Static Verification of Wireless Sensor Networks with Formal Methods

Alessandro Testa[1,2], Antonio Coronato[1]
(1) ICAR - CNR
Via P. Castellino 111, 80131 Napoli, Italy
alessandro.testa@na.icar.cnr.it,
antonio.coronato@na.icar.cnr.it

Marcello Cinque[2]
(2) DIS
Universita' di Napoli Federico II
Via Claudio 21, 80125 Napoli, Italy
a.testa@unina.it,macinque@unina.it

Juan Carlos Augusto[3]
(3) SEIS, Middlesex University
London, United Kingdom
jcaugusto.work@gmail.com

***Abstract*—Wireless Sensor Networks (WSNs) are widely recognized as a solution to build monitoring systems, even in critical environments. WSNs, however, are subjected to faults due to several causes (i.e. rain, EMF radiations, vibrations, etc..) and tools and methodologies for the design of dependable WSN-based systems are needed. Formal methods partially meet such needs by assessing the degree of correctness of design models and identifying potential system bottlenecks.**
The aim of this paper is to define a methodology for the static verification of WSN based systems using a formal language (*Event Calculus*). In particular we show how the formal specification can be used to verify the design of a WSN in terms of its dependability properties. To this aim, we define a set of correctness specifications that apply to a generic WSN, coupled with specific structural specifications describing the target network topology to evaluate. Finally, after having presented an automatic tool, designed to support the designer, we adopt this methodology to a case study.

***Keywords*-Reasoning, Formal Methods, Testing, Static Verification, Wireless Sensor Network**

I. Introduction

Wireless Sensor Networks (WSNs) are being more and more adopted in critical application scenarios where the level of trust on sensor nodes becomes very important, affecting the success of large-scale industrial applications. Examples of such scenarios are environmental monitoring (e.g. detection of fires in forests), structural monitoring of civil engineering structures [1] and in medical scenarios (e.g. health monitoring) [2], [3].
WSN are dynamic systems, exposed to several threats due to the unreliability of the wireless medium, the limited energy budget, the deployment into harsh environments, and the cheap hardware adopted [4]. Typically a node can fail due to battery exhaustion or due to network partition. If a node fails and stops to operate, it can cause an isolation towards the nodes connected to it.
Exploring all possible consequences of node failures at design time is difficult, since the number of cases to be considered grows exponentially with the number of sensor nodes. However, it is possible to verify the behavior of a given WSN topology statically, i.e., under given circumstances, in order to evaluate the consequences of the failure of a given set of nodes in terms of its impact on the correct functioning of the rest of the network. This type of verification is paramount to pinpoint dependability bottlenecks in the network (i.e., sensor nodes or communication links which are particularly critical for the correct functioning of the network) and to guide the designer on how to modify the WSN configuration in order to meet given design constraints.
In this paper, we propose the use of formal languages to conduct the static verification of WSN-based systems, in terms of their dependability properties. In particular, since WSN nodes behavior can be characterized in terms of an event flow (e.g. a node turns on or a node connects to another node, etc.), we adopt an event-based formal language, namely *Event Calculus*. By means of this formalism we can specify and reason about WSN nodes events and their effects on the whole network.
An open issue with formal specifications of WSNs is that they need to be adapted when changing the target WSN configuration, e.g., in terms of the number of nodes and topology. To address this problem, we conceive two logical sets of specifications: a general specification for WSN correctness properties that is valid for any WSN, and a structural specification related to the topology of the target WSN, designed in order to be generated automatically. To simplify the adoption of the specification and the computation of relevant dependability metrics, we also define a support tool designed to i) automatically generate the structural specifications given the target WSN topology, ii) perform reasoning, by means of an Event Calculus reasoner, starting from the correctness and structural specifications and from an initial event trace (i.e., to verify the WSN under given circumstances), and iii) compute dependability metrics, such as *connection resiliency* and *coverage*, starting from the event trace produced by the reasoner.
The rest of the paper is organized as follows. Section II presents related work, while in Section III we present the specifications of a WSN using Event Calculus; in particular we introduce the Event Calculus formalism and we define the global and structural specifications for a WSN. The proposed support tool is illustrated in Section IV whereas in Section V we describe the adoption of the approach with reference to a case study. Finally, conclusion and future work are provided in Section VI.

