Handling Complex Events in Surveillance Tasks

Daniele Bartocci and Marco Ferretti

Department of Computer Engineering and Systems Science
University of Pavia, Italy
daniele.bartocci@consorzio-cini.it, marco.ferretti@unipv.it

Abstract. In this paper we fully develop a fall detection application that focuses on *complex event* detection. We use a decoupled approach, whereby the definition of events and of their complexity is fully detached from low and intermediate image processing level. We focus on context independence and flexibility to allow the reuse of existing approaches on recognition task. We build on existing proposals based on domain knowledge representation through ontologies. We encode knowledge at the rule level, thus providing a more flexible way to handle complexity of events involving more actors and rich time relationships. We obtained positive results from an experimental dataset of 22 recordings, including simple and complex fall events.

Keywords: fall detection, complex event, rule engine.

1 Introduction

The automatic detection of events in video stream is one of the most promising applications of computer vision. Its application field includes surveillance, monitoring, human-computer interaction and content-based video annotation.

Event detection systems usually follow these major steps [1]: extraction of low-level features, recognition of actions or primitive events, high-level semantic interpretation of the events occurring.

Over the past decade many approaches have been proposed to perform the first two tasks (see [2] for a survey) which are often context dependent. Research on the latter stage of event detection is instead leaning towards description-based approaches to ensure independence from the application domain.

Recent works on this topic suggest the use of ontologies and rule languages to detach domain knowledge from the application code. The VERL language [3] has been proposed to define events, but it lacks a full implementation.

Snidaro et al. [4] used the standardized Ontology Web Language (OWL) and Semantic Web Rule Language (SWRL) to represent and maintain domain knowledge in surveillance applications. They further explored this approach [5] by replacing SWRL with a dedicated rule language, Jess. Their experimental architecture allows to infer events given an input of primitive events, and to define the system behavior when an event triggers.

We designed our framework for event detection on top of a similar architecture, which decouples image processing from reasoning. We also adopted their domain

G. Maino and G.L. Foresti (Eds.): ICIAP 2011, Part II, LNCS 6979, pp. 99–108, 2011.

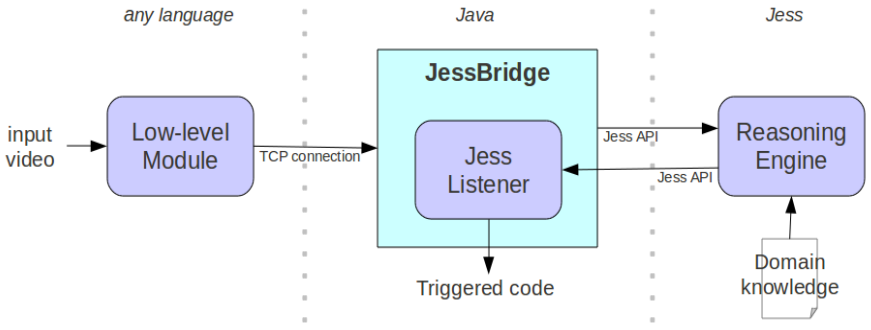

Fig. 1. Architecture overview

knowledge organization but propose a different approach in encoding to better address the definition and detection of *complex events*. Our goal is framework evaluation in terms of flexibility and complex event handling.

In order to achieve this goal we developed a complete application instance in the fall detection application field, tailoring a promising approach by Rougier et al. [6] to suit our framework. Potential falls are detected through Motion History Image (MHI) and shape analysis; the reasoning module distinguishes confirmed falls from false alarms and identifies *complex collective* falls.

The rest of the paper is organized as follows. The framework architecture is discussed in the next section. Section 3 covers the encoding of domain knowledge. The test application is presented, along with experimental results, in sections 4 and 5. Finally, conclusions are drawn in section 6.

2 Framework Architecture

We designed the framework architecture to efficiently implement an event detection system that separates action recognition from domain knowledge and reasoning. This allows for easy reuse of previous approaches on recognition task, while using a dedicated rule engine to carry out the reasoning task.

We adopted the Jess rule engine, a complete environment with its own scripting language to define and evaluate rules. Jess can act as a standalone application but it also exposes a powerful API to embed the rule engine into any Java application.

These features make it suitable for a simple yet effective client/server architecture, based on TCP connections. The client (low-level module) implements all signal processing routines needed to recognize actions or primitive events from the input video stream, sending the results to the server.

The server is the reasoning module; it acts as an interface to the rule engine, feeding it with the messages received from the client and behaving accordingly to the events detected.

