Formal Analysis of Policies in Wireless Sensor Network Applications

Marco Patrignani Nelson Matthys José Proença Danny Hughes Dave Clarke
IBBT-DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
3001 Leuven, Belgium
name.surname@cs.kuleuven.be

***Abstract*—Since wireless sensor network applications are ever growing in scale and complexity, managers require strong formal guarantees that any changes done to the system can be enacted safely. This paper presents the formalisation and analysis of the semantics of policies, tiny software artefacts used to orchestrate wireless sensor network applications. The semantics of policies is formalised in terms of traces augmented with information concerning the constraints under which traces are executed. These traces are composed according to the network topology and subsequently analysed using the mCRL2 model-checking tool. The analysis allows for the detection of semantical inconsistencies that may lead to dangerous or unwanted behaviour of the application based on the policy configuration. An analysis of policies in a real-world system is provided, showing how to verify security and resource usage properties.**

***Keywords*-Model Checking, Formal Methods, Policy-driven Middleware, Wireless Sensor Network Applications.**

I. Introduction

Wireless Sensor Networks (WSN) are evolving into long-lived infrastructures on which applications developed by different stakeholders are concurrently executed [1]. These applications are typically characterized by various sources of complexity and highly dynamic requirements. In order to cope with such difficulties, lightweight runtime reconfigurable WSN component models [2] and policy-based systems [3] have recently emerged. These approaches promote modularity both at development- and run-time, while supporting dynamic deployment and easing reconfiguration of functionalities.

The Component and Policy Infrastructure (CaPI) [4] is a prototype of WSN middleware featuring the combination of a lightweight component-based runtime and policy-driven management framework. Based on our experiences with CaPI in a number of real-world WSN applications [5], we discovered that errors in the composition of components with policies can compromise the resulting application. While components are used to provide reusable functional blocks, policies provide an abstraction to govern behavioural concerns, such as security or adaptability, separated from functional code. However, in contrast to components, which can safely be composed with other components that match their explicit interfaces, criteria for composing policies involve reasoning about their intended semantics and how they interact with each other. Hence, carelessly applying policies to orchestrate components may introduce application behaviour which is opaque, inefficient, render certain components unreliable for reuse in other compositions, or even compromise the entire WSN [5].

The main features of policies and their implications are summarised below:

- the functionality provided by each policy is specified via a set of domain-specific actions, whose execution is subject to the evaluation of conditional guards;
- the specification of a single policy can be semantically correct on its own, though the composition with other policies may lead to inconsistencies.

Thus, composing policies is error-prone and might lead to conflicts at runtime that are difficult to detect and resolve on a resource-constrained sensor node. Furthermore, the set of deployed policies may change over time, mandating the need for conflict checking prior to commitment of the change. The challenge of conflict checking is significant when considering distribution: a message traveling in the network is subject to all policies it encounters. In fact, a message triggers policies both on the sending node and on the receiving node. Thus, how messages are exchanged in the network is a crucial notion that must be considered in order to understand the semantics of policies across the system. It is by composing the semantics of policies of two different nodes that we can detect distributed semantical conflicts. Policy conflicts can compromise the functioning of the entire application, and since sensor nodes are often hard to reach and replace, there is need for a formal analysis prior to deployment that guarantees that the policy composition behaves as expected. One of the most important benefits of such a formal approach is that problems that can occur at run-time can be detected by analysing the formal model. Due to the resource constraints of sensor nodes and the high cost of communication, the analysis should maximally exploit the capabilities of the back-end, which has greater computational power and fewer limitations than the sensor nodes.

Marco Patrignani holds a Ph.D. fellowship of the Research Foundation Flanders (FWO). The research from Nelson Matthys is conducted in the context of the IWT-SBO-SymbioNets project No. 090062.

This paper makes three key contributions: firstly, and most importantly, it proposes the formalization of policies in CaPI-based WSN applications and gives their semantics in terms of a novel formalisation technique: constrained traces. Constrained traces are traces augmented with information concerning the constraints under which the trace is executed. Secondly, this paper shows how to compose constrained traces in order to obtain the semantics of policies deployed across a network of nodes. Such system-level semantics can be analysed by means of properties whose satisfiability implies the correctness of the application. A tailored set of properties that detect security threats and energy wastage is also presented. Finally, this paper presents preliminary results on the realisation of the aforementioned theory: a tool for the verification of properties of policies in CaPI. This last point provides tangible evidence of the validity of the presented approach, moving the results from a theoretical to a practical point of view.

