Model checking the ATerm library

Gabriels, J.M.A.M.

Award date:
2008

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

Model Checking the ATerm Library

By
J.M.A.M. Gabriels

Supervisors

Prof. Dr. M.G.J. van den Brand (TU/e)
Prof. Dr. J.C. van de Pol (UT)

Eindhoven, December 2007

Abstract

Important properties of good software are reliability and predictability. There is no place for strange and unwanted behavior in software such as arbitrary values or system crashes. In ANSI-C programs using the tree-like data structures of the ATerm library, improper use of globally defined ATerms can cause such behavior. This thesis describes a method for transforming ANSI-C source code, using the ASF+SDF Meta Environment, to a format that can be checked for these patterns of, sometimes hard to find, unwanted behavior using the CADP toolkit's model checker.

Acknowledgments

I would like to take the opportunity to express my thanks to all those who helped me in any way with my Master's project.

Special thanks go to Mark van den Brand and Jaco van de Pol, my supervisors, who helped me when I got stuck and provided helpful insight and tips that aided me in moving the project forward.

I also want to thank my colleagues and roommates Bas, Chantal, Dennis, Arjan, Ludo, Remko and Alexander who helped me along and were always prepared to brainstorm with me. Furthermore, my thanks go out to the members of the Software Engineering & Technology (SET) group at the Eindhoven University of Technology.

Last but not least, I want to thank my parents, friends and Renée for their support.

Eindhoven, December 2007

Joost Gabriels

Contents

1 Introduction **5**
- 1.1 Software quality . . . 5
- 1.2 Problem description . . . 6
- 1.3 Thesis outline . . . 7

2 Annotated Terms **8**
- 2.1 ATerms and accidental removal . . . 8
- 2.2 The problems illustrated . . . 9
  - 2.2.1 Global ATerms . . . 9
  - 2.2.2 ATerms in structs . . . 11
  - 2.2.3 Mismatch of protect and unprotect . . . 11
- 2.3 Conclusions . . . 13
- 2.4 Planned approach . . . 14

3 Parsing ANSI-C with ASF+SDF **16**
- 3.1 ASF+SDF Meta Environment . . . 16
  - 3.1.1 Syntax Definition Formalism . . . 16
  - 3.1.2 Algebraic Specification Formalism . . . 17
  - 3.1.3 Using the Meta Environment . . . 18
- 3.2 Structure of ANSI-C programs . . . 18
  - 3.2.1 Ambiguities . . . 20
  - 3.2.2 The C-subset . . . 22
- 3.3 Normalizing ANSI-C code . . . 23

4 The Control Flow Graph **27**
- 4.1 Structure and Definition . . . 27
- 4.2 Building the control flow graph . . . 28
- 4.3 Basic Statements . . . 29
  - 4.3.1 Declarations . . . 29
  - 4.3.2 Assignments . . . 30
- 4.4 Repetitions . . . 30
  - 4.4.1 While-Do repetition . . . 30
  - 4.4.2 Do-While repetition . . . 31
- 4.5 If-then-else alternative . . . 32
- 4.6 Global Definitions . . . 33
- 4.7 Flow-altering statements . . . 34
  - 4.7.1 Function Calls . . . 34
  - 4.7.2 The problem of recursion . . . 35

4.7.3 Return statements . . . 39
4.7.4 Break statements . . . 39
4.8 The nail warehouse . . . 41

5 The Dual Graph **44**
5.1 Structure of the dual graph . . . 44
5.2 Building the dual graph . . . 45
5.3 The nail warehouse revisited . . . 46

6 Abstraction **48**
6.1 The Abstract Graph . . . 48
6.2 Building the abstract graph . . . 49
6.2.1 Phase 1: Traversing the external declarations . . . 49
6.2.2 Phase 2: Abstracting declarations and statements . . . 49
6.3 Nail warehouse example . . . 53

7 Model Checking with CADP **56**
7.1 The CADP toolkit . . . 56
7.1.1 The Aldebaran Graph . . . 56
7.1.2 Reducing the models . . . 57
7.1.3 Regular alternation-free μ-calculus . . . 58
7.1.4 Evaluator . . . 59
7.2 Behavioral patterns . . . 60
7.2.1 Protect before use . . . 60
7.2.2 Matching protect and unprotect . . . 60
7.3 Checking the models . . . 61
7.3.1 The Nail Warehouse . . . 62
7.3.2 Test programs . . . 63

8 Results and Conclusions **69**
8.1 Results . . . 69
8.2 Conclusions . . . 69
8.3 Related Work . . . 72
8.3.1 Model Checking program source code . . . 72
8.3.2 Fact extraction . . . 74
8.4 Recommendations and future work . . . 74

A List of Tools **81**

B Using the ATerm library **82**
B.1 Nail Warehouse 1 . . . 83
B.2 Nail Warehouse 2 . . . 84
B.3 Test program 1 . . . 85
B.4 Test program 2 . . . 86
B.5 Test program 3 . . . 87
B.6 Test program 4 . . . 88
B.7 Test program 5 . . . 89
B.8 Test program 6 . . . 90
B.9 Test program 7 . . . 91
B.10 Test program 8 . . . 92

C Normalizing **93**
C.1 Normalizing (SDF) . . . 93
C.2 Normalizing (ASF) . . . 94

D Control Flow Graph **101**
D.1 Triple . . . 101
D.2 Tuple . . . 102
D.3 DeclStat . . . 103
D.4 Control Flow Graph structure (SDF) . . . 104
D.5 Control Flow Graph structure equations (ASF) . . . 107
D.6 Building the control flow graph (SDF) . . . 110
D.7 Building the control flow graph (ASF) . . . 114

E Dual Graph **126**
E.1 Dual Graph structure (SDF) . . . 126
E.2 Dual Graph structure equations (ASF) . . . 128
E.3 Building the dual graph (SDF) . . . 129
E.4 Building the dual graph (ADF) . . . 130