Figure 1 shows the overall framework architecture. The core component, called *JessBridge*, is written in Java language and fills the server role. Upon initialization it invokes a Jess engine, loading the domain knowledge from a configuration file.

When a connection is established with the client a buffer is used to store and forward incoming messages to Jess, which executes them. We assume that each row is a valid Jess command (stating an entity or action detected by the low-level module) or a system command (ex: connection break, engine re-initialization).

A separate Java class, which extends the *JessListener* interface (from Jess APIs), defines the system behavior. This class is the handler associated to the rule engine, and its methods will be invoked when an event is inferred or specific rules fire. Thanks to this handler we can execute any kind of triggered code: database updates, automated phone calls, activation of an alarm.

In terms of flexibility, the proposed architecture is platform independent and the whole reasoning task can be defined simply modifying the domain knowledge file and the listener class.

The architecture can be expanded by connecting many clients to the same rule engine using a separate thread for each TCP connection. This also allows the system to interact with other security devices such as smartcard readers and biometric sensors.

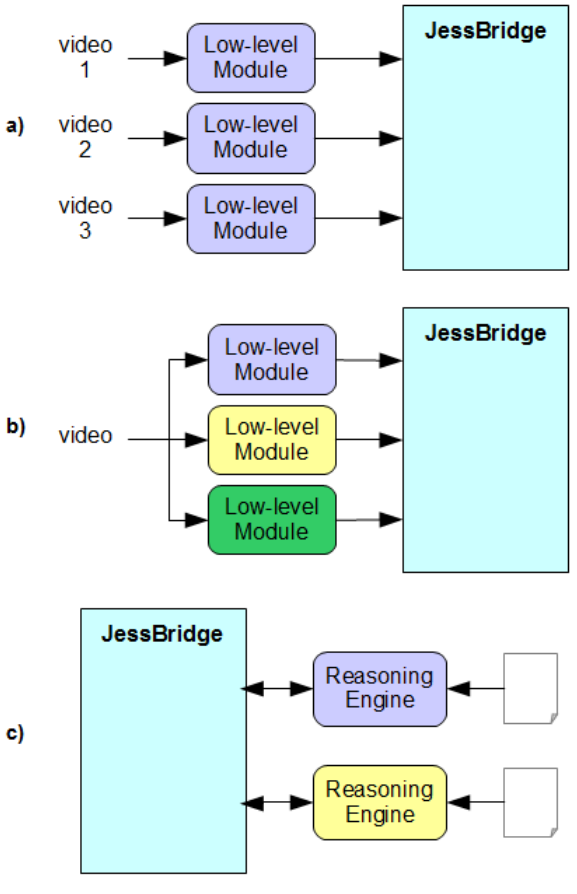

Fig. 2. Expanded architecture examples; *a)* multi-camera environment; *b)* different action classifiers; *c)* multiple rule engines

As shown in figure 2 each client may consist of the same low-level module connected to a different video stream (a common multi-camera environment). This way the reasoning module would be capable of inferring complex events based on simple events occurring at different locations. Otherwise the architecture may comprehend two or more dedicated low-level modules on the same stream, such as different action classifiers.

It is remarkable that the same server can handle more than one Jess instances, each with its very own domain knowledge. Such an architecture may be used for automatic rule learning and ontology evaluation.

3 Embedding Domain Knowledge

Domain knowledge is a key part of the event detection system. It ranges from the different types of entities that can be recognized to the relations and interactions that could take place between them. In our approach it also reflects in how the low-level module encodes its output. As discussed in [5], the knowledge needed by the reasoning engine can be split into the following categories, which differ in purpose and usage:

Static knowledge: taxonomic definitions of event detection's concepts (entities, actions and events). It may include some static instance of entities describing the passive environment such as: door, ATM, etc.
Dynamic knowledge: set of entities, actions and events detected or inferred by the system, each represented by a single instance of a concept in the static knowledge.
Rule knowledge: set of rules that define both simple and complex events for the application context.

Besides dynamic knowledge, contained in the rule engine working memory during the application execution, all definitions must be encoded (offline) in a proper way and loaded upon initialization. Unlike previous works we decided to encode taxonomical knowledge and rule knowledge in Jess language. Our different approach enables a simple solution to define complex events. Since common ontology languages (like OWL) can be easily translated in Jess language [5] if needed, this choice doesn't harm our flexibility goal.

