3.2154.222

An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis

Holger Peine

May 18, 1992

Abstract

This work is a first investigation of an observation noted as possibly promising:

> The problem of application-independent recognition of given elements from the architecture of an unknown software system to be analyzed can be conceived as a special case of the classification problem in a terminological reasoning system if supplied with a suitably defined taxonomy for software–elements.

This problem, however, has been solved in certain terminological reasoning systems (TRSs).

To the end of investigating this idea, the availability of a TRS was necessary (provided at DFKI by virtue of the $\mathcal{KRIS}$–system) as well as stating clearly the envisaged application-independent software-elements, followed by a concept taxonomy expressible in $\mathcal{KRIS}$ and delivering the desired results. Furthermore, a tool had to be developed to analyze software (i.e., the source code) and generate the input information for the taxonomy from that.

Stating application-independent complete and correct conditions for the role of an element within a software system turned out to be feasible for only a few basic concepts, because software employs at least up to now too few standardized concepts. The translation of the feasible concepts to $\mathcal{KRIS}$ resulted in problems of the expressive power of TRSs that were recognized as fundamental. The root of this problem spawned a new language construction for $\mathcal{KRIS}$.

Under the assumption of this new construction, a taxonomy of software-elements was formulated. However, as the incorporation of this construct, while recognized as feasible, would exceed the scope of this work and is therefore still to come, it has not been possible so far to test the formulated taxonomy.

Hoping this will become possible in the future, the tool for input generation was developed nonetheless. The chosen programming language to be processed is C, as there was an initial tool already available for it.

Thus the concluding judgment of this investigation is still to come.

Contents

Acknowledgement

This work was hosted within the German Research Center for Artificial Intelligence (DFKI) as a part of the AKA–WINO project. It was initiated by an informal cooperation with Siemens AG, Munich, Germany, dpt. ZFE IS SOF 13.

The author thanks the concerned members of both groups for their support, particularly Bernhard Hollunder for numerous advice and Dan Nesmith for corrections concerning the English language.

What the reader may expect

The present work is a first step in a process whose very viability it investigates. Further, the investigation does not conclude unequivocally, because it shows the task to demand much deeper work and also stronger tools than were possible here. Specifically, the investigation is incomplete in that its main concrete result cannot yet be verified, as it proved to require a new tool which is described but has not yet been implemented. Clearly it does not offer here a programming system or a formalism proved as useful here.

Therefore the reader be warned that he will find on the following pages *ideas*, *experiences* and *insights*, but *no directly usable results* and also less pleasing results than sobering ones.

In the text, an understandable and thus in places redundant development of the investigation was emphasized in contrast to a concise description in the style of a deduction.

At last, it is remarked here that the gender-sensitive pronouns "he", "she" etc. are used alternatingly by section.

1 The Field

This section gives an overview of the two fields whose possible connection is investigated in this paper. The reader familiar with one or both may skip the respective subsection(s).

1.1 Reverse Engineering

This section gives an overview of Reverse Engineering with regard to the need for it and to its basic conception.

1.1.1 Need

The notion of "cost for software" immediately brings to mind the cost for software development. Upon a little reflection one remembers the expense for software maintenance. This expense, however, constitutes the bulk of software expense in reality [GLKT90], as increasing software complexity elongates the time of use (and thus of maintenance) for economical reasons and the sheer amount of human expertise bottled-up there,as does the desire for continuous upward compatibility rather than installing something completely new. Furthermore, the rapidly increasing costs and risks of a new development suggest the extension of old software rather than designing something from scratch.

But there is also a corresponding shift within the activities of maintenance: Whereas the adaptation of the given software to increasing requirements was once the main task and understanding the current software was merely a less crucial preparation, today this understanding of the software to be maintained has come to consume about half of the maintainer's time [Hru90][GMN+87], and because of the growing complexity of software this fraction can be expected to increase even further.

What does it mean to *understand* a program one is assigned to maintain, one which was written by other people? This is a complicated effort and hard to verbalize at all, and will therefore probably never be completely automated, as it involves getting a "picture" of another person's mind and its way of conceiving and solving programming problems which have no uniform solutions. Nonetheless, there is a consensus that a fundamental part of this task is acquiring (from whatever sources available) a conception of the overall structure of the program, i.e., its main building blocks, their purpose and their interconnections, the policies of control and data flow and of all the "customs" followed there – in short, of the program's *architecture*. Understanding the concrete algorithms employed there is easier, as they are usually the best known and best documented pieces – and after all, we all recognize a polling loop and the like when we see them. The difficulty of this *architecture acquisition* however is the core reason for the expense consumed by understanding.

Why should it be so difficult to recapture a program's architecture? In fact, it need not necessarily be so, but in practice it is, and this is caused by the typical handling of the software life-cycle: The life-cycle ought to be completely reiterated from the requirements specification through design to the implementation and documentation, every time a new requirement is incorporated. But instead of that, reality looks about like this:

The original design of the software system was likely still clear and perhaps even amenable to maintenance – there was a "good" architecture. Unfortunately, there is still no general method (at least no generally agreed one) of representing and recording an ar-

chitecture (short of natural language), let alone the sad reality that *no information at all* was recorded in a form applicable to maintenance. Thus the architecture representation consists in an essential part of "folklore", i.e., things you are told or, even worse, demonstrated. Inevitably, information communicated in this way erodes over time: In the course of a software project (decades when including maintenance) programmers come and go, and the once-clear conception of the system's architecture fades a bit more with each new programmer, not completely informed about the system architecture and conducting extensions and supposed "improvements" (changes improving the system only *locally*) in a way which makes the software more and more complicated and entangled, although of course usually preserving its functionality. In short: An architecture that is hard to acquire will be increasingly eroded. The incomplete architecture conception in the maintainer's mind will lead her to modifications which violate the original architecture. Worse still, as the changes are conducted only locally, i.e., at the code level, and not at the design level, design and implementation soon diverge, thus invalidating any documentation. In the end, the code is the only reliable information about the system. The software life-cycle is interrupted, it ends at the code in a blind alley. Future maintenance is referred to the Sisyphean task of understanding other people's ill-documented code.

1.1.2 Aim and Concept

Of course the best remedy to an evil is always prevention, and thus the most efficient and elegant solution for making software systems easier to understand lies in better software development, including for example a rigorous conduct of the life-cycle with actual recycling of the formally represented documents of every stage, which would, however, first require developing such representation formalisms for life-cycle documents and probably much more – but all this, even if it were ever to become reality, would pertain only to new software. So what about the mass of existing software? It *must* be maintained, it *is* maintained by whatever means, and there is an urgent need for tools to assist this task. As stated above, the central difficulty is architecture acquisition. This should be achieved on a path that is most suggestively described as the **reversal of the software engineering process** from the existing system back to the roots of its original design. **Reverse Engineering**(RE) aims at the development of tools for this process. It is thus a subfield of software engineering.

The principle of RE can be summed up as follows: Software development results are identifiable and traceable, as the development process follows certain rules, i.e., the map: $architecture \rightarrow existing\_system$ is invertible to some extent. This immediately raises two questions:

- How much can be inverted ("reversed")? How much of the implicit architecture can be recaptured?
- How can the possible reversion be performed?

Both questions are recent areas of research. Theoretically, the term "existing system" embraces not only the source code, but also any kind of documentation and available information in general about the system. But since source code is the only universally formalized format of such information, all attempts of RE–*tools* have so far been based on a (possibly human assisted) source code processing (tools for utilizing natural lan-

guage documentation are still ahead of conception). Therefore, in the following, RE is understood only in this context of source code analysis.

Contributing to the first question, it is not too much of a gamble to predict that the vision of a fully automated RE-machine which is fed with the source code of a system and then prints its architecture (in some formalism yet to be defined) is not realistic, because the implementation of a software embodies not only software development knowledge, but also application-specific knowledge that cannot be reconstructed from the source: just imagine understanding the source of a compiler without knowing about grammars! All knowledge of this kind would have to be available to this RE-machine. Since such knowledge is far from being formalized, we cannot reasonably expect fully automated RE.

But this observation of the need for application-specific, but program-independent knowledge gives a hint to the second question, to a "divide and conquer"–approach to RE: After all, there *is* a place where this knowledge is present, and this is the human user, the "reverse engineer". She knows about the program's domain, but needs assistance in structuring the sheer mass of source code, and this in turn is where the RE-tool could come to assistance. Thus the operating mode of an RE-system should be interactive (as opposed to single tools which can of course be fully automated, e.g. a cross reference generator). To be of novel help in analyzing the source code, the system must however possess some knowledge of the mentioned "rules" of software development, specifically of software architecture.

The task of RE in the described context can thus be rephrased as making the rules of software architecture explicit and casting them into tools.

1.2 Terminological Reasoning Systems

This section outlines TRSs in general. A concrete example is given in Section 3.2.

1.2.1 Origin and Purpose

No long after the euphoric beginning of artificial intelligence (AI) in the late 1950s, serious problems were encountered which were soon recognized as instances of a general phenomenon: As soon as a program which nicely solved the problems its designers had in mind during construction was confronted with a slight variation of the problem, it failed with sometimes ridiculous results. This happened because the program "did not know what it was doing", i.e., it did not possess knowledge of the context of its task within a whole world of things – *the* whole world, in the extreme case. The vagueness of this knowledge requirement gives a glimpse of its tremendous difficulty – nevertheless, programs deserving the badge of intelligence need such knowledge, and thus one of the fundamental areas of research in AI has since then been finding means of expressing knowledge, or, more accurately, formalisms for *knowledge representation.*

In the 1970s, the research in natural language understanding isolated a special kind of knowledge needed to recognize the entities occurring in a sentence. This is necessary, e.g., to disambiguate words. To understand the different meanings of "arm" in the two otherwise virtually identical sentences "the girl's arm moved" and "the clock's arm moved", knowledge is necessary about *what a thing is*, e.g. what a girl and a clock are, or technically, by what concepts (i.e., abstractions) they are *subsumed* – in our example, say, HUMAN and DEVICE. The same kind of knowledge is needed to infer *general prop-*

erties of things, properties which may be crucial for understanding but are not explicitly mentioned in the sentence because they are well known to any human, e.g., that girls are children, clocks tell the time, children may not know how to read a clock, etc. Once again, such knowledge is a property of the involved concept and can be inferred if it is clear what concept subsumes the given entity. Such knowledge of concepts and their relationships is today called *conceptual knowledge* or *terminological knowledge*. The latter term is preferred in this paper.

Furthermore, it was observed that concepts are prominently related by the subconcept–superconcept–relationship (e.g. HUMANs are ANIMALs), thus forming a hierarchy of subsumption. This suggested an efficient implementation of concept properties by inheritance (see next subsection) and since the applicability of the concept idea in language processing was immediate, it initiated much programming with many concept definition methods coming under a confusing variety of names like semantic nets, frames, scripts, conceptual dependency graphs, units, or schemata (most of these embraced more than what is today meant by terminological logics, but were so ill-defined with respect to semantics that it is justified to list them here in the context of conceptual reasoning). More and more new features were added to the basic idea of conceptual entities, again confusing and intersecting to a large extent, while the expressive power of all these features remained as vague as their use. But all this had been just a way of programming, not a formalism for knowledge representation, because these methods did not offer what is demanded from a true formalism: uniformity, clarity, and generality. This was because they lacked formal semantics, their meaning being defined only in terms of their behaviour in their applications. Things asked for clarification now.

1.2.2 Terminological Languages

The need for uniformity and generality in terminological knowledge representation was soon realized, and in response more systematic methods were developed. The most prominent among them was the idea of KL-ONE[BS85], which can be called the father of today's terminological reasoning systems, together with its numerous modifications [NvL88], [Neb90], [BBMR89], [PS84], [Vil85], [BPGL85], [MB87], [KBR86], [Kob89].

KL-ONE offers *concepts* and *roles* (relationships between concepts). Beginning with a small set of primitive concepts, e.g. Procedure, Variable, and roles, e.g. uses, the language offers operators to form recursively more complex concepts and roles. Concepts can be combined, among others, by Boolean operators; however, the familiar notation with $\wedge$ and $\vee$ is not used here, as these should be reserved for combining *assertions* whose interpretation is a truth value, whereas the interpretation of a concept or a role will be explained as something different. Therefore square symbols like $\sqcap$ or $\sqcup$ are used here. Until the formal definition of their semantics, the reader is encouraged to rely on her intuition concerning the meaning of these symbols.

Examples of operators are conjunction,

GlobalProcedure := Procedure $\sqcap$ GlobalConstruct,

or quantifying restrictions on some, all or a certain number of the partners by a role,

CleanProcedure := Procedure $\sqcap$ $\forall$uses_construct: OwnConstruct,

meaning a procedure using only its own (local) constructs, or finally demanding conditions between partners by two different roles,

LocalVariable := Variable ⊓ (definedBy = usedBy).
(This demands equality between the definedBy-partner and the usedBy-partner.) Properties of concepts are expressed in KL-ONE by the presence of roles like Variable := ... ⊓ ∃has_type.
Roles can be formed by composition, conjunction or disjunction of other roles, also by inversion of another role like usedBy := uses^{-1},
or by restricting the allowed partners of another role, calls := uses.Procedure, making calls mean uses with the restriction that the used thing be a Procedure[1].

The definitions of the concepts imply subsumption relationships between them, e.g. a GlobalProcedure is a Procedure, so that all these concepts can then be ordered in a *subsumption hierarchy* by an algorithm in a process called *classification*. (Note that this hierarchy is generally not a tree, as a concept may possess several superconcepts, a GlobalProcedure is also a GlobalConstruct). The system is then ready to answer questions or, more generally, infer implicit knowledge about the represented domain, like subsumption relationships and concept properties. The intention behind these capabilities is not so much an interactive system questioned directly by a human user, but an inference component within a larger system solving a problem in the domain, such as understanding a sentence.

In order to also deal with concrete individuals as well as abstract concepts, in KL-ONE a distinction between a *T–Box* ("terminological box") and an *A–Box* ("assertional box") is made. A TBox is a collection of concept and role definitions (like those of Procedure and Variable above), whereas an ABox contains assertions about concrete individuals and concrete role–relationships between them (e.g. that init_Controller is a Procedure, init_Controller calls check_State, init_Controller is a GlobalConstruct etc.). These actual individuals are called *instances* of those concepts whose definitions they fulfill, and so init_Controller is an instance of GlobalProcedure (you had to look around shortly why this is so, hadn't you? – This gave you a glimpse of what classification is!). The services were consistently expanded to incorporate the ABox, offering classification and queries concerning individuals, too. The technical term for classifying an individual ("finding out what it is") is *realization*. The system could now infer information about individuals like init_Controller which is not explicitly present in the ABox, but may be stored with a concept like Procedure – say, that check_State is a Procedure, too: this may be the result of classifying check_State or of exploiting the calls role, perhaps because is annotated with the restriction that anything called must be a Procedure. The subsumption hierarchy of concepts is also called a *taxonomy*, a term sometimes also applied to the TBox.

