Formal modelling and analysis of socio-technical systems

Probst, Christian W.; Kammüller, Florian; Hansen, Rene Rydhof

Published in:
Semantics, Logics, and Calculi

Link to article, DOI:
10.1007/978-3-319-27810-0_3

Publication date:
2016

Document Version
Peer reviewed version

Citation (APA):
Probst, C. W., Kammüller, F., & Hansen, R. R. (2016). Formal modelling and analysis of socio-technical systems. In C. W. Probst, C. Hankin, & R. Rydhof Hansen (Eds.), *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays* (pp. 54-73). Springer. https://doi.org/10.1007/978-3-319-27810-0_3

Formal Modelling and Analysis of Socio-technical Systems

Christian W. Probst[1], Florian Kammüller[2], and René Rydhof Hansen[3]

[1] Technical University of Denmark
cwpr@dtu.dk
[2] Middlesex University, UK
f.kammueller@mdx.ac.uk
[3] Aalborg University, Denmark
rrh@cs.aau.dk

Abstract. Attacks on systems and organisations increasingly exploit human actors, for example through social engineering. This non-technical aspect of attacks complicates their formal treatment and automatic identification. Formalisation of human behaviour is difficult at best, and attacks on socio-technical systems are still mostly identified through brainstorming of experts. In this work we discuss several approaches to formalising socio-technical systems and their analysis. Starting from a flow logic-based analysis of the insider threat, we discuss how to include the socio aspects explicitly, and show a formalisation that proves properties of this formalisation. On the formal side, our work closes the gap between formal and informal approaches to socio-technical systems. On the informal side, we show how to steal a birthday cake from a bakery by social engineering.

1 Introduction

Applying formal methods [1] to an informal world is difficult. It often requires to loosen the precision of analysis results, or to overly restrict the aspects that can be modelled. This dilemma causes many approaches that try to understand events in the real world to abstract away its difficult parts. In this paper we present an application of formal methods to organisations to analyse socio-technical systems, and illustrate how aspects of the informal world can be handled in formal analyses.

Socio-technical systems, as the name implies, are a mix of social and technical aspects. Organisations are a good example for socio-technical systems, since they combine technical infrastructure and policies with human actors, who operate (in) this infrastructure and interact with it. An increasing number of attacks against organisations exploit this mix and involve attack steps on the "socio" part, for example, through social engineering. Security in socio-technical systems should therefor not only consider both individual parts, but also their interactions. The recent attack on a German steel mill [2] was a combination of both targeted phishing emails and social engineering attacks. The phishing helped the hackers extract information they used to gain access to the plant's office network and then its production systems. As a result, the technical infrastructure of the mill suffered severe damage. Traditional and well-established risk assessment methods often identify potential threats against socio-technical systems, but often abstract away the internal structure of an organisation and ignore human factors.

Actually, only few, if any, approaches to systematic risk assessment take such "human factor"-based attacks into consideration. Probably the strongest threat against socio-technical systems is the insider threat [3, 4]. Insiders have access to parts of the organisation's infrastructure and assets, and they are trusted to perform certain operations on these. Starting from a flow logic [5] based analysis of the insider threat, we discuss how to include the socio aspects explicitly, and show a formalisation that proves properties of this formalisation.

On the formal side, our work closes the gap between formal and informal approaches to socio-technical systems. On the informal side, we show how to steal a birthday cake from a bakery by social engineering.

Our work thereby closes the gap by developing models and analytic processes that support assessing both the socio *and* the technical side of organisations as socio-technical systems, thus combining human factors and physical infrastructure. Our approach simplifies the identification of possible attacks and provides qualified assessment and ranking of attacks based on the expected impact.

The rest of this chapter is structured as follows. After introducing the bakery example as our socio-technical system, Section 3 presents a formalisation of such systems followed by a flow logic-based analysis in Section 4. A discussion of the limitations of this formal approach when facing human actions leads to a more general identification of possible attacks in Section 5, followed by an attempt to formalise human behaviour in Section 6. After discussing related work Section 7, we conclude the paper with an outlook on future developments.

2 The Drama of the Birthday Cake in Three Pictures

In this section, we provide a case study of a very recent insider attack where a baker's wife socially engineered her husband the baker with the malicious intention to steal Hanne and Flemming's birthday cake. What is worse, is that she succeeded — due to the lack of formal analysis in this bakery. In the rest of this paper we will illustrate the attack and then show different formalisations to identify this attack. Figures 1, 2, and 3 illustrate the sequence of events that lead to the devastating outcome. The part of the bakery that is not illustrated is presented in the next section.

3 Modelling Socio-technical Systems

Our model represents the infrastructure of organisations, in this case the bakery, as nodes in a directed graph [6], representing *rooms*, access control points, and similar locations. *Actors* are represented by nodes and can possess *assets*, which

Fig. 1. The baker bakes a cake for Hanne and Flemming's birthday and protects it by putting it in the cake locker — but his wife sees it all.

Fig. 2. The baker's wife uses a social engineering attack on the baker to get his credentials: the key to the cake locker.

Fig. 3. Disaster: Hanne and Flemming's birthday cake vanished from cake locker!

model data and items that are relevant in the modelled scenario. Assets can be annotated with a value and a metric, *e.g.*, the likelihood of being lost. Nodes representing assets can be attached to locations or actors; assets attached to actors move around with that actor. Actors perform *actions* on locations, including physical locations or other actors. Thes actions are restricted by *policies* that represent both access control and the behaviour as expected by an organisation from its employees. Policies consist of required credentials and enabled actions, representing what an actor needs to provide in order to enable the actions in a policy, and what actions are enabled if an actor provides the required credentials, respectively.

Our modelling approach is based on Klaim [7]. In contrast to Klaim, we attach processes and actors to special nodes that move around with the process. This makes the modelling of actors and items carried by actors more intuitive and natural, but can easily be mapped back to original Klaim. The metrics mentioned above can represent any quantitative knowledge about components, for example, likelihood, time, price, impact, or probability distributions. The latter could describe behaviour of actors or timing distributions.

3.1 Semantics of Socio-Technical Models

In the following we briefly summarise the formal semantics of our socio-technial models. The calculus follows previous presentations closely and we will therefore not go deep into details here, merely refer to [8]. As already mentioned, the semantics is based on a variant of the Klaim calculus [7], called bacKlaim, which in turn is based on acKlaim [6, 8]. The Klaim calculus uses the *tuple space* paradigm, in which systems are composed of a set of distributed nodes that communicate and interact by reading and writing tuples in shared tuples spaces. The following presentation of bacKlaim is an adaptation and simplification of the calculus presented in [8].

