Formal Software Development in the Verification Support Environment (VSE)

Dieter Hutter Georg Rock Jörg H. Siekmann Werner Stephan Roland Vogt
Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
[German Research Center for Artificial Intelligence]
Stuhlsatzenhausweg 3
D-66123 Saarbrücken
{hutter,rock,siekmann,stephan,vogt}@dfki.de

Abstract

The paper presents a survey of the VSE system, a kind of CASE-tool for *formal* software development. It is a summary of a tutorial presentation describing methodology, formalisms, architecture, and proof support of the system. For illustration a commercial application from the IT-security domain is used.

Introduction

On their way of becoming a mature discipline formal methods have created a rich variety of logical formalisms, methodological approaches, and tools. While there seems to be a certain convergence with respect to formalisms we are faced with a rapidly growing number of different approaches to apply formal methods in special scenarios. At least for non-experts the situation is even more confusing with respect to tool support as there are tools for editing, (type-) checking, visualizing, and animating specifications, for validation by testing or model checking, and for interactive proof generation in various contexts.

As far as validation is concerned the main dichotomy today is between logical inference (or deduction) and model checking. In the logical (or axiomatic) approach mathematical models are described by statements of a logical language and conclusions are drawn by using some (implemented) inference mechanism. Typically in this area proof generation requires some kind of user-interaction.

Model-checker start with a system description and analyze properties expressed in a logical language by an exhaustive search through the state space. In particular if the system specification is given by a state transition system the (explicit) use of logic is reduced to a minimum. Model-checking is restricted to (practically) finite systems and mainly used for validation. Axiomatic approaches are more general in two aspects, they are applicable to all kinds of systems and cover the whole development process.

The Verification Support Environment (VSE) tool provides a fairly *general* methodology based on the axiomatic approach and offers comprehensive support and safety by *integrating* several services, in particular a powerful deduction component. It can perhaps best be viewed as a CASE-tool for formal software development.

VSE was developed in two phases for the German Information Security Agency (GISA) by consortia from industry and academia. The tutorial presents an introduction to VSE-II, the new version of the system. Although the system was released in February 1999 it was already used in a number of commercial projects at the time the tutorial was written. One of these, a security model for chip cards, will be used as a running example in this presentation.

In the next section we give an overview of the VSE methodology and the basic architecture of the system. After that there will be a closer look at the formalisms used in VSE.

The VSE Methodology

Like in conventional software development we have to distinguish between the requirements phase and the design phase. Formal software development starts with the construction of an abstract system model according to a requirements specification. The system model (given as an axiomatic theory) provides the basic terminology and the abstract solution. It does not necessarily reflect the architecture of the system to be developed. Strictly speaking validation in VSE comprises both the design of the requirements specification and the proof that the system model actually *satisfies* the given requirements. In practice the abstract system model and the requirements specification are developed hand in hand.

Following general engineering principles there is a strong emphasis on modularity as the main means to cope with complexity. Due to the additional effort of carrying out proofs the request for modularity is even stronger than in software engineering in general. So at least the abstract systems specification will be structured into sub-specifications ideally being as independent from each other as possible. Most proofs arising from the requirements are then carried out locally to these sub-specifications. The formal background of operators depend on the formalism under consideration. In many cases also the requirement specification will be composed out of several parts. There might be even proofs necessary to show, for example, that a certain requirement is sufficient to establish a more general property.

The initial solution represented by the abstract system model is the starting point of the *refinement process*. In each refinement step a specification is related to a more concrete one. The underlying theory gives rise to proof obligations that guarantee the correctness of the refinement step. Typically refinement steps involve a mapping that relates abstract notions to the concrete ones that are used in the refined system. For both formalisms supported by VSE there are *programming notions* that allow for the introduction of constructive solutions.

Formal development in VSE follows the invent-and-verify paradigm, that is the more concrete ones and also the mappings that constitute refinement steps are given by the user. Although this approach is more coarse grained than those based on transformation or synthesis it is far from being a post-mortem verification: In order to manage complexity independent sub-specifications are refined independently, and, there are several intermediate layers between the abstract specification and the level where all specifications are (efficiently) executable and code can be generated by the system.

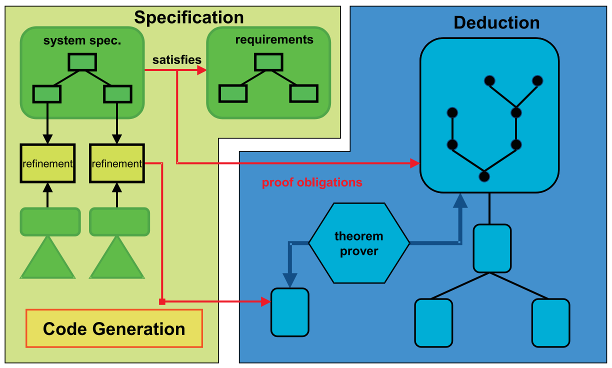

Figure 1: The VSE Methodology

Formalisms in VSE

Formal development in VSE is based on two formalisms: *abstract data types* are used to specify data structures and functional computations while a version of *temporal logic* is used to specify the dynamic behavior of systems with a persistent state. Although there is a fully developed methodology in its own right for abstract data types typically data types are used to provide values for state dependent (flexible) variables in state based systems. Functional computations are then used to model single (uninterruptable) steps of state based systems.