The logically next step to make KL-ONE a true knowledge representation formalism was the addition of formal semantics. Until then, it was impossible to define a notion of soundness and completeness for the employed algorithms for classification and the like. However, the analogy between the concept and role operators and the operators of first order predicate logics suggested that KL-ONE was indeed some restricted kind of first order logics and thus could be given formal semantics in the same spirit. Actually this was done in [BL84], giving a *model-theoretic semantics*, i.e., a set-theoretic interpretation over the domain of discourse as the basic set. A concept is interpreted as a subset of the domain, or, from a logical point of view, a unary predicate (namely the set of all

[1]This notation will be used through this paper. Note the difference between the dot and the colon and that concept names are capitalized, while role names are not.

individuals subsumed by the concept), e.g., Procedure is interpreted as { init_Controller, check_State, ... }. A role is interpreted as a binary relation over the domain (logically a binary predicate), e.g. calls as { (init_Controller, check_State), ... }. An individual symbol is of course interpreted as an individual element of the domain. Set-theoretic interpretations of the operators as mappings between subsets and relations completed the KL-ONE-semantics, interpreting e.g. a conjunction as the intersection of the interpretations of its conjuncts. This semantics makes it possible to call such an interpretation a model for an ABox w.r.t to a TBox if and only if it satisfies all of their axioms.

This model-theoretic semantics forms the basis of terminological reasoning today, which is now clearly recognized as a subset of first order predicate logics, thus earning its term language (terms formed with the above operators) the name terminological *logics*. The calculi completing the terminological languages (*TLs*) with an ABox, the necessary algorithms and the service interface will be called *terminological reasoning systems (TRSs)* in the following.

1.2.3 Today's Services and Performance

In the last decade, a number of KL-ONE-like knowledge representation systems have been developed (cited above), their primary difference being the characteristic selections of operators offered for concept and role construction. Of course they come with widely differing user interfaces and have been applied in different domains, but their algorithmic capabilities (not regarding efficiency!) relative to the set of operators are very similar, mostly classification and related services interfaced to the user by various retrieval functions.

The operator selection is thus the characteristic feature of a TRS – it implies the possible expressive power and also the achievable complexity bounds. All full-size TRSs include concept conjunction, restrictions on role partners ("value restriction", sometimes existential, sometimes universal, sometimes both), and restrictions on the upper and lower bounds of the number of role partners ("number restriction"). The inclusion of concept disjunction is controversial, as with negation which is sometimes limited in application to especially simple concepts. An especially controversial issue is the inclusion of the mentioned demands imposed on partners by two different roles ("role-value map"). Regarding the chosen role operators, again conjunction is not debated, as is role restriction, while the other operators like disjunction, inversion or composition are controversial.

What are then the criteria for deciding on an operator selection? Why not implement all feasible operators? An assessment of the descriptive power of a concrete selection from these is a difficult logical, and even linguistic, task. Concerning, in contrast, an assessment of the computational complexity of certain selections, it is observed that subsumption is the terminological analog to logical implication and subsumption decision may thus be viewed as a kind of theorem proving. This explains a computational behaviour that should not be a surprise to any logician: The complexity of classification as the central algorithm, which is basically subsumption decision, usually rapidly increases by adding operators. This ranges from a polynomial complexity when deciding in a language offering only concept conjunction, number restriction, and universal value restriction, to full undecidability, which is entailed by adding role-value maps or by allowing the composition of general roles. This monotonic function between expressive power and complexity demands an economic choice of operators and also accounts partly for the variety of languages, as

their developers had various preferences in this trade-off.

The complexity of classification should not be overweighted, however, as the subsumption hierarchy is precomputed when reading the concept definitions in the TBox. These do not usually change during operation, so that query answers can be rapidly retrieved from the precomputed hierarchy. In most applications, the system actually employs such a fixed set of concepts – however, it is conceivable to dynamically refine some concepts (say, because a new property has emerged that some instances of the concept possess and others do not). This would be done by differentiation of the existing concept to two new subconcepts, which are then classified into their proper place in the hierarchy. If this happens, the consistency of the new concept definitions with the old hierarchy must be checked, using an algorithm whose complexity is comparable to that of classification.

Terminological reasoning is actually still in its adolescence – now that the theoretical basis has been cleared, it must be integrated into real problem solving systems. This integration will show the direction for thoughtful enhancements of operators and improvements of the algorithms. The idea investigated in this paper is one such attempt at applying terminological reasoning to real world problems. Not surprisingly, it actually required a new operator to be introduced.

A concrete example for the state of the art in TRSs is the $\mathcal{KRIS}$–system described in section 3.2

2 The Idea

This section motivates the investigation and lays out the plan.

2.1 A Central Need of Reverse Engineering

In the section introducing RE, its aim was described as recapturing the obscured architecture of an existing software system. This was concisely cast to the formulation of *inversion of the maping: architecture → existing_system.* The image space of the inverse mapping will thus be the architecture, in other words, certain source code entities, e.g. a procedure, must be mapped to certain architectural elements, e.g. an accessor to an abstract data type. This poses a major question: How is this space of architecture structured? *What are the architectural elements used in software design?*

Obviously, the first step of a general approach to RE must therefore be laying down the form of its desired results – you have to be sure of what exactly you want to build before you start an engineering process. This trivial requirement is a major hurdle in RE, however, since it would require an "architectural" language specifying software architecture as universally and unambigously as a programming language specifies an algorithm. Regarding the difficulties of software engineering in managing architectures, such a language is, if possible at all, far ahead – remember that we are talking here about application-independent software architectures. But this comparison of architectures and programs is not completely discouraging: The way elementary concepts in programming languages like subroutines, loops, arrays, or pointers evolved piecewise in the very first days of machine language programming, years before they were abstracted and unified in the first high level programming language, this very way is analogously covered today as we try to isolate useful elements of software architectures – the concept of a module is a good

example for that.

We search for concepts of software elements then. And since the concrete aim of RE is to provide machine support in extracting these software elements from the existing system, the desired concepts must be formally defined in order to be algorithmically recognizable. Assuming a set of such concept definitions available, the task of recognizing these defined architecture elements could then be transferred from the human reverse engineer to the supporting machine. A *central need of RE* can thus be expressed like this: *A set of definitions of software architecture concepts is needed which allow architecture recognition from the source code.* Software elements should be recognized and classified under these concepts according to their role in the design. To automate this recognition, the concept definitions, while required to tower to a certain complexity and abstraction towards the architecture level, must be grounded on simple syntactic entities and relationships which can be easily extracted from the source code.

2.2 The Vision

Certainly we cannot expect to find definitions of the required kind for all such software concepts (Section 4.1 gives some reasons for this), therefore this work should be seen as an *exploration of how much is feasible.* Anyway, the process of recognition outlined above indeed appears as a process of classification, being a typical inference problem in TRSs: The basic idea to model the recognition is to extract source code items (which are considered as ABox-entities) and to classify them as instances of more and more abstract TBox-concepts (which denote architecture elements). Therefore it is an interesting approach to express the needed concepts as a TBox: the classification would be for free then.

As an example, consider the concept of a module-local variable being defined as such a variable that all procedures it is used by are defined in the same module as the variable. Then the inference might work like this:

Given the ABox	and the TBox
Variable v	ModuleLocalVariable :=
Module m	Variable ⊓ ∀ usedBy: *a procedure of the same module*
v definedBy m	(if you think this a bit vague, then feel
Procedure p	assented and wait for Section 4.2 !)
p definedBy m	
v usedBy p	

and the fact (somehow derived) that p is the only Procedure that v is usedBy,

then v is recognized (classified) as a ModuleLocalVariable, which in turn might be later classified for example as the physical storage of an abstract data object, and so on, climbing the abstraction hierarchy of software architecture.

Thus the following *vision* appears:

> A TRS equipped with knowledge of software architecture in the form of its TBox and with the data of a particular target program in its ABox can form the basis of an interactive RE-tool, performing classification and answering queries about properties of the particular concept instances.

Such a system might even allow the user to store his growing insight into the system by incrementally adding new differentiations of concepts in the form of subconcepts which are not completely ABox-derivable any more but involve domain-specific concepts and roles whose instances are supplied by the user.

2.2.1 A Piece of Evidence

How realistic is this vision?

Actually it is not a completely new idea, so that there is some evidence available. The "LaSSIE" system [DBSB90] was developed at AT&T for reverse engineering their telephone switching software Definity/75^{TM}, which contains about one million non-comment-lines of source code. LaSSIE uses the TRS KANDOR [PS84] for classification. LaSSIE's TBox models processes and functional units interacting in a switching system, such as calls, connnections and the like. It contains concept definitions like

$$\mathsf{Connect\_Action} = \mathsf{Network\_Action} \sqcap \forall \mathsf{performedBy}\text{: } \mathsf{BusController} \quad .$$

The ABox is populated with individuals from the Definity system. In this application domain, a large amount of knowledge could be expressed in KANDOR's TBox–concepts (about 200 concepts), and the idea is that these are used and slowly added to during further development of the software. The Definity programmers are encouraged to specify their work using the concepts, and even define new concepts if necessary. This standardized description of functional units in Definity makes it possible to compile a catalog of such units and provide a catalog browser accepting queries at different levels of abstraction (corresponding to concepts of different specificity). This facilitates software reuse. Because of its large and tailored knowledge base and because additionally LaSSIE is equipped with a natural language query interface, it is appreciated as a valuable tool by the reverse engineers. However, although there is also a part of the TBox describing programming conventions independent of the particular Definity software, like what kinds of files there are and what their interconnections are, and although even application-independent information like cross reference data is included, LaSSIE's aim is not to recover software architecture in general, but to provide a most detailed record of Definity's structure. LaSSIE is thus not a general, i.e., application-independent approach to software analysis, as its domain is not the space of software elements, but the world of switching programming.

While this system is therefore only partly comparable, it does give some evidence for the conjecture that TRSs are useful tools to draw valuable inferences in software analysis. However, software architecture is a less understood domain than switching, and therefore the working example of LaSSIE must be appreciated with caution with respect to our aim.

2.3 The Plan

The investigation of the described idea was planned to proceed as follows:

1. Formulating application-independent definitions of software-elements which
   - deliver useful information to the reverse engineer which otherwise he would indeed collect manually,
   - are mathematically unambigous,
   - correspond to the concepts used by the reverse engineer when thinking about software.
2. Translating them to a $\mathcal{KRIS}$-TBox such that
   - the semantics is preserved when mapping to the restricted TBox-language,
   - all the primitive concepts are easily extracted from the source code
3. Developing an ABox-generator which fills the primitive concepts with individuals from the analyzed source code

Section 4 describes the problems and results in realizing this plan.

3 The Starting Equipment

In both fields of this investigation, there exists software which was built upon here. This section gives overviews of the two applied systems. They may also serve as an example for the state of the art in the two fields.

3.1 The RE-Tool Arch/xpass

Arch ("Architecture Assistant") is an RE-tool for restructuring the modularity of existing software. It was developed at Siemens Corporate Research (SCR), Princeton, USA [Sch90]. Arch examines the grouping of procedures into modules, discovers potential modularization errors in the form of misplaced procedures, heuristically regroups procedures and indicates procedures violating the principle of information hiding [Par72].

As Arch groups procedures, it must have a guideline of what should be grouped together; a sort of similarity measure for procedures. As the primary relationship in the focus of Arch is *design dependency*, the employed procedure similarity measure is a *design similarity measure*, as opposed to a control flow or data flow dependency measure. A control flow dependency measure presumes that the flow of control is the backbone of software structure and therefore groups together procedures calling each other. Analogously, a data flow dependency measure groups procedures passing data among one another.

While these two dependencies, especially control flow, used to be (and often still are) the chief guidelines for modularizing software, software engineers nowadays agree that it is more fruitful to *group together procedures sharing design assumptions*, i.e., relying on common assumptions about certain structures in their outside world. This greatly facilitates maintenance, as it is easier to change a design decision when all procedures

relying on it can be found together, ideally in one module. As an example contrasting this approach to control flow dependency, a look at how to modularize a compiler is useful: control flow dependency would (as was numerously done) group together the procedures conducting the individual phases of the compiler, with lexical analysis in the first module and its procedures writing into a symbol table, which is then read and further written by the syntactic analysis in the second module, and so on. Note however that procedures from several modules use the symbol table and thus rely on its data structure. If the table structure were changed, all these modules would have to be examined to track down the required changes there. In contrast to this, design dependency would group all procedures depending on the symbol table structure in one module, all those dealing with the parse tree in another, and so on. Note the difference to data flow, too: Two procedures both writing but not reading the table are not data flow dependent, but they are certainly design dependent. Design dependency is not limited to shared data, as commonly used types or constants establish design links as well: Once again, the dependency is established by assumptions in whatever form about the procedure's outside world. This notion of dependency comes closer to the principle of information hiding, which is agreed to be essential for good software structure.

What is needed then is a measure of shared information. The above mentioned idea of collecting procedures' external assumptions leads to the basic principle of Arch: External assumptions are collected in sets which are then compared to define similarity by the amount of shared information in terms of common assumptions.

The external assumptions are called features in Arch, and a feature of some procedure is any non-local name (a name whose scope includes more than one procedure) appearing in the head or body of that procedure. Each feature is given a name which is unique throughout the whole program. Examples of features are calling a non-local procedure, declaring a local variable of non-local type, or using a non-local variable. The features "calls p", "uses t" and "uses v" would then be attached to the procedure. Each feature is associated a weight.

The similarity between two procedures is then basically defined as a ratio of the weighted numbers of shared and distinctive features.
Let P and Q be the (finite) feature sets of procedures p and q. Then

$$\begin{aligned} sim(p,q) &= \frac{weight(P \cap Q)}{weight(P \cap Q) + w_{distinct}(weight(P \setminus Q) + weight(Q \setminus P))} \\ \text{where } weight(X) &= \sum_{x \in X} w_x \quad \text{for } w_x > 0 \end{aligned}$$

Such a function is used (with some modifications introducing control parameters and corrective terms) by Arch to measure the design similarity of two procedures. The weights w_x of the individual features are of course intended to mirror the importance of a feature. Features can be given default weights based on their number of occurrences (a rare feature shared by two procedures hints at a close coupling between them), or can be automatically adjusted to agree with a start-up modularization taken from the old software structure or from the programmer, or can be hand-tuned in delicate cases. If the similarity of two procedures is intended, but cannot be inferred by any common feature, the user can add an artificial feature common exactly to them in order to enforce similarity.

Equipped with this distance measure, Arch provides two *principal services*: Clustering and maverick analysis. *Clustering* is the grouping of procedures into modules of high

internal similarity. This can be done in batch mode or interactively by asking for confirmation of proposed placements of procedures and proposed introduction or merging of modules. The process thus results in a new modularization of the examined system. A *maverick* is a procedure that appears to be in the wrong module because it is more similar to members of other modules than to those of its own module. Arch offers such a maverick along with its present and proposed module for inspection of their common and distinctive features. The user can then agree to Arch's replacement proposal or adjust the weights of some feature(s) to justify the presence of the alleged maverick in its present module.

All these services are embedded in a window environment, and dependencies are shown as pictures of design graphs, call graphs etc.

In its input, Arch needs all the features of all procedures. These features, basically declarations and uses, must be extracted from the source code in the fashion of a cross-reference listing. This is performed by a supporting tool called *xpass*, which is an independent *source code analyzer* delivering a feature listing in the format expected by Arch. Xpass is thus the programming language dependent part of the Arch system. SCR originally developed such a tool for the C programming language, corresponding tools for CHILL and Intel-80x86-Assembler followed, FORTRAN is being considered.

Extensive experiments at SCR have shown Arch to be a valuable tool in restructuring software modularization.

3.2 The Knowledge Representation System $\mathcal{KRIS}$

This subsection is based on [BH90].