In Jess terminology an instance is a *fact*, which is added dynamically to the working memory through an *assert* command. Each concept in the static knowledge is defined by a fact template with a list of *slot*s to express concept's properties; static instances can be added with the *deffacts* construct. The following example shows our basic template for the event concept:

```
(deftemplate event
    (slot type)
    (slot ID (default-dynamic (gensym*)))
    (multislot subjects)
    (slot t)      ;time
)
```

In this template, the `type` property is an instance of an event category, which can be defined as an enumeration or can be drawn from an ontology. The slot `ID` is system assigned when a fact is asserted in memory. This template can be used for *both* simple and complex events. The `multislot` property allows to associate to the event either a single or a list of values for event's `subjects`; this Jess feature is the key to handle *complex events* with a non-fixed number of subjects such as collective events. The `time` property is handled accordingly to the event category: in simple events (one subject only), it is the time at which the event is instantiated, in complex ones it can be suitably defined. In the following sections an example will be provided.

The *defrule* construct is used to specify the rule knowledge. Each rule has two parts: a pattern which is used to match facts in the working memory and the action to execute when the rule is triggered. Rules are usually composed of logical expressions that specify temporal and spatial constraints between entities and actions (or events). Instances in working memory are matched to verify these constraints. The activation of a simple rule can assert, modify or retract (delete from working memory) a fact concerning an entity or event. Examples of the *defrule* construct will be given and duly commented in section 4.2.

An appropriate flushing strategy has to be included in the rule set in order to manage instances' life cycles.

4 Test Application

To test our framework we developed a simple event detection system based on the proposed architecture. The steps required to deploy the complete application are limited to: low-level module implementation, specific domain knowledge design, definition of the listener class specifying the system behavior (in our test simply log messages). The application detects falls using an approach tailored on the work by Rougier et al. [6], as described in the next subsection. False alarms are filtered out by the reasoning module. Additionally the rule engine can identify and track collective falls.

4.1 Low-Level Module

Action recognition is achieved through the analysis of a mix of image features: a motion coefficient and the orientation and proportion of human shape. This module is implemented in C language using the OpenCV library.

First of all we need to identify moving people in the sequence. For this purpose the image is segmented with the background subtraction method described in [7], then each extracted blob is tracked between subsequent images using a predictor based on a Kalman filter, and processed individually.

We quantify motion using a coefficient based on the Motion History Image (MHI) H_τ of the blob, computed over 500ms:

$$C_{motion}(t) = \frac{\sum_{\forall (x,y) \in blob} H_\tau(x,y,t)}{\#\, pixel \in blob}\,. \tag{1}$$

Since motion is large when a fall occurs we filter out minor movements by thresholding C_{motion}. If a large motion is detected ($C_{motion} > 65\%$) we further analyze the image to discriminate a fall from other activities such as running. The blob is approximated by an ellipse using moments [8] calculating its orientation θ and semi-axis ratio ρ. We consider that a large motion is a fall if the standard deviation of one of these two features, computed for a duration of 1s, is over a fixed threshold (15 degrees on orientation and 0.9 on aspect ratio).

When a potential fall is detected a command is sent to the reasoning module to assert an *action:fall* fact. Furthermore the low-level module periodically notifies Jess whether the blob is moving (*action:move*), checking the ellipse's center coordinates.

4.2 Reasoning Module

In [6] false alarms and negligible falls are filtered out checking if the person is motionless on the ground after a possible fall. We detached this task from the image processing routines and implemented it in our reasoning module. In order to do so we refined the static domain knowledge to fit the application context.

The sole entity involved is the *agent*, a person who can perform the following actions: *move* or *fall*. The defined events include the simple event *confirmed-fall* and *recovery*, plus a complex *collective-fall* event. A *confirmed-fall* is an event that includes a fall, whereby the person involved remains still at ground, while a *recovery* event is a fall followed by a movement of the fallen subject, that actually recovers and moves away.

We provide a rule (depicted in figure 3) to infer an *event:recovery* when a movement is detected within 5 seconds after a fall – in Jess terms an *action:fall* and an *action:move* fact, with the same subject and appropriate time values, are found in the working memory. Likewise the complementary event (*confirmed-fall*) is inferred when no movement is detected.

We also used static instances to define inactivity zones, areas in which a fall is legitimate such as a bed or a couch. A dedicated rule with a higher priority than the previous two filters out those falls.