Abstract Data Types

Formal specification techniques treat data objects as mathematical objects of a certain domain. To get rid of the technical details of data types in real programming languages one either considers a single rich domain as it is done in Z, (Spivey 1992), or one abstracts away from incidental properties of a particular domain by considering whole *classes* of structures as it is done in the abstract data type approach, (Loeckx, Ehrich, & Wolf 1996; Astesiano, Kreowski, & Krieg-Brückner 1999). Here data objects are viewed as resulting from the nested application of certain functions, a (concrete) data typ being given by a collection of domains (carriers) and functions on these domains. To introduce abstraction one separates syntax from semantics and considers classes of *algebras* (or models) $\mathcal{A}$ which are interpretations of a fixed collection Σ of function *symbols* f as functions $f_{\mathcal{A}}$ on the carriers of $\mathcal{A}$. Classes of algebras are restricted by axioms of a logical language (over Σ) which usually is a sublanguage of first-order predicate logic. The various approaches to abstract data types differ in the classes of algebras that are considered and the corresponding description techniques that are used for specification.

In VSE full first-order logic is used to specify data types. In general all models $\mathcal{A}$ satisfying the axioms Ax, written $\mathcal{A} \models Ax$, are considered. Two models $\mathcal{A}_1$ and $\mathcal{A}_2$ of Ax will not necessarily be isomorphic, that is they differ not only in the concrete representation of data objects. This allows for a really abstract style of specification where one describes *what* a function does but not *how* this is realized. A well known example is encoding and decoding. On the abstract level it might suffice to know that $dec(enc(v)) = v$ leaving open a wide range of perhaps highly sophisticated implementations for later refinements.

But VSE also supports a more constructive style of specification, that allows to introduce recursive data structures like lists and trees. Classes of algebras are restricted by requiring that certain carriers are *generated* by constructors from some $\Sigma' \subset \Sigma$ which means that for each element a of the corresponding carrier A, there is a term τ over Σ' that denotes a, i.e. $a = [\![\tau]\!]$. In particular we consider *freely* generated structures where each element $a \in A$ has a *unique* representation in Σ'. Fig. 2 shows the specification of lists resp. the enrichment of an arbitrary data type with a single additional element. Generated clauses bring about *induction principles* that have to be used if inductive theorems or lemmata have to be proven. The axiomatic counterpart of these clauses is generated by the system when the deduction unit belonging to the specification is generated. In section techniques for inductive theorem proving are discussed.

```
THEORY TAny
  TYPES Any
THEORYEND

BASIC TMaybe
  PARAMS TAny
  Maybe = nothing           WITH isNothing
        | box(unbox: Any) WITH isBox
BASICEND

BASIC TList
  PARAMS TAny
  List = nil
       | cons(first: Any, rest: List)
BASICEND
```

Figure 2: Freely Generated Data Types

So far we have discussed elementary (unstructured) specifications. In VSE there are two ways of structuring data type specifications: *generic* specifications and the *import* of specifications. By importing (using) possibly several specifications *enrichment* and (disjoint) *union* can be modeled in VSE. Generic specifications provide an additional slot (parameter part) to describe the formal parameter including axioms. Upon actualization of a generic theory as part of the using slot of some other theory proof obligations are generated for these axioms. Fig. 2 is also an example for a parametrised specification. The parameter theory `TAny` contains a single definition for a type `Any`. So `TList` resp. `TMaybe` can be instantiated to form lists of elements of any kind resp. to enrich any data type with an additional (error) value.

Structured theories are not flattened in VSE. Each specification is an entity in its own right and linked to other specifications according to the used specification building operation. The semantic counterparts of these operations determine the translation into logical formulas (renaming) and also the flow of information between the units corresponding to the specification entities.

VSE implements an elaborated theory of data type refinements, (Reif 1992). Operations of an (abstract) export algebra are implemented by programs that use operations from some (more concrete) import algebra. The axioms of the export specification give raise to proof obligations that are assertions about the implementing programs. Properties of the import specifications are used in the course of verifying the assertions in a programming logic.

State-Based Systems

State-based systems are used to model reactive and concurrent systems. Their semantics is given by *behaviors*, i. e. infinite sequences of states ($\bar{s} = s_0, s_1, s_2, \ldots$), where a state $s \in S$ is a valuation of *flexible* variables $x \in X$, $s(x)$ being an element of a carrier of an abstract data type. Therefore specifications of state-based systems import (use) theories.

The basic ideas of the VSE approach to state-based systems are taken from TLA the Temporal Logic of Actions, (Lamport 1994a). System steps are specified by a collection of so called *actions* which are first-order formulas that in addition to logical (rigid) variables contain primed and unprimed flexible variables. Both, in syntax and semantics there is a straightforward relation to (basic) Z-schemata. Actions A denote (binary) relations on the set of states, $[\![A]\!] \subset S \times S$. Like in Z (flexible) variables not mentioned in A change arbitrarily.