3.2.1 Rationale

As explained in the end of the paragraph on history of TRSs (Section 1.2), the field received its formal grounding with the introduction of the model-theoretic semantics for concept and role terms. All TRSs developed so far could now be assessed on a uniform basis, the descriptive power of the various operators could be measured and investigations of soundness and completeness of the used algorithms were now possible.

The development of a new TRS offering a large set of operators with sound and complete algorithms, was the goal of the AKA-WINO project at DFKI. This project resulted in the TRS $\mathcal{KRIS}$ ("Knowledge Representation and Inference System").

An initial examination of the existing TRSs delivered the insight that all of them use sound but incomplete algorithms[2], which was not just poor programming, but often inevitable, if subsumption turned out to be undecidable for the respective operator sets. Sound *and* complete algorithms were only known for rather trivial TRSs until the development of $\mathcal{KRIS}$. The analysis of the exact reasons for undecidability revealed that composition and role-value maps on general roles causes undecidability, but there is a special kind of role which preserves decidability. Some roles are actually not full relations (i.e., n:m-relationships), but partial functions (n:1-relationships). These special roles are called *features* or attributes, and their computational behaviour is, as would be expected,

[2] A sort of exception is [PS84], who ensures completeness not by adjusting the algorithms to the logics but vice versa by using a four-valued semantics providing also for "unknown"-results of algorithms.

more tractable than that of full roles, namely when it comes to chaining them and imposing role-value maps upon them: This is decidable for features, but not for general roles.

Combining these insights, the AKA-WINO project then developed the $\mathcal{KRIS}$-system, which rests on sound and complete algorithms and yet offers a relatively rich choice of operators. Roughly, its rationale can be described as a TRS striving for a uniform approximation of a maximally expressive decidable termininological language. According to this policy, subsumption in $\mathcal{KRIS}$ is decidable, but highly intractable (P-space-hard). This was willingly accepted, because this deterrent *worst-case*-complexity does not give convincing evidence about the complexity of *average* TBoxes and ABoxes. Thus it was also one of the project goals to explore the average complexity.

Today, $\mathcal{KRIS}$ is a working system used as a testbed for various questions in terminological reasoning, offering tailored optimized algorithms for different subsets of the TBox-language. It is intended for a human user and therefore equipped with a graphical interface. Its query language has not yet been fully scaled up with its inference capabilities, as the work has so far focussed on logical and algorithmical questions. The query interface will be extended, however, also offering interfacing to an embedding system.

$\mathcal{KRIS}$ was implemented in Common-Lisp on a Symbolics machine and is being ported to Macintosh.

3.2.2 TBox and ABox

The $\mathcal{KRIS}$-TBox offers the following concept-forming operators (for short examples see Section 1.2), which are given in $\mathcal{KRIS}$-syntax here:

(**and** $C_1 \ldots C_k$)	conjunction
(**or** $C_1 \ldots C_k$)	disjunction
(**not** C)	negation
(**all** r C) (**all** f C)	value restriction
(**some** r C) (**some** f C)	existential restriction
(**atleast** n r) (**atmost** n r)	number restriction
(**equal** f_1 f_2)	equality role-value map (features only!), usually called "agreement"
(**not-equal** f_1 f_2)	the same for unequality ("disagreement")

The available role-forming operators are

(**and** $r_1 \ldots r_k$)	conjunction
(**restr** r C)	restriction

The feature-forming operators are

(**and** $f_1 \ldots f_k$)	conjunction
(**compose** $f_1 \ldots f_k$)	composition

All the above operators may be arbitrarily nested with one another if the resulting term is still well-formed in that it contains only concept (role; feature) terms in concept (role; feature) positions. There is, however one *exception*: The role argument of an **atmost** or **atleast** must not be a **restr**.

The following statements introduce terms (i.e., concepts, roles and features) by giving them a name, which is called a terminological axiom (as it involves only concepts, no individuals):

- Introduction of unrestrictedly interpretable primitive terms:
  (defprimconcept C) (defprimrole r) (defprimattribute f)
- Introduction of fully defined terms by their definition:
  (defconcept A A') (defrole r r') (defattribute a a')
  where the quoted terms denote term expressions formed with the listed term operators. This establishes a logical equivalence between t and t'.
- Introduction of partially defined (and thus still primitive) terms by necessary, but not sufficient conditions:
  (defprimconcept C C') (defprimrole r r') (defprimattribute f f')
  with the quoted terms as above. This establishes only a logical implication t $\Rightarrow$ t', but not vice versa.

A $\mathcal{KRIS}$-TBox is a finite sequence of such terminological axioms with the second argument (the defining term) constructed, if present, from the term-forming operators above. These definitions must not contain a cycle [3].

A $\mathcal{KRIS}$-ABox is a finite sequence of *assertional axioms* (making assertions about individuals) of the form

(assert-ind a C) (assert-ind a b r) (assert-ind a b f)

meaning that the individual denoted by a is an instance of concept C, (a name which is defined in the TBox), a is r-related to b, and that the f of a is b.

3.2.3 Semantics of $\mathcal{KRIS}$

As mentioned in the section on TRSs in general (sec. 1.2), their semantics is a model-theoretic one, interpreting concept and role terms as sets over the domain of discourse. A concept is interpreted as a subset of the domain, a role as a binary relation over the domain, and a feature as a partial function over the domain.

What conditions must hold then for these sets? The interpretation of unrestricted primitive terms is left open (being an instance of such a term must be explicitly stated in the ABox and cannot be inferred in any way). The interpretation of partially defined terms, however, is restricted by a superset (remember that only a necessary condition is given as a "definition" for these – the restricting superset is then the set of all individuals satisfying this condition). Thus an interpretation $\cdot^I$ for such a term must grant the condition

for every (defprimconcept A C) in the TBox: $A^I \subseteq C^I$.

(Analogously with roles and features). Finally, the interpretation of fully defined complex terms is recursively fixed by the interpretations of their constituent subterms, combined in a well-defined way depending on the operators as shown below. Any interpretation for these terms must grant

for every (defconcept A C) in the TBox: $A^I = C^I$.

[3] For a discussion of terminological cycles see [Baa90b] [Neb88]

Examples for operator semantics are as follows, with $\cdot^I$ denoting the interpretation function and Δ denoting the domain:

$$
\begin{array}{lcl}
(\textsf{and}\ \ C_1 \ldots C_k)^I & := & C_1^I \cap \ldots \cap C_k^I \\
(\textsf{not}\ \ \mathrm{C})^I & := & \Delta \setminus C^I \\
(\textsf{all}\ \mathrm{r}\ \ \mathrm{C})^I & := & \{a \in \Delta \mid \forall b : (a,b) \in R^I \Rightarrow b \in C^I\} \\
(\textsf{some}\ \mathrm{r}\ \ \mathrm{C})^I & := & \{a \in \Delta \mid \exists b : (a,b) \in R^I \wedge b \in C^I\} \\
(\textsf{atleast}\ \mathrm{n}\ \ \mathrm{r})^I & := & \{a \in \Delta \mid card(\ \{b \in \Delta |\ (a,b) \in r\}\ )\ \geq\ n\} \\
(\textsf{equal}\ \mathrm{f}\ \ \mathrm{g})^I & := & \{a \in dom(f^I) \cap dom(g^I) \mid f^I(a) = g^I(a)\} \\
(\textsf{restr}\ \mathrm{r}\ \ \mathrm{C})^I & := & \{(a,b) \in r^I \mid b \in C^I\} \\
(\textsf{compose}\ f_1 \ldots f_k)^I & := & f_k^I \circ \ldots \circ f_1^I
\end{array}
$$

Two remarks are in order concerning the interpretation of the ABox by $\mathcal{KRIS}$. Each individual constant symbol is assumed to denote a unique individual of the domain, which is well known from data bases as the "unique names assumption". In contrast to data bases, however, $\mathcal{KRIS}$ does not assume a closed-world semantics for the ABox ("there are no other relations than explicitly listed here"), but an *open-world semantics* to be able to model incomplete knowledge: An ABox containing nothing but the axiom (assert-ind a b r) does thus not exclude a being also s-related to c or whatever you like. Accordingly, $\mathcal{KRIS}$ does not infer that b is the only r-related individual of a. Thus "$\forall$"-statements and "$\neg$"-statements cannot be derived by simply checking all concerned ABox-individuals. Section 4.2 reports how this feature of $\mathcal{KRIS}$ can be a nuisance for some applications, and how it can be mended.

3.2.4 Reasoning Capabilities

As introduced in Section 1.2, the central algorithm of a TRS is classification: the ordering of a concept into its proper place in the subsumption hierarchy, and the recognition of an individual as an instance of some concept(s). While classification gives an intuitive picture of what the system delivers, it is useful to separate and examine more closely what services are actually performed. This analysis also leads the way to how the services are implemented in $\mathcal{KRIS}$. The following six problems are solved by $\mathcal{KRIS}$ and constitute the user's options (by a knowledge base we mean an ABox and a TBox):

- *Is the knowledge base consistent?*
  Obviously this requirement is essential prior to any inferences. This check is not trivial for large knowledge bases – indeed, the subsequent four services will turn out to be based upon it. The checking algorithm works by incrementally building a model, iterating over all the axioms.

- *Does the concept C subsume the concept D?*
  This is the central question. A positive answer allows one to infer for D all knowledge valid for C. The question can be rephrased by the simple observation from propositional logics that it is equivalent to the concept term and D not C)) being inconsistent, i.e., having no model – so this question turns out to be decidable by the above consistency check, too.

- *What does the subsumption hierarchy look like?*
  This is built up by placing each concept introduced by a terminological axiom

through repeated subsumption decision, in order to classify it more and more accurately until its proper place in the hierarchy is found.

- *Is the assertional axiom α implied by the knowledge base KB?*
  This is a question about an individual. If e.g. α = (assert-ind a C), then the user asks whether or not a is an instance of C. Since
  $KB \models C(a) \Leftrightarrow KB \vdash C(a) \Leftrightarrow (KB \Rightarrow C(a))$ valid $\Leftrightarrow KB \wedge \neg C(a)$ inconsistent
  (by soundness and completeness and the deduction theorem),
  this can once again be decided by the consistency check.
  If α is one of the other two ABox-axioms, the argument is the same.

- *What are the most special concepts subsuming the individual a?*
  This is our old task of finding out what a is, the classification of a (or better and technically correct, realization of a).

- *What are the individuals subsumed by the concept C?*
  This is the inverse service to the above.

Thus a $\mathcal{KRIS}$-session looks like this in principle: The user loads the TBox for the domain, which someone has carefully devised some time ago, and then feeds the ABox with the concrete data she wants to examine in the present session. The system is now ready to perform the indicated services (an initial universal consistency check might be the first action).

The reader familiar with data base queries will have noticed that limiting queries to assertional axioms is a strong restriction, as they can neither contain variables needed to form queries like "What x are r-related to a", nor can the result be further processed by demanding additional conditions etc., as can be done in relational algebra. $\mathcal{KRIS}$ is presently being extended to offer such services, too.

3.3 The Coupling

Arch generates information about the attribution of individual procedures to individual modules (which appear as ABox-items); it does *not* generate new software concepts like for instance a special kind of module or anything of this kind. It is thus a tool at the instance level, not at the concept level, so that there can be no dynamic connection to the planned TBox, whose concepts are consequently fixed in advance to Arch and independent of the data from the particular program under analysis. The concept taxonomy can of course still be incrementally refined by the human user, but not by Arch: *The information from Arch to $\mathcal{KRIS}$ will flow through the ABox.*

As the planned TBox receives the primitive relationships through the ABox, all issues like what procedure belongs to what module must have been decided at the time of reading the ABox. Running Arch beforehand to improve modularity is therefore optional and not required by the TBox – it just takes modularity "as is", i.e., as found in the ABox.

What *is* required, however, is an ABox containing the primitive syntactic relationships between the individuals. Arch needs quite similar information in the form of its feature data and uses the xpass tool to generate this. As a transformation of the feature data to ABox-items is a less laborious task than extracting the ABox data directly from the source code, this paper takes the way of building the required ABox-generator on top of xpass in the form of a such transformer from xpass's output to ABox-assertions.

Xpass, however, has its deficiencies and also bugs (described in Section 5.2), but they are limited to the submodular level and therefore judged acceptable with respect to the goal of the TBox, focussing on modules rather than procedures.

The coupling will thus look like this: The program's source code is fed to xpass, which outputs a feature list, which is then transformed to a $\mathcal{KRIS}$-ABox and finally evaluated with respect to the TBox.

4 The Process

This section reports the experiences in concept formulation and translation to $\mathcal{KRIS}$.

4.1 Concept Formulation

As our plan is to automate software element recognition by concept classification *deriving from source code information*, the first step is concept formulation: Laying down *what* elements are to be recognized, and *how* (i.e., by what exact definitions) they are to be recognized from the source.

Regarding the considerable literature on software engineering, there appears to be plenty of advice on how to proceed in developing software (the waterfall-model and the like), but remarkably less so on what general elements to build it from. Still, an experienced programmer has his own ways of thinking about software structure, his vocabulary of software entities. This was used, then, as the target of concept formulation. Now the idea might be tempting at the first glance that no more were needed than "just writing it down cleanly" (with more or less effort) to cast this vocabulary into a workable set of concepts. This optimism was detected to have been implicit in the original idea initiating this investigation, too.

However, when it comes to tying them down, descriptions (the term "definitions" cannot reasonably be applied here) of most software concepts will *evade any attempt to exactly catch their nature in terms of syntax.* This has at least three reasons:

- The majority of software concepts do not operate on the syntactic level, but on a level of purpose: Their distinctive characteristic is their dynamic effect on and use for the other system components at run time, and not their syntactic relations to them at compile time. Examples for these concepts are security check, control flow dispatcher, return code, exception handler, utility function, mailbox structure, computing function, flag variable, blackboard variable, demon, ...– certainly syntactic *descriptions* for these are conceivable, but no such *definitions*, as experience has shown that there is always some exceptional case not covered there.

  Since the characteristic of these concepts is their meaning as opposed to their form, it appears justified to call them *semantic concepts* here, as opposed to the source-code-definable *syntactic concepts*. Not surprisingly, especially the most fruitful concepts for architecture fall under this category of semantic concepts – just think of the concept of a dispatcher procedure calling the next action procedure, which cannot easily be distinguished from that of a generic computation procedure just calling the appropriate specialized procedure to fulfill its task.

- Many characterizing properties of software features (and not only of them) refuse to fit into the strict truth notion of logics: They are inherently vague, i.e., their truth must be measured by degrees (of unclear nature), rather than by a strict value of either true or false. The description of an "important procedure" as being "often called" would require a strict distinction of "often" of the kind that being called n times is not often, but $n+1$ times is – obviously not an adequate representation of "often".

  This problem is not new to AI and has been combatted with heuristic definitions (i.e., approximations where an occasional error is accepted) or with an extension of logical truth values from two distinct values to a continuum, as fuzzy logics tries to do. However, representing vague knowledge is still an area of current research, in particular "vague logics".

  Defaults are another kind of non-standard logics, which can also be useful for our task, as witnessed by the example "A 0/1-result means a Boolean value", which is usually true in C, but might also mean a numerical result.

  Concerning TRSs, there is a plan at DFKI to incorporate such and other non-standard logics into $\mathcal{KRIS}$ in a forthcoming project.