In keeping with tradition, the semantics of the bacKlaim calculus is split into three layers: nets, processes, and actions. Nets define the overall, distributed structure of the system by specifying where individual nodes and tuple spaces are located. Processes and actions define the actual behaviour of the nodes. The syntax of nets, processes, and actions is shown in Figure 4. In the bacKlaim calculus there are two actions for reading a tuple in a remote tuple space: **in** for destructive read and **read** for non-destructive read. Both these input actions allow for *template* specifications of the tuple(s) to be read, facilitating a simple form of pattern matching with variable binding. The syntax for templates is shown in Figure 5 and the corresponding semantics is shown in Figure 7.

One of the key differences between classic Klaim and bacKlaim is the explicit support for access control policies in the latter, through a *reference monitor* embedded in the semantics. Before going further into the semantics of bacKlaim, we first need to define these *access control policies*. In the bacKlaim calculus, the kind of access that is relevant to control, is whether or not a process at a given location is allowed to perform a specific action at a remote location. Thus

$\ell ::= l$	locality	$N ::= l ::^{\delta} P$	single node
$\mid\ u$	locality variable	$\mid\ l ::^{\delta} \langle et \rangle$	located tuple
		$\mid\ N_1 \parallel N_2$	net composition
$P ::= \mathbf{nil}$	null process	$a ::= \mathbf{out}\,(t)@\ell$	output
$\mid\ a.P$	action prefixing	$\mid\ \mathbf{in}\,(T)@\ell$	input
$\mid\ P_1 \mid P_2$	parallel composition	$\mid\ \mathbf{read}\,(T)@\ell$	read
$\mid\ A$	process invocation	$\mid\ \mathbf{eval}\,(P)@\ell$	migration
		$\mid\ \mathbf{move}\,(\ell)$	move

Fig. 4. Syntax of nets, processes, and actions.

$T ::= F \mid F, T$	templates	$et ::= ef \mid ef, et$	evaluated tuple
$F ::= f \mid !x \mid !u$	template fields	$ef ::= V \mid l$	evaluated tuple field
$t ::= f \mid f, t$	tuples	$e ::= V \mid x \mid \ldots$	expressions
$f ::= e \mid l \mid u$	tuple fields		

Fig. 5. Syntax for tuples and templates.

we can formalise access control policies as follows:

$$\begin{aligned} \pi &\subseteq \mathsf{AccMode} = \{\mathbf{i}, \mathbf{r}, \mathbf{o}, \mathbf{e}, \mathbf{n}, \mathbf{m}\} \\ \delta &\in \mathsf{Policy} = (\mathsf{Loc} \cup \{\star\}) \rightarrow \mathcal{P}\,(\mathsf{AccMode}) \end{aligned}$$

where the *access modes* correspond to the actions that can be taken in the semantics: **i** for (destructively) reading a tuple, **r** for (non-destructively) reading a tuple, **o** for outputting (writing) a tuple, **e** for remote evaluation of a process, and **n** for the capability to create new locations. The special '$\star$' location is used to denote default policies, i.e., access modes that are allowed from all locations not specifically mentioned.

We can now continue with the semantics for bacKlaim, by defining the reduction relation for processes and actions, shown in Figure 6. In general, a process is composed of sequences of actions, (sub-)processes that execute in parallel, or a recursive invocation through a place-holder variable. The actions a process can perform are: **out**, that writes a tuple to the specified tuple space; **in**, that reads a tuple (at the specified tuple space) matching the template and then *removes* the tuple in question; **read** that also reads a tuple (at the specified tuple space) matching the given template but does *not* remove the tuple; **eval** that evaluates the given process at the specified (remote) location. Finally, the **move** action relocates the node representing the actor or process. However, we only wish to allow certain moves between nodes, e.g., a node representing a (physical) actor should only be able to move between nodes representing physical localities. We

$$\frac{[\![t]\!] = et \qquad \boxed{(l,l') \in \mathcal{I} \wedge \mathbf{o} \in \delta(l')}}{l ::^{\delta} \mathbf{out}\,(t)@l'.P \parallel l' ::^{\delta'} P' \rightarrowtail_{\mathcal{I}} l ::^{\delta} P \parallel l' ::^{\delta'} P' \parallel l' ::^{\delta'} \langle et \rangle}$$

$$\frac{\mathit{match}([\![T]\!], et) = \sigma \qquad \boxed{(l,l') \in \mathcal{I} \wedge \mathbf{i} \in \delta(l')}}{l ::^{\delta} \mathbf{in}\,(T)@l'.P \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail_{\mathcal{I}} l ::^{\delta} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$\frac{\mathit{match}([\![T]\!], et) = \sigma \qquad \boxed{(l,l') \in \mathcal{I} \wedge \mathbf{r} \in \delta(l')}}{l ::^{\delta} \mathbf{read}\,(T)@l'.P \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail_{\mathcal{I}} l ::^{\delta} P\sigma \parallel l' ::^{\delta'} \langle et \rangle}$$

$$\frac{\boxed{(l,l') \in \mathcal{I} \wedge \mathbf{e} \in \delta(l')}}{l ::^{\delta} \mathbf{eval}\,(Q)@l'.P \ \parallel \ l' ::^{\delta'} P' \rightarrowtail_{\mathcal{I}} l ::^{\delta} P \parallel l' ::^{\delta'} Q \parallel l' ::^{\delta'} P'}$$

$$\frac{\boxed{\{(l,l''),(l'',l')\} \in \mathcal{I} \wedge \mathbf{m} \in \delta(l')}}{l ::^{\delta} \mathbf{move}\,(l')\,.P \ \parallel \ l' ::^{\delta'} P' \rightarrowtail_{\mathcal{I}'} l ::^{\delta} P \parallel l' ::^{\delta'} P'} \qquad \mathcal{I}' = \mathcal{I} \setminus (l, \_) \cup \{(l,l')\}$$

$$\frac{L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail_{\mathcal{I}} L' \vdash N_1' \parallel N_2} \qquad \frac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'}$$

Fig. 6. Reduction semantics for bacKlaim.

$$\mathit{match}(V,V) = \epsilon \quad \mathit{match}(!x,V) = [V/x] \quad \mathit{match}(l,l) = \epsilon \quad \mathit{match}(!u,l') = [l'/u]$$

$$\frac{\mathit{match}(F, \mathit{ef}) = \sigma_1 \qquad \mathit{match}(T, et) = \sigma_2}{\mathit{match}((F,T),(\mathit{ef}, et)) = \sigma_1 \circ \sigma_2}$$

Fig. 7. Semantics for template matching.

$$\begin{array}{ll} N_1 \parallel N_2 \equiv N_2 \parallel N_1 & (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \\ l ::^{\delta} P \equiv l ::^{\delta} (P \mid \mathbf{nil}) & l ::^{\delta} A \equiv l ::^{\delta} P \qquad \text{if } A \overset{\triangle}{=} P \end{array}$$

$$l ::^{\delta} (P_1 \mid P_2) \equiv l ::^{\delta} P_1 \parallel l ::^{\delta} P_2$$

Fig. 8. Structural congruence on nets and processes.