Temporal formulae φ describe sets of behaviors by using the (temporal) operators $\Box$ (always), $\Diamond$ (eventually), and **unless** which are interpreted in the usual way, (Manna & Pnueli 1991). The basic technique of specifying systems which needs some modifications discussed below uses formulas of the form $\phi_{init} \wedge \Box(A_1 \vee \ldots \vee A_n)$. The first order-formula ϕ_{init} defines the *initial states* while the actions $A_1 \vee \ldots \vee A_n$ describe the possible state transitions. Clearly $\bar{s} \in [\![\Box(A_1 \vee \ldots \vee A_n)]\!]$ iff for all i there exists a j, $1 \leq j \leq n$, such that $s_i[\![A_j]\!]s_{i+1}$.

State-based systems can be structured into parallel components. In simple cases we distinguish between *input* variables, *output* variables, and variables *local* to a component C. Input variables i are read-only, that is there must be no primed occurrences of i in the action definitions of C. Two components C_1 and C_2 are composed by connecting (certain) input variables and output variables. Since we want to model parallel composition by conjunction the specifications of C_1 and C_2 have to be modified. Both have to allow *stuttering* with respect to certain variables. $\Box(A_1 \vee \ldots \vee A_n)$ is changed to $\Box(A_1 \vee \ldots \vee A_n \vee (x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n))$ abbreviated by $\Box[A_1 \vee \ldots \vee A_n]_{\bar{x}}$, where $\bar{x}$ (typically) consists of the output variables and local variables of a component. Due to this modification system behaviors are always invariant under stuttering which means that the $\circ$ (next operator) makes no sense in this context.

Having introduced stuttering it can be guaranteed that with C_1 and C_2 also $C_1 \wedge C_2$ is consistent and defines the parallel composition of both components. Adding the additional constraint (as part of the description of the combined system) that output variables of C_1 and C_2 are not changed simultaneously, we obtain *interleaved* parallel executions.

If communication via *shared* variables is allowed, steps of parallel components have to be "colored" in order to preserve consistency. Steps of the component itself are distinguished from steps of the "environment" by using the predicate *active*.

Parallel composition is the main means to structure state-based specifications. As in the case of abstract data types there is no flattening, that is proofs are carried out local to the components of a combined system. However, in almost all cases we need *assumptions* about the environment of the component under consideration. VSE provides a correctness management that rules out (unsound) circular reasoning in using assumptions and also special deduction techniques for assumption guarantee proofs. These are based on the coloring of steps and applicable not only for safety properties ($\Box\phi$) but also for reactivity ($\Box\Diamond\phi$).

In most developments one wants the local variables of a system not to be visible to the outside world. *Hiding* is done by existential quantification on flexible variables. Roughly speaking one can say that x changes arbitrarily in all computations of $\exists x.\varphi[x]$ independently of φ.

Finally, like in most other temporal approaches specifications have to impose *fairness* constraints on the possible behaviors of a system. VSE supports the notions of *strong* and *weak fairness*. The latter means that an action A cannot be continously enabled (ready to execute) without being taken, which is expressed by the formula $\Box\Diamond \mathit{taken}(A) \vee \Box\Diamond\neg \mathit{enabled}(A)$.

A simple imperative programming language allows to model the sequential flow of control in basic components (cf. Sec.). Using a local flexible variable for *labels* these programs are translated automatically into action definitions including suitable fairness constraints.

Refinement is logically modeled by implication between temporal specifications. Stuttering steps in the abstract model allow single steps to be replaced by whole computations in the more concrete design. To treat hidden variables a refinement mapping is used during the refinement proof. For parallel components being refined separately we again need assumptions about the environment. In this case these are restricted to safety properties.

Since both, refinement and the satisfies relation (between system specification and safety/security requirements) is modeled by implication complex security models can be given by a structured specification of it's own and related to the abstract system model by refining one of the components. This is done in our chip-card example. The security model specifies the admissible traces using a state machine and also formalizes the knowledge obtained by an "observer". Fig. 3 shows a small part of the security model.

```
TLSPEC Objectives
  USING TSigCard
  DATA
    OUT
      oChannel: MInfo.Maybe
    IN
      iChannel: MInfo.Maybe
    INTERNAL
      authUser:  bool;
      secretKey: MInfo.Maybe
  VARS i : Info
  SPEC
    [] secretKey = secretKey';
    [] NOT inferable(oChannel,secretKey);
    [] ALL i:
       ( inferable(oChannel,sig(i,secretKey))
         -> authUser =  t )
  HIDE iChannel, authUser, secretKey
TLSPECEND
```

Figure 3: Excerpt of a state machine specification

Use Case: Digital Signatures Devices

The German digital signature act (Signaturgesetz – SigG) resp. the accompanying digital signature ordinance (Signaturverordnung – SigV) prescribes the certification of all related technical components according to standardized security criteria. Concretely, an evaluation based on the European Information Technology Security Evaluation Criteria (ITSEC) is demanded. In order to meet SigG conformance requirements, every signature component has to reach evaluation level E4 of ITSEC. In particular, this requires a formal security policy model which declaratively describes the important security features an IT product has to fulfill at an abstract level.

The main goal of our use case is the development of a formal security policy model for SmartCards supposedly being in conformity with SigG. The formal modeling is serves as a general reference and is publicly available for the development of products according to ITSEC level E4 and above. In this paper we are presenting a very small and incomplete excerpt of our formal model for digital signature devices (SigCards).