- Software development as it is today is simply not standardized enough, does not employ enough standard concepts – a state which is well expressed by the saying that software development is still crafting, even craftsmanship, instead of fabrication (as suggested by the term of software *engineering*). By the way, practitioners in the field complain that today's actual practice is even worse, as RE is still more inhibited by *bad* crafting of the masses of shallow-educated or autodidactical programmers than by the general shortcomings of (even good) crafting in comparison to fabrication.

  While the efforts for CASE nourish the hope that standardization in software production will increase, and the trend towards object orientation hopefully brings about standardization (by reuse) at the level of coding, only little of these two will percolate to the between layer of software architecture, again and again challenging the creative human. Still, we must put our hope in the development of more powerful software architecture concepts and the education of better software architects.

As a result of this, fewer concept definitions than initially hoped could be stated satisfactorily. Inherent vagueness of concepts cannot be tackled with a TRS (at least today). However, most of the wrecked hopes concerning feasible definitions are due to the omnipresence of semantic concepts: The desired notions are simply too strong to be inferred from the syntax. But this hints a way for the future, too: As a semantic concept is characterized by its dynamic effect on the system, this is what must be observed to recognize the concept: a run time simulation with data flow analysis as its central part might bring about better results.

Still, about twenty core concepts were laid down, mostly procedure and module classifications (see section 5.1 for details) with the emphasis on locality and data encapsulation. The last of the above three obstacles, lack of standard in programming, might set a limit to the value of even these achieved concepts, as their practical usability obviously depends on how much they are employed in real software. Unfortunately, it has not yet been possible to test this, as a new and not yet implemented TBox-operator for $\mathcal{KRIS}$ turned out to be essential, as described in the immediately subsequent section.

4.2 Translation to $\mathcal{KRIS}$

Of course concept formulation was conducted all the time with the target borne in mind of translating it to a logical language, and so the definitions were right from the beginning kept mathematically precise and purely syntactic, which precluded right away many attempts as described above. Still, the TL underlying $\mathcal{KRIS}$ is not full predicate calculus, and so some ways of expression, while fulfilling the requirements of being precise and syntactic, could still not be expressed in $\mathcal{KRIS}$. These were recognized as being not mere challenges for the skill of formulation in $\mathcal{KRIS}$, but as demanding fundamentally more from the language than it was intended for respectively than is possible in a TRS. One of these problems could be mended, one be contained to a negligible size – one, however, demanded the invention of a completely new construction (not an operator anyway) for a TRS, which was utterly indispensable for the definition of the central concept of locality in a program. The three problems and their solutions are described in the following three subsections.

4.2.1 Closed World Semantics

As mentioned in the section on the semantics of $\mathcal{KRIS}$ (sec. 3.2), an ABox is interpreted according to the *open world semantics*: the ABox information is assumed to be sound, but not complete – the possibility is taken into account that part of the information is yet unknown and not present in the ABox, so that the instances of the concepts and roles explicitly listed there are not presumed to embrace all assertions that hold in the domain. This is a conscious decision in the design of $\mathcal{KRIS}$, the reason being that *non-monotonicity* was to be avoided: Later additions to the ABox should not be allowed to invalidate previous conclusions footed on the now incorrect assumption of an exhaustive ABox. Accordingly, it is logically unsound to derive any such conclusion presuming completeness. Such conclusions are entailed, however, in every concept or role term using a universal value restriction (relying on *all* information being known) or a negation (relying on everything *not* listed being false). Thus, the following knowledge base would not allow $\mathcal{KRIS}$ to draw certain conclusions:

Given the ABox	and the TBox
Variable v	LocalVariable := Variable ⊓ ∃ defBy: Procedure
Variable u	NonlocalVariable := ¬LocalVariable
Procedure p	
p defines v	
v defBy p	LeafProcedure := Procedure ⊓ ∀ defines: Variable
	(a leaf in the procedure definition tree)

(and no other instances of these terms being in the ABox), then $\mathcal{KRIS}$ would conclude neither that u is a NonlocalVariable (as this involves a negation), nor that p is a LeafProcedure (this involves a universal restriction) – all that would be inferred is that v is a LocalVariable. Universal and negative information would have to be explicitly given in the ABox, like u: ¬∃defBy: Procedure, if such conclusions like the former two are desired.

Now it is certain that universal restrictions are indispensable for our purpose, as is easily demonstrated by the LeafProcedure-example above. The same is true for negation, as the two are well-known to be equivalent, once you have an existential quantifier, by

the interchangability of $\neg\exists x : \neg P$ and $\forall x : P$. So how can we achieve the validity of universal conclusions in an open world semantics? Once again, such a semantics allows only inferences founded on *positive* information, and not on the *absence* of information, and so we must provide some positive axioms in the ABox expressing that the listed instances are all there will ever be. This is similar to the approach of circumscription [McC80] to capture such a closedness property. In general, such axioms are not possible in first order logics, as they make statements about an infinite number of conceivable assertions which do *not* hold. However, there are two features of our task and $\mathcal{KRIS}$ allowing the construction of such "ABox-closing" axioms nonetheless. The first is the observation that the complication of later modifications of the ABox does not happen in our case: If Procedure p uses Variables $v_1 \ldots v_n$ and no others, this will hold for the time of the whole analysis, and so will all other relationships. *Our knowledge of the examined program is complete.* Additionally, since quantifiers in $\mathcal{KRIS}$ are allowed only to quantify roles, not concepts, we have to add "closing axioms" only for the interpretations of role terms. This gets us as far as achieving our goal if we were only allowed to state that "the number of possible instances of each role term in the ABox is exactly the number of its listed instances". This, however, *can* be expressed cleanly in $\mathcal{KRIS}$ by a corresponding number restriction.

The problem can thus be mended by counting all instances in the ABox for each individual and each of its roles and then adding a number restriction axiom for them, limiting the allowed number of role partners to exactly the counted number and thus excluding any possible others.[4] This effectively makes the ABox a closed world, and the open world semantics is then equivalent to the closed world one (with respect to the sole instances).

4.2.2 Terminological Cycles

An ABox is said to contain a terminological cycle if there is a concept C whose defining concept term (the second argument of the defconcept-axiom) contains (possibly through several nested definitions) a reference to C itself. Obviously this is not a definition in the sense of explaining C in terms of other things already known, and yet it is a natural way of expression: A Procedure may call other Procedures (a cycle of length 1), a Procedure defines a Variable which can again be passed_to a Procedure (a cycle of lenght 2), etc. The recursive structure of programs accounts for many such cycles in a TBox capturing them. This observation suggests that terminological cycles in a TBox can actually be conceived as an instance of the familiar concept of mutually recursive definitions in a programming language. Analogously to these, terminological cycles introduce difficult computational problems (see [Neb89], [Baa90b]) into a TBox.[5]. This is why $\mathcal{KRIS}$ does not allow terminological cycles and rejects a TBox containing one.

Since there is no work-around to mimic terminological cycles, the initial TBox had to be cleaned of them, as it had been allowed cycles like the above examples to find out how far one could get at all. In order to contain the practical impact of this loss of precision occurring in some concept defitions, the definitions were rearranged so as to limit the knowledge expressed by occurrences of terminological cycles to checks of

[4] See the `close_ABox_world`-tool in appendix B for the details.

[5] The knowledge calling for terminological cycles can often be equivalently expressed by transitive closure of roles (see [Baa90a]).

comparatively trivial conditions, which are anyway obliged to by all correct programs, e.g. that procedures call only procedures (and not modules, say) or that variables are defined in some place, each variable has exactly one type etc. These remaining occurrences of cycles were then simply omitted in the TBox. They are not checked anymore, and thus an incorrect program would no longer result in an inconsistent knowledge base. However, this is a negligible restriction, since it would not make much sense to analyze a program that still contained compiler errors.

4.2.3 Reference to Individuals

This is a fundamental problem, rooted in the very principle of terminological reasoning and therefore questioning in general the applicability of TRSs for our purpose.

Problem:

A TRS owes its name to dealing with terms, i.e., collective abstractions of things in contrast to the things themselves. It provides means of defining concepts, meaning *sets of things*, by certain relationships to other such sets of things. However, it does not allow relationships containing references to *a* certain thing, to an *individual* in the sense used so far. There are good reasons for this restriction, which is after all basically nothing other than the distinction between TBox and ABox, which accounts for most of both the simplicity and elegance of the language and the efficient implementation. Reference to concepts only is often sufficient, as in

$$\mathsf{Variable} := \ldots \sqcap \exists \mathsf{hasType}\colon \mathsf{Type},$$

where the concrete type of a variable does not matter – just any instance of Type will do to classify an instance of Variable. There are concept definitions, however, which do require a relationship not just to some unspecified instance of another concept (appearing as a reference to this concept in general, like Type), but to a special instance distinguished by a certain link originating from the individual instance to be classified (referred to as the "examined instance" in the following, as the classification might fail after all). This link restricts the partners in the relationship (which is expressed as a role, of course) to a single special one. As a first example, think of the concept of a (directly) recursive procedure as "a procedure which calls itself". A first attempt might get as far as

$$\mathsf{RecurProc} := \mathsf{Procedure} \sqcap \exists \mathsf{calls}\colon \mathsf{Procedure}$$

– but this does not express what was desired: a RecurProc does not merely call *some* other Procedure, but a very special one, namely itself. The partner in the calls-relationship must be restricted to this special procedure. In this case, the distinguishing link from the examined individual to the partner instance is identity. So there ought to be some construction P in the place of the called Procedure, denoting the "right" procedure. With the definition as it is, certainly a procedure p calling p would satisfy the definition, but so would a procedure q calling p (and no others), too. Now there is no way to "narrow" this definition down to this P, to the "right" procedure – for a first attempt at such a P, one might imagine somehow constraining the interpretation of the second Procedure in the definition of RecurProc to subsuming exactly one instance[6], or allowing a constant symbol in its place. This, however, would not work either, because the "right" procedure is not

[6]The TRS CLASSIC offers a facility to explicitly fix the interpretation of a concept to a literally enumerated set of individuals, which must, however, not appear in the ABox

fixed at the time of concept definition, but depends on the examined instance: When examining p, the procedure required to be called is p, at other times it is q etc.

The mentioned link from the examined instance to the distinguished partner instance generally need not be identity, as in the previous example. As a second example, consider the concept of a (module-) local variable defined as a variable with all procedures using it being defined in the same module. Assuming a TBox-language allowing the agreement on features (like $\mathcal{KRIS}$) and a feature moduleOf to designate the defining module of a variable or a procedure, an attempt gets as far as

$$\mathsf{LocalVariable} := \mathsf{Variable} \sqcap \forall(\mathsf{usedBy.Procedure})\text{: } (\mathsf{moduleOf} = M)$$

with once again M standing for the "right" module, for the module of the examined instance, the instance of Variable just being classified "on the left hand side of the definition".

Solution: The SELF-construction

These two examples are just a few from quite a number of problems with references to individuals in definitions, which turned up in the translation of the TBox to $\mathcal{KRIS}$. How can we specify the "right" role-partners in these definitions? Since the only means in a $\mathcal{KRIS}$-TBox to get hold of an individual is by a feature (remember they are functions and thus have only one possible value, i.e., an individual), and since feature values can be constrained by agreements, it was recognized that the only possible solution to this problem of constraints linking certain individuals could run as follows:

> The link constraint must be formulated as an equality between certain characteristics (which are individuals) of the two instances. It can then be expressed through an agreement between features, with one feature denoting ("pointing to") the characteristic of the distinguished partner instance (often, this is the just the partner itself) and the other feature somehow denoting the characteristic of the examined instance (possibly itself).

This solution allowed all problems with individual references to be reduced to a standard structure given below which is quite close to that of the examples from above. Following the indicated solution, we use a new feature procedureOf to transcribe the first example, mapping things in the scope of a procedure to that procedure, in particular a procedure (quite consequently) to itself. This is the feature pointing to the intended partner instance (here, the procedure itself). The first example can then be transcribed to

$$\mathsf{RecurProc} := \mathsf{Procedure} \sqcap \exists\mathsf{calls}\text{: } (\mathsf{procedureOf} = P)$$

with P standing for the "right" procedure, which is now of the standard structure of references to individuals in its simplest form (when the link is identity). The procedureOf-feature links the examined instance (the calling procedure) to the distinguished partner instance (the called procedure). In this case, these two are identical, and this is reflected by the procedureOf-feature reducing to the identity mapping when applied to procedures.

If the link between the examined instance and its partner is different from simple identity, the solution suggests to try to express it as equality on certain characteristics reached by one or more intermediate features, which are applied to examined instance and/or its partner. The second example illustrates this: The link required from the potential instance of LocalVariable to its partner instance of Procedure through the usedBy-role is not identity (how could it be?), but "having the same moduleOf". Accordingly, we use

the moduleOf-feature as an intermediate applied to both the partner procedure and the examined instance and arrive at the final transcription of the second example

$$\mathsf{LocalVariable} := \mathsf{Variable} \sqcap \forall(\mathsf{usedBy.Procedure})\text{: } (\mathsf{moduleOf} = \mathsf{moduleOf}(V))$$

with V "standing for" the examined instance – quite as the P in the transcription of the first example. This now suggests the definition of a new concept operator defined as follows: Let f_i, g_j be features. Then

$$\mathsf{C} := (\ \mathsf{f}_1 \circ \ldots \circ \mathsf{f}_n = \mathsf{g}_1 \circ \ldots \circ \mathsf{g}_k(\mathsf{SELF})\ )$$
$$\mathsf{D} := (\ \mathsf{f}_1 \circ \ldots \circ \mathsf{f}_n \neq \mathsf{g}_1 \circ \ldots \circ \mathsf{g}_k(\mathsf{SELF})\ )$$

are concepts, with $n, k \geq 0$ and SELF "standing for" the instance which is currently examined whether it is an instance of C respectively D. Both C and D are called a SELF-*expression*. Such an expression is thus a generalized agreement concept. It may be used in a value restriction to express a link constraint, such as

$$\forall \mathsf{r}\text{: } (\ \mathsf{f}_1 \circ \ldots \circ \mathsf{f}_n = \mathsf{g}_1 \circ \ldots \circ \mathsf{g}_k(\mathsf{SELF})\ ) \quad .$$

[7] This is the mentioned standard form of our problem.

Nearly all individual reference problems occurring in the TBox of software concepts which was compiled with the help of SELF could be expressed as such standard form definitions (with both n and k never greater than 1).

Now that we have standardized our individual reference problem to this form, we must admit that $\mathcal{KRIS}$ (as all the other complete TRSs) does presently not offer a way of expression equivalent to this SELF, the individual just being tried as a potential candidate of an instance of C. Such a way of expression is, however, indispensable for our task, as is clearly witnessed by the preceding examples. Therefore,

> a new operator applicable within concept definitions named SELF was invented which always denotes the individual just being classified under the concept.

SELF is allowed only in the lexical context of (dis)agreements matching the standard form from above. Considering the operators available in $\mathcal{KRIS}$, this is anyway the only way to utilize an individual in a concept definition.

SELF is a kind of *individual variable* in concept definitions, not a concept, but not a constant symbol either, as its interpretation varies depending on the examined instance. But it is a special kind of individual variable, occuring only in the context of denoting "the individual just being classified". Concretely, it allows just as few individual variables as possible to build our TBox.

Individual variables would, if allowed in full generality, imply serious semantic and computational problems up to utter undecidability. This narrowing of the desire for individual variables as far down as to the case of the SELF-construction thus conforms to the principle in $\mathcal{KRIS}$ of maximizing expressive power while preserving decidability.