II. Related Work

A Wireless Sensor Network is composed of N nodes, each one equipped with a processor, a power supply unit, including batteries, a sensor board with one or more sensors (e.g., light sensor, accelerometer, etc.), a radio board enabling the wireless multi-hop communication. The WSN includes a sink which gathers data from WSN nodes.

Several studies adopt analytical models to assess the performance and dependability of WSNs. These models are based on a mathematical representation of the WSN characteristics and are solved by means of simulation.

In [5] authors define a mathematical model of the energy consumption of nodes to forecast the network lifetime.

Another approach for assessing WSNs makes use of behavioral simulators, i.e., tools able to reproduce the expected behavior of a system by means of a code-based description. Several simulators for WSNs have been proposed in literature, such as [6], [7] and [8].

Behavioral simulators, compared to the analytical models, allow to reproduce the expected behavior of WSN nodes on the basis of the real application planned to execute on nodes. However, it is not always possible to observe non-functional properties of WSNs by means of simulative approaches, since models need to be redefined and adapted to the specific network to simulate. In [9] authors introduce an approach for the automated generation of WSN dependability models, based on a variant of Petri nets.

Recently, different formal methods and tools have been applied for the modeling and analysis of WSNs, such as [10], [11], and [12].

In [10] authors apply a formal tool to wireless sensor networks. Authors propose a formal language to specify the WSN and a tool to simulate it. However, the formal specification has to be rewritten if the WSN under study changes.

In [11] authors address the problem of verifying the correctness of sensor networks through a case study. They do not consider wireless sensors and they use Approximate Probabilistic Model Checker (APMC), a tool that allows to approximately check the correctness of extremely large probabilistic systems, to verify it. Instead, we rely on deterministic checking of correctness properties.

In [12] authors describe a model-driven performance engineering framework for WSNs (called Moppet). This framework uses the Event Calculus formalism to estimate the performance of WSN applications in terms of power consumption and lifetime of each sensor node. Similarly to [12], we propose to use Event Calculus, focusing however not only on lifetime, but mainly on dependability properties, such as resiliency and coverage, and by enabling the automatic generation of the structural specification, which is important to foster the adoption of the approach.

III. WSN Specifications

In this section we introduce the Event Calculus formalism and we define the specification of both WSN correctness properties and WSN structure, using Event Calculus. Finally, we define the metrics considered in this paper.

A. Event Calculus

In the context of the formal methods, *Event Calculus* acquires particular interest [13].

Event Calculus was presented by Robert Kowalski and Marek Sergot in 1986 [14] and it was extended by Murray Shanahan and Rob Miller in the 1990s [15].

It is a logical language for representing and reasoning of the events and their effects [16]. The basic elements of Event Calculus are represented by the concept of fluent, event and predicate [13]. You can specify the value of fluents at some given time points, the events that took place at given time points, and their effects.

In the event calculus, there is a single time line on which actual events occur. A set of actual event occurrences is named *narrative* so for this reason the event calculus is *narrative-based.*

Over the years the event calculus has evolved considerably from its original version adding new predicates. We just report main predicates of Event Calculus: *Initiates*, *Terminates*, *HoldsAt* and *Happens*.

Let us suppose that e is an event, f is a fluent and t is a timepoint.

We have,

$$Initiates(e, f, t)$$

means that, if the event e is executed at time t, then the fluent f will be true after t.

The predicate

$$Terminates(e, f, t)$$

has a similar meaning, with the only difference being that when the event e is executed at time t, then the fluent f will be false after t.

The predicate **HoldsAt** is used to tell which fluents hold at a given time point.

For example,

$$HoldsAt(f, t)$$

means that the fluent f is true at time t.

The predicate **Happens** is used when the event e occurs at timepoint t.

$$Happens(e, t)$$

1) Event Calculus Reasoners: A number of techniques can be used to perform automated reasoning in the event calculus, including logic programming in Prolog, answer set programming, satisfiability solving, and first-order logic automated theorem proving.

Several event calculus reasoning programs have been built that exploit satisfiability (SAT) solvers.
The *Discrete Event Calculus (DEC) Reasoner* [17] [18] uses SAT solvers to perform various types of event calculus reasoning including deduction, abduction, post-diction, and model finding. It was implemented by Erik Mueller [19] and its syntax is explained in [20] (e.g. the meaning of the symbols used in the formulas).
The DEC Reasoner has been used in this work to check the correctness properties written in Event Calculus.