The idea underlying of the SigCard technology is to provide evidence that no one can generate a digital signature except the legal card-holder. Speaking in technical terms this means that a signature key is stored on the SigCard which must be kept strictly confidential. There should be no way to infer the secret key from any output generated by the SigCard.

```
THEORY TFundamental
  TYPES
    Info; Subject;
    Bucks = FREELY GENERATED BY
      buckAutomaton | buckInput | buckOutput
  FUNCTIONS
    id: Subject -> Subject;
    learns: Subject, Info -> Subject;
    encode: Info, Info -> Info;
    decode: Info, Info -> Info
  PREDICATES
    knows : Subject, Info;
    inferable: Info, Info
  VARS
    i,k: Info;
    s: Subject
  AXIOMS
    id(s) = id(learns(s,i));
    knows(learns(s,i),i);
    inferable(i,k) <->
      EX s: knows(learns(s,i),k) AND
            NOT knows(s,k)
THEORYEND
```

Figure 4: Fundamental data types

In order to be able to formally specify the security policy of SigCards we have to start with some fundamental definitions. In Fig. 4 we introduce the abstract data types `Information` and `Subject`. An element of type `Subject` is equipped with a certain amount of knowledge (elements of type `Info`) which is indicated by the predicate `knows`. Subjects can change by learning (function `learns`) some new piece of information. The incomplete collection of axioms shown in Fig. 4 states that

- the `identity` of a subject is independent of its knowledge,
- a subjects knowledge base contains all information previously learned, and
- the notion of information extraction relates two pieces of information by the existence of a subject being able to infer one from the other.

Finally, the theory `TFundamental` introduces a certain number of tokens of type `Bucks` needed later on to express the flow of control between otherwise independent state machines as well as two functions needed for establishing confidential communication channels.

Since we want to reason about the input and output channels of a SigCard we have to distinguish between two principal states: Either the channel contains a specific piece of information or it contains nothing meaningful. This observation implies that `nothing` cannot be of type `Info`. Therefore, we enrich the theory `TFundamental` with the help of the type `Maybe` (cf. Fig. 2). In the resulting theory `TSigCard` (cf. Fig. 5) we extend the fundamental functions and

```
THEORY TSigCard
  USING TFundamental;
    MInfo = TMaybe[Info]
  FUNCTIONS
    noInfo: MInfo.Maybe;
    noKey:  MInfo.Maybe;
    learns: Subject, MInfo.Maybe -> Subject
  PREDICATES
    knows: Subject, MInfo.Maybe
  VARS i,j: MInfo.Maybe;
       s:   Subject
  AXIOMS
    FOR noInfo: noInfo = MInfo.nothing
    FOR noKey:  noKey  = MInfo.nothing
    FOR knows: DEFPRED
    knows(s,i) <->
      SWITCH i IN
      CASE nothing: TRUE
      CASE box:
        TFundamental.knows(s,unbox(i))
      NI
    FOR learns: DEFFUNC
    learns(s,i) =
      SWITCH i IN
      CASE nothing: s
      CASE box:
        TFundamental.learns(s,unbox(i))
      NI
THEORYEND
```

Figure 5: Data type enrichment

predicates operating on type `Info` to the additional value `nothing`.

Now, we are well prepared to abstractly specify the behavior of a SigCard as a collection of cooperating state machines. The central component of the formal security policy model is the specification of the `Automaton` component. It represents the principle operating states of the SigCard and maintains state changes by appropriately reacting to all security related events. Although this temporal logic specification is the most important part of our use case concerning the security aspect, it is of minor interest for illustrating the concepts of our specification language. Therefore, we omit it here and concentrate on two specifications handling the input resp. output channels of the SigCard.

The purpose of the input stream component is to model the input channel as a part of the SigCard interface (cf. Fig. 6). It specifies a state machine that endlessly performs a loop consisting of two alternating actions. Since there is no hand shaking in the communication protocol we have to distinguish different inputs by requiring a certain gap between incoming pieces of information. The action `waitForNoInfo` simply recognizes such a gap. It doesn't change any state variable and therefore acts like a guard for the second action (`prepareInput`).

At this point the state machine stutters until the buck stops here, i. e. until the overall control flow enables the action. Then the next piece of information arriving at the input channel (`iChannel`) is read, decrypted, and transmitted to the other components via the state variable `iData`. This step is completed by transferring the control back to the `Automaton` component.

```
TLSPEC InputStream
  USING TSigCard; natural
  DATA      OUT    iData: MInfo.Maybe
            IN     iChannel: MInfo.Maybe;
                   iKey: Info
    SHARED INOUT  buck: Bucks
  ACTIONS
    prepareInput ::=
        buck = buckInput
    AND buck' = buckAutomaton
    AND iChannel /= noInfo
    AND iData' =
        box(decode(iKey,unbox(iChannel)))
    waitForNoInfo ::=
        iChannel = noInfo
    AND UNCHANGED(iData, buck)
  SPEC
    INITIAL
      iData = noInfo
    TRANSITIONS BEGIN
      WHILE true DO BEGIN
        waitForNoInfo;
        prepareInput
      END OD
    END {iData, buck}
TLSPECEND
```

Figure 6: Model of the input stream

The state machine `OutputStream` (cf. Fig. 7) models the output channel part of the SigCard interface. It is the dual of the `InputStream` component. After receiving the buck, it encrypts the output data (`oData`) and writes it to the `oChannel`. Due to its duality properties the next step is to clear the output channel by generating the value `noInfo`. Note, that the semantics of our specification language allows the component to perform arbitrary many stuttering steps between two actions.