The remainder of the paper is organised as follows. §II presents the formalisation of the semantics at policy and at system levels, §III introduces the idea behind the analysis conducted on a system of policies. §IV contains details of the prototype implementation that realises the formalization and analysis of a system. §V describes and motivates the assumptions and restrictions of the presented work, §VI discusses related work, §VII presents future work and concludes.

II. System Semantics

This section introduces policies in the CaPI middleware and formalises their semantics at both node and system levels.

Example 1 (Policies): Table I presents two different policies P_1 and P_2 deployed on a computational node N_1. N_1 contains a component that emits temperature messages,

Table I
Example policies deployed on node N_1.

$P_1@N_1$	$P_2@N_1$

`on (T) as e`
`  if (true)`
`  encrypt e;`
	`on (°C) as e`
	`  if (lowBatt())`
	`  deny e;`

which the application must communicate in an encrypted fashion. In order to do so, policy P_1 is deployed: it encrypts all temperature messages (`T`) produced on N_1. To save power, a filtering policy P_2 is deployed: it discards all centigrades messages when the battery is low (assume a built-in function `lowBatt()`). Unfortunately, the deployment of $P_1; P_2$ on node N_1 leads to a waste of resources, since messages will be encrypted and subsequently discarded.

A. Informal Semantics

Figure 1 provides a simplified example of the CaPI middleware in which WSN applications are realised through a combination of components and policies. For a detailed overview of CaPI, the interested reader is referred to [4]. Components exchange information with each other via mes-

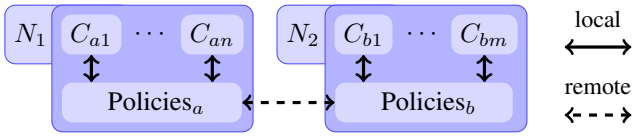

Figure 1. Communication between components and policies.

sages, either *locally* or *remotely*, as defined by the application composition. Local communication occurs within the same computational node, while remote communication occurs between two different nodes. All messages possess a type, e.g., temperature or humidity, and type-specific contents; CaPI-policies specify the type of messages they are applicable to. Message exchange in CaPI is governed by the policies on every node. Thus, every message might trigger a number of policies during its lifetime, on the node where the message is generated, and on the destination node, in case of a remote communication. By calculating all combinations in which policies are triggered by messages in the network we obtain the full system semantics.

The semantics of the system needs to consider all valid sequences of actions performed by policies. Intuitively the semantics of a policy consists of all possible sequences of domain-specific actions that can be executed by that policy. Policies can be seen as a tree-shaped control flow graph. Conditional branches create two possible sequences of actions, based on whether the conditional branch is executed or not. The semantics of composed policies is given by the composition of such graphs. Each root-to-leaf path represents a possible valid sequence of actions executed by various policies. However, paths resulting from inconsistent choices during the branching operations are filtered while calculating the semantics. The semantics of the system is a collection of all root-to-leaf paths: a set of sequences of actions. All sequences of the set are analyzed to verify that the system does not have semantical inconsistencies. If the analysis fails, then a policy may execute undesired behaviour at runtime.

B. Semantics of Policy Programs

We refer to the formalisation of policies that are deployed on a CaPI node as a *policy program*. Policy programs rely on the notions of branches, domain-specific actions and composition operations, which are common to several works related to policies [6], [7].

Figure 2 presents the syntax of policy programs. Assume the presence of a tree-structured taxonomy of message types $\mathcal{T}$, and a countably infinite set of logical predicates $\mathcal{L}$. Message types t are elements of the tree-structured taxonomy $\mathcal{T}$. Guards g are logical predicates: elements of the set $\mathcal{L}$,

Policy Program $P ::=$	$\texttt{on}(t)\{P\}$	*Type* $t \in \mathcal{T}$
	$\mid \texttt{if}(g)\{P\}$	*If-Guard* $g \in \mathcal{L}$
	$\mid a$	*Action* $a \in \mathcal{A}$
	$\mid P; P'$	

Figure 2. Syntax of policy programs related elements.

side-effect free expressions that evaluate to a boolean result. Consider the finite set of actions $\mathcal{A} ::= \{\texttt{allow}, \texttt{deny}, \texttt{encrypt}, \texttt{decrypt}, \texttt{sign}, \texttt{verify}, \texttt{persist}, \texttt{delete}\}$. All elements of $\mathcal{A}$ are actions that policy programs can execute, the formal analysis is focussed on such actions. For the sake of simplicity, parameters have been omitted.

