Abstract Interpretation based Verification of Logic Programs

Giorgio Levi

Dipartimento di Informatica
Università di Pisa

levi@di.unipi.it

http://www.di.unipi.it/di/groups/lp/

Abstract Interpretation

[Patrick and Radhia Cousot, late 70's]

- a general theory for approximating the semantics of discrete dynamic systems
  - the abstract semantics is an approximation of the concrete semantics
  - concrete values are replaced by approximated properties, modeled by an abstract domain
  - hierarchies of semantics
  - definition (and reconstruction) of static analyses (including type systems)
  - useful for
    - proving the correctness of an analysis algorithm
    - deriving optimal abstract semantics from the abstract domain
  - abstract domain design methodologies (e.g. domain refinements) to systematically improve the precision of the domain

Abstract Interpretation and Verification

- abstract interpretation was originally intended as a method for automatically generating program invariants
- we are interested in one specific approach to the generation of abstract interpretation-based partial correctness conditions
- the idea
  - the semantics $[\![P]\!]$ of P is the least fixpoint of a semantic evaluation function T_P on the concrete domain $(\mathbb{D}, \leq)$
  - the class of properties is formalized as an abstract domain $(\mathcal{A}, \sqsubseteq)$, related to $(\mathbb{D}, \leq)$ by a Galois connection $\alpha : \mathbb{D} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathbb{D}$
  - the abstract semantic evaluation function T_P^α is systematically derived from T_P, α and γ
  - the abstract semantics is a correct approximation of the concrete semantics by construction and no additional correctness theorems need to be proved
  - an element $\mathcal{S}$ of the domain $(\mathcal{A}, \sqsubseteq)$ is the specification of the abstraction of the intended concrete semantics
  - the partial correctness of P w.r.t. a specification $\mathcal{S}$ can be expressed as

    $$\alpha([\![P]\!]) \sqsubseteq \mathcal{S}$$

  - a sufficient condition is

    $$T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$$

    * $\mathcal{S}$ is a *pre-fixpoint* of the abstract semantic evaluation function T_P^α

The inductive verification method based on $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$

- inherits the nice features of abstract interpretation
  - we can define a verification framework, parametric with respect to the (abstract) property we want to model
  - given a specific property, the corresponding verification conditions are systematically derived from the framework and guaranteed to be indeed sufficient partial correctness conditions
  - the verification method is complete, if the abstraction is precise (complete according to abstract interpretation theory)
- the method does not require to compute fixpoints

Using abstract interpretation to make $T_P^{\alpha}(\mathcal{S}) \sqsubseteq \mathcal{S}$ effective

- we need
  - a concrete fixpoint semantics, where we can observe the property to verify
  - a finite representation of the intended abstract behaviour (specification)
- abstract interpretation theory provides results and techniques which can be used to tackle the problems below
  - how to systematically design a semantics, which models suitable observable behaviours and exhibits suitable properties related to precision and compositionality
  - how to reconstruct the existing notions of partial correctness and related verification methods (for logic programs) simply in terms of different choices of the concrete semantics
  - how to choose the abstract domain to model the property, so as to make effective the verification method (finite specification)
    - abstract domains developed for static program analysis (types, groundness, etc.)
    - assertions in a suitable specification language

Summary of the talk

- the issue of completeness
- designing semantics by abstract interpretation
- abstract diagnosis
- reconstruction of verification methods
- assertions and specification languages
- extensions and future developments

The issue of completeness

- G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. *Proceedings of PLILP'98*

- given an inductive proof method, if a program is correct with respect to a specification $\mathcal{S}$, the sufficient verification condition might not hold for $\mathcal{S}$
- if the method is complete, then, when the program is correct with respect to $\mathcal{S}$, there exists a property $\mathcal{R}$, stronger than $\mathcal{S}$, which verifies the condition
- the method derived from condition $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$ is complete if and only if the abstraction is precise with respect to T_P, that is if $\alpha(lfp(T_P)) = lfp(T_P^\alpha)$
- an easier to prove sufficient condition for precision is full precision, that is $\alpha \circ T_P = T_P^\alpha \circ \alpha$
  - there exist methods to systematically enrich a domain of properties so as to obtain an abstraction which is fully precise

Designing semantics by abstract interpretation: the concrete semantics

- M. Comini and M.C. Meo. Compositionality properties of *SLD*-derivations. *Theoretical Computer Science*, 1999.