F Abstract Graph **132**
F.1 Label . . . 132
F.2 Abstract Graph structure (SDF) . . . 132
F.3 Abstract Graph structure equations (ASF) . . . 134
F.4 Building the abstract graph (SDF) . . . 135
F.5 Building the abstract graph (ASF) . . . 137

G Aldebaran graph **148**
G.1 Quote . . . 148
G.2 AutEdge . . . 149
G.3 Aldebaran graph structure (SDF) . . . 150
G.4 Aldebaran graph structure equations (ASF) . . . 151
G.5 Building the Aldebaran graph (SDF) . . . 151
G.6 Building the Aldebaran graph (ASF) . . . 152

Chapter 1

Introduction

Consider the following toy example:

Example 1.0.1. *Consider a warehouse in nails. This particular warehouse sells just one type of nail and at any given time, it can store up to 500,000 nails. The warehouse policy is to order more nails from the manufacturer when the number of nails in storage is less than 1,000.*

Today, the inventory shows 400,025 nails and an order for 400,000 nails has come in from a DIY-store. This order should not be a problem since there are enough nails in inventory. The order is placed, the nails are sold and ... the system crashes, no order for new nails is created and the inventory number shows 352,028 nails. What went wrong? Why the strange inventory number? Why did the system crash?

(End of Example)

1.1 Software quality

Much has been said on the topic of reliability of software and much effort has been put into the development of techniques to improve the reliability of software. One of the characteristics of reliable software is its correctness. In other words, that the constructed software is correct with regards to its specification and that it simply does what it was designed to do. Another is the predictability of software. If a value is assigned to a variable, and we read it again without any action on it in the mean time, we expect it to still have that same value. Reliability and predictability leave no room for strange, sometimes hard to explain, behavior like occurred in the nail warehouse.

As software systems grow bigger and more paths and branches are added to create new functionality, the complexity increases. Over time it can be difficult to have a clear overview of the software. This is not difficult just because of the size, but also because of the complexity of the code and individual styles of programmers. Debugging software to find out what causes some strange action that "should not have happend" (such as an inventory value of 352,028 nails) can take a very long time. Therefore, being able to transform the software and

the safety requirements into a form where they can automatically be checked is highly desirable and can answer questions like: "Does this model of the software violate the requirements?" and "Is there some branch that results in faulty actions?"

Model checkers that check models against a set of safety requirements exist and have proven useful in discovering errors in systems. An example of such a model checker is the CADP evaluator [4] that has been used successfully in verifying hardware systems and communication protocols. (See [2]). Most model checkers do not work with actual programs as input, but use models. Each model checker uses its own kind of model language. The CADP evaluator can work with a graph structure is while the the SPIN model checker [20, 5] works on model programmed in the Promela language [5].

1.2 Problem description

When using the ATerm library [32] in computer programs, unwanted behavior can occur. The ATerm library is a simple and effective means of representing tree-like structures, that can be used for exchange of complex data structures between applications. The ATerm library uses maximal subterm sharing and has an automated garbage collector. When using the ATerms, unwanted behavior can be triggered by improper use of the protect and unprotect functions that the ATerm library provides. An unexpected garbage collection of unprotected globally defined ATerms can cause arbitrary values upon referencing. Furthermore, a mismatch between a protect and unprotect action can cause a memory leak or system crash.

This thesis, describes the research and development of a proof-of-concept that tries to answer the following question:

> "How can ANSI-C source code (using the ATerm library) be checked for patterns of unwanted behavior using Model Checking techniques."

The proof-of-concept will consist of a description and implementation of a method that will check, by using model checking techniques, a given ANSI-C program using the functions of the ATerm library against a set of safety requirements that ensure proper use of global ATerms. The ATerm library's protect/unprotect scheme provides a clear set of "behavioral patterns" to check for in the programs. The research can be divided into several sub questions:

1. What are the problems with globally defined ATerms in combination with automatic garbage collection?
2. What kind of behavior is unwanted and how can it be recognized?
3. How can the program be represented in such a way, that it captures the flow of the program?
   (a) What parser for ANSI-C should be used?
   (b) What intermediate structures should be used?

4. What information is necessary and what information can be abstracted away?

5. How can we check this representation with model checking techniques to find the cases of unwanted behavior?

   (a) What model checker should be used?

   (b) How should the patterns of unwanted behavior be represented?

1.3 Thesis outline

Chapter 2 explains what the ATerm library is and illustrates the problems that can come from carelessness in using globally defined ATerms. Chapter 3 introduces the SDF formalism and tools with which the ANSI-C language can be transformed and in chapter 4, the process of transformation from programming language to control flow graph is explained. In chapter 5, the control flow graph is transformed into a dual graph to match the graph type of the CADP model checker. This model will then be abstracted in chapter 6. All unnecessary information (i.e. that is not concerned with ATerms) will be removed so that with well known state space reduction techniques, the size and complexity of the model can be minimized. Chapter 7 describes the actual model checking process and gives examples of the results obtained. Certain requirements of proper use are checked with the abstracted and minimized models to see if improper use can be detected. Finally, chapter 8 will hold the results and conclusions of the project.

Chapter 2

Annotated Terms

ATerms or "Annotated Terms" (as described in [32] and [33]) is a simple and efficient means of representing tree-like structures that is platform and language independent. ATerms are efficient because of the use of maximal subterm sharing which makes them ideal for sharing complex data structures between applications.

Instead of recreating the same object over and over again, ATerms are reused. This way, only new terms will be made which reduces the memory consumption. The ATerm library does this by checking a hash table with pointers to the ATerms to see if the ATerm has already been created. ATerms are widely used in a range of applications such as the Toolbus [11], JITty [30], mCRL2 [18], Stratego [12], in the ASF+SDF Meta Environment [1, 37, 24] and may more applications such as development environments and term rewrite engines. See [33] for a list of more applications that use ATerms.