The evaluation of a policy program depends on the type of the message that triggers its execution. Let *msg-type* be such a type. A policy program can be an *on-statement* $\texttt{on}(t)\{P\}$ consisting of an *on-guard* t and a body P. The on-statement checks whether *msg-type* is a subtype of t according to the type hierarchy $\mathcal{T}$, continuing as P when it is. Analogously, a policy program can be a *conditional branch* $\texttt{if}(g)\{P\}$ consisting of a *guard* g and a body P. The if-statement checks g, executing the body P when the guard evaluates to true. Finally, a policy program can be an action $a \in \mathcal{A}$ or the sequential composition of different programs $P; P'$.

Figure 3 presents the syntax elements related to the semantics of both policy programs and of the system. We write $\overline{x}$ as a shorthand for $x_1, \ldots, x_n$ for $n \geq 0$, and ϵ denotes an empty sequence. Constrained traces are pairs

Constrained Trace $\tau ::=$	$(\overline{\ell}; C)$
Label $\ell \in$	$\mathcal{A} \cup \{\lightning\}$
Accumulator $C ::=$	$(\overline{g}, \overline{t})$

Figure 3. Syntax of constrained traces.

consisting of a sequence of labels $\overline{\ell}$ and an accumulator C, representing a set of constraints. Labels ℓ are elements of $\mathcal{A}$ or $\lightning$, indicating respectively the execution of an action or the exchange of a message between two nodes. An accumulator C is a pair of the form $(\overline{g}, \overline{t})$, consisting of a list of logical constraints $\overline{g}$ and a lists of type constraints $\overline{t}$. Logical constraints model which logical statements g must hold for the computation to be executed; type constraints model for which types the computation is applicable. An accumulator C is compatible with a guard g or with a message type t, written $C \frown g$ and $C \frown t$, when it is possible to extend C with g or t without being inconsistent. Since C keeps a record of the choices involved in the previously encountered branches, only traces with consistent choices at on- and if-statements, according to the $\frown$ relation, are considered by the semantics. Together these constraints describe one possible path through the control flow of a policy program.

Accumulator and Guards. Logical expressions g often depend on runtime values in order to determine whether they evaluate to true or false. Since a static analysis has no access to the runtime values computed by the application, such values are abstracted away by means of the predicate abstraction technique [8]. With predicate abstraction, logical expressions that are syntactically equal are assumed to evaluate to the same truth value at runtime. For example, two concatenated policy programs may contain the same expression `(e.val<9)` in their if-guard. When the expression is encountered for the first time, a bifurcation is created, one accumulator storing `(e.val<9)`, the other storing `¬(e.val<9)`, because there is no way to determine how the guard will evaluate at runtime. When the expression is encountered for the second time, no bifurcations are created because an assumption on the evaluation of `(e.val<9)` is stored in the accumulator. If `(e.val<9)` is found in the accumulator, then the predicate is supposed to evaluate to true, if `¬(e.val<9)` is found, then it is supposed to evaluate to false.

Compatibility of guards and accumulator builds on the notion of predicate abstraction. Predicates that may evaluate to true are stored in the accumulator, those that may evaluate to false are stored preceded by a logical negation ($\neg$). A logical expression is compatible with an accumulator if it can be added to the logical expressions that are already in the accumulator and the resulting set is satisfiable. When a new guard g is encountered, both g and $\neg g$ are tested to be compatible with the accumulator. If g is compatible with the accumulator, then it can evaluate to true at runtime. Conversely, if $\neg g$ is compatible with the accumulator, then it can evaluate to false at runtime.[1] Addition of logical expressions to the accumulator is treated standardly in a set-oriented fashion.