- a framework based on abstract interpretation
- the concrete semantics models SLD-trees and is formalized both denotationally and operationally
  - $\mathcal{B}[\![\boldsymbol{G}\, in\, P]\!]$ is the operational semantics of goal G in program P (the set of its SLD-derivations)
  - T_P is the (denotational) semantics evaluation function for a set of definite clauses P
  - $[\![P]\!] = lfp(T_P)$ is the denotational semantics of P
  - $\mathcal{Q}[\![\boldsymbol{G}\ in\ P]\!]$ is the denotational derivation of the semantics of the goal G from $[\![P]\!]$
- properties of the concrete semantics
  - equivalence between the operational semantics and the denotational semantics
    $\forall G, P.\, \mathcal{B}[\![\boldsymbol{G}\ in\ P]\!] = \mathcal{Q}[\![\boldsymbol{G}\ in\ P]\!]$
  - existence of an AND-compositional goal-independent denotation $[\![P]\!]$
    - we can precisely derive, by means of $\mathcal{Q}$, the behaviour of every (conjunctive) goal from $[\![P]\!]$

The abstraction framework

- M. Comini, G. Levi and M.C. Meo. A Theory of Observables for Logic Programs. *Information and Computation*, to appear

- an observable is a Galois insertion between the domain of SLD-trees and an abstract domain describing the property to be modeled
- the abstract denotational semantic functions T_P^α and $\mathcal{Q}^\alpha$, together with the abstract denotational semantics $[\![P]\!]^\alpha = lfp(T_P^\alpha)$, are systematically derived from the concrete ones, by replacing the concrete semantic operators by their optimal abstract versions
- the taxonomy of classes of observables
  - an observable belongs to a class if it satisfies a set of conditions relating the concrete semantic operators and the Galois insertion
  - for observables belonging to a given class, we know how to automatically derive the "best" semantics and which are the properties of such a semantics (precision, relation between concrete operational semantics and abstract denotational semantics, existence of a goal-independent denotation, compositionality)

Denotational and semi-denotational observables

- denotational observables
  - the optimal abstract semantics can be defined denotationally, by taking the optimal abstract version T_P^α of T_P
  - precise abstract denotational semantics
    $[\![P]\!]^\alpha = \alpha([\![P]\!])$
  - AND-compositional abstract (goal-independent) denotation
    $\forall G, P. \alpha(\mathcal{B}[\![\boldsymbol{G}\ in\ P]\!]) = \mathcal{Q}^\alpha[\![\boldsymbol{G}\ in\ P]\!]$
  - includes correct answers substitutions, computed answer substitutions, call patterns and resultants
  - denotational observables are precise and lead to complete verification methods
- semi-denotational observables
  - intended to model the properties useful for static program analysis
  - we just loose precision to achieve termination
    $\alpha([\![P]\!]) \sqsubseteq [\![P]\!]^\alpha$
  - AND-compositionality guarantees that we can be as precise as possible, when using denotational definitions
  - the (abstract) semantics for any goal G computed denotationally is as precise as the operational one, i.e.
    $\forall G, P. \mathcal{B}^\alpha[\![\boldsymbol{G}\ in\ P]\!] = \mathcal{Q}^\alpha[\![\boldsymbol{G}\ in\ P]\!]$
  - includes the domain $depth(k)$, the domain $\mathcal{POS}$ for groundness analysis and other optimal domains, designed by using the refinement operators