Another thing that makes the ATerm library so efficient is an automated garbage collection mechanism that cleans up ATerms that are no longer used in order to free memory that would otherwise be unavailable. Providing some sort of control over this process, the ATerm library provides functions to explicitly protect ATerms from untimely removal by the garbage collector and unprotect them to free memory.

Unfortunately, even though this mechanism is very effective, forgetting to protect certain types of ATerms can sometimes cause strange behavior in programs.

2.1 ATerms and accidental removal

Any time a new ATerm is created, the library checks if it can reuse an old ATerm from a free-objects-list. This list holds removed ATerms who's memory can be reused and new ATerms that are not yet in use. If this is not the case, the garbage collector will start its “mark-sweep” algorithm to clean up unused ATerms. It does this by first marking all ATerms as “dead” and then checking which ATerms are reachable from a known set of root objects on the

execution-stack. These reachable objects are marked as "live" and all remaining "dead" objects are moved into a free-objects-list. ATerms in this list are not immediately disposed of, but will again be reused upon creation of a new ATerm.

When using ATerms as global variables in a program, the programmer must always keep in mind that without explicit protection, these ATerms are unprotected and will be marked "dead" in the next garbage collector run. Explicit protection of the globally declared ATerms will put them in a data structure that is accessible by the garbage collector. Only this way, will the global ATerms will be marked as "live" again.

Referencing a garbage-collected ATerm can result in retrieving an arbitrary meaningless value without any warning concerning its removal. Because the ATerm is removed, this is like retrieving a value through a null-pointer. ATerms that are declared within functions of the program do not have to be protected and unprotected by the programmer. These local ATerms are automatically protected for the duration of the function (see [15]).

When an ATerm is moved into the free-objects-list, its value can still be retrieved, until the creation of a new ATerm reuses its memory. This adds to the confusion when forgetting to protect and unprotect global ATerms causes strange, seemingly unrelated, errors between functions.

A global ATerm, that is not protected is a candidate for removal in the next garbage collection. Other candidates are structs and globally declared arrays that hold ATerms. The ATerms in these self-made data structures and arrays must also be protected explicitly in order to make sure that they will not be removed in the next sweep.

Besides arbitrary values, carelessness in protecting and unprotecting global ATerms poses another problem. To successfully protect an ATerm, the protect and unprotect actions must match. That is, every protect is eventually followed by an unprotect and every unprotect is eventualy preceded by a protect. Programs that have protect actions without matching unprotect actions will keep unused memory protected and create a memory leak. If a trace through the program exists where an unprotect action is done without a protect preceding it, the program will give a segmentation fault and give a core dump.

2.2 The problems illustrated

2.2.1 Global ATerms

Figure 2.1 shows a code fragment that sketches the faulty use of a globally declared ATerm.

The code fragment shows a global ATerm of type integer being declared in (1), defined with the value *42* in (2) and referenced in (3) without any protection against garbage collection. Since there is no telling when the garbage collector will start its sweep, no guarantee can be given whether the global ATerm will

```
ATermInt global;                                (1)

int main(int argc, char* argv[])
{
        ...
        global = ATmakeInt(42);                 (2)
        ...
        ATprintf("%t",global);                  (3)
        ...
}
```

Figure 2.1: Code fragment of improper use of global ATerms

still exist when it is referenced.

To demonstrate the possibility of strange behavior, several test programs have been made which intentionally force the garbage collector to remove a global ATerm. This provides the possibility to see what goes wrong and how that corresponds with the model checker's output. These test programs have been included in appendix B and will return in chapter 7.

The ATerm library has the option to give verbose information on the garbage collector. Using this verbose option, the code in figure 2.1 was used in a test program (Appendix B.3) with the following output:

```
Value of global ATerm = 701476
2 garbage collects,

stack depth: 2147483647/0/0 words

reclamation percentage: 2147483647/0/0
```

Figure 2.2: Verbose ATerm output illustrates removal

In the actual test program (Appendix B.3), a global ATerm was given the value 42. After that, 400,000 new ATerms were created. The garbage collector ran twice and removed the unprotected global ATerm. The final act of printing the value of the global ATerms gives an arbitrary value.

The code fragment in figure 2.3 shows the proper use of a global ATerm. After declaring the global ATerm in (1), it has to be protected in (2) before it is defined in (3). Both defining and referencing the ATerm in (4) will now be guaranteed to be successful. The protection function puts the global ATerms in a data structure that is accessable by the garbage collector during its sweep. This way, it can be labeled as "live" and will not be removed. The programmer must make sure to unprotect the ATerm in (5) after he is done with it. This will make sure the ATerm can be collected upon the next sweep. By unprotecting, no more memory is retained than necessary.

```
ATermInt global;                                (1)

int main(int argc, char* argv[])
{
        ...
        ATprotect(&global);                     (2)
        global = ATmakeInt(42);                 (3)
        ...
        ATprintf("%t",global);                  (4)
        ...
        ATunprotect(&global);                   (5)
        ...
}
```

Figure 2.3: Code fragment of proper use of global ATerms

Explicitly protecting the global ATerm as done in figure 2.3 (Appendix B.4) now shows:

```
Value of global ATerm = 42
2 garbage collects,

stack depth: 2147483647/0/0 words

reclamation percentage: 2147483647/0/0
```

Figure 2.4: Verbose ATerm output shows success

2.2.2 ATerms in structs

A similar situation arises with the use of ATerms in structs and globally defined arrays. The code fragment in figure 2.5 shows a struct in which an ATerm is used but not protected. This ATerm again must be explicitly protected by the programmer to protect it from being collected.