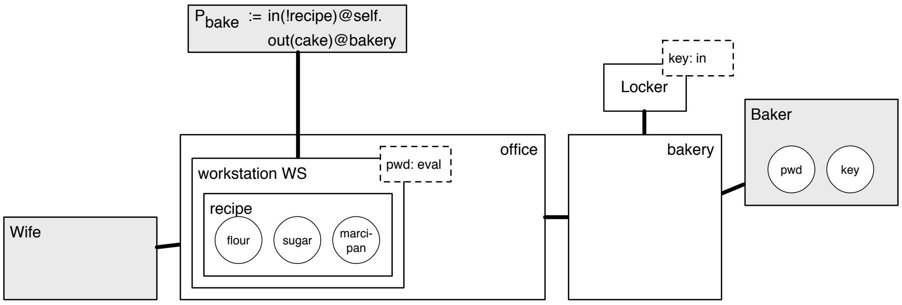

Fig. 9. Graphical representation of the crime scene, the bakery. The rectangles represent actors, locations, assets, and processes. The baker is (still) in possession of the key and the password for the computer.

formalise this in the form of the so-called *infrastructure* of the underlying nets:

$$\mathcal{I} \in \mathsf{Infrastructure} = \mathcal{P}\left(\mathsf{Locality} \times \mathsf{Locality}\right)$$

Essentially, the infrastructure is a graph, relating the pairs of nodes between which moves are allowed (still subject to access control rules).

In addition to the reduction relation, the semantics also incorporates a structural congruence, simplifying (re-)presentation of, computation with, and reasoning about processes and nets. The congruence is shown in Figure 8.

3.2 The Bakery Model

The bakery example introduced in Section 2 is based on the baker and his wife, and of course the cake. The assets in this example are the key to the cake locker, the cake itself, and, to add to the excitement, a computer with the recipe for the cake. The recipe is input to a process on the computer that outputs a cake in the bakery.[4] We assume the baker to have an (internalised) policy that forbids the cake to leave the bakery prematurely. Figure 9 shows the formalisation of the bakery, consisting of the baker shop, the office, the cake locker, and the outside world. The baker has the key and the Password to his computer. The policies in the model require, *e.g.*, the key to enter the cake locker and the password to log into the computer. Actor nodes also represent processes running on the corresponding locations. The process at the computer represents the “creation” of a cake, that is output at the bakery.

---

[4] To simplify treatment we assume the bakery to be high-tech. A different approach would have been to model the baking process at the baker or the bakery, requiring the recipe as input.

$$
\begin{array}{lll}
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{N}} l ::^{\delta} P & \text{iff} & (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{\lfloor l \rfloor} P \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{N}} l ::^{\delta} \langle et \rangle & \text{iff} & \langle et \rangle \in \hat{T}(\lfloor l \rfloor, \delta) \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{N}} N_1 \parallel N_2 & \text{iff} & (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{N}} N_1 \wedge (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{N}} N_2 \\[4pt]
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} \mathbf{nil} & \text{iff} & \mathit{true} \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} P_1 \mid P_2 & \text{iff} & (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} P_1 \wedge (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} P_2 \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} A & \text{iff} & (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} P \qquad \text{if } A \triangleq P \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} a.P & \text{iff} & (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} a \wedge (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{l} P \\[4pt]
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} \mathbf{out}\,(t)@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') \colon \; (\mathbf{o} \in \delta(\hat{l}) \Rightarrow \hat{\sigma}[\![t]\!] \subseteq \hat{T}(\hat{l})) \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} \mathbf{in}\,(T)@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') \colon \; (\mathbf{i} \in \delta(\hat{l}) \Rightarrow \hat{\sigma} \models_1 T : \hat{T}(\hat{l}) \triangleright \hat{W}_{\bullet}) \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} \mathbf{read}\,(T)@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') \colon \; (\mathbf{r} \in \delta(\hat{l}) \Rightarrow \hat{\sigma} \models_1 T : \hat{T}(\hat{l}) \triangleright \hat{W}_{\bullet}) \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} \mathbf{eval}\,(Q)@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') : (\mathbf{e} \in \delta(\hat{l}) \Rightarrow (\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{P}}^{\hat{l}} Q) \\
(\hat{T},\hat{\sigma},\hat{\mathcal{I}}) \models_{\mathrm{A}}^{l} \mathbf{move}\,(\ell') & \text{iff} & \forall \hat{l} \in \hat{\sigma}(l) : \forall \hat{l}' \in \hat{\sigma}(\ell') : (\mathbf{m} \in \delta(\hat{l}') \Rightarrow (\hat{l},\hat{l}') \in \hat{\mathcal{I}})
\end{array}
$$

Fig. 10. Flow Logic specification for control flow analysis of bacKlaim.

$$
\begin{array}{lll}
\hat{\sigma} \models_i \epsilon : \hat{V}_{\circ} \triangleright \hat{V}_{\bullet} & \text{iff} & \{\hat{et} \in \hat{V}_{\circ} \,|\, |\hat{et}| = i\} \sqsubseteq \hat{V}_{\bullet} \\
\hat{\sigma} \models_i V, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \{\hat{et} \in \hat{V}_{\circ} \,|\, \pi_i(\hat{et}) = V\} \sqsubseteq \hat{V}_{\bullet} \\
\hat{\sigma} \models_i l, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \{\hat{et} \in \hat{V}_{\circ} \,|\, \pi_i(\hat{et}) = V\} \sqsubseteq \hat{V}_{\bullet} \\
\hat{\sigma} \models_i x, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \{\hat{et} \in \hat{V}_{\circ} \,|\, \pi_i(\hat{et}) \in \hat{\sigma}(x)\} \sqsubseteq \hat{V}_{\bullet} \\
\hat{\sigma} \models_i u, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \{\hat{et} \in \hat{V}_{\circ} \,|\, \pi_i(\hat{et}) \in \hat{\sigma}(u)\} \sqsubseteq \hat{V}_{\bullet} \\
\hat{\sigma} \models_i !x, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \hat{V}_{\circ} \sqsubseteq \hat{V}_{\bullet} \wedge \pi_i(\hat{W}_{\bullet}) \sqsubseteq \hat{\sigma}(x) \\
\hat{\sigma} \models_i !u, T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet} & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_{\bullet} \triangleright \hat{W}_{\bullet} \wedge \hat{V}_{\circ} \sqsubseteq \hat{V}_{\bullet} \wedge \pi_i(\hat{W}_{\bullet}) \sqsubseteq \hat{\sigma}(u)
\end{array}
$$

Fig. 11. Flow Logic specification for pattern match analysis [8].