Abstract diagnosis of logic programs

- M. Comini, G. Levi, M.C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 1999

- extends declarative debugging to a debugging framework parametric w.r.t. the abstraction
- uses the sufficient condition $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$ to prove partial correctness (and to detect incorrectness bugs)
- the specification $\mathcal{S}$ handles properties which are abstractions of computed answers
  - $\mathcal{S}$ can be viewed as a postcondition
- a similar approach where different approximations (modeled by abstract interpretation) can be used in the semantics and in the specification
  - F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynsky, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. *Proceedings of AADEBUG'97*
- from abstract diagnosis to verification
  - more general specifications, including preconditions

A verification framework

- G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. *Proceedings of PLILP'98*

- there exist a lot of verification methods for logic programs, defined by using ad-hoc constructions
- they can be reconstructed as instances of the framework, by simply choosing different observables (abstractions of the *SLD*-derivations traces semantics)
- their reconstruction in terms of abstract interpretation allows us to compare the different techniques and to show the essential differences
- the approach can be explained in terms of two steps, both modeled by abstractions
  1. abstraction needed to derive the semantics which models the proof method
  2. abstraction needed to model a specific class of properties
     - to get finite specifications

Reconstruction of some verification methods

- success-correctness
  - post-conditions only
  - the observable is computed answers (s-semantics)
    - the method is complete, since the observable is precise
  - if we take, in the second abstraction step, properties closed under instantiation (as in Drabent's (1997) method), the verification condition boils down to the one of Clark's (1979) and Deransart's (1993) methods
- I/O correctness
  - specifications are pairs of pre- and post-conditions (the post-condition holds whenever the pre-condition is satisfied)
  - the observable is a simple modification of computed answers
    - the method is complete, since the observable is precise

Stronger preconditions: call correctness

- I/O and call correctness
  - specifications are still pairs of pre- and post-conditions (the pre-conditions are satisfied by all the procedure calls)
  - the observable is call patterns
    - the method is complete, since the observable is precise
  - the verification condition, obtained by unfolding $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$ with the call patterns T_P^α, is a slight generalization of the one defined by the Drabent-Maluszynski method (1988)
    - by taking, in the second abstraction step, properties closed under instantiation, we reconstruct the Bossi-Cocco condition (1989)
    - by further abstractions (modes, types, etc.), we reconstruct the hierarchy of verification conditions defined by Apt-Marchiori (1994)

Towards finite specifications: the second abstraction step

- the second abstraction is concerned with the choice of an abstract domain to model the property
- there exist two possibilities
  1. abstract domains designed for static analysis ($depth(k)$, modes, types, groundness dependencies, etc.)
     - we can reason on the domain of properties by efficient abstract computation steps and prove the sufficient condition by using the operations on the abstract cpo
     - as in static analysis, in general we loose precision (and therefore the completeness of the verification method)
     - we succeed in getting finite specifications
     - if we model the property by semi-denotational observables, even if the method is not complete, the abstract T_P^α, used in condition $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$, introduces the same amount of approximation of the "best" goal-dependent abstract operational semantics
  2. properties can be specified as assertions in a suitable specification language

Assertions and specification languages

- specifications are usually given by means of assertions, i.e. formulas in a suitable formal specification language
  - in our case a specification is (extensionally defined as) the intended abstract semantics
- assertions can be viewed as an abstract domain, as shown by the Cousot's in the early papers on abstract interpretation
- we show the case of success-correctness only, where the concrete domain consists of sets of atoms
  - similar constructions can be given for the other notions of correctness

Success correctness with assertions

- G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. *Proceedings of PLILP'98*