SELF is a construction which has not yet been considered in terminological reasoning. An implementation is considered feasible and will be attempted at DFKI.

---

[7] This could also be expressed by an agreement involving full roles and also their inverses:

$$(\mathsf{r} = (\mathsf{f}_1 \circ \ldots \circ \mathsf{f}_n)^{-1} \circ \mathsf{g}_1 \circ \ldots \circ \mathsf{g}_k)$$

However, such general agreements entail undecidability.

Problems with the Solution:

SELF is decidable, but its nature is not obvious. If you were not contented with the "definition" of SELF as "the instance just being classified", then you are exactly right. What *is* this SELF at all? We have observed that it is not a constant, as its interpetation changes depending on what instance is being classified at the moment, and it is not a feature either, as it would then apply to the concept which is to be restricted in the agreement, i.e., the r-partner of the C in the standard form – and this is not what was intended. SELF might be described as a *metalogical constant*, as it must be substituted by an individual during evaluation of the SELF-expression. *After* this substitution, SELF has the effect of a constant symbol denoting the substituted individual; but as the substitution happens at the meta-level in the evaluation process, not at the level of logics, this justifies the description of SELF as a meta-logical constant.

SELF is, unfortunately, not a logical expression at all, as it violates the familiar principle that the interpretation of a closed expression does not depend on its occurence context, once you have chosen a model. E.g. in $\mathcal{KRIS}$, the interpretation of a concept is always the set of individuals subsumed by the concept definition, however deeply nested in an embedding expression the concept appears. The same is true in first order logics: The interpretation of a term is always the same individual from the domain, and the truth value of a formula does not depend on how many other formulae are attached to it by logical operators. SELF, however, is different: As SELF denotes the individual just being classified under the concept at the left hand side (l.h.s.) of the definition containing the SELF-expression, this would change if the right hand side (r.h.s.) of the definition were substituted into a larger embedding definition, since the SELF would now denote the instance being classified at the l.h.s. *of the embedding concept definition*, and no longer the correct instance from before. An example will show this easily: Assuming the concept definitions

ProcWithOwnVar := Procedure $\sqcap$ $\exists$usesVar: (procedureOf = SELF)
ModWithPWOV := Module $\sqcap$ $\exists$defines: ProcWithOwnVar

then the SELF-expression (procedureOf = SELF) has different interpretations according to its lexical context:
All by itself, it denotes

$$\{x|\ \text{procedureOf}(x) = a\}$$

if a is examined;
its denotation, however, *within the definition of* ProcWithOwnVar (as in the example) when examining p is

$$\{x|\ \text{procedureOf(x)} = p\}$$

(this is what was intended for), whereas its denotation if the r.h.s of ProcWithOwnVar were *lexically substituted within the definition of* ModWithPWOV (this happens when expanding a TBox) would then (examining m) be

$$\{x|\ \text{procedureOf(x)} = m\}$$

In the last case, SELF duly evaluates to m (after all, the individual being examined now), disregarding that within the old definition context it evaluated to p. Thus an algorithm which performs a preliminary bottom-up substitution of all concept occurrences by their definitions (as $\mathcal{KRIS}$ presently does) would never realize m as an instance of ModWithPWOV, even if m defines p and p is an instance of ProcWithOwnVar. Therefore the concepts containing SELF-expressions cannot not be substituted as usual, but the algorithm must

keep track of their original concept definition context (perhaps by substituting the SELF by a pointer to this concept) to evaluate the SELF correctly.

A problem of different nature is soon encountered when actually writing definitions containing SELF-expressions: SELF's context dependency tends to result in lengthy concept definitions, because splitting the definition would cut off a deeply nested SELF from its context.

All this is admitted to appear still awkward (though precise) in meaning, and "logically impure". Future investigation will further clarify the semantics, possibly the syntax and also the implementation of SELF-expressions.

Conclusions about SELF:

The expressive range (in the pragmatic aspect) of TRSs in general and also of $\mathcal{KRIS}$ in particular is still under investigation (not the least, in this paper), but it is certain that reference to individuals in concept definitions is both needed for many applications and impossible to express in present TLs.

This introduction of individual references into a TRS described in this section by the (at least in $\mathcal{KRIS}$, only possible) "back door" of features constitutes, in a way, a pollution of the idea of terminological reasoning. It is however, a carefully limited one. The introduction of full general individual variables into concept definitions would have turned them into a kind of Prolog-rules (with the ABox constituting the facts), with all the semantic and computational difficulties well-known from there. This would certainly not be a wise direction for the further development of TRSs. The proposed enhancement SELF, however, constitutes a *carefully limited variable introduction*, powerful enough to solve considerably more problems than before, but still retaining decidability. It is thus a proposal worth investigating and incorporating into $\mathcal{KRIS}$. This incorporation will be conducted at DFKI.

5 The Results

The last section reported the difficulties encountered in stating descriptions of software elements as $\mathcal{KRIS}$-definitions. While these experiences account for reductions in the scope that was initially hoped to be covered, the plan laid out in Section 2.3 was carried out and resulted in the $\mathcal{KRIS}$-TBox and the ABox-generator which will be presented in this section. Their source code can be found in the appendix.

As the SELF-construction still awaits implementation, the TBox could not be tested. Accordingly, the ABox generator could only be tested by careful inspection of its output ABox, and not by loading this into $\mathcal{KRIS}$.

5.1 TBox

The mentioned reduction of scope means that the TBox contains fewer concepts than hoped for. In particular, such concepts as desribed as "semantic" in Section 4.1 could not be incorporated into the TBox, for the reasons explained there. A number of "syntactic" concepts, however, could be translated to the TBox almost completely, *assuming the* SELF-*construction*. They will be presented in the following.

5.1.1 Basic Structure

What does the TBox look like then? Remember that it was to build concepts at the architecture level from simpler ones at the source code level. Accordingly, it is founded on five basic concepts (all primitive, of course) at the source code level,

Module, Procedure, Variable, Type, Constant

which are related by two basic roles at this level:

defines and uses.

All other roles are derived from these two by specialization and inversion.

As the fundamental notion of software architecture is *locality* (deriving many others like abstraction, information hiding etc.), the concepts of LocConstruct and (complementarily) GlobConstruct were defined as the first step towards the architecture level. In programming languages with adequate scope control facilities in definitions, this might still be source code extractable and not ask for a defined (in contrast to a primitive) concept at all; but as the target language of this investigation is C, which offers little scope control (and even this is usually ignored, concerning module-local items), locality of a source code item (like a variable) is defined in the TBox not from its definition in the source, but from its *use* throughout the program. Corresponding to the program being structured in modules and these again in procedures, *three levels of locality* are distinguished:

- GlobConstruct: being accessible within the whole program
- ModLocConstruct: the same within exactly one whole module
- ProcLocConstruct: the same within one procedure only

All three concepts are mutually disjoint. The concept LocConstruct is then the disjunction of ModLocConstruct and ProcLocConstruct. Their concrete definitions are as follows, with usedBy and defBy meaning the inverses of uses and defines:

LocConstruct	:=	∀usedBy: (modOfIs = modOfIs(SELF)) i.e., anything which is used only from things defined somewhere within the same module (the modOfIs-feature gives the module containing [possibly within a procedure definition] the definition of an item).
ModLocConstruct	:=	LocConstruct ⊓ ∃ defBy: Module
ProcLocConstruct	:=	LocConstruct ⊓ ∃ defBy: Procedure

These levels of locality resulted in corresponding localizations of the five basic concepts, like ModLocVariable or ProcLocType, and of the basic roles like exports := defines.GlobConstruct. Wherever reasonable, this was done in full generality, with the locality concept of a language like Modula-2 in mind, thus in places including language constructions not possible in C (LocMod, ProcLocProc, ProcLocConst).

The localized concepts were then used to define the actual "goal" concepts of the TBox, capturing a *classification of modules* and, less complex, a *classification of procedures.*

Procedures are classified according to their side effect behaviour (SideEffectGuarded, SideEffectCausing), their frequency of use (AdHocProc, UtilityProc, DeadProc) as a weak approximation of the semantic concept of importance, and their calling behaviour (In-

terfaceProc, SystemIOProc, DirectRecurProc). These procedure concepts[8] do not require complex definitions and may thus (with the exception of InterfaceProc) be called archicture level concepts with some generosity only. Still, they are very handy in practice and save looking around in the source code.

Including the module concepts presented in the subsequent section, the TBox contains about 60 concept on the whole, among these about 40 auxiliaries like localizations, and 20 goal concepts defining the procedure and module classification. Seven roles and four features were used to define them, all of which derive from defines and uses by restriction and inversion.

5.1.2 Module Classification

The classification of modules constitutes the core content of the TBox; only here the representation of architecture level concepts succeeded to a satisfactory extent. Some results of software engineering isolating certain kinds of modules ([Nag90]) could be utilized here. Besides concepts similar to the procedure concepts above, like UtilityMod, AdHocMod, MainMod, four fundamental purposes of modules within an architecture could be translated to $\mathcal{KRIS}$-definitions with practically sufficient accuracy:

- AbstrDataTypeMod: This kind of module realizes the familiar concept of an abstract data type (ADT), i.e., it exports a type with a hidden internal structure and a collection of procedures operating on it such that the type is completely determined by the behaviour of these procedures and these procedures are the only ones accessing its internal structure.

  Such a module exports the type itself, the accessor procedures, and a constructor function returning an object of that type. It can thus be seen as a template generating objects of that type. Additionally, in order to prevent the module interface from being littered with other procedures unrelated to the ADT, all exported procedures are required to use the abstract type (this makes them true accessor procedures). This led to the following definition:

  AbstrDataTypeMod :=
  Module ⊓ ∀(exports.Procedure): ∃uses:
  (GlobType ⊓ (modOfIs=SELF) ⊓ ∃isTypeOf:
  (GlobProc ⊓ (modOfIs=SELF)))

  Here the two SELF-expressions ensure that the abstract type (the GlobType) and its constructor function (the GlobProc) really are in this module.

  An ADTAccessor is a GlobProc defined in such a module.

  Note that this definition is only a *sufficiently* accurate translation of the ADT notion: The definition does not enforce all exported procedures using this same one type – there might be several GlobType-instances exported from this module. Restricting the number of exported GlobTypes to one is not possible because atmost does not accept a restr as its role argument in the present version of $\mathcal{KRIS}$. Additionally, this definition does not grant that the internal representation of a variable instance of the GlobType defined in some other module is not accessed there (violating the abstraction). To prevent this, there would have to be a specialization of

[8] Additionally there are concepts for procedures attached to certain kinds of modules; they are explained with the respective module.

uses, call it enters, meaning an such an invasive access to the internal representation, e.g. through dereferencing (if the ADT is a pointer) or component selection (if it is a structure). Then the GlobType could be further restricted by adding the conjunct

$$\forall(\text{isTypeOf.Variable}): \forall\text{enteredBy}: (\text{modOfIs}=\text{SELF}).$$

meaning that invasive accesses are restricted to come from within the own module. Since xpass does not deliver information as detailed as such an enters, this conjunct was not added to the definition of AbstrDataTypeMod.

- AbstrDataObjMod: Such a module realizes an abstract data object (ADO), which is quite similar in purpose to an ADT, except that it is not a template for generating objects, but an object itself. It has thus no constructor function, but only accessors. Since the physical object, the AbstrDataObjVar (defined as a ModLocVar used by a GlobProc of that module), is hidden in the module, it is protected and no problems with illegal accesses from other modules arise. Again, only procedures really using the AbstrDataObjVar are allowed in the export interface, resulting in the definition
  AbstrDataObjMod :=
  $$\text{Module} \sqcap \exists\ \text{defines}: \text{AbstrDataObjVar} \sqcap \forall(\text{exports.Procedure}): (\exists\ \text{uses}: \text{AbstrDataObjVar})$$
  Again, this definition does not prevent several AbstrDataObjVars from being defined in this module.

  ADOAcessor is defined analogously to the ADT.

  An interesting specialization of AbstrDataObjMod is
  $$\text{VirtualDeviceMod} := \text{AbstrDataObjMod} \sqcap \exists\text{defines}: (\text{Procedure} \sqcap \exists\text{calls}: \text{SystemIOCall}).$$
  This defines a virtual device through some of its procedures performing physical I/O (with SystemIOCall being a primitive concept). We then have
  $$\text{VirtualIOProc} := \text{Procedure} \sqcap \exists\text{defBy}: \text{VirtualDeviceMod}.$$

- FunctionalMod: This kind of module realizes a collection of procedures which are functions in the mathematical sense, i.e., they depend exclusively on their parameters. This is neatly expressed by the concept of being both SideEffectGuarded and not SideEffectCausing. A common example for this is a module realizing a floating point library. Often these modules also export constants, so the definition reads
  $$\text{FunctionalMod} := \text{Module} \sqcap \forall\text{exports}: (\text{Constant} \sqcup (\text{SideEffectGuarded} \sqcap \neg\text{SideEffectCausing}))$$
  with
  $$\text{SideEffectGuarded} := \text{Procedure} \sqcap \forall\text{reads}: (\text{procOfIs}=\text{SELF})$$
  and
  $$\text{SideEffectCausing} := \text{Procedure} \sqcap \exists\text{writes}: (\text{procOfIs}\neq\text{SELF})$$

- DeclarationMod: This is a sort of module that contains only a list of (global) type and constant definitions and nothing local. Its definition is
  $$\text{DeclarationMod} := \text{Module} \sqcap \forall\text{exports}: (\text{Type} \sqcup \text{Constant}) \sqcap \forall\text{defines}: \text{GlobConstruct}$$

5.2 ABox-generator

The ABox-generator transforms the output of the source-code analyzer *xpass* (see Section 3.2) into assertional axioms ready to be loaded into $\mathcal{KRIS}$. Its precision of source code information thus depends on how precise xpass's output is. Unfortunately, xpass has several precision deficiencies due to its original purpose in the Arch system, which does not care for a procedure's internal structure and therefore discards such information. Judged relatively to xpass's output, however, the ABox-generator is correct, meaning that it makes explicit all relationships between source code items implied in the xpass output. It also utilizes all pieces of information extractable at all from there, and it provides hooks for future addition of the missing information to the xpass-output which it will process correctly without any modifications necessary.

Annoyingly, xpass also contains some bugs (listed below), which cannot be corrected in the ABox-generator.

5.2.1 Way of Processing

The generator consists of six programs for the UNIX string-processing language *awk* chained by a pipe, five of which[9] are auxiliaries of no more than a few dozen lines, and one, `make_ABox`, is a program of about one thousand lines of awk source doing the bulk of the job. Awk was chosen because of its built-in regular expression scanner, string processing routines, and hash tables, which were needed to read the various kinds of input lines and to store all the reference information between them and between program items. These ready-to-use facilities were expected to outweigh the shortcomings of a line-oriented language like awk, which provides no procedures, only two (even implicit) types, and no data structures except arrays and hash tables. The experience of writing `make_ABox`, however, shows that it is not a good idea to implement a task of a size even as managable as this one in awk. Especially awk's property of reading its input lines strictly sequentially causes numerous nuisances and temporal storage overhead when the input contains as many mutual references as in this case, e.g. when reading a `v_use`-line, it is not yet clear until a corresponding `v_use_within` appears whether this use is within a function, and, if so, what function uses it.