Working with VSE

The VSE systems basically consists of two components a front-end for editing and visualizing specifications and a deduction component. Both components are fully integrated, in particular with respect to the correctness management discussed below.

The front-end of the system provides a comprehensive view on (partial) developments by a so called *development graph*. Nodes in this graph correspond to the various specification entities, like theories (data type specifications), temporal specifications, mappings, modules, and procedures. These nodes are connected by links of certain types. Figure 9 shows the development graph for the formal model discussed above. The node called `ICC_Policy` contains the temporal specification of a (generic) SigCard.

The import of theories into other theories or temporal specifications is given by using-links. In our example `ICC_Policy` and also other specifications use the theory

```
TLSPEC OutputStream
  USING TSigCard; natural
  DATA     OUT    oChannel: MInfo.Maybe
           IN     oData: MInfo.Maybe;
                  oKey: Info
    SHARED INOUT  buck: Bucks
  ACTIONS
    prepareOutput ::=
        buck = buckOutput
    AND buck' = buckAutomaton
    AND oData /= noInfo
    AND oChannel' =
        box(encode(oKey,unbox(oData)))
    generateNoInfo ::=
        oChannel' = noInfo
    AND UNCHANGED(buck)
  SPEC
    INITIAL
      oChannel = noInfo
    TRANSITIONS BEGIN
      WHILE true DO BEGIN
        prepareOutput;
        generateNoInfo
      END OD
    END {oChannel, buck}
TLSPECEND
```

Figure 7: Model of the output stream

TICC_Policy which is the root of a complex theory structure. The subgraph corresponding to this structure, containing among others the theories shown above, is hidden in Figure 9. Other temporal specifications use additional theories.

Parallel composition of state-based systems is done by linking a node that represents the combined system to the nodes that represent the components. The combined node contains the full description of the parallel system including global aspects like interleaving and coloring. The behavior of ICC_Policy is given by the parallel execution of several components, including InputStream and Output-Stream from above. As already mentioned a component called Automaton is governs the flow of control between the other components by setting the (shared) variable *buck*. Components ("objects") O2 - O12 encapsulate the various state-transitions of the system.

Requirements specifications (safety/security models) are linked to system models by so-called satisfies-links. In our example ICC contains the formal security requirements for the card.

Theories, mappings, modules, and procedures (the actual programs that implement abstract operations) are linked together to form refinement steps for data types. Refinements of temporal specifications consist of two specification nodes and a refinement mapping between them.

If one clicks on a node of the development graph a window containing the corresponding specification comes up. Among others the user can now inspect or edit the specification by importing a file (generated by ones favorite editor) or, in particular in the beginning by using the built-in syntax directed editor.

If a connected part of the development graph is completed a routine for static analysis (type check) can be invoked. It checks restrictions like for example the one mentioned for input variables of state-based components. In the central repository specifications that have been checked successfully are marked as being (syntactically) valid. They can be translated to logical axioms in the deduction component. In the graphical representation type checked components are marked by ✓.

Certain links in the development graph, most importantly satisfies links and refinement links, give rise to *proof obligations*. After clicking on such a link the deduction component is started and the user finds himself located within the (deduction) unit that corresponds to the deduction problem defined by the link. So clicking on the link between ICC_Policy and ICC would generate reload a deduction unit that contains as a proof obligation the implication between two (large) temporal formulae. Proof obligations and axioms are generated by the system from the specifications that constitute the development. In particular the translation into temporal formulae can be rather complex. It can be thought of as implementing the intended semantics of the various constructs of the VSE specification language. For example the (sequential) programs used in the specification of InputStream and OutputStream are translated into action systems that use labels to mimic the flow of control.

Deduction units encapsulate data that are relevant for deduction, like axioms, proof obligations, lemmata, simplifier rules, and (partial) proofs. Proof generation as described below is always done within the local context of a deduction unit. The structure of the development graph is mirrored in the deduction component in the sense that units have access to other units according to the links given by the graph. For example, a proof about a combined state-based system can be conducted in a way that the essential lemmata are proven local the the units corresponding to the components of the system and then exported to the node corresponding to the entire system where they are combined to the final result. Proof obligations as well as the logical form of specifications that can be viewed as their semantics are generated by the system upon the first invocation or later incremental changes. Deduction units are stored in a separate repository in a form that is close to the internal representation of the theorem prover.

Carrying out the deductive work local to units that correspond to parts of the specification is of great advantage for an efficient management of change. If parts of a large specification are changed after some deductive work has already been done the correctness management detects those units that are affected by the change. It also supports the user in reestablishing a consistent state of the development. In particular existing (but invalid) proofs can be used to generate new ones. In addition to this global correctness management there is a bookkeeping mechanism for local lemma-bases that detects circularities in the dependency graph. As mentioned above there are particular techniques (the organization of lemma bases into levels) for the treatment of assumptions.

Deductive Support

Formal developments give rise to various proof obligations. We have, for instance, to verify that the formal requirement specification satisfies a given security policy or that an implementation is indeed a refinement of the given requirement specification. Specification and verification phases are intertwined. Since failed proofs may reveal problematic or even erroneous specification parts, verification plays an important part as a formal validation of abstract (i.e. non-executable) specifications.