- a first order language $\mathcal{L} = \langle \Sigma, \Pi, V \rangle$
  - the signature of $\mathcal{L}$ includes functions, constants and variables occurring in the program
- a term-interpretation $\mathcal{I} = \langle Terms(\Sigma, V), \Sigma_{\mathcal{I}}, \Pi_{\mathcal{I}} \rangle$
  - the set of non-ground terms viewed as an $\mathcal{L}$ structure
- $\mathcal{F}$ is a set of formulas of $\mathcal{L}$, which describe properties of predicate arguments (*assertions*)
- an atom $p(t_1, \ldots, t_n)$ *satisfies* the formula $\Phi[x_1, \ldots, x_n]$ of $\mathcal{F}$, iff for each σ,

$$\mathcal{I} \models_{\sigma[x_1,\ldots,x_n \setminus t_1,\ldots,t_n]} \Phi[x_1, \ldots, x_n]$$

- the set of assertions $\{\Theta_p\}_{p \in Pred}$ associates an assertion Θ_p to each predicate p of P
- P is *success-correct* with respect to $\{\Theta_p\}_{p \in Pred}$ iff $\forall p(t) \in Atoms$, $p(t) \overset{\theta}{\rightsquigarrow} \square$ implies that $p(t)\theta$ satisfies Θ_p

Assertions as abstract domains

- a partial order on $\mathcal{F}$ induced by implication under the interpretation $\mathcal{I}$

  $$\Theta \preceq_{\mathcal{I}} \Phi \text{ iff } \mathcal{I} \models \Theta \Rightarrow \Phi$$

- a domain

  $$\mathcal{A}_{\mathcal{I}} = (Pred \rightarrow \mathcal{F}/_{\equiv_{\mathcal{I}}}, \leq)$$

  whose elements can be represented as sets $\{\Theta_p\}_{p \in Pred}$, where each Θ_p is a formula of $\mathcal{F}$ with free variables corresponding to arguments of p

  - the order is the pointwise extension of the order between formulas of $\mathcal{F}$

- a function from $\mathcal{A}_{\mathcal{I}}$ to $\mathcal{C}$:

  $$\gamma_{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}).\{p(\mathbf{t}) \in Atoms \mid p(\mathbf{t}) \textit{ satisfies } \Theta_p\}.$$

  - if $\mathcal{F}$ is a complete lattice closed under conjunction, $\gamma_{\mathcal{I}}$ is meet-additive
  - by standard abstract interpretation results, it induces a Galois connection between $\mathcal{C}$ and $\mathcal{A}$

A sufficient condition for success-correctness

- the optimal abstraction on $\mathcal{A}_{\mathcal{I}}$

$$T_P^{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}).\bigvee_{c\in P}\bigwedge\{\Phi \mid \mathcal{I} \models \bigwedge_{i=1}^{n}\Theta_{p_i}[\mathbf{x}_i\backslash\mathbf{t}_i] \Rightarrow \Phi[\mathbf{x}\backslash\mathbf{t}]\}$$

$$c = p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$$

- P is *success-correct* with respect to $\{\Theta_p\}_{p\in Pred}$, iff

$$\alpha_{\mathcal{I}}(lfp(T_P^{\mathcal{C}})) \leq \Theta$$

where $\alpha_{\mathcal{I}}$ is the adjoint function of $\gamma_{\mathcal{I}}$

- by unfolding condition $T_P^{\mathcal{I}}(\Theta) \leq \Theta$, we obtain a sufficient condition
  P is success-correct with respect to $\{\Phi_p\}_{p\in Pred}$ if, for each clause

$$p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$$

the following condition holds

$$\mathcal{I} \models \bigwedge_{i=1}^{n}\Phi_{p_i}[\mathbf{x}_i\backslash\mathbf{t}_i] \Rightarrow \Phi_p[\mathbf{x}\backslash\mathbf{t}]$$

  - Deransart's verification method with assertions

Verification with assertions

- if the relation $\models$ is decidable, we have an effective way to prove the sufficient condition
  - examples of decidable specification languages for logic programs
    - a language of properties by Marchiori (1996), which allows us to express groundness, freeness and sharing of terms
    - a decidable extension, including polymorphic types, by Volpe (1998)
- as in the case of extensional properties, the method is complete if the abstraction is precise
- an interesting alternative specification language:
  - logic programs as specifications
  - the verification conditions can often be proved by standard program transformation techniques
    - folding, unfolding, goal replacement