The ATerm is part of a struct (1) and is declared in (2). It will eventually be defined in (3) and may or may not be present when the ATerm is referenced in (4). In the code fragment in figure 2.6, the struct is properly used.

Like the example in figure 2.3, the ATerm is (globally) declared inside the struct in (1) and (2), explicitly protected in (3). Defining it in (4) and using it in (5) can be done without any problems. After which it must be freed by unprotecting it in (6).

2.2.3 Mismatch of protect and unprotect

Forgetting to unprotect a protected global ATerm will cause a memory leak and calling unprotect while no preceding protect was called causes a segmentation

```
struct foo {
        ...
        ATermInt bar;                                   (1)
        ...
} myStruct;                                             (2)

...

int main(int argc, char* argv[])
{
        ...
        myStruct.bar = ATmakeInt(42);                   (3)
        ...
        printf("%d", A"TgetInt(myStruct.bar));          (4)
        ...
}
```

Figure 2.5: Code fragment of improper use of a global ATerm in a struct

```
struct foo {
        ...
        ATermInt bar;                                   (1)
        ...
} myStruct;                                             (2)
...
int main(int argc, char* argv[])
{
        ...
        ATprotect(&myStruct.bar);                       (3)
        myStruct.bar = ATmakeInt(42);                   (4)
        ...
        printf("%d", A"TgetInt(myStruct.bar));          (5)
        ...
        ATunprotect(&myStruct.bar);                     (6)
        ...
}
```

Figure 2.6: Code fragment of proper use of a global ATerm in a struct

fault and results in a core dump. To investigate this, four test programs were written (Appendix B.7 through B.10.). Consider the code fragment in figure 2.7.

After declaration of the ATerm as global in (1), a choice is made by referring to some random value produced by the ANSI-C pseudo random number generator. In one branch, the global ATerm is protected prior to defining it in (2) and (3), in the other it is not. It is directly defined in (4). After both branches are finished, a final use action on the ATerm is performed in (5) whereafter the

```
ATermInt global;                                                    (1)

int main(int argc, char* argv[])
{
        ...
        if(randomValue < 50) {
                ATprotect(&global);                                 (2)
                global = ATmakeInt(371);                            (3)
                ...
        }
        else {
                global = ATmakeInt(42);                             (4)
                ...
        }
        ATprintf("Value of global ATerm = %t\n",global);            (5)
        ATunprotect(&global);                                       (6)
        ...
}
```

Figure 2.7: Code fragment with mismatch in the else branch.

ATerm is unprotected in (6). Both the result of the mismatch (no protect in the else branch) and the definition of an ATerm without protection (also in the else branch) can now be seen by checking the output in figure 2.8.

```
Value was at least 50
Value of global ATerm = 352040
random number was: 84
Segmentation fault (core dumped)
```

Figure 2.8: Output shows mismatch and arbitrary value

Not only did the else branch violate the demand that the global ATerm must be protected prior to use (hence the strange arbitrary value 352040), but the mismatch between protect and unprotect in that same branch caused the segmentation fault.

2.3 Conclusions

As a conclusion: carelessness in the protection of globally defined ATerms can cause strange behavior or even result in a segmentation fault. Just imagine the implications when the mentioned arbitrary values are used as guard conditions. The question how to check whether all globally defined ATerms are protected prior to define or use actions, and all protected ATerms are eventually unprotected and vice versa is the main topic addressed in this thesis. As a summary, consider the informal description of proper use of the ATerm library in figure 2.9.

```
Proper use of the ATerm library:

(1)   After declaration, All define and use actions must be between
      protect and unprotect.

      Declaration -> Protect -> (Define or Use)* -> Unprotect

(2)   After a global ATerm is unprotected it must first be protected
      again before it can be defined again. After defining, the ATerm
      can be used again.

      Unprotect -> ... -> Protect -> Define -> (Define or Use)* -> Unprotect

(3,4) Every protect is eventually followed by an unprotect and vice versa

      Protect -> ... -> Unprotect
```

Figure 2.9: Summary: Proper use of the ATerm library

Any violation of these patterns should result in an error in the output of the model checker.

2.4 Planned approach

Given the problems with global ATerms described in this chapter, a planned approach can be made. Firstly, the programs need to be parsed in order to be able to transform them into a format that can be used as input for a model checker. The ASF+SDF Meta Environment will be used for both parsing and transforming. The Meta Environment is a framework for language development and source transformations and provides means to use custom intermediate languages and data structures. Its build-in support for the ANSI-C language and in-house expertise at the Eindhoven University of Technology make it the first choice. Alternatively, flex and bison [7] or perhaps the GNU C compiler (GCC) could be used to construct a parser for the ANSI-C language.

Before anything else is done, the ANSI-C code must first be normalized to reduce the number of language constructions that need to be checked. After that, the "flow" of the programs needs to be extracted to be able to say anything about the order in which statements are executed. Two intermediate data structures will be used to capture that flow. One structure has the action labels on the nodes, the other on the edges. These two kinds of graphs open up a broader range of model checkers and analysis tools that may be used in the future. The ASF+SDF Meta Environment will be used to transform the programs into control flow graphs (with labels on the nodes) and dual graphs (with labels on the edges). These two transformations will be done in sequence. First the ANSI-C code is transformed into a node-labeled control flow graph which is then transformed into an edge-labeled dual graph.

Using this dual graph, all non-global ATerm information can be left out, leaving only the global ATerm behavior. This abstraction step will label all edges in the dual graph with a set of labels that denotes how an ATerm is used. The resulting graph can be used, after a layout transformation, with the CADP evaluator. This on-the-fly model checker can take an edge-labeled graph as a model. The CADP toolkit will be used for several reasons. Firstly, the CADP toolkit is freely available for academic use. Secondly there is again in-house expertise and thirdly, it accepts graphs as input instead of a model programmed in a language specific to that model checker. The use of a graph provides a more generic representation that may be used in future research. A domain-specific language would perhaps narrow down the possibilities. Alternatively a different model checker could be used. If so, transformations from ANSI-C code to the domain-specific language of the model checker should be constructed. As an alternative, SPIN [20] or mCRL2 [18] could be used.