B. Correctness Specifications

We define a set of global specifications in Event Calculus aimed to verify the WSN behavior when undesirable events occur, like a failure of a node (both transient and permanent) or the connection/disconnection of a node from another node or from the whole network. So, we want to check the consequences of these events on the whole network.
For instance, let us consider the figure 1 and let us suppose that node *i* permits the data transmission between the sink node and the subnet *A*. We want to check when the subnet *A* is isolated from the rest of network. We suppose that node *i* is connected with node *j* and *k*. If node *i* fails, the nodes *j* and *k* (and all the nodes of the subnet *A*) are alive but isolated and so the whole subnet *A* is isolated.
More in general, if a subnet depends on a node and this node becomes isolated then all of the nodes of the subnet are isolated.

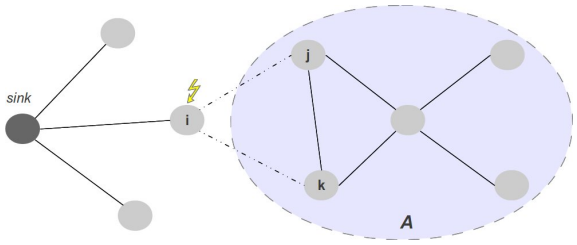

Figure 1: Isolation of a WSN subnet

The correctness specifications are reported in the listing 1. They define the basic set of events that can happen in the WSN, the second level events that can occur as a consequence (e.g., isolation), and the fluents used to specify the correctness properties.

Listing 1: Correctness Specifications of a WSN

```
sort sensor:integer
sort to_sensor:integer
sort from_sensor:integer

predicate Neighbor(sensor,from_sensor)

event Start(sensor)
event Stop(sensor)
event Connect(sensor, to_sensor)
event Disconnect(sensor, from_sensor)

event Join(sensor)
event Isolate(sensor)

fluent IsAlive(sensor)
fluent IsLinked(sensor,from_sensor)
fluent IsReachable(sensor)

[sensor] HoldsAt(IsAlive(sensor),0).
[sensor] HoldsAt(IsReachable(sensor),0).
[sensor,from_sensor] Neighbor(from_sensor,
     sensor) <-> HoldsAt(IsLinked(sensor,
    from_sensor),0).

[time,sensor] Initiates(Start(sensor),
    IsAlive(sensor), time).
[time,sensor] Terminates(Stop(sensor),
    IsAlive(sensor), time).
[sensor,to_sensor,time] Initiates(Connect(
    sensor,to_sensor), IsLinked(sensor,to_
    sensor), time).
[sensor,from_sensor,time] Terminates(
    Disconnect(sensor,from_sensor),
    IsLinked(sensor,from_sensor), time).
[sensor,time] Initiates(Join(sensor),
    IsReachable(sensor),time).
[sensor,time] Terminates(Isolate(sensor),
    IsReachable(sensor),time).

[time] !Happens(Isolate(1),time).
[time] !Happens(Join(1),time).

[time, sensor]
HoldsAt(IsAlive(sensor),time) -> !Happens(
    Start(sensor),time).

[time,sensor]
!HoldsAt(IsAlive(sensor),time) -> !Happens
    (Stop(sensor),time).

[time, sensor]
HoldsAt(IsReachable(sensor),time) -> !
    Happens(Join(sensor),time).

[time,sensor]
!HoldsAt(IsReachable(sensor),time) -> !
    Happens(Isolate(sensor),time).

[sensor,from_sensor, time]
!Neighbor(from_sensor, sensor) -> !Happens
    (Connect(sensor, from_sensor),time) &
    !Happens(Disconnect(sensor, from_
    sensor),time).

[time, sensor, to_sensor]
HoldsAt(IsLinked(sensor, to_sensor),time)
    -> !Happens(Connect(sensor, to_sensor)
    ,time).

[time,sensor, from_sensor]
!HoldsAt(IsLinked(sensor, from_sensor),
    time) -> !Happens(Disconnect(sensor,
```

```
      from_sensor),time).
53
54 [sensor,from_sensor, time]
55 Neighbor(from_sensor,sensor) & HoldsAt(
      IsReachable(sensor),time) & HoldsAt(
      IsAlive(sensor),time) & ( !{from_
      sensor2} ( HoldsAt(IsAlive(from_sensor
      2),time) & HoldsAt(IsReachable(from_
      sensor2),time) & HoldsAt(IsLinked(
      sensor,from_sensor2),time)) & Neighbor
      (from_sensor2,sensor) ) ->
56 Happens(Isolate(sensor),time).
57
58 [sensor,from_sensor, time]
59 ( !HoldsAt(IsReachable(sensor),time) &
      HoldsAt(IsAlive(sensor),time) &
      HoldsAt(IsAlive(from_sensor),time) &
      HoldsAt(IsReachable(from_sensor),time)
       ) & HoldsAt(IsLinked(sensor,from_
      sensor),time) & Neighbor(from_sensor,
      sensor) ->
60 Happens(Join(sensor),time).
61
62 [sensor,from_sensor, time]
63 ( (HoldsAt(IsAlive(from_sensor),time) &
      HoldsAt(IsReachable(from_sensor),time)
       & HoldsAt(IsLinked(sensor,from_sensor
      ),time) ) | !HoldsAt(IsReachable(
      sensor),time) | !HoldsAt(IsAlive(
      sensor),time) ) & Neighbor(from_sensor
      ,sensor) ->
64 !Happens(Isolate(sensor),time).
65
66 [sensor,from_sensor,time]
67 ( HoldsAt(IsReachable(sensor),time) | !
      HoldsAt(IsAlive(sensor),time) | !
      HoldsAt(IsLinked(sensor,from_sensor),
      time) | !HoldsAt(IsAlive(from_sensor),
      time) | !HoldsAt(IsReachable(from_
      sensor),time)) & Neighbor(from_sensor,
      sensor) ->
68 !Happens(Join(sensor),time).
```