Accumulator and Types. Unlike guards, types do not depend on runtime values. The possible types of messages that a node can produce are statically known and given in the node description, as mentioned in §V. This list of types is the initial knowledge of the accumulator, since messages of all possible types produced on a node can flow through the policy programs of such a node. As policy programs are traversed, the list of types contained in the accumulator shrinks, giving a better description of what types of messages trigger the execution of the policy program.

A type t ($\neg t$) is compatible with an accumulator C if C contains at least one type t' that could (not) trigger the execution of the policy program. Intersection of a type t ($\neg t$)

[1]Note that the two cases are not mutually exclusive. For example, knowing that *"the pen is red"*, *"the book is blue"* and *"the book is not blue"* are compatible with the knowledge.

and an accumulator C means reducing C to all the types t' that are (not) children of t in the tree-structured taxonomy $\mathcal{T}$. The formalization of these intuitions is presented in Figure 4.

$$
\begin{aligned}
(\overline{g},\overline{t}) \frown t &\iff \exists t' \in \overline{t} \mid t' \prec t \\
(\overline{g},\overline{t}) \frown \neg t &\iff \exists t' \in \overline{t} \mid t' \not\prec t \\
(\overline{g},\overline{t}) \cap t &= (\overline{g}, \{t' \in \overline{t} \mid t' \prec t\}) \\
(\overline{g},\overline{t}) \cap \neg t &= (\overline{g}, \{t' \in \overline{t} \mid t' \not\prec t\})
\end{aligned}
$$

Figure 4. Compatibility relation and intersection for types.

The formal semantics of policy programs, presented in Figure 5, yields a set of constrained traces given a policy program and an accumulator. At every on- and if-statement

$$
\begin{aligned}
[\![\texttt{on}(t)\{P\}]\!]_C =& \{\tau \mid \tau \in [\![P]\!]_{C \cap \{t\}}, C \frown t\} \cup \\
& \{(\epsilon; C \cap \{\neg t\}) \mid C \frown \neg t\} \\
[\![\texttt{if}(g)\{P\}]\!]_C =& \{\tau \mid \tau \in [\![P]\!]_{C \cup \{g\}}, C \frown g\} \cup \\
& \{(\epsilon; C \cup \{\neg g\}) \mid C \frown \neg g\} \\
[\![a]\!]_C =& \{(a; C)\} \\
[\![P; P']\!]_C =& \{(\overline{a}\,\overline{a}'; C') \mid (\overline{a}; C'') \in [\![P]\!]_C, \\
& \qquad (\overline{a}'; C') \in [\![P']\!]_{C''}\}
\end{aligned}
$$

Figure 5. Semantic function of policy program.

two sets of traces are built, depending on whether the on- and if-guards succeed or fail. This branching is determined by a *compatibility* relation $\frown$, described below. Addition of elements to the accumulator and the compatibility relation are described in detail below. At every action a, the semantics function returns the constrained trace made of a and the current accumulator C. When considering a sequential composition of policy programs $P; P'$ the function returns the set of all possible traces composed as follows. The first part of the trace $\overline{a}$ is the one derived from the semantics function applied to P with the starting accumulator C. The second part of the trace $\overline{a}'$ is the one derived from the semantics function applied to P' with all possible accumulators C' associated with a constrained trace of P.

Example 2 (Policy composition): Figure 6 shows the application of function $[\![\cdot]\!]$ from Figure 5 to the policy program $P_1; P_2$ obtained from Example 1. Syntactic simplifications are applied to P_1 and P_2. The starting accumulator is set to ($^\circ$C,H) in order to model node N_1 producing both centigrade ($^\circ$C) and humidity (H) messages. Notice that for the sake of brevity when a part of the accumulator is empty, it is omitted entirely. The resulting set of traces is exactly the sequence