The problems with ATerms described in this chapter can be turned into patterns. These patterns or safety properties can be expressed in a temporal logic usable with the CADP evaluator. By automatically checking the patterns against the abstracted model, the CADP evaluator can create an error trace visualizing the violation in protect / unprotect use. Using the output of the model checker, it should be clear where the unwanted behavior in ANSI-C programs, using the ATerm library, comes from.

Chapter 3

Parsing ANSI-C with ASF+SDF

Imperative programming languages such as ANSI-C, have the property that they are structured in a way such that the declarations and statements are generally evaluated in a sequential order. Not only simple assignments, but also repetitions such as while and for are considered statements themselves. The order in which these declarations and statements are executed allows us to capture the so-called "flow" of the program in a directed graph. This chapter describes what formalisms and tools use used to translate ANSI-C programs into an intermediate structure that captures this flow.

3.1 ASF+SDF Meta Environment

For the transformations in this project, the ASF+SDF Meta Environment [1, 37, 24, 34] is used. The Meta Environment is a framework for language development, source code transformations and analysis. It uses Syntax Definition Formalism (SDF) and Algebraic Specification Formalism (ASF) to transform and analyze languages and various other structures.

3.1.1 Syntax Definition Formalism

The Syntax Definition Formalism (SDF) can be used to describe the syntax of programming languages, domain specific languages and data structures. SDF has a modular structure and allowes for both lexical and context-free definitions. An example of the basic SDF definition of a list is given in figure 3.1.

Presented in figure 3.1 is the syntax definition of a List container. Since SDF uses modular syntax definitions, other syntax definitions can be inserted with "imports" at (1). The exports keyword presents the context-free syntax below to the outside world. At (2), the structure of a list is defined. The X is used as parameter which acts as a generic variable. It is instantiated with a certain datatype when a list is created (e.g. imports List[Integer] for a list of integers). Such a module is called parameterized. The list starts with a "[" and ends with a "]". Between the brackets, {X ","}* stands for zero or more elements of type

```
module containers/List[X]

imports                                                          (1)
        basic/Booleans
        basic/Integers

exports
  context-free syntax
        "[" {X ","}* "]"      -> List[[X]]                       (2)

  context-free syntax
        length( List[[X]] )  -> Integer
        head( List[[X]] )    -> X
        tail( List[[X]] )    -> List[[X]]
        elem( X, List[[X]] ) -> Boolean                          (3)
        empty(List[[X]])     -> Boolean
        cons( X, List[[X]] ) -> List[[X]]
        X ":" List[[X]]      -> List[[X]]
        concat(List[[X]], List[[X]]) -> List[[X]]                (4)

hiddens
  imports
        basic/Comments
        basic/Whitespace
  variables                                                      (5)
        "X"[0-9\']*                  -> X
        "X*"[0-9\']*                 -> X*
        "X,*"[0-9\']*                -> {X ","}*
```

Figure 3.1: SDF definition of a list

X separated by comma's.

Functions on the list structure are also described here. For example (3) defines a function *elem* that checks whether an element is present in the list. It does this, by taking an X and a list of X's and delivers a boolean value. The function is described by an ASF specification. Another function is described in (4). This function called *concat* concatenates two lists. The variables that can be used in the ASF equations are defined in (5). Note that "X,*" is a just string that represents a comma separated list.

For an extended overview of SDF's capabilities and in depth details, see [36].

3.1.2 Algebraic Specification Formalism

The Algebraic Specification Formalism (ASF) defines the semantic properties of the definitions given in the SDF definition. It does this by providing a set of equations that describe how structures in the given SDF definition can be

manipulated.

```
[list-elem]
  elem (X, [ X,*1, X, X,*2 ]) = true                    (1a)

[default-list-elem]
    elem (X, [X,*]) = false                             (1b)

[list-concat] concat([X,*1], [X,*2]) = [X,*1 , X,*2]   (2)
```

Figure 3.2: ASF equations for the elem and concat functions

Figure 3.2 (1a) and (1b) illustrates the ASF equation of a function that checks if an element occurs in a given list. It does this by means of pattern matching. It finds out if it can find precisely that X in the list by checking whether the list can be split up in such a way that zero or more elements are in front and zero or more elements are behind precisely that X. If this is the case, the value true is returned, otherwise, false will be the answer. In (2), a function is describes that concatenates two lists of type X.

For an extended overview of ASF's capabilities and in depth details, see [35].

3.1.3 Using the Meta Environment

Using the ASF+SDF Meta Environment, that combines the power of SDF and ASF, provides the possibility to analyze and manipulate traditional languages, such as ANSI-C, domain-specific languages and data structures. Given the SDF definition of the list and the ASF equations, a term can be made that holds the initialized function with values that need to be checked.

Actually using the *elem* and *concat* function is illustrated in figure 3.3.

```
elem(3, [1,2,3,4,5])

concat([1,2],[3,4])
```

Figure 3.3: Using the elem and concat function

Since the list can be split into the values 1 and 2 in front, 4 and 5 behind and 3 in the middle, the result of the *elem* function will be true. The result of *concat* will be the list $[1, 2, 3, 4]$.

3.2 Structure of ANSI-C programs

The book "The C Programming Language" by Kernighan and Ritchie [23] is considered to be the leading reference on the ANSI-C standard and covers the various language constructions. The ASF+SDF Meta Environment includes the

SDF definitions of the ANSI-C (c89) standard. These definition files were used for parsing the C programs in this project.