1) Sorts: In the first part of the specification (lines 1 to 3), we define three types:

- *sensor*
  It is the reference sensor for events and fluents.
- *to_sensor*
  This sort is used for events and fluents with two parameters in input. For instance, we use *to_sensor* in case of connection (i.e. a sensor connects **to** another sensor).
- *from_sensor*
  It has the dual meaning of *to_sensor*.

2) Predicate: In line 5 we pre-declare the predicate *Neighbor(sensor, from_sensor)* to control if there is a connection between two nodes: it means that a *sensor* receives data from a *from_sensor*. In practice, this predicate is declared in the structural specification (see III-C).

3) Events: We distinguish events of first level (lines 7 to 10) from events of second level directly caught from the WSN by means of specific monitors (lines 12-13).
The set of basic events includes:

- *Start(sensor)*
  This event occurs when a sensor switches on.
- *Stop(sensor)*
  This event occurs when a sensor switches off, e.g., due to a failure.
- *Connect(sensor, to_sensor)*
  This event occurs when a sensor connects to another sensor.
- *Disconnect(sensor, from_sensor)*
  This event occurs when a sensor disconnects from another sensor for an unknown reason (i.e. sensor movements or channel fading).

Events of second level are generated by the reasoner on the basis of our specifications and of the sequence of first level events actually occurred.

- *Join(sensor)*
  This event occurs when there is at least a connection between a sensor and one or more sensors. This event is an effect of the *Connect(sensor, to_sensor)* event.
- *Isolate(sensor)*
  This event occurs when a sensor is completely isolated from the network. In case of point-to-point connection, if there is a *Stop(sensor)* or *Disconnect(sensor, from_sensor)* event, then also an *Isolate(sensor)* event occurs. If a sensor has n neighbors, then it will be isolated if and only if all the n neighbors are not alive or isolated or not connected to the sensor.

4) Fluents: The values of the fluents (defined in the lines 15, 16 and 17) are determined by the events in according to the axioms illustrated in the lines 23-28. They are:

- *IsAlive(sensor)*
  It is true when a *Start* event occurs for a sensor; it is false when a *Stop* event occurs (lines 23-24).
- *IsLinked(sensor, to_sensor)*
  It is true when an *Connect* event occur and so a sensor is connected to a sensor from which it receives data; it is false when a *Disconnect* event occurs (lines 25-26).
- *IsReachable(sensor)*
  It is true when a sensor is reachable and there is at least one path between the sensor and the sink sensor; it is false when the sensor is completely isolated from all its neighbor nodes and so from the whole network (lines 27-28).

5) Initial conditions: From line 19 to line 21 we set some initial conditions. The reasoner considers these conditions when it starts the reasoning.
In particular, we state that at the beginning every sensor is

alive, is reachable and is linked with another sensor on the basis of the WSN topology.