Example 1: a decidable assertion language

the clause

```
sort(Xs,Ys) :- perm(Xs,Ys), ord(Ys).
```

the precondition

$$sort(X,Y) \mapsto list(X) \wedge ground(X)$$
$$perm(X,Y) \mapsto list(X) \wedge ground(X)$$
$$ord(X) \mapsto list(X) \wedge ground(X)$$

the postcondition

$$sort(X,Y) \mapsto list(Y) \wedge ground(Y)$$
$$perm(X,Y) \mapsto list(Y) \wedge ground(Y)$$
$$ord(X) \mapsto \mathbf{true}$$

the verification condition (call correctness)

$$list(Xs) \wedge ground(Xs) \Rightarrow list(Xs) \wedge ground(Xs)$$
$$list(Xs) \wedge ground(Xs) \wedge list(Ys) \wedge ground(Ys) \Rightarrow$$
$$list(Ys) \wedge ground(Ys)$$
$$list(Xs) \wedge ground(Xs) \wedge list(Ys) \wedge ground(Ys) \wedge \mathbf{true} \Rightarrow$$
$$list(Ys) \wedge ground(Ys)$$

Example 2: LP description of a reactive system

```
c1:  e00( [ (null,null) | X] ) :- e00( X ).
c2:  e00( [ (10,null) | X] ) :- e10( X ).
c3:  e00( [ (water,beep) | X] ) :- e00( X ).
c4:  e00( [ (coffee,beep) | X] ) :- e00( X ).

c5:  e10( [ (null,null) | X] ) :- e10( X ).
c6:  e10( [ (10,null) | X] ) :- e20( X ).
c7:  e10( [ (water,water) | X] ) :- e00( X ).
c8:  e10( [ (coffee,beep) | X] ) :- e10( X ).

c9:  e20( [ (null,null) | X] ) :- e20( X ).
cA:  e20( [ (water,water) | X] ) :- e10( X ).
cB:  e20( [ (coffee,coffee) | X] ) :- e00( X ).
cC:  e20( [ (coffee,null) | X] ) :- warm( X ).

cE:  warm( [ (null,coffee) | X] ) :- e00( X ).
```

a vending machine

Assertion language, specifications, verification conditions

```
sublist(Xs,Ys) :- sublistX(Xs,Ys).
sublist(Xs,[Y|Ys]) :- sublist(Xs,Ys).

sublistX([],Xs).
sublistX([Y|Xs],[Y|Ys]) :- sublistX(Xs,Ys).

match(Xs,X,Ys) :- matchX(Xs,X,Ys).
match(Xs,X,[Y|Ys]) :- match(Xs,X,Ys).

matchX([],X,[X|_]).
matchX([],X,[Y|Ys]) :- matchX([],X,Ys).
matchX([Y|Xs],X,[Y|Ys]) :- matchX(Xs,X,Ys).
```

$$e00(X) \mapsto \texttt{sublist}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], X)$$
$$e10(X) \mapsto \texttt{sublistX}([(\texttt{10}, \_), (\texttt{coffee}, \_)], X)$$
$$e20(X) \mapsto \texttt{sublistX}([(\texttt{coffee}, \_)], X)$$
$$warm(X) \mapsto TRUE$$

$$e00(X) \mapsto \texttt{match}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), X)$$
$$e10(X) \mapsto \texttt{matchX}([(\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), X)$$
$$e20(X) \mapsto \texttt{matchX}([(\texttt{coffee}, \_)], (\_, \texttt{coffee}), X)$$
$$warm(X) \mapsto \texttt{matchX}([\,], (\_, \texttt{coffee}), X)$$