In transforming source code into an intermediate structure, the ASF+SDF Meta Environment allows for sequentially processing of one statement after another and build up a control flow graph at the same time. The process of transforming ANSI-C code to a control flow graph is explained in chapter 4. The statements and declarations in the source code are matched against the Meta Environment's syntax definition of the ANSI-C programming language.

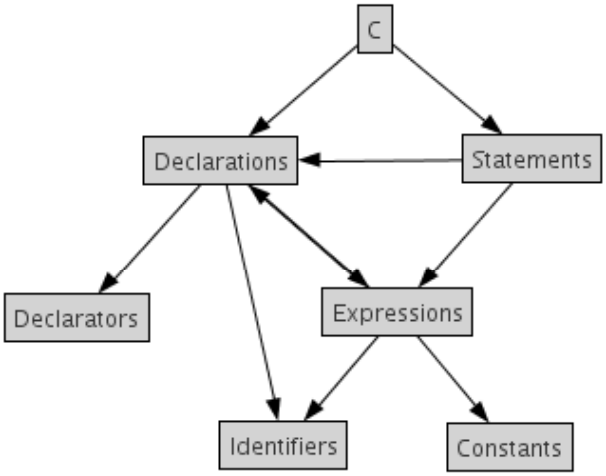

Figure 3.4: Import graph of the C language in the ASF+SDF Meta Environment

Figure 3.4 shows modular structure in the import graph of the C language as used for this project in the Meta Environment. In the module C, the entire ANSI-C program is defined as a TranslationUnit which is build up out of Declarations (global variables) and FunctionDefinitions. Illustrating how the modular structure allows for the syntax of the entire programming language, consider the following example of a FunctionDefinition:

Example 3.2.1. *FunctionDefinitions are build from:*

```
Specifier* Declarator Declaration* "{" Declaration* Statement* "}"
-> FunctionDefinition.
```

Compare that to the following function definition in C: (with int as Specifier, main as Declarator and the parameters as Declaration*)*

```
int main(int argc, char** argv) { int i = 0; i = i * 2; return 0; }
```

(End of Example)

3.2.1 Ambiguities

ANSI-C is an ambiguous language. The SDF definition that comes with the Meta Environments reads:

> "DONT: do not try to fully disambiguate this grammar because that would ruin its declarative nature. C is ambiguous. However, the expression grammar should be fully unambiguous (Expression)."

Several ambiguities have arisen during the transformations. Two examples shall be given. The first ambiguity is found in the parsing of parameters in function definitions. Consider figure 3.5.

```
int foo(int counter) { ... }
```

Figure 3.5: Ambiguity in using parameters

Using the visual parse tree in the Meta Environment, the ambiguity can be visualized. In figure 3.6, the colored triangle denotes an ambiguity. Both the left or the right branch are possible ways of parsing the parameter. A choice needs to be taken between parsing *int counter* as a specifier (left arrow) or as a parameter (right arrow).

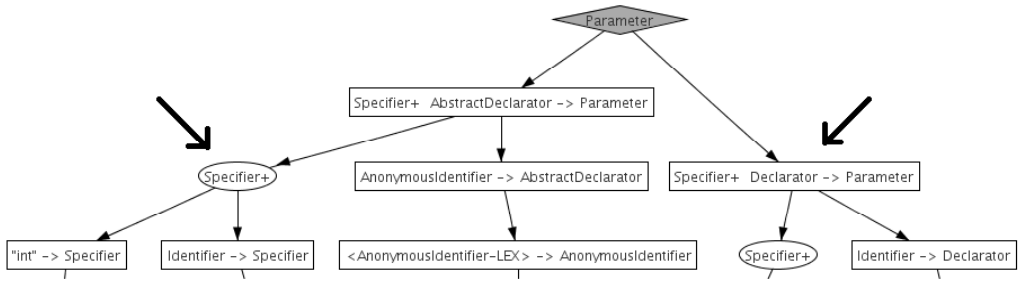

Figure 3.6: Visualization of parameter ambiguity

With regard to the ambiguity in figure 3.6, the right branch is preferred because it describes the most common way of using parameters, namely with both the type and name of the variable. The following adaptation has been made to the SDF syntax definition of the parameter.

```
Specifier+ AbstractDeclarator  -> Parameter {avoid}
```

This removes the ambiguity because it will avoid using this rule and always use the preferred one. Avoiding this rule, rules out an older way of describing parameters, namely like illustrated in figure 3.7. This poses a restriction but because the remaining rule describes the most commonly used way of using

parameters, it may be the best way to eliminate the ambiguity. The formal parameters of a function are not interesting for this project. In a later stage of the project, the parameters passed to a function will be examined in the function call statement.

```
int foo(counter) int counter; { ... }
```

Figure 3.7: Alternative use of parameters

A second ambiguity is found in parsing something of the form illustrated in figure 3.8.

```
while (i > 0) {
        foo(counter);
        i--;
}
```

Figure 3.8: Foo, declaration or statement?

The problem in figure 3.8 is that the function call to *foo* with argument *counter* can be parsed in two ways, either as declaration or as statement as is illustrated in figure 3.9. Note that this ambiguity only occurs if a function call with no return value is used as first statement or when exclusively preceded by declarations in a repetition or alternative branch.