4 Flow Logic-based Analysis of Processes

The first analysis for catching the thief is a flow logic analysis similar to [8]. This analysis takes a sequence of actions and performs a static control flow analysis to compute and assess its effect by a conservative approximation of the possible flow between actors, processes, and tuple spaces. Following the Flow Logic framework [9], we specify a *judgements* for nets, processes, and actions that determine whether or not an analysis estimate correctly describes all configurations that are reachable from the initial state. The definitions are shown in Figure 10.

The tuple spaces and variable values are collected in $\hat{T}$ and $\hat{\sigma}$. For space reasons we do not consider the **newloc** action that dynamically creates new locations. Similar to [8] we could use canonical names. For the pattern matching we reuse the Flow Logic specification, shown in Figure 11 from [8].

$$\begin{aligned} P_{baker} &:= \textbf{move}\,(\mathit{office})\,.\textbf{eval}\,(P_{bake})@\mathit{WS}.\textbf{move}\,(\mathit{bakery})\,.\textbf{in}\,(\mathit{cake})@\mathit{bakery}.\\ &\quad\ \textbf{move}\,(\mathit{Locker})\,.\textbf{out}\,(\mathit{cake})@\mathit{Locker}.\textbf{move}\,(\mathit{bakery})\\ P_{wife} &:= \textbf{move}\,(\mathit{bakery})\,.\textbf{in}\,(\mathit{key})@\mathit{baker}.\\ &\quad\ \textbf{move}\,(\mathit{Locker})\,.\textbf{in}\,(\mathit{cake})@\mathit{Locker}.\textbf{move}\,(\mathit{bakery}) \end{aligned}$$

Fig. 12. The two processes for the Flow Logic analysis. The baker bakes the cake and brings it to the Locker, and his wife picks the key from him, goes also to the Locker, and steals the cake.

Having specified the analysis it remains to be shown that the information computed by the analysis is correct. In the Flow Logic framework this is usually done by establishing a *subject reduction* property for the analysis:

Theorem 1 (Subject Reduction). *If* $(\hat{T}, \hat{\sigma}, \hat{\mathcal{I}}) \models_N N$ *and* $L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'$ *then* $(\hat{T}, \hat{\sigma}, \hat{\mathcal{I}}) \models_N N'$.

Proof. (Sketch) By induction on the structure of $L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'$ and using auxiliary results for the other judgements.

4.1 Analysing the Bakery Example

Before we conclude this section, we quickly want to see whether the flow logic analysis can help the baker in protecting the cake. We consider the two processes shown in Figure 12; the first process represents the baker going to the office and starts the "bake" process on the workstation. As a result, the cake appears in the bakery, the baker goes there and picks up, goes to the Locker and puts it down, and then returns to the bakery. The wife meets the baker somewhere, in our case in the bakery, picks the key from his pocket, goes to the Locker, gets the cake, and returns to the bakery.

The result of the flow logic analysis of the two processes shown in Figure 12 is that the cake will be at the baker, the bakery, and the Locker. However, it will also be at the wife, which is what the baker wanted to prevent, knowing her sweet tooth. This means that from the flow logic analysis, the baker can learn that his wife has stolen the cake.

5 Attack Generation

Unfortunately, there is a problem with the processes described in the previous section. Processes are a suitable abstraction for programs, but we are in general not able to obtain processes describing human behaviour. If the baker knew, which actions his wife had performed, he also would know that she stole cake—without any analysis or tool support.

If we cannot obtain processes to identify attacks, we need a different method to do so. In this section we present a recent development to attack generation based on the negation of policies [10, 11]. The policies we consider describe global system states that should be fulfilled at all times; our approach identifies sequences of actions that results in a policy violation.

In the bakery example, the baker could have the global policy that only the birthday children or he should get the cake, or more concretely, that his wife should *not* have the cake. Since she is determined to obtain the cake, she would violate this global policy by obtaining it.

We choose attack trees as a succinct way of representing attacks. In attack trees [12, 13], the root represents a goal, and the children represent sub-attacks. Sub-attacks can be alternatives for reaching the goal (disjunctive node) or they must all be completed to reach the goal (conjunctive node). We assume an implicit, left to right order for children of conjunctive nodes. For example, an attacker first needs to move to a location before being able to perform an action. Leaf nodes represent the basic actions in an attack. The operators $\oplus_{\vee}$ and $\oplus_{\wedge}$ combine attack trees by adding a disjunctive or conjunctive root, respectively.

In the remainder of this section we present the rules for generating attack trees from models. The rules take as arguments the infrastructure $\mathcal{I}$ and an actor component $\mathcal{A}$, which stores reached locations, obtained data, and acquired identities for the attacker. The rules either succeed and return an (possibly empty) attack tree, or they block if no valid result can be computed. Our approach for invalidating a policy consists of four basic steps:

Identify Attackers: Choose the policy to invalidate, and identify the possible actors who could invalidate it.

Target Locations: Identify a set of locations where the prohibited actions can be performed.

Goto Target Location: Generate attacks for reaching target locations. This will identify and obtain required assets to perform any of these actions, and obtain all assets required to reach the target location.

Move to Target Location and Perform Action: Finally, move to the location identified in the second step and perform the action.

In the first step we identify possible attackers and locations where the action violating the global policy can be applied (see Figure 13). These are the goals for

$$\frac{\begin{array}{c} \sigma = \mathit{unify}_{\mathcal{I}}(\mathit{Actors}, \mathit{credentials}) \\ \mathit{attackers} = \mathit{getAttacker}_{\mathcal{I}}(\mathit{actor}, \sigma) \qquad \mathit{goals} = \mathit{applicableAt}_{\mathcal{I}}(\mathit{credentials}, \mathit{enabled}, \sigma) \\ \mathcal{I}, \mathit{attackers}, \mathit{goals} \vdash_{\mathit{goal}} \mathit{trees} \qquad\qquad \mathcal{T} := \oplus_{\vee}\, \mathit{trees} \end{array}}{\mathcal{I}, \mathit{not}(\mathit{actor}, \mathit{credentials}, \mathit{enabled}) \vdash_P \mathcal{T}}$$

Fig. 13. Attack generation starts from the global policy $\mathit{not}(\mathit{actor}, \mathit{credentials}, \mathit{enabled})$. Attack trees are generated for all possible policy violations. As every attack tree represents a violation of the policy, the resulting attack trees are combined by an *or* node.