During a formal development in an industrial setting, hundreds or even thousands of proof obligations arise and each of these obligations has to be verified in order to complete the formal development. As a result, we have to be able to prove somehow each particular proof obligation (provided it is valid). In an industrial setting, we can make the observations that on one hand the arising proofs are too complex to be done fully automatically. On the other hand, the proofs are too longish to be done by hand. Thus there is a need for an integrated approach of *interactive* (i.e. user guided) and *automated* (i.e. machine guided) theorem proving.

As an overall goal of the deductive support, we have to minimize the time, a proof engineer has to spent in verifying all these proof obligations. Notice, that this issue does not necessarily mean that we aim at the highest possible degree of automation as this may result in longish attempts to prove unprovable subproblems. In contrast, VSE aims at a more sophisticated interaction between the machine and its human supervisor. On a strategic level user and machine have to interact in both directions. On the one hand the user acts as an "intelligent" heuristic to be used by the machine and on the other hand the machine provides a variety of high-level strategies to manipulate its behavior. Both, machine and user have to agree on a common strategic language to tackle the arising proof obligations.

Modularity is a key issue in VSE as it is the most essential measure to reduce complexity of specifications and arising proof obligations. Similar to the B-tool (Abrial 1991), the deductive process is done *within* a structured specification, represented by the development graph in VSE. Proof obligations are always tackled locally to a specific deductive unit. Thus the amount of axioms, available in the proof process, is reduced to the locally visible part of the overall specification.

To support different paradigms of programming, VSE provides a variety of different logics. For instance Dynamic Logic is used to verify properties of sequential programs while a temporal logic of actions (Lamport 1994b) forms the basis to reason about concurrent programs. Unlike systems like ISABELLE, VSE does not provide a general logical framework with some generic proof support. In contrast, VSE aims at an optimized proof support for the logics under consideration (i.e. DL, TLA, and FOL) and incorporates lots of logic specific knowledge into the proof engine.

Logic Engine

In order to provide a uniform framework for the various logics under consideration, VSE is based on corresponding sequent calculi (Gentzen 1935). Proofs and also proof attempts are explicitly represented into (partial) proof trees. Analytic proof search allows us to decompose complex proof obligations by application of calculus rules in a structure preserving way, extending the partial proof tree. Step by step, the theorem under consideration is reduced into "smaller" subproblems by the application of calculus rules. It is well-known that in classical logic the sequent calculus has several drawbacks compared to machine-oriented calculi like resolution or tableau. In order to apply specific calculus rules, we sometimes have to guess appropriate instantiations of the rules to push the proof further. Machine-oriented calculi try to avoid such a situation by a least-commitment strategy. VSE incorporates various techniques on top of the sequent calculus to imitate such strategies.

The first source of in-efficency is the removal of $\exists$-quantifiers in Gentzen's calculus as it requires the selection of an appropriate instantiation. VSE introduces *meta-variables* as placeholder for these guesses and uses unification (instead of matching) for rule application. Thus, these meta-variables are more and more constrained by the ongoing proof. To keep track of the Eigenvariable condition, VSE uses the concept of *skolem-terms* which reduce this Eigenvariable condition to an occur-check problem in unification.

The next source of inefficiency is related to the problem of finding an appropriate sequence in which quantifiers of a sequence have to be eliminated. Since quantifier rules are usually not permutive, finding the correct order may be crucial for finding the overall proof. Borrowing ideas from the resolution calculus, VSE provides a calculus rule (Autexier, Mantel, & Stephan 1998) for a simultaneous elimination of several quantifiers. This rule also minimizes the number of arguments of skolem functions, as skolem-terms do not depend on the meta-variables of other formulas.

As another example consider the contraction rule which does not permute with the introduction of meta-variables. Thus, VSE provides so-called *copy-schemata* which allows one to postpone guessing the appropriate number of copies beforehand. Similar to ISABELLE's use of λ-terms to encapsulate local variables, a copy-schema also restricts the scope of meta-variables to specific part of the sequent. Any copy of this scheme will result in a renaming of the locally bind meta-variables.

Since the complexity of arising proof obligations can often be reduced by applying given definitions as simplification rules, there is a need to modify the internal structure of a proof obligation. In order to simulate paramodulation-like deduction, VSE provides composed proof rules which are based on the *CUT*-rule.

Simplification

In formal methods, simplification techniques are indispensable to reduce the size and complexity of arising proof obligations. VSE-SL provides various specification concepts which support an automatic generation of simplification rules. In many cases these concepts allow for the definition of "executable" programs within the various logics supported by VSE.

```
BASIC TList
  PARAMS TElement
    List = nil WITH isEmpty |
    cons(first : element,
         rest : list) WITH isNotEmpty
BASICEND
```

Figure 8: Specification of TList

On predicate logical level functions and predicates can be defined in an algorithmic way (cf. for instance, `knows` or `learns` in Figure 5). On one hand such a algorithm is translated into a set of conditional equations or equivalences. On the other hand the system checks the termination of the denoted algorithm and, in case it is successful, generates a set of simplification rules which mimics a symbolic execution of this program.

Dynamic Logic incorporates an abstract programming language. Means for an symbolic execution are encoded into specific calculus rules dealing with the basic constructs of this language. In VSE special tactics are available which imitate also the symbolic execution of such programs and — in combination with fix-point induction — provide a powerful mechanism to prove properties about programs (Heisel, Reif, & Stephan 1986).