6) Definition of the correctness properties: In lines 30-31 we assert that the node 1, being hypothetically the sink node, is never isolated and then it cannot receive Isolate and Connect events.
In lines 33-52 we show conditions in which events generally can or cannot occur. For example if a node is alive, it cannot be started as well as if a node is isolated it cannot occur an *Isolate* event.
In lines 54-68 we show the axioms that determine when defined events specifically happen.
For example in the lines 54-56 we define conditions for having an *Isolate* event. A sensor can be isolated if it is initially reachable from every sensor, alive and, considering each neighbor sensor, there is not connection among them or every neighbor sensor is not reachable.

C. Structural Specification

In this section we define the structurual specifications related to the topology of a WSN. Differently from the specifications described in previous sub-section III-B, these specifications vary on the basis of the structure of the WSN. To this aim, we use the predicate *Neighbor* (pre-declared in the previous sub-section, line 5 of the specification) to specify how nodes are linked in the topology. For instance,

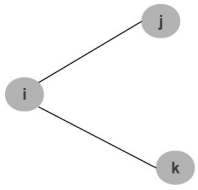

Figure 2: Example of topology of a WSN

considering the topology in figure 2, let us suppose node *i* is connected with *j* and *k* and let us consider a tree graph where sink node (root node) is the node *i* and the nodes *j* and *k* are child nodes. The resulting specification is reported in listing 2, where *sensor1* is the parent node (*i*) and *sensor2* are the child nodes (*j* and *k*).

Listing 2: Example of Neighbor predicate Specification

```
1 [sensor1,sensor2] Neighbor(sensor1,sensor
    2) <-> (
2 (sensor1=i & sensor2=j) |
3 (sensor1=i & sensor2=k)
4 ).
```

The role of the *Neighbor* predicate is very important to understand when an axiom can be applied. Let us examine the axiom related at a possible isolation (lines 54-56 of listing 1) and let us apply it for the figure 2. The described implication is true when, given a couple of nodes (*sensor*, *from_sensor*), the conditions about isolation are true and there is a link between nodes (in this case, between node *j* and *i* or node *k* and *i*). This, for instance, can never be true for the couple of nodes *j* and *k*, since there is not a physical link between them.

D. Initial Event Trace

In order to verify the behavior of the target WSN under given circumstances, we need to indicate an initial event sequence (Event Trace). For example, observing listing 3, by means of *Happens* predicates, we can declare that at time-point 1 sensor j stops, at timepoint 3 sensor k stops, etc. In this way, by means of the reasoner, we can observe the consequences of any initial sequence of events of interest for the designer, e.g., to test the robustness of the designed topology against the temporary unavailability (failure/recovery) of a given set of nodes, or to quantify to what extent the modification of the topology can be beneficial for the network.

Listing 3: Example of initial event trace

```
1 Happens(Stop(j),1).
2 Happens(Stop(k),3).
3 Happens(Start(k),4).
4 Happens(Disconnect(k,i),5).
5
6 completion Happens
```

Finally in the structural specifications we have to include the number of the sensors of the topology and the number of timepoints to select the observation time.

E. Metrics

In this paper we focus on the following dependability metrics, defined in [9]:

- *Connection Resiliency* represents the number of node failures and disconnection events that can be sustained while preserving a given number of nodes connected to the sink.
- *Coverage* is the time interval in which the WSN can operate, while preserving a given number of nodes connected to the sink.

As described in the following, these metrics are evaluated by analyzing the event trace generated by the reasoner, given the target WSN and the initial event trace.

IV. The Support Tool

A Java-based tool has been designed and implemented to simplify the application of the proposed methodology.
By means of a graphical user interface (GUI), shown in figure 3, the user can simply specify i) the topology of the target WSN, in terms of a connectivity matrix (topology section of the GUI), ii) the time horizon to consider, in terms of the number of timepoints (EC section), and iii) the initial event trace, by pressing the *New Event Trace* button in the EC section. Figure 4 shows an example of dialog