$$
\begin{aligned}
[\![P_1]\!]_{\circ\texttt{C,H}} =& [\![\texttt{on (T) \{if (true)\{encrypt;\}\}}]\!]_{\circ\texttt{C,H}} \\
=& \{[\![\texttt{if (true)\{encrypt;\}}]\!]_{\circ\texttt{C}}, (\epsilon; \texttt{H})\} \\
=& \{[\![\texttt{encrypt;}]\!]_{\circ\texttt{C}}, (\epsilon; \texttt{H})\} \\
=& \{(\texttt{encrypt;}^\circ\texttt{C}), (\epsilon; \texttt{H})\} \\
[\![P_2]\!]_{\texttt{H}} =& \{\epsilon; \texttt{H}\} \\
[\![P_2]\!]_{\circ\texttt{C}} =& [\![\texttt{on (}^\circ\texttt{C)\{if (lowBatt())\{deny;\}\}}]\!]_{\circ\texttt{C}} \\
=& \{[\![\texttt{if (lowBatt())\{deny;\}}]\!]_{\circ\texttt{C}}\} \\
=& \{[\![\texttt{deny;}]\!]_{\texttt{lowBatt(),}^\circ\texttt{C}}, (\epsilon; \neg\texttt{lowBatt(),}^\circ\texttt{C})\} \\
=& \{(\texttt{deny; lowBatt(),}^\circ\texttt{C}), (\epsilon; \neg\texttt{lowBatt(),}^\circ\texttt{C})\} \\
[\![P_1; P_2]\!]_{\circ\texttt{C,H}} =& \{(\epsilon; \texttt{H}), (\texttt{encrypt; }\neg\texttt{lowBatt(),}^\circ\texttt{C}), \\
& \qquad (\texttt{encrypt deny; lowBatt(),}^\circ\texttt{C})\}
\end{aligned}
$$

Figure 6. Semantics of node N_1 obtained via the application of function $[\![\cdot]\!]$ to policy program $P_1; P_2$.

of operations that can be executed by the policies on node N_1 when triggered by a message.

C. Semantics of Policy Programs at System Level

Since policy programs are triggered by messages when they leave a node and when they enter one, knowledge about how messages are exchanged in the network is crucial to calculate the semantics of the whole system.

The network semantics is formalized in Figure 7. To

$$
\begin{aligned}
[\![\mathsf{local}; \mathsf{remote}]\!] = & \{[\![P]\!]_{C_0} \mid P \in \mathsf{local}\} \cup \\
& \{(\overline{a} \not\downarrow \overline{a}'; C) \mid (P, P') \in \mathsf{remote}, \\
& \qquad (\overline{a}; C') \in [\![P]\!]_{C_0}, (\overline{a}'; C) \in [\![P']\!]_{C'}\}
\end{aligned}
$$

Figure 7. Semantic function of policies in a sensor network.

avoid modelling components, assume local: a set of policy programs, and remote: a set of pairs of policy programs. Elements of local are policies applied during local communication. Pairs in remote are policy programs executed on the sending and on the receiving node when a message is sent across the network. The system semantics consist of the set of all possible (consistent) constrained traces from when a message is created until it is received by a component. The starting accumulator C_0 contains the empty logical constraint and the set of all types the node can produce.

Example 3 (System semantics): Consider a node N_2 that communicates with node N_1 from Example 1 on messages of type centigrade ($^\circ$C) and humidity (H). Intuitively, there are components on N_1 that produce such messages and components on N_2 that receive them. These details are abstracted away; the only required information is that there is communication from N_1 to N_2 on messages of type $^\circ$C

and H and that N_1 produces both centigrade and humidity messages. Table II presents policies that may be deployed on N_2: P_4 decrypts all temperature messages, while P_5 decrypts all kind of messages (Any). If P_1 is deployed on N_1

Table II
EXAMPLE POLICIES DEPLOYED ON NODE N_2.

$P_4@N_2$	$P_5@N_2$
on (T) as e if (true) decrypt e;	on (Any) as e if (true) decrypt e;

and P_5 is deployed on N_2, we obtain the following system $S_1 \equiv \mathsf{local} = \emptyset; \mathsf{remote} = (P_1; P_5)$. The semantics of S_1 is calculated in the upper part of Figure 8. The second trace of

