-activity 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 the off-the-shelf NuSMV model-checker[1].

2 Related work

We are in the line of synchronous models and languages. However, ADeL is dedicated to non computer scientist end users and applied to a domain not usually targeted by synchronous approaches, namely human activity recognition. Other models such as Message Sequence Chart (MCS) [1] or Live Sequence Chart (LSC)[3] can also be used to describe activities. These languages may be given formal semantics liable to analysis. However, their formal verification, especially with model-checking, may be delicate.

3 Results

Our first experiments showed that the code generated by the ADeL compiler, basically composed of Boolean equations, is easy to integrate in a recognition system, produces compact code, and is efficient at run time. We are currently validating the ADeL graphical tool with doctors to describe medical activities.

[1] `http://nusmv.fbk.eu/`

References

[1] Haitao Dan, Robert M. Hierons, and Steve Counsell. A framework for pathologies of Message Sequence Charts. *Inf. Softw. Technol.*, 54(11):1283–1295, nov 2012.

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

[3] 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.

[4] Annie Ressouche and Daniel Gaffé. Compilation modulaire d'un langage synchrone. *Revue des sciences et technologies de l'information, série Théorie et Science Informatique*, 4(30):441–471, June 2011.

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