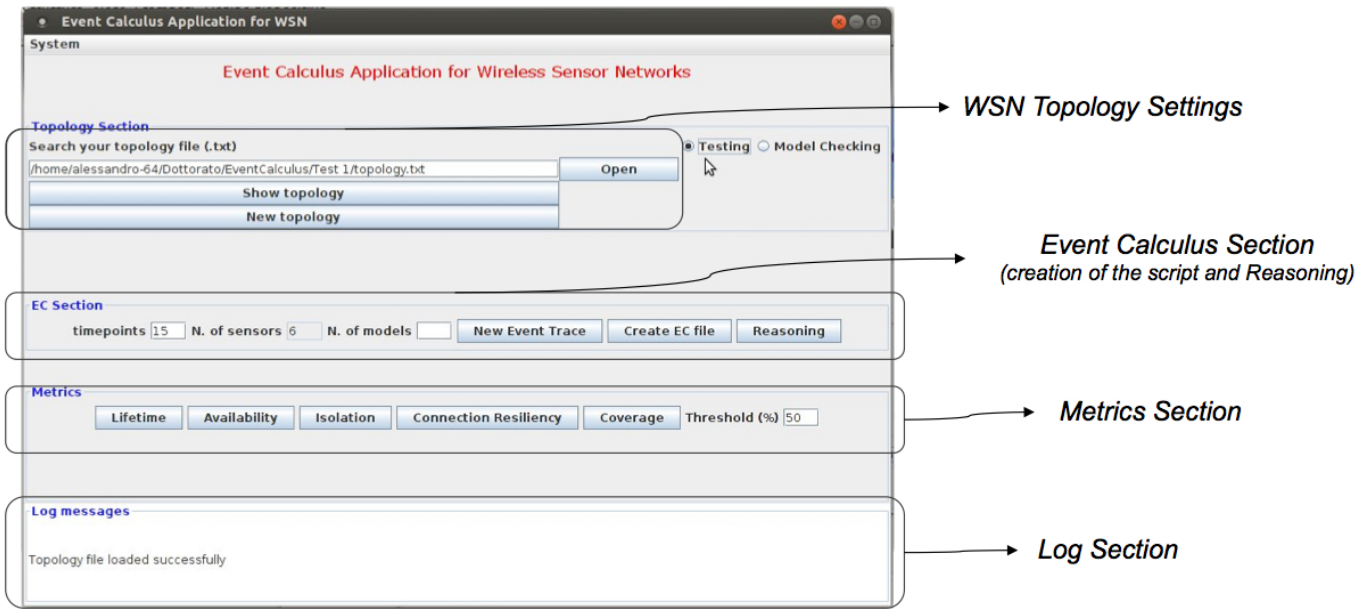

Figure 3: Application GUI

associated with such button, where the user can specify the initial event trace. Hence, the user does not need to know

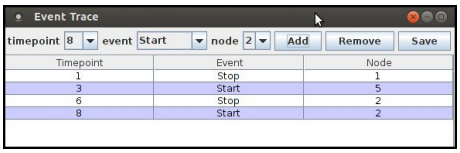

Figure 4: Example of an Initial Event Trace specified by the user

the details about the underlying formalism, which is hidden by the tool.

Once user inputs are inserted, the tool is able to automatically generate the structural specification and the initial event trace in terms of Event Calculus specification files. These files, together with the general correctness specification, are then provided by the tool as inputs to the DECReasoner, which produces the event trace as the result of the reasoning (this happens when the user presses the Reasoning button in the EC section). The obtained trace is finally analyzed by the tool to evaluate the dependability metrics defined in section III-E, namely coverage and connection resiliency (when the user presses the respective buttons in the Metrics section of the GUI). Other metrics, such as availability, lifetime, and isolation, will be treated in next releases of the tool.

A. Metrics computation

By means of a parser that analyzes the trace produced by the DECReasoner, the tool calculates the metrics defined in section III-E.

The computation of *Coverage* and *Connection Resiliency* depends on a threshold parameter, to be indicated as a percentage by the user in the GUI (see the "Threshold" text field in the Metrics section in figure 3). The threshold expresses the fraction of failed and isolated nodes that the user can tolerate, given its design constraints. For instance, over a WSN of 20 nodes, a threshold set to 100% means that all the 20 nodes have to be connected, whereas 50% means that the user can tolerate at most 10 isolated nodes. Considering the threshold value, we calculate the Coverage analyzing the *IsReachable(sensor)* and *IsAlive(sensor)* fluents found in the event trace produced by the reasoner: if a *-IsReachable(x)* or a *-IsAlive(sensor)* fluent is found in the event trace, this means that node x became isolated or it stopped. For example in the case of coverage at 50%, for a WSN with 7 nodes, there is coverage when at least 4 nodes are not isolated (i.e., they are reachable). Hence, as soon as 4 different nodes are no more reachable or alive (looking at the fluents), the network is no more covered. The coverage can be then evaluated as the interval $[0, t]$, being t the time point of the last failure or disconnection event before the isolation (e.g., the time point of the event that caused the isolation of a number of nodes exceeding the threshold).