$$
\begin{aligned}
[\![P_1]\!]_{\circ \mathtt{C},\mathtt{H}} &= \{(\mathtt{encrypt}; {}^{\circ}\mathtt{C}), (\epsilon; \mathtt{H})\} \text{ (see Figure 6)} \\
[\![P_5]\!]_{\mathtt{H}} &= \{(\mathtt{decrypt}; \mathtt{H})\} \\
[\![P_5]\!]_{\circ \mathtt{C}} &= \{(\mathtt{decrypt}; {}^{\circ}\mathtt{C})\} \\
[\![S_1]\!] &= \{(\overline{a} \not\downarrow \overline{a}'; C) \mid (\overline{a}; C') \in [\![P_1]\!]_{C_0}, (\overline{a}'; C) \in [\![P_5]\!]_{C'}\} \\
&= \{(\not\downarrow \mathtt{decrypt}; \mathtt{H}), (\mathtt{encrypt} \not\downarrow \mathtt{decrypt}; {}^{\circ}\mathtt{C})\} \\
[\![S_2]\!] &= \{(\not\downarrow; \mathtt{H}), (\mathtt{encrypt} \not\downarrow \mathtt{decrypt}; {}^{\circ}\mathtt{C})\}
\end{aligned}
$$

Figure 8. Semantics of systems $S_1 \equiv \mathsf{local} = \emptyset; \mathsf{remote} = (P_1; P_5)$ and $S_2 \equiv \mathsf{local} = \emptyset; \mathsf{remote} = (P_1; P_4)$.

$[\![S_1]\!]$ is an undesired one since a decrypt appears without a matching encrypt. Such trace would not be present if P_4 is deployed on N_2 in place of P_5. The semantics of configuration $S_2 \equiv \mathsf{local} = \emptyset; \mathsf{remote} = (P_1; P_4)$ is presented in the lower part of Figure 8.

III. SYSTEM ANALYSIS VIA MODAL LOGIC

Desirable properties over the traces given by the semantics of Figure 7 are described using mCRL2's modal logic [9].

Figure 9 provides examples of formulas used to detect semantical inconsistencies across different scenarios. Square brackets require that all possible paths match the enclosed formula, and angle brackets require the existence of a path that matches the formula. Let us provide two examples that show how such formulas are to be read. Formula (enc-before-dec) states that any sequence of actions (square brackets) that is not an encrypt ((! encrypt) *) followed by a decrypt (.decrypt) may never happen happen (false). Formula (dec-after-enc) states that any sequence of actions (square brackets) that is something (true*) followed by an encrypt (.encrypt), followed by something (.true*), followed by a network communication (.$\not\downarrow$), followed by anything that is not a decrypt (. (!decrypt) *), *may* be followed (angular brackets) by something (true*) followed by a decrypt (.decrypt).

In order to enforce the correct execution of security protocols, encrypt and decrypt must occur in the right order. Formula (enc-before-dec) states that decrypt cannot occur without a preceding encrypt , and Formula (dec-after-enc) states that after every encrypt must be followed by a decrypt . Similar reasoning applies to actions sign and verify, and to delete and persist; some formulas are omitted for the sake of brevity. For power management issues, expensive security-related operations should not be allowed on a message that is going to trigger a deny, as stated by Formula (no-waste-deny) and Formula (no-waste-deny-aft). Other formulas can be used to enforce the correct order of actions in the communication between two components, including application-specific properties.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

A prototype implementation has been developed following the aforementioned theory.[2] This prototype expects two inputs: (1) a full description of the application, containing the nodes on which it is deployed, the CaPI policies present on every node, and the application topology in terms of communicating nodes, and (2) a list of properties that well-formed applications must satisfy. In return, the prototype (a) creates an abstract model of a WSN which conforms to our formalisation and (b) uses the mCRL2 toolset [9] to verify the properties specified in input (2). The prototype shall be incorporated in the network administration tool used to manage CaPI applications.

Abstract Model. Policies deployed in a CaPI application are flattened in a single policy program. Some arguments and actions that do not influence the analysis are ignored and guards are simplified using the predicate abstraction mechanism described in §II-B. Predicates that are syntactically equivalent are therefore considered equal in our model.

Model Checking. Firstly, an mCRL2 process representing all possible traces is extracted from the abstract model. Secondly, checks whether the given properties hold with respect to the mCRL2 process are made. When a property fails, a second analysis is performed to provide a more accurate description of the problem. The first process is faster for mCRL2 to analyse but it contains traces without the related accumulators, thus there is no useful information for the application manager regarding the circumstances under which faults occur.