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{goto}(\mathit{location}) \land \mathit{perform}(\mathit{action}) \vdash_{GP} \mathcal{T}}{\mathcal{I}, \mathcal{A}, (\mathit{location}, \mathit{action}) \vdash_{\mathit{goal}} \mathcal{T}}$$

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \vdash_{\mathit{goto}} \mathcal{T}_{\mathit{goto}}, \mathcal{A}' \qquad \mathcal{I}, \mathcal{A}', \mathit{perform}(a) \vdash_{\mathit{perform}} \mathcal{T}_{\mathit{action}}, \mathcal{A}''}{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \land \mathit{perform}(a) \vdash_{GP} \mathcal{T}_{\mathit{goto}} \oplus_{\land} \mathcal{T}_{\mathit{action}}, \mathcal{A}''}$$

Fig. 14. For each identified goal (consisting of a location and an action) an attacker moves to the location and performs the action. The rules result in an attack tree and a new state of the attacker, which includes the obtained keys and reached locations.

$$\frac{\begin{array}{c}\mathit{paths} = \mathit{getAllPaths}_{\mathcal{I}}(\mathcal{A}, l) \qquad \mathcal{I}, \mathcal{A}, \mathit{paths} \vdash_{\mathit{path}} \mathit{trees}, \mathcal{A}' \\ \mathcal{T} := \oplus_{\lor} \mathit{trees}\end{array}}{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \vdash_{\mathit{goto}} \mathcal{T}, \mathcal{A}'}$$

$$\frac{\begin{array}{c}\mathit{missing} = \mathit{missingCredentials}_{\mathcal{I}}(\mathcal{A}, \mathit{path}) \qquad \mathcal{I}, \mathcal{A}, \mathit{missing} \vdash_{\mathit{credential}} \mathit{trees}, \mathcal{A}' \\ \mathcal{T} := \oplus_{\land} \mathit{trees}\end{array}}{\mathcal{I}, \mathcal{A}, \mathit{path} \vdash_{\mathit{path}} \mathcal{T} \oplus_{\land} \mathcal{N}^{\text{pass path}}, \mathcal{A}'}$$

Fig. 15. Going to a location and performing an action results in two attack trees. The function *getAllPaths* returns all paths from the current locations of the actor to the goal location l, and the resulting attack trees are alternatives for reaching this location.

the attacker, and are the basis for generating attack trees (Figure 14). For each goal we generate a tree for moving to the location and another one for performing the action. While moving to the location new credentials may be required, which recursively invoke the attack generation again. The resulting new knowledge is added to the actor component $\mathcal{A}$.

The rules in Figure 15 and Figure 16 generate attack trees for moving around, performing actions, and obtaining credentials, resulting in attack trees for every single action of the attacker. The function *missingCredentials* uses the unification described above to match policies with the assets available in the model. The attack generation then generates one attack for each of these assets and combines the resulting trees with a disjunctive node.

Attack generation also considers triggering processes to obtain assets. We do not present this interaction between actors and processes for space reasons, as it follows the rules presented above.

5.1 Post-Processing Attack Trees

The generated attack trees only represent the factual attack steps for reaching the final goal. The trees do not contain any annotation or metrics about the likelihood of success of actions such as social engineering, or the potential impact of actions. Also the likelihood of a given attacker to succeed or fail is not considered.

$$\frac{i \notin \mathit{identities} \implies \mathcal{T} = \mathcal{N}^{\text{obtain identity } i}}{\mathcal{I}, (\mathit{identities}, \mathit{locations}, \mathit{assets}), \mathit{identity}\ i \vdash_{\mathit{credential}} \mathcal{T}, (\mathit{identities} \cup \{i\}, \mathit{locations}, \mathit{assets})}$$

$$\frac{\begin{array}{c}\mathcal{A} = (\mathit{identities}, \mathit{locations}, \mathit{assets}) \wedge a \notin \mathit{assets} \implies \\ \mathit{goals} = \mathit{availableAt}_{\mathcal{I}}(a) \qquad \mathcal{I}, \mathcal{A}, \mathit{goals} \vdash_{\mathit{goal}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\vee} \mathit{trees}\end{array}}{\mathcal{I}, \mathcal{A}, \mathit{asset}\ a \vdash_{\mathit{credential}} \mathcal{T}, \mathcal{A}'}$$

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{predicate}\ p(\mathit{arguments}) \vdash_{\mathit{predicate}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\vee} \mathit{trees}}{\mathcal{I}, \mathcal{A}, \mathit{predicate}\ p(\mathit{arguments}) \vdash_{\mathit{credential}} \mathcal{T}, \mathcal{A}'}$$

Fig. 16. Depending on the missing credential, different attacks are generated. If the actor lacks an identity, an attack node representing an abstract social engineering attack is generated, for example, social engineering or impersonating. If the missing credential is an asset, the function *availableAt* returns a set of pairs of locations from which this asset is available, and the according **in** actions. If the missing credential is a predicate, a combination of credentials fulfilling the predicate must be obtained.

Computing qualitative and quantitative measures [14, 15] on attack trees is orthogonal to our approach. beyond the scope of this work. The generated attack trees also often contain duplicated sub-trees, due to similar scenarios being encountered in several locations, for example, the social engineering of the same actor, or the requirement for the same credentials. This is not an inherent limitation, but may clutter attack trees. Similar to [16], a post-processing of attack trees can simplify the result.

5.2 Attack Tree for the Bakery Example

Figure 17 shows part of the attack tree generated for the bakery example. The first attack is the one described in the previous section and shown in Figure 12: the wife steals the key from the baker and gets the cake from the Locker after it has been baked. A variant of this attack is that she breaks the Locker door open. In the second attack, she social engineers the baker to bake the cake, and then picks up the cake in the bakery *before* the baker does so. In the third attack, she gets the password to the work station from the baker, and then starts P_{bake} herself. Finally, she can social engineer the baker to give her the cake, maybe promising him to share it. All attacks, where assets are stolen also occur in a variant where actors with access to the asset are social engineered to obtain the asset and give it to the attacker.

6 Analysis of Socio-Technical Attacks in Isabelle

We now consider a third approach to modelling socio-technical systems using the interactive theorem prover Isabelle. We first illustrate the attack and then we discuss how the socio-technical model can be transfered to the modeling and verification of insider threats using our Isabelle framework [17]. Finally, we extend the Isabelle technique here further to Isabelle Attack Trees.