The Connection Resiliency can then be easily evaluated as the number of failure and disconnection events (namely, *Stop(sensor)* and *Disconnect(sensor, from_sensor)* events) that happen within the coverage interval, excluding the last failure/disconnection event, that is, the one that actually leads the number of isolated nodes to overcome the threshold. For example, if we have coverage in the interval [0,

6], and during this period 3 failure/disconnection events can be counted, than the Connection Resiliency is 2, that is, the WSN was able to tolerate 2 failures or disconnections while preserving more than 50% of the nodes connected.
In the next section we show the computation of these two metrics by considering the event trace generated in the context of a case study.

V. Case Study

Let us consider the wireless body sensor network reported in figure 5 and proposed by Quwaider et al. in [21].
This Wireless Body Sensor Network (WBSN) is constructed

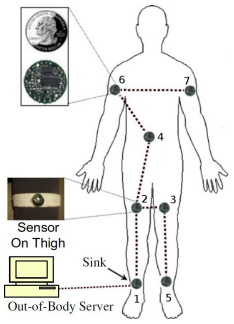

Figure 5: Wireless Body Sensor Network topology

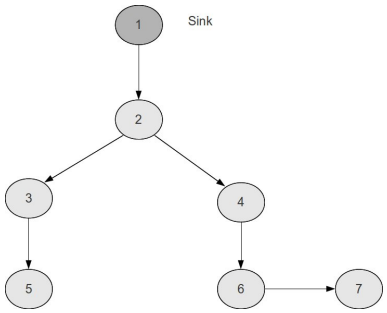

Figure 6: WBSN topology

by mounting seven sensor nodes attached on two ankles, two thighs, two upper-arms and one in the waist area. In figure 6 it is reported the topology of the WBSN, in which the arrows indicate the relationships between couples of nodes (i.e. node 2 is linked with node 1, node 3 and 4 are linked with node 2, etc.). Node 1 is the sink of the network. Each node consists of a 900 MHz Mica2Dot MOTE (running Tiny-OS).
The objective of the case study is to observe the behavior of this WBSN in 10 timepoints, supposing that, after one timepoint, a disconnection between node 5 and node 3 occurs, and that node 4 stops at timepoint 3. In particular, we are interested to evaluate if the reasoning performed by the DECReasoner on our specifications is correct. For a coverage threshold set at 50%, we should observe a coverage interval equals to $[0, 3]$ (i.e., when node 4 stops at time point 3, 4 nodes are not reachable, namely 4, 5, 6 and 7), and a connection resiliency equals to 1 (i.e., only the disconnection event is tolerated).
In the following, we show the specifications generated by the tool and the trace produced by the DECReasoner by providing the WBSN connectivity matrix and the initial event trace as inputs to the tool, as explained in the previous section.

A. Event Calculus Specifications obtained by the tool

1) Structural specification: analyzing the connectivity matrix of the WBSN topology, the tool establishes how the nodes are linked (see Listing 4).

Listing 4: Neighbor predicate definition

```
1 [sensor1,sensor2] Neighbor(sensor1,sensor
      2) <-> (
2 (sensor1=1 & sensor2=2) |
3 (sensor1=2 & sensor2=3) | (sensor1=2 &
      sensor2=4) |
4 (sensor1=3 & sensor2=5) |
5 (sensor1=4 & sensor2=6) |
6 (sensor1=6 & sensor2=7)
```

2) Initial event trace: the initial event trace produced by the tool in our case is reported in Listing 5.

Listing 5: Initial event trace

```
1 Happens(Disconnect(5 , 3),1).
2 Happens(Stop(4),3).
3
4 completion Happens
```

B. Outcome and Metrics computation

Listing 6 reports the outcome (also called the *narrative*) produced by the DECReasoner when invoked by the tool. The event trace confirms our expectations. We can observe that after the stop of node 4, nodes 6 and 7 becomes not reachable. Considering that node 5 was already not reachable, this means that a total of 4 nodes are isolated. The coverage is computed as the time point of the last failure event causing such isolation, that is 3. Consequently, the connection resiliency is computed by counting the number of failure and disconnection events in the interval $[0, 3]$, excluding the last event; hence, it is equal to 1.