Small experiments have been performed, revealing promising results. The prototype scales linearly wrt the number of connections between network nodes and to the number of policies with identical if-guards. The prototype scales exponentially wrt the number of different if-guards, although real case scenarios suggest that the number of different if-guards is typically low. On a configuration with 110 policies distributed on 80 communicating nodes, the prototype finds the constrained traces and analyses 13 formulas against them

[2]Available at http://distrinet.cs.kuleuven.be/software/dissent/.

```
[(!encrypt)*.decrypt ] false;                                          (enc-before-dec)
[true*.encrypt.true*.↯.(!decrypt)*]<true*.decrypt > true;              (dec-after-enc)
[(!sign)*.verify ] false;                                              (sig-before-ver)
[true*.sign.true*.↯.(!verify)*]<true*.verify > true;                   (ver-after-sig)
[true*.(sign + verify + encrypt + decrypt).true*.deny ] false;         (no-waste-deny)
[true*.deny.true*.(sign + verify + encrypt + decrypt)] false;          (no-waste-deny-aft)
```

Figure 9. Example of formulas capturing undesired properties in CaPI applications.

in around 44 seconds on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 4GB 1333MHz DDR3 RAM. An additional 0.049 seconds are required for each trace that fails a property check in order to run the second analysis.

V. Discussion

This section describes the assumptions and limitations of the presented work.

The presented model assumes that certain information is known by the application manager. Details about the application topology are required, namely, which node communicates with which other node. Information concerning all policies deployed on all nodes and all types of messages produced by all nodes is also required.

The model is only concerned with application events, namely, messages exchanged between components; low-level messages, such as routing control, are not considered in this work. Since the focus is on the semantics of policies, components are abstracted away; future work will address component verification. We model a static network configuration, focussing on security and resource usage problems. Modeling and analyzing the set of policy actions in charge of dynamical reconfiguration of the network is left for future work.

We chose not to model wireless interference since a scenario where no packets are lost is the one that generates the most interesting traces. Lossy communication can generate partial traces: only the ones starting from the sender node. Such traces are not interesting to model when analyzing security and resource-optimization properties. For example, from the security point of view an administrator wants to know if all encrypted messages are decrypted upon reception, if such messages are lost they lose interest. Partial traces will be interesting to model when dynamic reconfiguration actions are taken into account, as a lost message can still trigger actions and reconfigure a part of the network. The same holds for the concept of interleaving among traces, they will be more interesting when reconfiguration action are taken into account.

VI. Related Work

Detection of conflicts in policies in distributed systems has been studied by Lupu and Sloman [10], by means of a static analysis, but their work is not concerned with the composition of policies across the network. Furthermore their policies regard authorization, thus conflicts involve granting and denying access. Conflicts in the presented work are more subtle since one must consider the intended semantics of the actions of policies in order to state what undesired behaviour is.

Surveys on the existing work in the area of modeling techniques for WSN are presented by Jacoub *et al.* [11] and by Stanley-Marbell *et al.* [12]. None of the surveyed models treats the semantic composition of policies across the network, as they are mostly used for simulations concerned with resource analysis. Little work exists in the field of formal verification of WSN applications. Sharma *et al.* [13] verify correctness of the operations that send and receive messages in Insense, a π-calculus based implementation of components for WSN application. Zheng *et al.* [14] adopt a labelled transition system approach to formalising WSN applications written in NesC and provide verification of deadlock-freeness, state reachability and liveness properties expressed in linear temporal logic. Mottola *et al.* [15] implemented Anquiro, a model checker for WSN software which relies on linear temporal logic (LTL) to verify properties of the software. These works analyse a different aspect of a WSN application, they are orthogonal to ours. WSNs have been formalised and verified for properties often involving lifetime and power analysis [16] or properties of algorithms, such as OGDC [17] or the S-MAC medium access control protocol [18]. These works show the validity of formal method approaches in WSN settings that do not focus on properties of the deployed application.