The input lines to `make_ABox` originally come from xpass. This tool parses all source files of the examined program (its command interface is identical to that of the C-compiler cc) and delivers its output in the form of a sequence of lines like

```
v_decl,34,i,process.c,11,EXTDEF
v_decl,112,i,check.c,11,EXTERN
v_use,120,i,check.c,130
f_decl,248,getData,check.c,86,STATIC
v_decl,253,v,check.c,90,AUTO
```

indicating that a globally visible variable[10] `i` was defined in file `process.c` at line 11, that such a variable is imported in file `check.c`, used in file `check.c` at line 130, that a

[9] `extract_has_decls`, `advance_has_decls`, `unique_names`, `eliminate_false_usesISS`, `close_ABox_world`

[10] In C practice, this does not necessarily imply that `i` is intended as a global variable; many C programmers use "static" only for procedure local permanent variables and not to distinguish module-local ones from global ones. This is why the TBox was explained to use not *declaration*, but *use* to determine the level of locality.

function `getData` was statically (i.e., module-locally) defined at line 86 in file `check.c`, and that an automatic variable `v` (C jargon for a produre local) was defined at line 90 in `check.c`. The number at the second place is an index sequentially counting all lines (except those like `v_refers...` which only serve as connectives between other lines). This permits references like

```
v_refers,112,34
v_use_within,120,248
v_has_type,112,7
```

to express that `i` was used by the function `getData` and that it has the type whose type definition line has the index 7. This also uniquely identifies the `i` imported at index 112 as the one defined at index 34 in `process.c`. Similar lines are given for definitions, declarations and uses of macros and types.

As awk reads its input lines sequentially, it saves much work to avoid forward references where this is easily possible. Such a job is done by `advance_has_decls`, which first uses `extract_has_decls` to collect lines referencing certain declaration lines from the xpass output file and then attaches them to the front of the file.

The next pipe stage is performed by `unique_names`, which reads the whole file to gather all information about the declaration context of all program items, and then attaches new lines like

```
unique_name,248,check.c_getData
unique_name,253,check.c_getData_v
```

connecting declaration indices with unique names constructed by prefixing with the module name and, if it is a procedure local, with the procedure name, too.

At this point, the data is still in the form of such lines, though enriched as described. The next step, however, is `make_ABox` itself printing the $\mathcal{KRIS}$-axioms, which is followed by the final `eliminate_false_usesISS` removing certain axioms which had to be printed at their time of occurence, but were invalidated by later axioms.

Now `make_ABox` reads its input line by line, and it has a matching awk-clause for each kind of possible line.[11] Program items are stored in various arrays and tables to resolve references, and finally assertional axioms are printed[12], e.g.

```
(assert-ind i Var)          ; that i is global must be inferred by KRIS
(assert-ind process.c Module)
(assert-ind process.c i defines)
(assert-ind i process.c defBy)
(assert-ind check.c Module)
(assert-ind process.c_getData ModLocProc)
(assert-ind process.c process.c_getData defines)
(assert-ind process.c_getData process.c defBy)
(assert-ind check.c_getData i uses)
(assert-ind i check.c_getData usedBy)
(assert-ind check.c i usesISS)          ; see next subsection for usesISS
```

Obviously a program produces a large number of such axioms (see Appendix C to get an

[11] awk-programs consist of a set of clauses, each of them a pair */regexp/* {*actions*}, where *regexp* is a regular expression matched with the input line and *actions* is a sequence of variable manipulations (with loops and alternatives) is executed on a successful match.

[12] Actually, the axioms are printed in the more readable format "`process.c defines i`". The `format_ABox`-utility can be used to change that into $\mathcal{KRIS}$-format.

idea of how large), more than one third of which, however, are due to the explicit listing of inverse roles, which can regrettably not be expressed by a general terminological axiom like usedBy := uses^{-1}.

The axioms can be further reduced by removing duplicates (note, however, that this should not be done by the popular UNIX-pipe sort|uniq, as this destroys the rough sequential correspondence of the axioms to the C-source).

The very last operation performed is usually piping the assertional axioms through `close_ABox_world`, which attaches the necessary number restriction axioms (see Section 4.2).

5.2.2 Policies of Source Interpretation

There are several questions concerning the intended meaning of the roles occurring in the assertional axioms and what axioms are printed at all. Here is a list of how the generator decides on such questions:

- Every source file defining a variable or a function is considered to be a module. Other files are appointed modules only when there is no corresponding `.c`-file with the same base name (see next item).

- Special care was taken to follow the customs in C programming concerning exported types and macros: Since the scopes of `typedef` and `#define` are restricted to the file which contains them, types and macros exported from a file `t.c` are implemented in C by putting their definitions into a header file like `t.h`, which is then `#included` by all client modules importing the types and macros. According to this, the actual module of such types is `t.c`, and not `t.h`, which is not considered a module at all.

- Whenever a variable or a function is used, an additional axiom indicating the use of its type is printed along with the use axiom.

- If a function `f` in module `m` uses an item `i` imported from a module different from `m`, this use is "propagated" to `m` in the form of an axiom `m` usesISS `i`, meaning that `m` uses `i` "in one of `m`'s substructures" (e.g. `f`). This is to keep `m`'s import interface clean.

- Uses of struct components are printed as uses of the whole struct-variable – after all, they *are* a part of it. A distinction of component uses would require a new role connecting a structure member to its embedding variable, say isPartOf; but, since structs may be nested, the transitive closure of isPartOf would be needed to find the embedding variable. Transitive closures of roles, however, are not possible in $\mathcal{KRIS}$.

- Procedure-local variables defined as "static" are printed as module-locals: This is motivated by the definition of an AbstrDataObjMod (see Section 5.1), because such a variable realizes an internal memory of a function which might turn it into a sort of AbstrDataObjMod, which requires a module-local variable to be recognized. This decision causes no harmful effects elsewhere, but must be kept in mind when reading the ABox.

5.2.3 Deficiencies

As was warned before, xpass neglects certain information, which propagates through `make_ABox` to the ABox. These are:

- For an external function or variable `e` imported locally by a function `l`, there is no "`f/v_decl e`" and "`f/v_decl_within e l`" printed, if `e` was already imported by the module of `l`. This is a neglible deficiency, as the local import was then unnecessary anyway.
- For parameters of a predefined type (int etc.), no `v_decl/v_decl_within` is printed.
- Variables declared in a procedure which have a predefined type are ignored completely! This is a clear consequence of xpass's purpose for Arch: Such variables are irrelevant for the external assumptions of a procedure.
- For a function returning a predefined type, no `f_has_type` is printed.
- There is no thing as `type_def_within` printed, although procedure-local types can be defined in C. Xpass prints them as usual, i.e., as module-locals.
- There is no thing as `type_uses_type` printed, although it would be helpful to know about type-type-dependencies, as between a pointer, array, or struct type and its base respectively component type.

  There *is*, however, a line `v_in_type` which is to indicate the component-struct relationship, but unfortunately it is buggy.
- There is no `type_uses_macro` or `macro_uses_macro`.

All these missing lines are serviced nonetheless in `make_ABox`, i.e., the corresponding clauses are fully implemented there, awaiting future use. It is conceivable to write auxiliary tools extracting this missing information from the C-source. This would, however, have exceeded the scope of this work.

5.2.4 Bugs

Additionally to this missing information, there are some plain bugs in xpass, too:

- A use of a procedure-local variable of a predefined type is falsely printed as a use of a global variable of the same name, if such a global variable exists.
- The same happens with a procedure-local type.
- `#undef`s are foolishly understood as macro uses and should be avoided altogether.
- Structure components of the same name from two different struct-types are not distinguished: no `v_decl` etc. appears for the second set of components, and uses of them are falsely printed as uses of the first set of components.
- Procedure-local "static" variables are printed as module-locals (i.e., no `v_decl_within` appears for them). But this happens to be just what the TBox expects about statics (see the last of the policies above) – so this bug is lucky for our purpose here!
- Pointers and array variables are treated as if being variables of the base type only.

6 The Insights

This closing section sums up the experiences gained in this investigation, gives a tentative evaluation of the original idea of applying TRSs to application-independent software analysis, and speculates about its possible future development. Again, all this is handicapped by the missing experiments with the TBox applied to real software.

The probably most prominent insight can be stated as follows: The architecture of real present software systems is generally not automatically extractable by source code analysis. While this goal was never really thought to be *completely* achievable (see Section 1.1.2), there was hope that this might be possible at least to a large extent. However, this investigation should be seen as evidence contributing to the conjecture that such an extraction is presently possible to a very limited extent only.

Still, even this limited extent, as outlined by the TBox, can be of practical help to reverse engineers.

What experiences support this sobering general judgement?

Both theory and (even more) practice of software architecture are presently not mature, let alone unified. Practice may even be called irregular. Real software applies too few standard concepts even of the small set evolved so far, due to deficiencies in software development practice and programmer education, other (conflicting) parameters of software being preferred, lack of adequate development tools and probably many more reasons. Therefore the practical value even of the modest module classification developed in this paper must be estimated with caution: Most programmers would probably not commit themselves to writing only modules whose architectural function is as clear as in this classification, but would rather mix them for ease of expression, efficiency, different preferences or simply carelessness.

Furtheron, all widely-used programming languages are insufficiently elaborated from the software architecture point of view (scope control, modularity, different kinds of modules) and thus do not offer enough architecture information in the source code. Fortunately, there is strong evidence that this will improve in the course of the general progress of programming, but also notably so by the spreading of object-oriented languages.

The observation that the meaning of many programming concepts can only be captured in run-time terms like "purpose" rather than in terms of compile-time like "reference" severely limits the static approach of source code analysis. Most concepts of software architecture belong to the former terms and thus will not be captured by source code analysis. They might, however, be tackled by the dynamical analysis of a run-time simulation. Work in this area is, however, only just beginning.

When trying to express such concepts of purpose in terms of syntax nonetheless, the ability to express vague knowledge or default knowledge becomes essential. This fundamentally handicaps a logical language like a TL. Leaving aside hypothetical extensions of logics, heuristic algorithms seem to be a good method to achieve better results here, as is witnessed by the example of Arch. These cannot be incorporated into any existing TRS, but there is work beginning in that direction. Another path out of this problem is that of allowing those concepts which are not source code definable (like vague concepts and "purpose" concepts) as high-level primitives in a TBox. While they could of course not be automatically recognized, this would at least provide the reverse engineer with a language to record his hand-extracted knowledge about such elements of the program in the same

format as the automatically generated information. Possibly even more complex concepts can then be built on top of these high-level primitives, subject to automatic classification again, once the user has supplied the ABox assertions naming instances of the high-level primitives.

Putting aside all theoretical difficulties in knowledge, its representation, and implementation, a practical RE-tool will also need the full facilities of a relational data base, in particular concerning the query interface and defining ad-hoc-expressions. Since inference is of course still the bulk of the job, it may be worthwile to examine how well deductive data bases instead of TRSs would perform on this job.

It must be noted that all this pertains *directly* to application-independent software analysis only, as was the approach of our investigation here. *Application-specific approaches* (like [DBSB90]) will achieve a larger codex of knowledge and consequently more inferences. However, the difficulties reported in Section 4.1 apply in principle to them as well. But the limits can be stretched: the achievable scope of representation of course depends on how much of the application domain is formally defined. For any application domain amenable to a computer approach at all, such a formalization is partly possible and delivers new concept definitions. Application-specific software analysis therefore seems to have a more promising future. The formally defined fraction of the domain might be considerably increased by an addition to the chores of software development: The developer would define logical descriptions (preferably concept terms for a TRS) of the implemented concepts in parallel to developing these concepts themselves. This would provide a TRS for RE with optimal, "first hand" knowledge. The fate of similar suggestions concerning program proofs, however, probably reduces this suggestion to a naive hope.

So where are we arrived now? What is left of the initial idea, and what have we achieved?

In general, the observation that a part of software analysis is inference, specifically that architecture acquisition can be supported by automatic concept recognition, this observation is believed here to have shown correct and worth further work. In particular, however, the value of this investigation is judged to consist less of formulating a few of such concept definitions, than of giving a necessary clarification of the vague idea as it was at the beginning. We now know better how to tackle the task and what can be done and, perhaps even more useful, what cannot be done with the described means.

The experiences made here with software as a complex structure which is, though formal in syntax and semantics, most prominently *human-created*, are just another piece of evidence for the insight that generally the achievable usefulness of a formal inference system (not restricted to TRSs) for understanding such human-created complex structures is less determined by that system's inference and representation capabilities than by our exact knowledge of how and what for we create these structures at all.

As so often in computer science, our fundamental task here is not to devise better algorithms (indispensable, but coming second), but first to find better languages of our own thinking.

References

[Baa90a] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. Technical report, German Research Center for Artificial Intelligence (DFKI), DFKI, Postfach 2080, D-6750 Kaiserslautern, Germany, 1990.

[Baa90b] F. Baader. Terminological cycles in KL-ONE-based knowledge representation languages. In *Proceedings of the 8th National Conference of the AAAI*, pages 621–626, Boston, Mas., 1990.

[BBMR89] A. Borgida, R.J. Brachman, D.L. McGuiness, and A. Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the International Conference on Management of Data*, Portland, Oregon, 1989.

[BH90] F. Baader and B. Hollunder. $\mathcal{KRIS}$: Knowledge representation and inference system – system description. Technical report, German Research Center for Artificial Intelligence (DFKI), DFKI, Postfach 2080, D-6750 Kaiserslautern, Germany, November 1990.

[BL84] R.J. Brachman and H.J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the 4th National Conference of the AAAI*, pages 34–37, Austin, Tex., 1984.

[BPGL85] R.J. Brachman, V. Pigman-Gilbert, and H. Levesque. An essential hybrid reasoning system: Knowledge and symbol level account of krypton. In *Proceedings of the 9th IJCAI*, pages 532–539, Los Angeles, Cal., 1985.

[BS85] R.J. Brachman and J. Schmolze. An overview of the KL-ONE-knowledge representation system. *Cognitive Science*, 9(2):171–216,, April 1985.

[DBSB90] P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard. Lassie: a knowledge-based software information system. In *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, April 1990.

[GLKT90] W. Gruber, K. Lebsanft, Th. Keller, and H.G. Tempel. Reverse engineering. Technical Report BeA002/91, Siemens AG, Dept. ZFE IS SOF13, Munich, Germany, December 1990.

[GMN+87] W. Gertke, S. Mittrach, G. Normann, G. Schulz, and S. Zorn. Studie zur Situation bei der Wartung und Pflege großer Software-Systeme im Haus. Technical report, Siemens AG, Dept. ZFE, Munich, Germany, July 1987.

[Hru90] P. Hruschka. Wiederverwendbarkeit in komplexen COBOL-Systemen. In R. Thurner, editor, *Re-Engineering – ein integrales Wartungskonzept zum Schutz von Software-Investitionen. Strategien – Methoden – Werkzeuge*, Hallbergmoos, Germany, 1990. AIT Angewandte Informationstechnik GmbH.

[KBR86] T.S. Kaczmarek, R. Bates, and G. Robins. Recent developments in NIKL. In *Proceedings of the 5th National Conference of the AAAI*, pages 578–587, Philadelphia, Pa., 1986.

[Kob89] A. Kobsa. The SB-ONE knowledge representation workbench. In *Preprints of the Workshop on Formal Aspects of Semantic Networks*, Two Harbors, Cal., February 1989.