For the specification of transitions in concurrent programs, VSE provides a pseudo-programming language. Transition specifications (cf. Figures 6,7 for examples), which are written in this language, are automatically translated into temporal logical formulas using an explicit representation of an program counter. Again, simplification of various proof obligations are available which make use of the specific knowledge about the behavior of the program counter.

Besides the automatic generation of simplification rules, VSE allows the user to define its own simplification routine by specifying a set of simplification rules. While the user is responsible to ensure the termination of the arising simplification procedure, the soundness of the each rule has to be verified inside the system.

VSE supports also built-in theories like integers or natural numbers by providing theory-specific (decision) procedures. VSE incorporates a decision procedure for linear arithmetic which is similar to the approach presented by Boyer and Moore (Boyer & Moore 1979).

Induction

Specifying generated datatypes give rise to the use of induction principles when verifying properties about these datatypes. The notion of generatedness in a model-theoretical view corresponds to induction axioms in a proof-theoretical view. Thus, defining an abstract datatype `TList` in Figure 8 as being generated results in providing an structural induction scheme

$$(\Psi(nil) \wedge \forall X : TList\ \Psi(rest(X)) \rightarrow \Psi(X)) \\ \rightarrow \forall X : TList\ \Psi(X)$$

as part of the axiomatization.[1]

Inductive reasoning introduces additional search control problems to those already present in first-order theorem proving. The need for the selection of an appropriate induction order or for the speculation of intermediate lemmata results in infinite branching points in the search space because (technically speaking) the cut rule cannot be eliminated in inductive theories. To overcome some of these problems, VSE uses recursively defined functions to introduce new well-founded orderings to the prover. As mentioned already in the previous paragraph, each algorithmic function or predicate definition is checked for termination. If the system is able to prove its termination, the recursion scheme encoded into the algorithm gives rise for a new (destructor stylish) induction scheme.

When proving induction formulas, VSE make use of syntactical differences between induction hypothesis and induction conclusion as additional control knowledge to guide the proof of the induction step. The application oriented heuristic, that the induction hypothesis should be used when proving the induction step, is translated into a syntactical requirement that in each proof step, the hypothesis should be homomorphically embedded into the conclusion. Furthermore, the underlying logics are enriched by additional annotations to encode these embeddings to allow for a formal reasoning on these constraints.

Tactical Theorem Proving

As mentioned before, VSE aims at an integration of interactive and automatic theorem proving.

Heuristics to tackle arising proof obligations follow the paradigm of a *structured deduction* which allows one to decompose automatically large proof obligations into simpler tasks and to synthesize an overall proof from the arising partial solutions. This enables the use of specialized methods to solve specific problems and also eases the speculation of lemmata needed to prove a theorem.

In VSE, the knowledge about how to tackle specific proof situations is encoded into a bundle of individual tactics. The accumulation of various tactics imposes an emerging functionality which is able to prove complex theorems in a goal directed way. All these tactics operate on a common representation of the actual proof state which is reflected in a proof tree annotated by additional tactical knowledge. Tactics may prune or refine this proof tree and represent the *algorithmic* part of the proof search. In VSE, proof decisions can be withdrawn by backtracking steps chronologically as well as by pruning arbitrary branches of the proof tree. The approach combines a high degree of automation with an elaborate interactive proof engineering environment. VSE uses annotated terms (Hutter 1997) as a framework to maintain various planning information during an actual proof search. They provide a simple mechanism to propagate knowledge arising within the proof search. Within this framework tactics are able to communicate their achieve-

[1] However, it should be noted that due to the intrinsic weakness of mechanizable logics this axiomatization is not categorical for infinite datatypes.