VII. Conclusion and Future Work

Extensions to the presented work will be made on several fronts. Firstly, the complete set of actions available in CaPI will be analysed, including parameters of each action, leading to formulas of the form `!(encrypt (t, e.key, hash))*.false`, which can be processed by `mCRL2`. As mentioned in §V, the excluded actions deal with dynamic aspects of the system e.g. `activate` and `deactivate` components and policies. In this setting we will exploit some of the more advanced features of `mCRL2`,

such as the verification of liveness and fairness properties, in order to verify the reachability of certain configuration states. Secondly, additional reasoning could be performed on logical guards in order to detect logical implication, e.g. $temp > 6$ implies $temp > 7, 8, \cdots$. This would reduce the set of traces considered by the prototype. Additionally, in order to model interleaving a concurrency model shall be taken into account.

This paper presented the formalisation and analysis of the semantics of policies orchestrating a WSN application. The semantics of policies is formalised in terms of traces augmented with information concerning the constraints under which the trace is executed. The CaPI middleware has been used as case study to provide tangible proof that the underlying theory is solid and leads to interesting results.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for the useful and very insightful remarks on an early draft of the work.

REFERENCES

[1] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, "Supporting concurrent applications in wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 139–152.

[2] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. Picco, and S. Zachariadis, "Reconfigurable component-based middleware for networked embedded systems," *International Journal of Wireless Information Networks*, vol. 14, pp. 149–162, 2007.

[3] Y. Zhu, S. L. Keoh, M. Sloman, and E. Lupu, "A lightweight policy system for body sensor networks," *Network and Service Management, IEEE Transactions on*, vol. 6, no. 3, pp. 137 –148, september 2009.

[4] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen, "A component and policy-based approach for efficient sensor network reconfiguration," in *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks, 2012*, 2012, to appear.

[5] N. Matthys, C. Huygens, D. Hughes, J. Ueyama, S. Michiels, and W. Joosen, "Policy-driven tailoring of sensor networks," in *Sensor Systems and Software, Revised Selected Papers. LNICST*, vol. 51. Springer, 2010.

[6] G. Russello, L. Mostarda, and N. Dulay, "A policy-based publish/subscribe middleware for sense-and-react applications," *Journal of Systems and Software*, vol. 84, no. 4, pp. 638–654, 2011.

[7] D. W. Marsh, R. O. Baldwin, B. E. Mullins, R. F. Mills, and M. R. Grimaila, "A security policy language for wireless sensor networks," *Journal of Systems and Software*, vol. 82, pp. 101–111, 2009.

[8] M. Fitting and R. L. Mendelsohn, *First-Order Modal Logic*. Kluwer Academic Press, 1998.

[9] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. v. Weerdenburg, W. Wesselink, T. Willemse, and J. v. d. Wulp, "The mcrl2 toolset," in *Proceedings of WASDeTT*, 2008.

[10] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 852–869, 1999.

[11] J. K. Jacoub, R. Liscano, and J. S. Bradbury, "A survey of modeling techniques for wireless sensor networks," in *Proc. of the 5th International Conference on Sensor Technologies and Applications*, ser. SENSORCOMM 2011, 2011, pp. 103–109.

[12] P. Stanley-Marbell, T. Basten, J. Rousselot, R. S. Oliver, H. Karl, M. Geilen, R. Hoes, G. Fohler, and J.-D. Decotignie, "System models in wireless sensor networks," Eindhoven University of Technology, Tech. Rep., 2008.

[13] O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison, and J. Sventek, "Towards verifying correctness of wireless sensor network applications using insense and spin," in *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 223–240.

[14] M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu, "Towards a model checker for nesc and wireless sensor networks," in *Proceedings of the 13th international conference on Formal methods and software engineering*, ser. ICFEM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 372–387.

[15] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi, "Anquiro: enabling efficient static verification of sensor network software," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, ser. SESENA '10. New York, NY, USA: ACM, 2010, pp. 32–37.

[16] S. Coleri, M. Ergen, and T. J. Koo, "Lifetime analysis of a sensor network with hybrid automata modelling," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, ser. WSNA '02. New York, NY, USA: ACM, 2002, pp. 98–104.

[17] P. C. Ölveczky and S. Thorvaldsen, "Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude," in *Proceedings of the 9th IFIP WG 6.1 international conference on Formal methods for open object-based distributed systems*, ser. FMOODS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 122–140.

[18] P. Ballarini and A. Miller, "Model checking medium access control for sensor networks," in *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ser. ISOLA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 255–262.