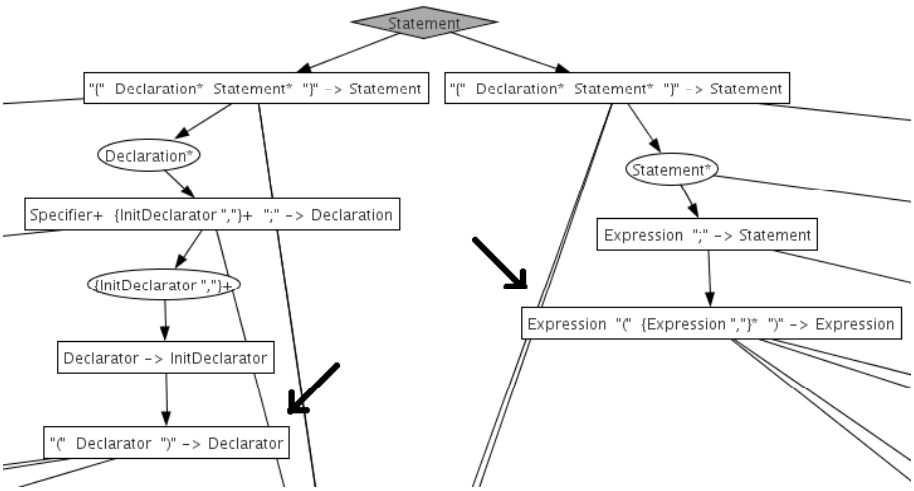

Figure 3.9: Visualization of ambiguity with parenthesis

To work around this problem, the following line in the SDF definition is adapted with a *reject* and a *bracket* tag which rejects the use of parenthesis

around a declarator.

```
"(" Declarator ")" -> Declarator {reject, bracket}
```

This adaptation makes sure that $foo(counter)$; is interpreted as a statement instead of a declaration. The downside of this restriction is that this rejection of parenthesis rules out type casting. Due to time constraints, no better alternative could be investigated.

Note 3.2.1. *Adapting the SDF definition of C like this is to be considered a workaround! This does not actually fix the problem, but postpones it. A real solution should be found in the use of amb-constructors. The workaround was chosen because it provided a way to focus on how to check the programs and move the project forward towards its goal.*

(End of Note)

3.2.2 The C-subset

For the proof of concept, the attention will be focused on a subset of ANSI-C, that is representative for showing that model checking source code against behavior specifications is feasible and an interesting field of research.

The subset consists of the following language constructions:

- Declarations (with and without initializations, both global and local).
- Assignments
- Repetitions (For, While-Do and Do-While).
- Alternative statements (If-then-else).
- Function calls (non-recursive).
- Increment and decrement operators (e.g. i++, i- -).
- Arrays.
- Structs.
- Break statements.
- Return.
- Basic pre-processor statements (#include and #define for constants).

What does the subset leave out? Language constructions such as:

- Pointers and aliases.
- Type casts (due to ambiguity elimination).
- Block definitions (i.e. { Declaration* Statement* } other than function definitions).

- Switch statements.
- Continue statements.
- Goto statements.
- Exit statements.
- Break n statements (braking to the n-th loop).
- Function calls (recursive).
- Pre-processor statements (#define for macros).

During the project, an iterative strategy of development was applied. As a minimum subset, the first four items plus the return statement were implemented. With this set, a set of test programs was developed that were subjected to the language transformations and checked against the safety properties. After that, the remaining items on the list were implemented and checked. The items on the second list with exception of the goto statement, pointers and recursion were not implemented due to time constraints. The use of this subset means that real industrial applications cannot be checked at the moment. Chapter 4 and 8 will go into some of the problems that need to be solved in order for this solution to be able to cope with real applications.

Because the control flow graph is built using an iterative algorithm, "Goto" statements would require reference points throughout the graph for linking the focus to any number of possible points, making an unstructured graph. Goto statements complicate the building process and are therefore not included in the C subset.

Also the use of aliases and pointers complicates things. By using a pointer to global ATerm for definitions and referencing, the variable name used in the declaration and definition or use action are no longer the same. As will be evident later, this poses a problem with checking what information concerns ATerms and what does not, since ATerm related information is filtered on the variable name used in the declaration. The same holds true for aliases. If an alias is used to reference, define or even protect or unprotect a globally defined ATerm, the action will not be filtered out because of a mismatch in variable name. For this reason, pointers and aliases are not included in the subset.

3.3 Normalizing ANSI-C code

The ANSI-C programming language allows the programmer some freedom in the way a statement or declaration is presented. The If-then-else structure for example, can be written in six different ways depending on the number of statements in each branch. If only one statement is present in a branch, the brackets may be omitted and of no else statement is needed, the entire else-branch may be omitted.

This freedom is also present with the for-statement. If the second argument (the condition) is omitted, the loop is infinite and will not terminate. However,

the first argument of the repetition (i.e. the initialization) will still apply and needs to be considered. The first and third argument may be left out without a problem (there are 4 ways of doing this), and will simply be omitted from the flow.

To handle all the variations of declarations and statements, the ANSI-C code is normalized into normal form that reduces the number of variations to consider to a minimum.

Consider figure 3.10 for some of the source-to-source transformations for the if-then-else structure:

```
[elimif-0]
  &Statement*3 := walkStats(&Statement1)
  ====>
  elimif( if (&Expression) &Statement1 ) =
    if (&Expression) { &Statement*3 } else { skip(); }

[elimif-1]
  &Statement*3 := walkStats(&Statement*)
  ====>
  elimif( if (&Expression) { &Statement* } ) = if
    (&Expression) { &Statement*3 } else { skip(); }

[elimif-2]
  &Statement*3 := walkStats(&Statement1),
  &Statement*4 := walkStats(&Statement2)
  ====>
  elimif( if (&Expression) &Statement1 else &Statement2 ) =
    if (&Expression) { &Statement*3 } else { &Statement*4 }

[elimif-3]
  &Statement*3 := walkStats(&Statement*),
  &Statement*4 := walkStats(&Statement)
  ====>
  elimif( if (&Expression) { &Statement* } else &Statement ) =
    if (&Expression) { &Statement*3 } else { &Statement*4 }

  ...
```

Figure 3.10: Source-to-source transformations of the if-then-else structure

The result is that only one variation needs to be processed. The complete if-then-else structure with brackets and else-branch. If no else-branch was present. the *skip*( ) function is inserted.