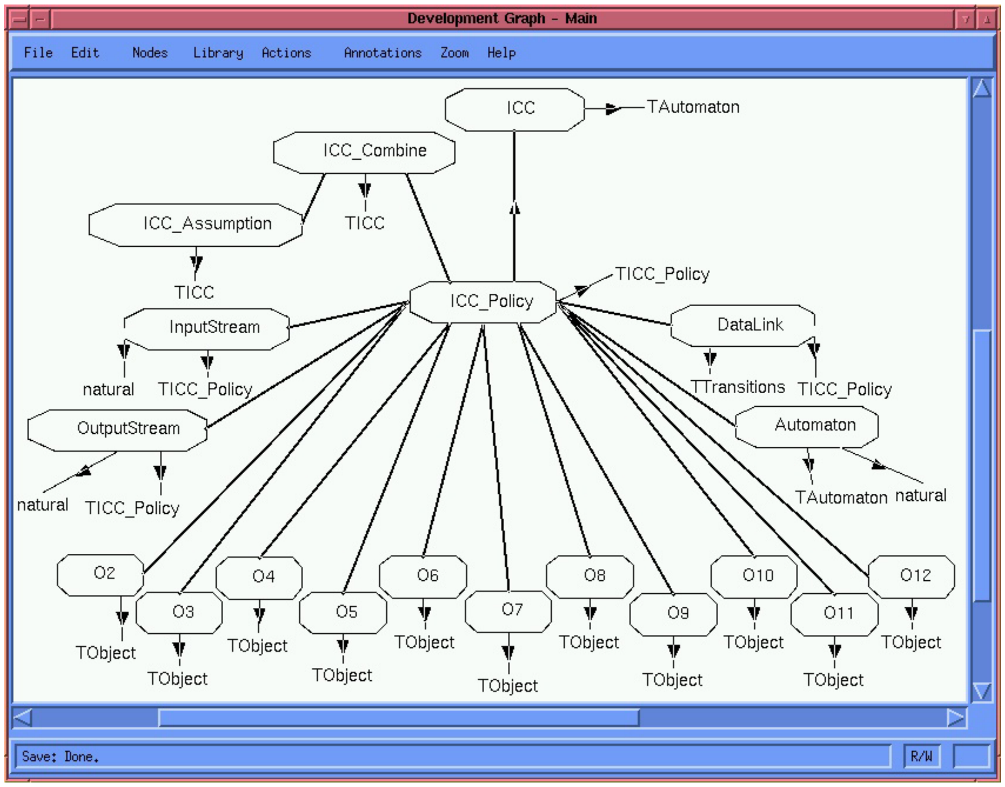

Figure 9: Development Graph

ments to following tactics and the progress of the proof is represented by a syntactical representation of differences between the actual state and possible goal states.

User Interaction

The explicit representation of partial proofs by proof trees allows the user to grasp the current proof state as well as the stepwise construction of a proof. VSE follows the principle of *direct manipulation* (Nelson 1980; Rutkowski 1982). Both, the formal development and and also a partial proof are used as task objects and are continuously presented to the user. Figure 9 shows parts of the development graph presented to the user. Context-sensitive pop up menus provide a variety of different actions operating on the proof sketch and, thus, corresponding to different possibilities to refine such a sketch. The partial proof as an object of interest is displayed so that actions which will modify the proof are directly placed in the high-level task domain.

At each stage of the proof synthesis, the human user can revise the proof sketch specified so far and give advice on how to fill in the gaps. On the one hand the user should be able to construct any possible proof within the given calculus and for example even enforce the application of a single deduction step. On the other hand proofs should be found almost automatically with as little user interaction as necessary. Therefore, sometimes only *some* strategic advice by a human user is necessary to let the system find the proof. Thus, VSE incorporates (human-like) strategic knowledge about proof search in order to facilitate users' hints. Conversely, the intentions of the (machine-like) built-in strategies have to be known by the user in order to efficiently assist the machine in the search for a proof. The essential means by which this contradictory mix of demands for the system are satisfied in VSE is the notion of a proof sketch.

The representation of a proof sketch allows the user to edit it in several ways: he may give more precise hints for a proof sketch, which includes directives to apply a certain rule as the next step if the system gets stuck, or to revise hints, e.g. by advising the system to use so-called bridge lemmas. He may add or delete subgoals, which includes the use of lemmas within proofs, and the possibility to correct the systems' assumptions on intermediate steps. Finally, in extreme cases he may even replace the whole sketch including subgoals and hints.

Conclusion

Since it's first release in 1994, VSE has been successfully applied in various projects for formal program development. Based on these experiences, it matured into a tool which offers specification and verification techniques for sequential and concurrent programming paradigms. The development graph serves as a logical database to maintain a structured formal development and as a graphical interface to represent the actual state of a formal development. Various proof techniques are integrated in VSE to ease the verification of arising proof obligations. Furthermore an elaborated correctness management takes care of the validity of such proofs once the specification is changed.

References

Abrial, J.-R. 1991. The B method for large software, specification, design and coding (abstract). volume 552 of *Lecture Notes in Computer Science*, 398–405. Springer-Verlag. Volume 2: Tutorials.

Astesiano, E.; Kreowski, H.-J.; and Krieg-Brückner, B., eds. 1999. *Algebraic foundations of systems specification*. IFIP state-of-the-art reports. Berlin: Springer.

Autexier, S.; Mantel, H.; and Stephan, W. 1998. Simultaneous quantifier elimination. In *KI'98: Advances in Artificial Intelligence*. Springer LNAI 1504.

Boyer, R. S., and Moore, J. S. 1979. *A Computational Logic*. Academic Press, London, England.

Gentzen, G. 1935. Untersuchungen über das logische schließen. *Mathem. Zeitschr.* 39:176–210, 405–431.

Heisel, M.; Reif, W.; and Stephan, W. 1986. An interactive verification system based on dynamic logic. In *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *LNCS*, 131–140. Springer.

Hutter, D. 1997. Colouring terms to control equational reasoning. *Journal of Automated Reasoning* 18:399–442. Kluwer-Publishers.

Lamport, L. 1994a. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3).

Lamport, L. 1994b. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3):872–923.

Loeckx, J.; Ehrich, H.-D.; and Wolf, M. 1996. *Specification of Abstract Data Types*. Chichester;New York;Brisbane: Teubner.

Manna, Z., and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems*. Springer.

Nelson, T. 1980. Interactive systems and the design of virtuality. *Creative Computing* 6(11).

Reif, W. 1992. Correctness of generic modules. In Nerode, and Taitslin., eds., *Symposium on Logical Foundations of Computer Science*, volume 620 of *LNCS*. Springer.

Rutkowski, C. 1982. An introduction to the human applications standard computer interface, part 1. *BYTE* 7(11):291–310.

Spivey, J. M. 1992. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, 2nd edition.