Lastly a complex event, *collective-fall*, is defined as two or more confirmed falls occurring in a short time span. Two rules are required to detect a collective event properly. The first asserts the event when the minimum constraints are satisfied – in our example two confirmed falls with different subjects and close time values (less than 6 seconds between falls), as expressed by the first two constraints of the rule:

```
(defrule collective_fall_create
    (event (type confirmed-fall)            ;1st constraint
           (subjects ?id1)
           (t ?t1))
    (event (type confirmed-fall)            ;2nd constraint
           (subjects ?id2&~?id1)
           (t ?t2&:(< (abs (- ?t1 ?t2)) 6)))
    (not (event  (type collective-fall)     ;3rd constraint
                 (subjects $? ?id1|?id2 $?)))
    =>
    (assert (event (type collective-fall)   ;rule effects
                   (subjects ?id1 ?id2)
                   (t (time))))
)
```

The two *confirmed-fall* facts hold a single subject value – namely `id1` and `id2` – since they are simple events, while the complex *collective-fall* event asserted as an effect of rule activation will hold the complete list of subjects involved. The third constraint avoids the assertion of a new instance of the complex event if another subject suffers a fall.

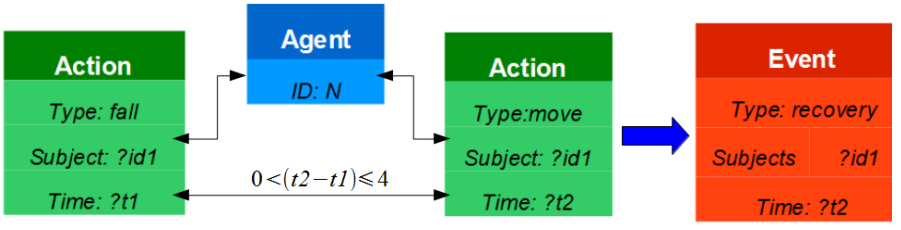

Fig. 3. Representation of the *event:recovery* inference rule

The second rule modifies the *collective-fall* event instance by expanding its subjects list whenever a new *confirmed-fall* event happens within a chosen time interval:

```
(defrule collective_fall_add
       (event (type confirmed-fall)            ;1st constraint
              (subjects ?idNew)
              (t ?tNew))
       ?e <-  (event (type collective-fall)    ;2nd constraint
              (subjects $?list&:(not (member$ ?idNew $?list)))
              (t ?t&:(< (abs (- ?t ?tNew)) 6)))
       =>
       (modify ?e (subjects $?list ?idNew))    ;rule effects
   )
```

The second constraint of the rule specifies that the new fall subject is matched against the *multislot* list to see if it is already in the list; if not, it is added if the temporal constraint is met. In the previous rule we defined the `time` property associated to the complex event as the current system timestamp. This means that the temporal constraint in the second rule will match any *confirmed-fall* asserted during 6 seconds following the *collective-fall* detection.

5 Experimental Results

To evaluate the test application, thereby the framework viability, we wanted to compare our approach on this subject with existing ones, such as [6]. We proceeded to record a dataset representing normal activities and simulated falls, including collective falls (missing in [6]). The application is designed to work with a single uncalibrated camera in a low-cost environment, so our video sequences were acquired using a simple USB webcam. Despite the low quality images (high compression artifacts and noise) we obtained positive results, as will be shown in the sequel.

The dataset (see Table 1) is composed of 22 sequences representing normal activities and simulated falls – harmful and recovered. The number of actors ranges from one (in 12 sequences) to four, exemplifying both simple and complex scenes. Figure 4 shows two snapshots taken from the recorded dataset. Obtained results are shown in table 2.

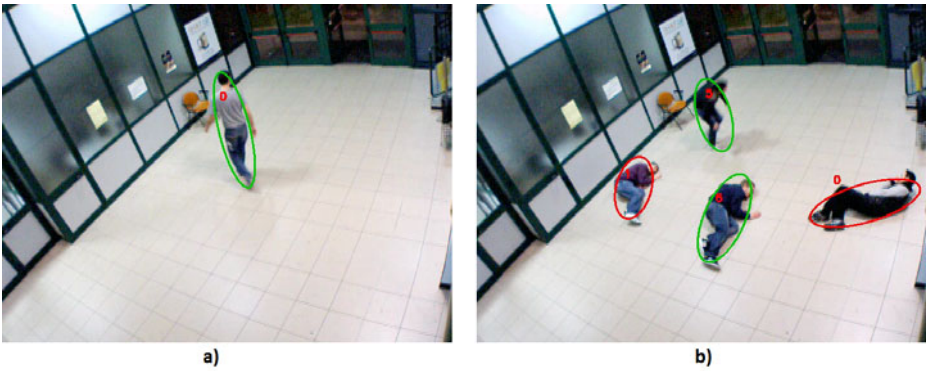

Fig. 4. Excerpts from the dataset; a) simple, single actor scene; b) complex, collective fall. Red ellipses represent already confirmed falls. The third person lying down will soon be tagged as "fallen" too.