I/O correctness verification conditions

$$\texttt{sublist}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], [(\texttt{10}, \texttt{null})|X]) \wedge$$
$$\texttt{matchX}([(\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), X) \Longrightarrow$$
$$\texttt{match}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), [(\texttt{10}, \texttt{null})|X])$$

Example 3: algorithm vs. specification

the insertion sort, is correct w.r.t. a specification given by the declarative (inefficient) specification of sort

```
c1:  isort([],[]).
c2:  isort([X|Xs],Ys) :- isort(Xs,Zs), insert(X,Zs,Ys).

c3:  insert(X,[],[X]).
c4:  insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).
c5:  insert(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

Assertion language, specifications, verification conditions

```
sort(Xs,Ys) :- perm(Xs,Ys), ord(Ys).

ord([]).
ord([X]).
ord([X,Y|Xs]) :- X =< Y, ord([Y|Xs]).

perm(Xs,[Z|Zs]) :- select(Z,Xs,Ys), perm(Ys,Zs).
perm([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Zs]) :- select(X,Xs,Zs).

intlist([]).
intlist([X|Xs]) :- integer(X), intlist(Xs).
```

$$isort(X,Y) \mapsto \texttt{intlist}(X)$$
$$insert(X,Y,Z) \mapsto int(X) \wedge \texttt{intlist}(Y) \wedge \texttt{ord}(Y)$$

$$isort(X,Y) \mapsto \texttt{intlist}(Y) \wedge \texttt{sort}(X,Y)$$
$$insert(X,Y,Z) \mapsto \texttt{intlist}(Z) \wedge \texttt{sort}([X|Y],Z)$$

a verification condition (call correctness)

$$\texttt{intlist}([X|Xs]) \wedge \texttt{intlist}(Zs) \wedge \texttt{sort}(Xs,Zs) \wedge \texttt{intlist}(Ys) \wedge$$
$$\texttt{sort}([X|Zs],Ys) \Longrightarrow \texttt{intlist}(Ys) \wedge \texttt{sort}([X|Xs],Ys)$$

which can be proved by first proving a property of **perm**, i.e.,
$$\texttt{perm}(Xs,Zs) \wedge \texttt{perm}([X|Zs],Ys) \Longleftrightarrow \texttt{perm}([X|Xs],Ys)$$

Discussion on verification with assertions

- some open problems related to assertion abstract domains
  - how to define more expressive (decidable) specification languages
  - the precision issue (specification language and assertions)
  - how to compare (from the viewpoint of precision) standard "static analysis" abstract domains to assertions
  - how to define refinement operators for domains defined by assertions
  - how to effectively derive, for a given assertion language, the abstraction function $\alpha_{\mathcal{I}}$, which is needed to design the optimal semantic operators
    - this would lead to a notion of abstract execution on the domain of assertions

Extension to other logic languages and observables

- the semantic framework for definite logic programs based on abstract interpretation has been extended to other languages
  - Prolog (selection and search rule, cut, a large set of primitives)
    - F. Spoto. Operational and Goal-Independent Denotational Semantics for Prolog with Cut. *Journal of Logic Programming*, to appear
    - F. Spoto and G. Levi. Abstract Interpretation of Prolog Programs. *Proc. AMAST'98*
  - Concurrent Constraint Programming
    - R. Moreno. Abstracting Properties in Concurrent Constraint Programming. *Proc. LPAR'99*
  - Hereditary Harrop Formulas (first-order subset of λ-Prolog)
- it is staightforward to adapt the verification techniques to the above languages
- both the semantic framework and an interesting extension of the verification techniques have been defined to model finite failure and infinite derivations in definite logic programs
  - R. Gori and G. Levi. On the verification of finite failure. *Proc. PPDP'99*

Future developments

- our approach to verification can be applied to any language once we have a fixpoint semantics adequate for the proof method

- there exist several useful semantic frameworks for various languages, developed as a foundation for static program analysis

- an interesting area we have just started to look at is the verification of security related properties for mobile systems