[MB87] R. McGregor and M. Bates. The LOOM knowledge representation language. Technical Report ISI/RS-87-188, Univ. of Southern California, Information Science Institute, Marina del Rey, Cal., 1987.

[McC80] J. McCarthy. Circumscription – a form of nonmonotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[Nag90] M. Nagl. *Methodisches Programmieren im Großen*. Springer Verlag, 1990.

[Neb88] B. Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, 1988.

[Neb89] B. Nebel. Terminological cycles: Semantics and computational properties. In *Proceedings of the Workshop on Formal Aspects of Semantic Networks*, Two Harbors, Cal., February 1989.

[Neb90] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Number 422 in Lecture Notes in AI. Springer, 1990.

[NvL88] B. Nebel and K. von Luck. Hybrid reasoning in BACK. In *Methodologies for Intelligent Systems*, pages 260–269. North Holland, 1988.

[Par72] D. L. Parnas. Information distribution aspects of design methodology. In *Information Processing 71*. North Holland, 1972.

[PS84] P. Patel-Schneider. Small can be beautiful in knowledge representation. In *Proceedings of the IEEE-workshop on Principles of Knowledge-Based Systems*, pages 11–16, Denver, Col., 1984.

[Sch90] R. W. Schwanke. An intelligent tool for re-engineering software modularity. Technical Report SCR-91-TR-319, Siemens AG, Corporate Research Dept., Princeton, USA, 1990.

[Vil85] M.B. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the 9th IJCAI*, pages 547–551, Los Angeles, Cal., 1985.

A Source Code of the TBox

```
;    This is the T-Box of the recognized software concepts in KRIS-format:

;  SOME ROLES:


(defprimrole defines)

(defprimrole uses)

(defprimrole usedBy)
;  the inverse of uses

(defprimrole isTypeOf usedBy)
;  returns the functions and Variables of a Type



;  ALL THE FEATURES:

(defprimattribute defBy)
;  returns the defining Procedure or Module

(defprimattribute modOfIs)
;   returns the Module of a Construct
;   note: modOfIs(m) equals m  for all Modules m

(defprimattribute procOfIs)
;  analogously with Procedures

(defprimattribute typeOfIs)
;   returns the Type of a Function or Variable
;   as a matter of fact, this ought to be a subfeature of uses - however, since
;   uses is a role and typeOf is feature, KRIS does not allow this relationship
;   to be expressed within the TBox. The ABox-generator has to ensure that uses
;   is duly annoted with every typeOf.


;   SOME CONCEPTS:

;   THESE 5 CONCEPTS FORM THE BUILDING MATERIAL OF THE T-BOX:

(defconcept Module (equals modOfIs SELF))
;   the definition part is just a security check - a correct ABox grants this.

(defconcept Procedure (equals prodOfIs SELF))
;   the definition part is just a security check - a correct ABox grants this.

(defconcept Function (and Procedure (some typeOfIs Type))

(defprimconcept Constant)

(defprimconcept Type)

(defprimconcept Variable)
```

```
;  SOME ROLES:

(defprimrole reads (restr uses Variable))

(defprimrole writes (restr uses Variable))



;  MORE CONCEPTS:

(defconcept ActiveConstruct (or Module Procedure))

(defconcept PassiveConstruct (or Variable Type Constant))

(defconcept SubmoduleConstruct (or PassiveConstruct Procedure))

(defconcept GlobMod (and Module (not (some defBy *top*))))

(defconcept LocMod  (and Module (some defBy Module)))

(defprimconcept GlobConstruct
   (or (and SubmoduleConstruct (some usedBy (not-equals modOfIs modOfIs(SELF))))
       GlobMod))
;  this is primitive because even a variable that is never used from outside
;  can be a GlobVar if the programmer defined it as global.


(defconcept LocConstruct (or LocMod (forall usedBy
                                          (equals modOfIs modOfIs(SELF)))))

(defconcept ProcLocConstruct (and LocConstruct (some defBy Procedure)))

(defconcept ModLocConstruct (and LocConstruct (some defBy Module)))

(defconcept Construct (or GlobConstruct LocConstruct))
;   this is intended to be the root of the concept taxonomy

(defprimconcept Predefined GlobalConstr)

(defconcept PreDefType (and Type Predefined))

(defprimconcept BooleanType PreDefType)

(defconcept GlobProc  (and Procedure GlobConstruct))

(defconcept LocProc  (and Procedure LocConstruct))

(defconcept ProcLocProc  (and Procedure ProcLocConstruct))

(defconcept ModLocProc  (and Procedure ModLocConstruct))

(defconcept GlobType  (and Type GlobConstruct))

(defconcept LocType  (and Type LocConstruct))
```

```
(defconcept ModLocType  (and Type ModLocConstruct))

(defconcept ProcLocType  (and Type ProcLocConstruct))

(defconcept GlobVar  (and Variable GlobConstruct))

(defconcept LocVar  (and Variable LocConstruct))

(defconcept ProcLocVar  (and Variable ProcLocConstruct))

(defconcept ModLocVar  (and Variable ModLocConstruct))

(defconcept GlobConst  (and Constant GlobConstruct))

(defconcept LocConst  (and Constant LocConstruct))

(defconcept ProcLocConst  (and Constant ProcLocConstruct))

(defconcept ModLocConst (and Constant ModLocConstruct))


(defconcept AbstrDataObjVar (and ModLocVar (some usedBy InterfaceProc)))
;   the Var that is encapsulated in an AbstrDataObjMod


(defprimconcept InputParameter ProcLocVar)
;  we cannot ensure here that this parameter is actually used for input only -
;  it may also be a pointer which is derefenced and then written into!

(defprimconcept OutputParameter ProcLocVar)

(defconcept Parameter (or InputParameter OutputParameter))

(defconcept StackVar (and ProcLocVar (not Parameter)))


;  SOME ROLES

(defrole definesLocally (restr defines LocConstruct))

(defrole exports (restr defines GlobConstr))

(defprimrole calls (restr uses Procedure))

(defprimrole usesISS (restr uses GlobConstruct))
;  a Module usesISS ("in substructure") an extern construct iff the use occurs
;  within a Procedure of the Module and not within the Module's declaration
;  or statement part (note: there is no Module statement part in C anyway).

(defprimrole usedISSBy)
;   the inverse of usesISS



;  THE GOAL CONCEPTS


;  PROCEDURE CLASSIFICATION:
```

```
(defconcept InterfaceProc (and GlobProc (forall usedBy
                                        (not-equals modOfIs modOfIs(SELF)))))

(defconcept AdHocProc (and Procedure (atmost 1 usedBy )))

(defconcept UtilityProc (and Procedure (atleast 2 usedBy)))

(defconcept DeadProc  (and Proc (atmost 0 usedBy)))


(defconcept SideEffectGuarded (and Procedure (forall reads
                                                    (equals procOfIs SELF))))

(defconcept SideEffectCause (and Procedure (some writes
                                                  (not-equals procOfIs SELF))))

(defprimconcept SystemIOCall (and Procedure Predefined))

(defconcept SystemIOProc (and Procedure (some calls SystemIOCall)))

(defconcept TestProc  (and Procedure (some typeOfIs BooleanType)
                           (not SideEffectCausing)))
(defconcept DirectRecurProc
            (and Procedure (some calls (equals procOfIs SELF))))


;  MODULE CLASSIFICATION:


(defconcept UtilityMod (or (atleast 2 usedBy)
                           (some defines (atleast 2 usedISSBy))))


(defconcept MainMod (and Module (not (some
                     (restr defines (or (some usedBy (not-equals modOfIs SELF))
                                        (some usedISSBy
                                             (not-equals modOfIs SELF))))))))
; the constructs of in this module are not used from other modules, so it must
; be the module containing the top-level procedure.


(defconcept AdHocMod (and Module (atmost 1 usedBy)
                                    (forall defines (atmost 1 usedISSBy))))


(defconcept DeclarationMod (and Module (forall exports (or Type Constant))
                                        (forall defines GlobConstr)))


(defconcept FunctionalMod (and Module
                              (forall exports (or (and SideEffectGuarded
                                                       (not SideEffectCausing))
                                                  Constant))))
;  no GlobTypes allowed to distinguish it from AbstrDataTypeMod

;  ModLocVars and uses of GlobVars need not be forbidden, as they would not
;  have any effect anyway: the procedures are all SideEffectGuarded, and
```

```
;  there is no other channel for information to flow out of the module
;  except the procedure parameters.

(defconcept AbstrDataObjMod (and Module
                                 (some defines AbstrDataObjVar)
                                 (forall (restr exports Procedure)
                                         (some uses AbstrDataObjVar))))


(defconcept AbstrDataTypeMod
            (and Module
                 (forall (restr exports Procedure)
                         (some uses (and GlobType
                                         (equals modOfIs SELF)
                                         (some isTypeOf
                                               (and GlobProc
                                                    (equals modOfIs SELF)))))))


(defconcept VirtualDeviceMod (and AbstrDataObjMod (some defines SystemIOProc)))


; PROCEDURES ATTACHED TO THESE MODULES:


(defconcept VirtualIOProc (and GlobProc (some defBy VirtualDeviceMod)))

(defconcept ADTAccessor   (and GlobProc (some defBy AbstrDataTypeMod)))

(defconcept ADOAccessor   (and GlobProc (some defBy AbstrDataObjMod)))
```