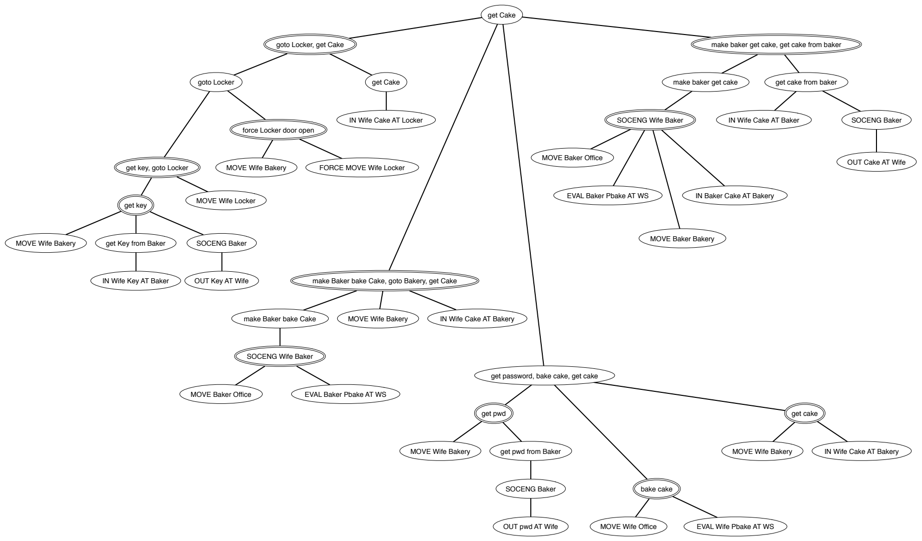

Fig. 17. Attack tree generated for the bakery example. The double-lined borders indicate disjunctive nodes.

6.1 Social Explanation for Insider Threats in Isabelle

In earlier work [17], we have used the process of sociological explanation based on Max Weber's *Grundmodell* and its logical interpretation to explain insider threats by moving between societal level (macro) and individual actor level (micro). The interpretation into a logic of explanation is formalized in Isabelle's Higher Order Logic thereby providing a tool to prove global security properties with machine assistance [17]. Isabelle/HOL is an interactive proof assistant based on Higher Order Logic (HOL). It enables specification of so-called object-logics for an application. Object-logics comprise new types, constants and definitions and reside in theory files, e.g., the file `Insider.thy` contains the object-logic we define for social explanation of insider threats below. We construct our theory as a conservative extension of HOL guaranteeing consistency. I.e., we do not introduce new axioms that could lead to inconsistencies.

We first provide here only the elements of this Insider theory necessary as a basis for attack trees and for modeling the bakery application. For a more complete view, please refer to [17] and the related online Isabelle resources [18].

In the Isabelle/HOL theory for Insiders, we express policies over actions `get`, `move`, `eval`, and `put`.

```
datatype action = get | move | eval | put
```

We abstract here from concrete data – actions have no parameters. Policies describe prerequisites for actions to be granted to actors given by pairs of predicates (conditions) and sets of (enabled) actions.

```
type_synonym policy = ((actor ⇒ bool) × action set)
```

We integrate policies with a graph into the infrastructure providing an organisational model where policies reside at locations and actors are adorned with additional predicates to specify their 'credentials'.

```
 datatype infrastructure = Infrastructure
"node graph" "location ⇒ policy set" "actor ⇒ bool"
```

These local policies serve to provide a specification of the 'normal' behaviour of actors but are also the starting point for possible attacks on the organisation's assets. The `enables` predicate specifies that an actor `a` can perform an action `a'`∈ `e` at location `l` in the infrastructure `I` if `a`'s credentials (stored in the tuple space `tspace I a`) imply the location policy's (stored in `delta I l`) condition `p` for `a`.

```
enables I l a a' ≡
 ∃ (p,e) ∈ delta I l. a' ∈ e ∧ (tspace I a ⟶ p(a))
```

For the application to the bakery senario, we only model two identities, `Baker` and `Wife` representing the baker and his wife. We define the set of bakery actors as a local definition in the locale `scenarioBakerNN`. We show here in a first instance the full Isabelle/HOL syntax but in all subsequent definitions we omit the `fixes` and `defines` keywords and also drop the types for clarity of the exposition. The double quotes **''s''** create a string in Isabelle/HOL.

```
 fixes bakery_actors :: identity set
 defines bakery_actors_def: bakery_actors ≡ {''Baker''}
```

The graph representing the infrastructure of the bakery case study contains only the minimal structure: (1) Kitchen, (2) Cake locker, (3) Home.

```
  bakery_locations ≡ {Location 1, Location 2, Location 3}
```

The global policy is 'no one except bakery employees can get anything from the cake locker'.

```
  global_policy I a ≡  a ∉ bakery_actors ⟶
                       ¬(enables I (Location 2) (Actor a) get)
```

Next, we have to provide the definition of the infrastructure. We first define the graph representing the organisation's locations and the positions of its actors. Locations are wrapped up with the datatype constructor `NL` and actors using the corresponding constructor `NA` to enable joining them in the datatype `node` and thus creating the following `node graph` as a set of pairs between locations or actors.

```
  ex_graph ≡ Graph {(NA (''Baker''), NL (Location 3)),
                    (NL (Location 3), NL(Location 1)),
                    (NL (Location 2), NL(Location 1)),
                    (NA (''Wife''), NL (Location 1))}
```

Policies are attached to locations in the organisation's graph using a function that maps each location to the set of the policies valid in this location. The policies are again pairs. The first element of these pairs are credentials which are defined as predicates over actors, *i.e.*, boolean valued functions describing, for example, whether an actor inhabits a role, or, whether an actor possesses something, like an identity or a key. The second elements are sets of actions that are authorised in this location for actors authenticated by the credentials.

```
local_policies ≡
 (λ x.  if x = Location 1 then
   {(λ x. (ID x ''Baker'')∨(ID x ''Wife''), {get,put}), (λ x. True, {move})}
                 else (if x = Location 2 then
                          {((λ x. has (x, ''key'')), {get,put,move})}
                       else (if x = Location 3 then
                          {((λ x. True, {get,put,move}))}
                           else {})))
```

The final component of any infrastructure is the credentials contained in a `tspace`. We define the assignment of the credentials to the actors similarly as a predicate over actors that is true for actors that have the credentials.

```
ex_creds ≡ (λ if x = Actor ''Baker'' then has (x,''key'') else False)
```

Finally, we can put the graph, the local policies, and the credential assignment into an infrastructure.

```
Bakery_scenario ≡ Infrastructure ex_graph local_policies ex_creds
```

Note, that all the above definitions have been implemented as local definitions using the locale keywords `fixes` and `defines`. Thus they are accessible whenever the locales `scenarioBakerNN` is invoked but are not axioms that could endanger consistency. We now also make use of the possibility of locales to define local assumptions. This is very suitable in this context since we want to emphasize that the following formulas are not general facts or axiomatic rules but are assumptions we make in order to explore the validity of the infrastructure's global policy. The first assumption provides that the precipitating event has occurred which leads to the second assumption that provides that Charly can act as an insider.

```
assumes Bakers_Wife_precipitating_event: tipping_point (astate ''Wife'')
assumes Insider_Wife : Insider ''Charly_comp'' {''Charly_priv''}
```