Listing 6: Outcome of the DECReasoner

```
0
1
Happens(Disconnect(5, 3), 1).
2
-IsLinked(5, 3).
Happens(Isolate(5), 2).
3
-IsReachable(5).
Happens(Stop(4), 3).
4
-IsAlive(4).
Happens(Isolate(6), 4).
```

```
5
-IsReachable(6).
Happens(Isolate(7), 5).
6
-IsReachable(7).
7
8
9
10
```

VI. Conclusions and Future Work

This paper has described a methodology for the verification of WSN based systems using the Event Calculus. Using Event Calculus formalism we have presented global specifications for a generic WSN that are integrated with structural specifications for a particular topology. A Java-based tool, designed to automate this approach, has been proposed. Finally a WBSN is been considered as case study. Albeit simplistic, the case study has been performed to show the correctness of the trace produced by the DECReasoner when processing our specifications.

As future work we want to extend this approach also to perform *Model Checking* and to compute other typical dependability metrics such as *lifetime*, *availability*, *isolation* (for a generic node). Also, other experimental campaigns have to be performed on more complex topologies. Finally, we think that this methodology is useful to do *Runtime Verification* which allows the system to self assess its dependability level, to detect faults and to recover from failures while running.

References

[1] Marcello Cinque, Domenico Cotroneo, Catello Di Martino, Stefano Russo, and Alessandro Testa. Avr-inject: A tool for injecting faults in wireless sensor nodes. In *IPDPS*, pages 1–8, 2009.

[2] Yang Hao and Robert Foster. Wireless body sensor networks for health-monitoring applications. *Physiological Measurement*, 29(11):R27–R56, November 2008.

[3] M. Paksuniemi, H. Sorvoja, E. Alasaarela, and R. Myllyla. Wireless sensor and data transmission needs and technologies for patient monitoring in the operating room and intensive care unit. *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 5182–5185, 2005.

[4] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 214–226, New York, NY, USA, 2004. ACM.

[5] Jae-Joon Lee, Bhaskar Krishnamachari, and C.-C. Jay Kuo. Impact of energy depletion and reliability on wireless sensor network connectivity. In *In Proceedings of the SPIE Defense and Security*, 2004.

[6] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[7] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[8] The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/.

[9] Catello Di Martino, Marcello Cinque, and Domenico Cotroneo. Automated generation of performance and dependability models for the assessment of wireless sensor networks. *IEEE Trans. Comput.*, 61(6):870–884, June 2012.

[10] P.C. Olveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in real-time maude. *Parallel and Distributed Processing Symposium, International*, 0:157, 2006.

[11] Akim Demaille. Probabilistic verification of sensor networks. In *In Proc. 4th IEEE Int. Conf. on Comput. Sci., Research, Innovation and Vision for the Future (RIVF06*, pages 45–54. IEEE Computer Society, 2006.

[12] Pruet Boonma and Junichi Suzuki. Moppet: A model-driven performance engineering framework for wireless sensor networks. *Comput. J.*, 53(10):1674–1690, 2010.

[13] Murray Shanahan. The Event Calculus Explained. *Lecture Notes in Computer Science*, 1600:409–??, 1999.

[14] R Kowalski and M Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, January 1986.

[15] Rob Miller and Murray Shanahan. Reasoning about discontinuities in the event calculus. In *in Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96*, pages 63–74. Morgan Kaufmann, 1996.

[16] F. Van Harmelen, V. Lifschitz, and B. Porter. *Handbook Of Knowledge Representation*. Foundations of Artificial Intelligence. Elsevier, 2008.

[17] Erik T. Mueller. Event calculus reasoning through satisfiability. *Journal of Logic and Computation*, 14:2004, 2004.

[18] Erik T. Mueller. A tool for satisfiability-based commonsense reasoning in the event calculus. In *FLAIRS Conference'04*, pages –1–1, 2004.

[19] E. Mueller. Decreasoner. *http://decreasoner.sourceforge.net*.

[20] Erik T. Muller. Discrete event calculus reasoner documentation. 2008.

[21] Muhannad Quwaider and Subir Biswas. Dtn routing in body sensor networks with dynamic postural partitioning. *Ad Hoc Netw.*, 8(8):824–841, November 2010.