Table 1. Dataset sequences

Sequence ID	Name	Agents	# frames	Confirmed falls	Recovery	Other actions
1	simple_fall_1	1	158	1		
2	recovery	1	158		1	
3	simple_fall_2	1	130	1		
4	simple_fall_3	1	113	1		
5	chair_fall_1	1	283	1		1
6	chair_fall_2	1	180	1		1
7	chair_liedown	1	203			2
8	crouching	1	184	1		1
9	liedown_recovery	1	199		1	1
10	pick_up_1	1	176	1		1
11	pick_up_2	1	155		1	2
12	spin_fall	1	223	1	1	
13	wandering	2	178	1		1
14	run_liedown	2	113		1	1
15	indifferent	2	136	1		1
16	run_away	2	81	1		
17	chair_fall_3	3	197	1		
18	collective_1	3	146	2		
19	collective_2	4	171	4		
20	collective_3	4	168	3	1	
21	collective_4	4	162	3	1	
22	hostile	4	171			2

Table 2. Fall detection results

	Detected	**Not detected**
Harmful falls	True positive: **15**	False negative: 2
Lures	False positive: 3	True negative: **19**

We get an overall good rate of event detection and false detection, with a sensitivity of 88.24% (ratio of true positive vs (true positive + false negative)) and a specificity of 86,36% (ratio of true negative vs (false positive + true negative)). Even if our dataset includes many sequences of higher complexity, our results are similar to those obtained by Rougier et al. [6]. We deduce that our decoupled approach, based on a rule engine, is suitable for event detection. Moreover our framework can reuse and expand previous approaches, without worsening the results.

Moreover, the collective fall event was included in 4 video sequences and detected in 3 of them, whenever the *confirmed-fall* events were correctly inferred. Due to complex blob intersections and subsequent tracking errors, the collective event could not be detected in one of those sequences, since only one out of three harmful falls were correctly recognized as *confirmed-fall*.

In summary, results show that we can correctly detect and handle complex events thanks to the rule engine but the higher the constraints complexity, the higher the chance of false negatives because of misclassifications and errors in the low-level module.

6 Conclusions

In this work we have shown an implementation of a complete framework for an event detection system which can handle complex events. We focused on flexibility and context independence. The approach is based on a client/server architecture which decouples action recognition from domain knowledge and reasoning. We deployed domain knowledge with a Jess-based rule engine, which performs the reasoning task.

We also developed a test application in the fall detection application field obtaining preliminary positive results and confirming the correctness of the approach. The experiments carried out emphasize the dependence of the reasoning task on a reliable recognition module. A more robust system could be obtained by strengthening the inference engine, accompanied by a probabilistic output from the recognition module. A possible approach is to extend Jess with fuzzy rule evaluation, thus modeling uncertainty in inferences using probabilistic ontologies.

References

1. Turaga, P., Chellappa, R., Subrahmaniam, V.S., Udrea, O.: Machine Recognition of Human Activities: A survey. IEEE Transactions on Circuits and Systems for Video Technology 18, 1473–1488 (2008)
2. Poppe, R.: A survey on vision-based human action recognition. Image and Vision Computing 28(6), 976–990 (2010)

3. Francois, A.R., Nevatia, R., Hobbs, J., Bolles, R.C., Smith, J.: VERL: an ontology framework for representing and annotating video events. IEEE MultiMedia Magazine 12(4), 76–86 (2005)
4. Snidaro, L., Belluz, M., Foresti, G.L.: Representing and recognizing complex events in surveillance applications. In: Proc. of the 2007 IEEE Conference on Advanced Video and Signal Based Surveillance, pp. 493–498 (2007)
5. Snidaro, L., Belluz, M., Foresti, G.L.: Modelling and Managing Domain Context for Automatic Surveillance Systems. In: Sixth IEEE International Conference on Advanced Video and Signal ased Surveillance (AVSS), pp. 238–243 (2009)
6. Rougier, C., Meunier, J., St-Arnaud, A., Rousseau, J.: Fall Detection from Human Shape and Motion History Using Video Surveillance. In: Proc. of the 21st International Conference on Advanced Information Networking and Applications Workshops, pp. 875–880 (2007)
7. Kim, K., Chalidabhongse, T., Harwood, D., Davis, L.: Real-time foreground-background segmentation using codebook model. Real-Time Imaging 11(3), 172–185 (2005)
8. Chaudhuri, B.B., Samanta, G.P.: Elliptic fit of objects in two and three dimensions by moment of inertia optimization. Pattern Recognition Letters 12(1), 1–7 (1991)