So far, we have specified the model. Based on these definitions and assumptions we can now state theorems about the security of the model and interactively prove them in our Isabelle/HOL framework. We can now first prove a sanity check on the model by validating the infrastructure for the "normal" case. For the baker as a bakery actor, everything is fine: the global policy does hold. The following is an Isabelle/HOL theorem `ex_inv` that can be proved automatically followed by the proof script of its interactive proof. The proof is achieved by locally unfolding the definitions of the scenario, e.g., `Bakery_scenario_def` and applying the simplifier.

```
lemma ex_inv:
 global_policy Bakery_scenario (''Baker'')
by (simp add: Bakery_scenario_def global_policy_def bakery_actors_def)
```

However, since the baker's `Wife` is at tipping point, she will ignore the global policy. This insider threat can now be formalised as an invalidation of the global company policy for `''Wife''` in the following "attack" theorem named `ex_inv1`.

```
theorem ex_inv1:
 ¬ global_policy Bakery_scenario ''Wife''
```

The proof of this theorem consists of a few simple steps largely supported by automated tactics. Thus `Wife` can get access to the cake leading to devastating outcomes (see Figure 3). The attack is proved above as an Isabelle/HOL theorem. Applying logical analysis, we thus exhibit that under the given assumptions the organisation's model is vulnerable to an insider. This overall procedure corresponds to the approach of invalidation of a global policy based on local policies for a given application scenario [10].

However, to systematically derive the actual attack vector the present paper provides a more constructive approach. We will next see how we can extend the Isabelle Insider framework to this.

6.2 Attack Trees in Isabelle

We now extend the theory Insider by Attack trees. The base attacks figure in an attack sequence (see Section 5). We represent them in Isabelle/HOL as a datatype and a list over this datatype.

```
datatype baseattack = None | Goto "location"
                    | Perform "action" | Credential "location"
type_synonym attackseq = "baseattack list"
```

The following defintion of attack tree, really defines the nodes of an attack tree. The simplest case is when a node in an attack tree is a base attack. Attacks can also be combined as the "and" of other attacks as defined in Section 5. This prescribes that the third element of type `attree` is a `baseattack` (usually a `Perform action`) that represents this attack, while the first element is an attack sequence and the second element is a label describing the attack (here a string).

```
datatype attree = BaseAttack "baseattack" ("𝒩 (_)") |
                  AndAttack "attackseq" "string" "baseattack" ("_ ⊕∧^(_) _")
```

As the corresponding projection functions for `attree` we define `get_attseq` and `get_attack` returning the entire attack sequence or the final base attack, respectively.

The following inductive predicate `get_then_move` shows how we represent the static analysis rules for the derivation of attack sequences. It translates the two rules of Figure 15 and formalizes how the impossible base attack `Goto l'` can be achieved by first going to location `l` and getting the credential from there. Logically, this is justified if an actor `a` can get to location `l'` in the extended infrastructure `add_credential I a s` where he possesses the credential `s` – as is expressed by the third `enables` proviso.

```
⟦ enables I l a move; enables I l a get;
  enables (add_credential I a s) l' a get ⟧
⟹ get_then_move I s
   (get_attackseq ([Goto l, Credential l, Goto l'] ⊕_∧^{get−move} Perform get))
   (Goto l')
```

An attack tree is constituted from the above defined nodes of type `attree` but children nodes must be refinement of their parents. Refinement means that some portion of the attack sequence has been extended according to rules like the above `get_then_move`. We formalize this constructor relation of the attack trees by the following refinement. The rules `trans` and `refl` make the refinement a preorder; the rule `get_moveI` shows how the `get_then_move` rule is integrated: If we replace the attack `a` by the `get_then_move` sequence `l` we get refine the attack sequence `A` into `A'` (the auxiliary function `sublist_rep` replaces a symbol in list by a list).

```
inductive
refines_to :: "[attree, infrastructure, attree] ⇒ bool" ("_ ⊑_(_) _")
where
get_moveI: ⟦ get_then_move I s l a;
              sublist_rep l a (get_attseq A) = (get_attseq A');
              get_attack A = get_attack A' ⟧ ⟹ A ⊑_I A' |
trans: ⟦ A ⊑_I A'; A' ⊑_I A'' ⟧ ⟹ A ⊑_I A'' |
refl : A ⊑_I A
```

The refinement of attack sequence allows the expansion of top level abstract attacks into longer sequences. Ultimately, we need to have a notion of when a sufficiently refined sequence of attacks is valid. This notion is provided by the final inductive predicate `is_and_attack_tree`. It integrates the base cases where base attacks can be directly logically derived from corresponding enables properties; it states that an attack sequence is valid if all its constituent attacks are so and it allows to transfer validity to shorter attacks if a refinement exists.

```
inductive
is_and_attack_tree :: [infrastructure, actor, attree] ⇒ bool ("_, _ ⊢ _")
where
att_act: enables I l a a' ⟹ I , a ⊢ 𝒩(Perform(a')) |
att_goto: enables I l a (move) ⟹ I, a ⊢ 𝒩(Goto l) |
att_cred: enables I l a (get) ⟹ I, a ⊢  𝒩(Credential l) |
att_list: ⟦ ∀ a ∈ (set(as)). I, a' ⊢ 𝒩(a) ⟧ ⟹ I, a' ⊢ as ⊕_∧^s a'' |
att_ref: ⟦ A ⊑_I A'; I, a ⊢ A' ⟧ ⟹ I, a ⊢ A
```

The Isabelle/HOL theory library provides a lot of list functions. We can thus simply define the "or" of attack trees by folding the above validity over a list of attacks.

`I, a` $\vdash_{G \oplus_{\vee}}$ `al` $\equiv$ `fold (`λ `x y. (I, a` $\vdash$ `x)` $\vee$ `y) al False`

To validate this formalisation of the attack trees, we now show how the bakery scenario attack can be derived.

First, we prove the following `get_then_move` property.

lemma `get_move_lem: get_then_move Bakery_scenario ''key''`
  `(get_attseq ([Goto (Location 1), Credential (Location 1), Goto (Location 2)]`
  $\oplus_{\wedge}^{get-move}$ `Perform get))`
  `(Goto (Location 2))`

After reducing with the defining rule of `get_then_move` above, proof requires resolving three "enables" subgoals; the final one uses the `add_credential` for `Wife`. This lemma rather immediately implies the following refines property.