B Example Program with its ABox

Here is the main module `client.c`, which uses the ADT-module `wordcount.c` and the ADO-module `wordcountTable.c`. The ABox ouput by the generator follows them.

```
/*           F I L E   wordcount.h                */

#define WORDLEN       32

#define INVALID       -1

typedef int           Bool;
#define TRUE          1
#define FALSE         0

typedef struct {
           char word[WORDLEN];
           int  count;
       }
           WordCount;

extern WordCount makeWC();
```

```
extern void      deleteWC();
extern void      printWC();

extern int       getWCount(), setWCount();


/*              F I L E   wordcount.c               */

/*  realizes an abstract data type "WordCount" asscociating a counter with
    a string                                                              */

#include "wordcount.h"

WordCount makeWC(name, initCount)
   char *name;     /* initialization parameters */
   int initCount;
{
   WordCount *newWCp;

   newWCp = (WordCount *) malloc(sizeof(WordCount));
   strncpy(newWCp->word, name, WORDLEN);
   newWCp->count = initCount;
   return *newWCp;
}


void deleteWC(wc)
   WordCount wc;
{
   free(&wc);
}


void printWC(wc)
  WordCount wc;
{
  printf("# of %s's = %d\n", wc.word, wc.count);
}


int getCount(wc)
  WordCount wc;
{
  return wc.count;
}


int setCount(wcp, newCount)    /* returns the old counter */
  WordCount *wcp;
  int       newCount;
{
  int oldCount;

  oldCount = wcp->count;
  wcp->count = newCount;
  return oldCount;
}
```

```
Bool valid(wc)
  WordCount wc;
{
  return wc.count != INVALID;
}


Bool fits(wc, name)        /* compares the word to a search name */
  WordCount wc;
  char *name;
{
  return !strncmp(wc.word, name, WORDLEN);
}

/*                  F I L E  wordcountTable.h            */

#define MAX_WCTABLE_LEN 100

extern void      initWCTable();
extern void      printWCTable();
extern int       lengthWCTable();
extern Bool      putWC();
extern WordCount getWC();

/*                F I L E  wordcountTable.c           */

/*  realizes an abstract data object (the table) as an arry of instances of
    the abstract data type "WordCount"                                    */

#include "wordcount.h"
#include "wordcountTable.h"

static int currTableLen;
static WordCount table[MAX_WCTABLE_LEN];       /* the table */
  /* As a matter of fact, one would usually cluster these two variables in
     one struct variable. This would correspond to their semantics in a better
     way, and it would also make "wordcountTable.c" an "AbstrDataObjMod"
     in the sense  i n t e n d e d  by the appropriate TBox-definition
     (i.e. there is exactly one "AbstrDataObjVar" referred to by all exported
     procedures. However, since xpass's struct-handling is bugged, and since
     the TBox-definition (see there!) does not enforce there being exactly one
     "AbstrDartaObjVar", we can leave the table in two distinct variables
     as it is here, and "wordcountTable.c" will be classified as an
     "AbstrDataObjMod" nevertheless!                                  */

static WordCount invalidWC; /* dummy signalling unknown entry */


void initWCTable()
{
  invalidWC = makeWC("", INVALID);   /* initialize the dummy */
  currTableLen = 0;
}


void printWCTable()
{
```

```
  int i;

  for(i=0; i < currTableLen; i++)
     printWC(table[i]);
}


int lengthWCTable()
{
  return currTableLen;
}


WordCount getWC(name)  /* returns an invalid WordCount, if name unknown */
   char *name;
{
   int i;

   for(i=0; i < currTableLen; i++)
     if(fits(table[i], name))
       return table[i];
   return invalidWC;  /* not found! */
}

Bool putWC(name, value) /* returns success or failure (due to full table)   */
   char *name;          /* and makes a new entry if there was none for name */
   int  value;
{
   int i;

   for(i=0; i < currTableLen; i++)
     if(fits(table[i], name))
       break;
   if(i < currTableLen) /* found */
     setCount(&table[i], value);
   else /* not found: make a new entry, unless table full */
     if(currTableLen == MAX_WCTABLE_LEN) /* table full */
       return FALSE;
     else /* make a new entry */
       table[currTableLen++] = makeWC(name, value);
   return TRUE;
}


/*           F I L E   client.c             */

/*  uses the abstract data object defined in "wordcountTable" and the
    abstract data type "WordCount"                                    */

/* #include "stdio.h" */
#include "wordcount.h"
#include "wordcountTable.h"

main()  /* fills an external WordCount-Table with counted occurences */
        /* of words from the stdin-stream, and finally prints it, as */
        /* soon as the word "stop" appears                           */
{
  char name[WORDLEN];
```

```
  WordCount wc;

  initWCTable();
  scanf("%s", name);
  while(strncmp(name, "stop", WORDLEN))   /* while name != stop */
  {
    if(valid(wc = getWC(name)))  /* we had this word (name) before */
      putWC(name, getCount(wc) + 1); /* increment its counter */
    else
      putWC(name, 1);       /* this is the first occurence of this name */
    scanf("%s", name);      /* get next word */
  }
  printWCTable();
}

            HERE COMES THE A-BOX OF THE PROGRAM:


BooleanType int
BooleanType BOOL
BooleanType BOOLEAN
BooleanType Bool
BooleanType Boolean
PreDefType int
PreDefType char
PreDefType short
PreDefType long
makeWC uses ./wordcount.h_WordCount
makeWC typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf makeWC
putWC uses ./wordcount.h_Bool
putWC typeOfIs ./wordcount.h_Bool
./wordcount.h_Bool isTypeOf putWC
getWC uses ./wordcount.h_WordCount
getWC typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf getWC
Module client.c
client.c modOfIs client.c
Procedure main
client.c defines main
main modOfIs client.c
main procOfIs main
main defBy client.c
main uses ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedBy main
client.c usesISS ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedISSBy client.c
client.c_main_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf client.c_main_wc
client.c_main_wc uses ./wordcount.h_WordCount
main uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy main
client.c usesISS ./wordcount.h_WordCount
./wordcount.h_WordCount usedISSBy client.c
StackVar client.c_main_wc
main defines client.c_main_wc
client.c_main_wc procOfIs main
client.c_main_wc defBy main
```

```
client.c_main_wc modOfIs client.c
main uses initWCTable
initWCTable usedBy main
main uses scanf
scanf usedBy main
main uses strncmp
strncmp usedBy main
main uses ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedBy main
client.c usesISS ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedISSBy client.c
main uses valid
valid usedBy main
main uses client.c_main_wc
client.c_main_wc usedBy main
main uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy main
main uses getWC
getWC usedBy main
main uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy main
main uses putWC
putWC usedBy main
main uses ./wordcount.h_Bool
./wordcount.h_Bool usedBy main
main uses getCount
getCount usedBy main
main uses client.c_main_wc
client.c_main_wc usedBy main
main uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy main
main uses putWC
putWC usedBy main
main uses ./wordcount.h_Bool
./wordcount.h_Bool usedBy main
main uses scanf
scanf usedBy main
main uses printWCTable
printWCTable usedBy main
Module wordcountTable.c
wordcountTable.c modOfIs wordcountTable.c
ModLocVar wordcountTable.c_currTableLen
wordcountTable.c defines wordcountTable.c_currTableLen
wordcountTable.c_currTableLen modOfIs wordcountTable.c
wordcountTable.c_currTableLen defBy wordcountTable.c
ModLocVar wordcountTable.c_table
wordcountTable.c defines wordcountTable.c_table
wordcountTable.c_table modOfIs wordcountTable.c
wordcountTable.c_table defBy wordcountTable.c
wordcountTable.c_table typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcountTable.c_table
wordcountTable.c_table uses ./wordcount.h_WordCount
ModLocVar wordcountTable.c_invalidWC
wordcountTable.c defines wordcountTable.c_invalidWC
wordcountTable.c_invalidWC modOfIs wordcountTable.c
wordcountTable.c_invalidWC defBy wordcountTable.c
wordcountTable.c_invalidWC typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcountTable.c_invalidWC
```

```
wordcountTable.c_invalidWC uses ./wordcount.h_WordCount
Procedure initWCTable
wordcountTable.c defines initWCTable
initWCTable modOfIs wordcountTable.c
initWCTable procOfIs initWCTable
initWCTable defBy wordcountTable.c
initWCTable uses wordcountTable.c_invalidWC
wordcountTable.c_invalidWC usedBy initWCTable
initWCTable uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy initWCTable
initWCTable uses makeWC
makeWC usedBy initWCTable
initWCTable uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy initWCTable
initWCTable uses ./wordcount.h_INVALID
./wordcount.h_INVALID usedBy initWCTable
wordcountTable.c usesISS ./wordcount.h_INVALID
./wordcount.h_INVALID usedISSBy wordcountTable.c
initWCTable uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy initWCTable
Procedure printWCTable
wordcountTable.c defines printWCTable
printWCTable modOfIs wordcountTable.c
printWCTable procOfIs printWCTable
printWCTable defBy wordcountTable.c
printWCTable uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy printWCTable
printWCTable uses printWC
printWC usedBy printWCTable
printWCTable uses wordcountTable.c_table
wordcountTable.c_table usedBy printWCTable
printWCTable uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy printWCTable
Procedure lengthWCTable
wordcountTable.c defines lengthWCTable
lengthWCTable modOfIs wordcountTable.c
lengthWCTable procOfIs lengthWCTable
lengthWCTable defBy wordcountTable.c
lengthWCTable uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy lengthWCTable
Procedure getWC
wordcountTable.c defines getWC
getWC modOfIs wordcountTable.c
getWC procOfIs getWC
getWC defBy wordcountTable.c
getWC uses ./wordcount.h_WordCount
getWC typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf getWC
getWC uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy getWC
getWC uses fits
fits usedBy getWC
getWC uses wordcountTable.c_table
wordcountTable.c_table usedBy getWC
getWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy getWC
getWC uses wordcountTable.c_table
wordcountTable.c_table usedBy getWC
```

```
getWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy getWC
getWC uses wordcountTable.c_invalidWC
wordcountTable.c_invalidWC usedBy getWC
getWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy getWC
Procedure putWC
wordcountTable.c defines putWC
putWC modOfIs wordcountTable.c
putWC procOfIs putWC
putWC defBy wordcountTable.c
putWC uses ./wordcount.h_Bool
putWC typeOfIs ./wordcount.h_Bool
./wordcount.h_Bool isTypeOf putWC
putWC uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy putWC
putWC uses fits
fits usedBy putWC
putWC uses wordcountTable.c_table
wordcountTable.c_table usedBy putWC
putWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy putWC
putWC uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy putWC
putWC uses setCount
setCount usedBy putWC
putWC uses wordcountTable.c_table
wordcountTable.c_table usedBy putWC
putWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy putWC
putWC uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy putWC
putWC uses ./wordcountTable.h_MAX_WCTABLE_LEN
./wordcountTable.h_MAX_WCTABLE_LEN usedBy putWC
putWC uses ./wordcount.h_FALSE
./wordcount.h_FALSE usedBy putWC
wordcountTable.c usesISS ./wordcount.h_FALSE
./wordcount.h_FALSE usedISSBy wordcountTable.c
putWC uses wordcountTable.c_table
wordcountTable.c_table usedBy putWC
putWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy putWC
putWC uses wordcountTable.c_currTableLen
wordcountTable.c_currTableLen usedBy putWC
putWC uses makeWC
makeWC usedBy putWC
putWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy putWC
putWC uses ./wordcount.h_TRUE
./wordcount.h_TRUE usedBy putWC
wordcountTable.c usesISS ./wordcount.h_TRUE
./wordcount.h_TRUE usedISSBy wordcountTable.c
Module wordcount.c
wordcount.c modOfIs wordcount.c
Procedure makeWC
wordcount.c defines makeWC
makeWC modOfIs wordcount.c
makeWC procOfIs makeWC
```

```
makeWC defBy wordcount.c
makeWC uses ./wordcount.h_WordCount
makeWC typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf makeWC
wordcount.c_makeWC_newWCp typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp uses ./wordcount.h_WordCount
makeWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy makeWC
StackVar wordcount.c_makeWC_newWCp
makeWC defines wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp procOfIs makeWC
wordcount.c_makeWC_newWCp defBy makeWC
wordcount.c_makeWC_newWCp modOfIs wordcount.c
makeWC uses wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp usedBy makeWC
makeWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy makeWC
makeWC uses malloc
malloc usedBy makeWC
makeWC uses strncpy
strncpy usedBy makeWC
makeWC uses wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp usedBy makeWC
makeWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy makeWC
makeWC uses ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedBy makeWC
makeWC uses wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp usedBy makeWC
makeWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy makeWC
makeWC uses wordcount.c_makeWC_newWCp
wordcount.c_makeWC_newWCp usedBy makeWC
makeWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy makeWC
Procedure deleteWC
wordcount.c defines deleteWC
deleteWC modOfIs wordcount.c
deleteWC procOfIs deleteWC
deleteWC defBy wordcount.c
wordcount.c_deleteWC_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_deleteWC_wc
wordcount.c_deleteWC_wc uses ./wordcount.h_WordCount
deleteWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy deleteWC
Parameter wordcount.c_deleteWC_wc
deleteWC defines wordcount.c_deleteWC_wc
wordcount.c_deleteWC_wc procOfIs deleteWC
wordcount.c_deleteWC_wc defBy deleteWC
wordcount.c_deleteWC_wc modOfIs wordcount.c
deleteWC uses free
free usedBy deleteWC
deleteWC uses wordcount.c_deleteWC_wc
wordcount.c_deleteWC_wc usedBy deleteWC
deleteWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy deleteWC
Procedure printWC
```

```
wordcount.c defines printWC
printWC modOfIs wordcount.c
printWC procOfIs printWC
printWC defBy wordcount.c
wordcount.c_printWC_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_printWC_wc
wordcount.c_printWC_wc uses ./wordcount.h_WordCount
printWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy printWC
Parameter wordcount.c_printWC_wc
printWC defines wordcount.c_printWC_wc
wordcount.c_printWC_wc procOfIs printWC
wordcount.c_printWC_wc defBy printWC
wordcount.c_printWC_wc modOfIs wordcount.c
printWC uses printf
printf usedBy printWC
printWC uses wordcount.c_printWC_wc
wordcount.c_printWC_wc usedBy printWC
printWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy printWC
printWC uses wordcount.c_printWC_wc
wordcount.c_printWC_wc usedBy printWC
printWC uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy printWC
Procedure getCount
wordcount.c defines getCount
getCount modOfIs wordcount.c
getCount procOfIs getCount
getCount defBy wordcount.c
wordcount.c_getCount_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_getCount_wc
wordcount.c_getCount_wc uses ./wordcount.h_WordCount
getCount uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy getCount
Parameter wordcount.c_getCount_wc
getCount defines wordcount.c_getCount_wc
wordcount.c_getCount_wc procOfIs getCount
wordcount.c_getCount_wc defBy getCount
wordcount.c_getCount_wc modOfIs wordcount.c
getCount uses wordcount.c_getCount_wc
wordcount.c_getCount_wc usedBy getCount
getCount uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy getCount
Procedure setCount
wordcount.c defines setCount
setCount modOfIs wordcount.c
setCount procOfIs setCount
setCount defBy wordcount.c
wordcount.c_setCount_wcp typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_setCount_wcp
wordcount.c_setCount_wcp uses ./wordcount.h_WordCount
setCount uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy setCount
Parameter wordcount.c_setCount_wcp
setCount defines wordcount.c_setCount_wcp
wordcount.c_setCount_wcp procOfIs setCount
wordcount.c_setCount_wcp defBy setCount
wordcount.c_setCount_wcp modOfIs wordcount.c
```

```
setCount uses wordcount.c_setCount_wcp
wordcount.c_setCount_wcp usedBy setCount
setCount uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy setCount
setCount uses wordcount.c_setCount_wcp
wordcount.c_setCount_wcp usedBy setCount
setCount uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy setCount
Procedure valid
wordcount.c defines valid
valid modOfIs wordcount.c
valid procOfIs valid
valid defBy wordcount.c
valid uses ./wordcount.h_Bool
valid typeOfIs ./wordcount.h_Bool
./wordcount.h_Bool isTypeOf valid
wordcount.c_valid_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_valid_wc
wordcount.c_valid_wc uses ./wordcount.h_WordCount
valid uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy valid
Parameter wordcount.c_valid_wc
valid defines wordcount.c_valid_wc
wordcount.c_valid_wc procOfIs valid
wordcount.c_valid_wc defBy valid
wordcount.c_valid_wc modOfIs wordcount.c
valid uses wordcount.c_valid_wc
wordcount.c_valid_wc usedBy valid
valid uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy valid
valid uses ./wordcount.h_INVALID
./wordcount.h_INVALID usedBy valid
Procedure fits
wordcount.c defines fits
fits modOfIs wordcount.c
fits procOfIs fits
fits defBy wordcount.c
fits uses ./wordcount.h_Bool
fits typeOfIs ./wordcount.h_Bool
./wordcount.h_Bool isTypeOf fits
wordcount.c_fits_wc typeOfIs ./wordcount.h_WordCount
./wordcount.h_WordCount isTypeOf wordcount.c_fits_wc
wordcount.c_fits_wc uses ./wordcount.h_WordCount
fits uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy fits
Parameter wordcount.c_fits_wc
fits defines wordcount.c_fits_wc
wordcount.c_fits_wc procOfIs fits
wordcount.c_fits_wc defBy fits
wordcount.c_fits_wc modOfIs wordcount.c
fits uses strncmp
strncmp usedBy fits
fits uses wordcount.c_fits_wc
wordcount.c_fits_wc usedBy fits
fits uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy fits
fits uses ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN usedBy fits
```

```
wordcount.c uses strncmp
strncmp usedBy wordcount.c
wordcount.c uses printf
printf usedBy wordcount.c
wordcount.c uses free
free usedBy wordcount.c
wordcount.c uses strncpy
strncpy usedBy wordcount.c
wordcount.c uses malloc
malloc usedBy wordcount.c
wordcountTable.c uses setCount
setCount usedBy wordcountTable.c
wordcountTable.c uses fits
fits usedBy wordcountTable.c
wordcountTable.c uses ./wordcount.h_Bool
./wordcount.h_Bool usedBy wordcountTable.c
client.c uses getCount
getCount usedBy client.c
client.c uses valid
valid usedBy client.c
client.c uses ./wordcount.h_Bool
./wordcount.h_Bool usedBy client.c
client.c uses strncmp
strncmp usedBy client.c
client.c uses scanf
scanf usedBy client.c
./wordcountTable.h uses getWC
getWC usedBy ./wordcountTable.h
./wordcountTable.h uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy ./wordcountTable.h
./wordcountTable.h uses putWC
putWC usedBy ./wordcountTable.h
./wordcountTable.h uses ./wordcount.h_Bool
./wordcount.h_Bool usedBy ./wordcountTable.h
./wordcountTable.h uses lengthWCTable
lengthWCTable usedBy ./wordcountTable.h
./wordcountTable.h uses printWCTable
printWCTable usedBy ./wordcountTable.h
./wordcountTable.h uses initWCTable
initWCTable usedBy ./wordcountTable.h
./wordcount.h uses setWCount
setWCount usedBy ./wordcount.h
./wordcount.h uses getWCount
getWCount usedBy ./wordcount.h
./wordcount.h uses printWC
printWC usedBy ./wordcount.h
./wordcount.h uses deleteWC
deleteWC usedBy ./wordcount.h
./wordcount.h uses makeWC
makeWC usedBy ./wordcount.h
./wordcount.h uses ./wordcount.h_WordCount
./wordcount.h_WordCount usedBy ./wordcount.h
./wordcountTable.h uses
 usedBy ./wordcountTable.h
./wordcount.h uses
 usedBy ./wordcount.h
Type ./wordcount.h_WordCount
wordcount.c defines ./wordcount.h_WordCount
```

```
./wordcount.h_WordCount modOfIs wordcount.c
./wordcount.h_WordCount defBy wordcount.c
Type ./wordcount.h_Bool
wordcount.c defines ./wordcount.h_Bool
./wordcount.h_Bool modOfIs wordcount.c
./wordcount.h_Bool defBy wordcount.c
Macro ./wordcountTable.h_MAX_WCTABLE_LEN
wordcountTable.c defines ./wordcountTable.h_MAX_WCTABLE_LEN
./wordcountTable.h_MAX_WCTABLE_LEN modOfIs wordcountTable.c
./wordcountTable.h_MAX_WCTABLE_LEN defBy wordcountTable.c
Macro ./wordcount.h_FALSE
wordcount.c defines ./wordcount.h_FALSE
./wordcount.h_FALSE modOfIs wordcount.c
./wordcount.h_FALSE defBy wordcount.c
Macro ./wordcount.h_TRUE
wordcount.c defines ./wordcount.h_TRUE
./wordcount.h_TRUE modOfIs wordcount.c
./wordcount.h_TRUE defBy wordcount.c
Macro ./wordcount.h_INVALID
wordcount.c defines ./wordcount.h_INVALID
./wordcount.h_INVALID modOfIs wordcount.c
./wordcount.h_INVALID defBy wordcount.c
Macro ./wordcount.h_WORDLEN
wordcount.c defines ./wordcount.h_WORDLEN
./wordcount.h_WORDLEN modOfIs wordcount.c
./wordcount.h_WORDLEN defBy wordcount.c

        THIS IS THE END OF THE EXAMPLE PROGRAM'S A-BOX
```