`([Goto (Location 2)]` $\oplus_{\wedge}^{get-cake}$ `Perform get)`
$\sqsubseteq_{Bakery-scenario}$
`([Goto (Location 1), Credential (Location 1), Goto (Location 2)]`
 $\oplus_{\wedge}^{get-move}$ `Perform get)`

We can show this refined attack as valid mainly showing that each step in it is valid.

lemma `final_attack: Bakery_scenario, Actor ''Wife''` $\vdash$
`([Goto (Location 1), Credential (Location 1), Goto (Location 2)]`
 $\oplus_{\wedge}^{get-move}$ `Perform get)`

The last lemma together with the refinement gives us finally that the top level abstract attack is a valid attack.

theorem `bakery_attack:`
`Bakery_scenario, Actor ''Wife''` $\vdash$ `([Goto (Location 2)]` $\oplus_{\wedge}^{get-cake}$ `Perform get)`

7 Related Work

System models such as ExASyM [6, 8] and Portunes [19] also model infrastructure and data, and analyse the modelled organisation for possible threats. However, Portunes supports mobility of nodes, instead of processes, and represents the social domain by low-level policies that describe the trust relation between people to model social engineering. Pieters *et al.* consider policy alignment to address different levels of abstraction of socio-technical systems [20], where policies are interpreted as first-order logical theories containing all sequences of actions and expressing the policy as a "distinguished" prefix-closed predicate in these theories. In contrast to their use of refinement for policies we use the security refinement paradox, *i.e.*, security is *not* generally preserved by refinement.

Attack trees [21] specify an attacker's main goal as the root of a tree; this goal is then disjunctively or conjunctively refined into sub-goals until the reached subgoals represent basic actions that correspond to atomic components. Disjunctive refinements represent alternative ways of achieving a goal, whereas conjunctive refinements depict different steps an attacker needs to take in order to achieve a goal. Techniques for the automated generation of attack graphs mostly consider computer networks only [22,23]. While these techniques usually require the specification of atomic attacks, in our approach the attack consists in invalidating a policy, and the model just provides the infrastructure and methods for doing so.

8 Conclusion

Modelling socio-technical systems with formal methods is a difficult undertaking. Due to the unpredictability of human behaviour, formal methods are often too restrictive to capture essential aspects. This results in the human factor often being ignored in these formalisations, since it cannot be represented in the model used.

In this work we have presented different techniques for modelling and analysing systems *including* human factors using recent advances in system models. Our approach supports all kinds of human factors that can be instantiated once an attack has been identified. The presented techniques address different aspects of analysing socio-technical systems. The flow-logic based approach (Section 4) supports analysis of observed actions; this can be compared to an a posteriori analysis to identify what has happened, or in combination with logged information, what might have happened. The attack generation (Section 5) identifies *all possible* attacks with respect to the model; this constitutes an a priori analysis of the modelled system. Finally, the formalisation with Isabelle (Section 6) provides a different view on system models and attacks, and a proof that the contributes the soundness of attack generation.

The attacks generated by the last two techniques include all relevant steps from detecting the required assets, obtaining them as well as any credentials needed to do so, and finally performing actions that are prohibited in the system. The generated attacks are precise enough to illustrate the threat, and they are general enough to hide the details of individual steps. The generated attacks are also complete with respect to the model; whenever an attack is possible in the model, it will be found.

Acknowledgment. Part of the research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318003 (TRE$_S$PASS). This publication reflects only the authors' views and the Union is not liable for any use that may be made of the information contained herein.

References

1. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2004)
2. BBC News: Hack attack causes 'massive damage' at steel works. Available from `http://www.bbc.com/news/technology-30575104`. (2014) Last visited October 15, 2015.

3. Cappelli, D.M., Moore, A.P., Trzeciak, R.F.: The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud). Addison-Wesley Professional (2012)
4. Hunker, J., Probst, C.W.: Insiders and insider threats—an overview of definitions and mitigation techniques. Journal ofWireless Mobile Networks, Ubiquitous Computing, and Dependable Applications **2**(1) (2011) 3–25
5. Nielson, H.R., Nielson, F., Pilegaard, H.: Flow logic for process calculi. ACM Computing Surveys **44**(1) (2012) 3
6. Probst, C.W., Hansen, R.R.: An extensible analysable system model. Information Security Technical Report **13**(4) (2008) 235–246
7. de Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering **24**(5) (May 1998) 315–330
8. Probst, C., Hansen, R., Nielson, F.: Where can an insider attack? In Dimitrakos, T., Martinelli, F., Ryan, P., Schneider, S., eds.: Formal Aspects in Security and Trust. Volume 4691 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 127–142
9. Nielson, H., Nielson, F.: Flow logic: A multi-paradigmatic approach to static analysis. In Mogensen, T.A., Schmidt, D.A., Sudborough, I.H., eds.: The Essence of Computation. Volume 2566 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002) 223–244
10. Kammüller, F., Probst, C.W.: Invalidating policies using structural information. In: Proceedings of the 2nd International IEEE Workshop on Research on Insider Threats (WRIT'13). (May 2013) 76–81
11. Kammüller, F., Probst, C.W.: Combining generated data models with formal invalidation for insider threat analysis. In: Proceedings of the 3rd International IEEE Workshop on Research on Insider Threats (WRIT'14). (May 2014) 229–235
12. Schneier, B.: Secrets and Lies: Digital Security in a Networked World. John Wiley & Sons (2004)
13. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: Dag-based attack and defense modeling: Don't miss the forest for the attack trees. Computer Science Review **13-14** (2014) 1–38
14. Aslanyan, Z., Nielson, F.: Pareto efficient solutions of attack-defence trees. In: Proceedings of the 4th International Conference on Principles of Security and Trust (POST 2015). (2015) 95–114
15. Buldas, A., Lenin, A.: New efficient utility upper bounds for the fully adaptive model of attack trees. In Das, S.K., Nita-Rotaru, C., Kantarcioglu, M., eds.: Proceedings of the 4th International Conference on Decision and Game Theory for Security (GameSec 2013). Volume 8252 of Lecture Notes in Computer Science., Springer (2013) 192–205
16. Vigo, R., Nielson, F., Nielson, H.R.: Automated generation of attack trees. In: Proceedings of the 27th Computer Security Foundations Symposium (CSF), IEEE (2014) 337–350
17. Kammüller, F., Probst, C.W.: Modeling and verification of insider threats using logical analysis. IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence. (2016) Accepted for publication.
18. Kammüller, F.: Isabelle formalisation of an insider threat framework with examples entitled independent and ambitious leader (2015) Available from `https://www.dropbox.com/sh/rx8d09pf31cv8bd/AAALKtaP8HMX642fi04Og4NLa?dl=0`.

19. Dimkov, T.: Alignment of Organizational Security Policies – Theory and Practice. University of Twente (2012)
20. Pieters, W., Dimkov, T., Pavlovic, D.: Security policy alignment: A formal approach. IEEE Systems Journal **7**(2) (2013) 275–287
21. Salter, C., Saydjari, O.S., Schneier, B., Wallner, J.: Toward a secure system engineering methodology. In: Proceedings of the 1998 Workshop on New Security Paradigms (NSPW). (September 1998) 2–10
22. Phillips, C., Swiler, L.P.: A graph-based system for network-vulnerability analysis. In: Proceedings of the 1998 workshop on New security paradigms NSPW 98. Volume pages. (1998) 71–79
23. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02). Volume 129. (2002) 273–284