Some of the source-to-source transformation for the for loop are presented in figure 3.11. These transformations transform the for loop into a while-do struc-

ture possibly preceded by the initializing statement (i.e. the first expression in the for-loop)

```
...

[elimfor-1]
  for ( ; &Expression2 ; &Expression3 ) { &Statement* } := &Statement,
  &Statement*1 := walkStats(&Statement*),
  &Statement+1 := while (&Expression2) { &Statement*1 &Expression3 ; }
  ====>
  elimfor(&Statement) = &Statement+1

[elimfor-2]
  for ( &Expression1 ; ; &Expression3 ) { &Statement* } := &Statement,
  &Statement*1 := walkStats(&Statement*),
  &Statement+1 := &Expression1 ; while (1) { &Statement*1 &Expression3 ; }
  ====>
  elimfor(&Statement) = &Statement+1

[elimfor-3]
  for ( &Expression1 ; &Expression2 ; ) { &Statement* } := &Statement,
  &Statement*1 := walkStats(&Statement*),
  &Statement+1 := &Expression1 ; while (&Expression2) { &Statement*1 }
  ====>
  elimfor(&Statement) = &Statement+1

  ...
```

Figure 3.11: Source-to-source transformations of the for structure

Multiple declarations also pose a variety in the source code since any number of variables of the same type may be declared in one line. Consider the transformation in figure 3.12.

```
    int i, j, k, l, ... ;
=>
    int i; int j; int k; int l, ... ;
```

Figure 3.12: Transformations for multiple declarations

In the scope of this project, structs are only of interest if instances of that struct are declared globally. If this is the case, then the global struct name combined with the individual variable names in the struct are themselves considered global.

When normalizing structs, two things happen. Firstly, the struct is transformed into as many global declarations as it has struct elements. Secondly, if

foo is the globally declared struct and *bar* is the name of the variable, throughout the entire source code the occurrence of *foo.bar* is replaced by the new variable *foo_bar*. This name must not already be in use. Unfortunately, there is no way to enforce this at the moment. Consider the source-to-source transformations for the ATerm variable in figure 3.13.

```
    struct name { ...; ATerm bar; ... ;} foo;
=>
    ...; ATerm foo_bar; ...

    foo.bar
=>
    foo_bar
```

Figure 3.13: Transformations of a struct to global variables

Chapter 4

The Control Flow Graph

To be able to transform the ANSI-C source code into a suitable data structure to hold the flow of the programs, the *Control Flow Graph* is introduced. The control flow graph creates an abstraction of the program that represents its behavior that may contain more paths that are possible in the run-time execution. Not the source code itself, but the behavioral "model" of it is used in model checking.

4.1 Structure and Definition

To define the control flow graph, described in [8, 40], the definition by Wilhelm and Maurer [40] is adopted for use in this project:

Definition 4.1.1. *A Control Flow Graph of a procedure is a node-labeled directed graph $CFG = (N, E, s, t)$ with N being the list of Nodes, E being the list of Edges, s being the start node and t being the stop node. For every primitive statement p of the procedure, there exists a node $n_p \in N$ which is labeled by this statement. Start node s only has outgoing edges and stop node t only has incoming edges.*

(End of Definition)

As is suggested by the definition, the Control Flow Graph is implemented as a tuple of a node list and an edge list and the start symbol will be fixed to 0. In addition to the fixed start symbol, also a fixed start symbol will be defined. This node will be labeled 1. This enables the technique of subgraph insertion.

For each of the language constructions in the subset of the ANSI-C programming language, a subgraph can be made to represent it. This subgraph is, as the name suggests, a part of the whole graph and describes the "flow" of the individual constructions. By incrementally inserting subgraphs into a skeleton framework, the Control Flow Graph of the entire program is obtained as is proposed by Wilhelm and Maurer in [40]. The insertion can be defined as follows:

Definition 4.1.2. *A subgraph $CFG' = (N', E', s', t')$ is inserted into a control flow graph $CFG = (N, E, s, t)$ between nodes n and $n' \in N$ making a new control flow graph $CFG'' = (N'', E'', s, t)$ with: $N'' = N \cup N'$ and $E'' = (E \setminus \{(n, n')\}) \cup E' \cup \{(n, s'), (t', n')\}$.*

(End of Definition)

This definition allows subgraphs to be made for individual constructions and later be placed in a "context" (which is the whole control flow graph) by linking all incoming edges and outgoing edges to the unique entry point and exit point respectively.

Note 4.1.1. *The return and break statement make the language construction it is in violate the unique exit point when building a subgraph. The transformation of these two flow altering statements will be addressed in paragraph 4.7.3 and 4.7.4.*

(End of Note)

As mentioned before, for this proof of concept, we focussed our attention on a subset of ANSI-C. All language constructs in this subset are described in this section.

4.2 Building the control flow graph

Prior to constructing subgraphs from declarations and statements, a skeleton structure to insert these subgraphs in is needed. The skeleton graph consists of a start node (with node number 0), a stop node (with node number 1) and an edge between them as illustrated by figure 4.1.

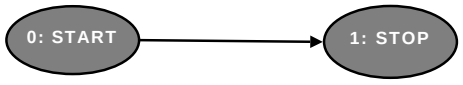

Figure 4.1: Skeleton Control Flow Graph

Inserting a subgraph into the (skeleton) control flow graph after an arbitrary number of iterations as illustrated in figure 4.2, is done, as defined by definition 4.1.2, by:

1. Determining where the subgraph must be placed (e.g. between n and n').
2. Removing the old edge between n and n'.
3. Creating an edge between n and the unique entry node of the subgraph.
4. Creating an edge between the unique exit point of the subgraph and n'.

Since the building process starts with a control flow graph that has a path from start to stop, inserting a subgraph in the way described, will make sure of a new control flow graph with a similar path. This way the end result will

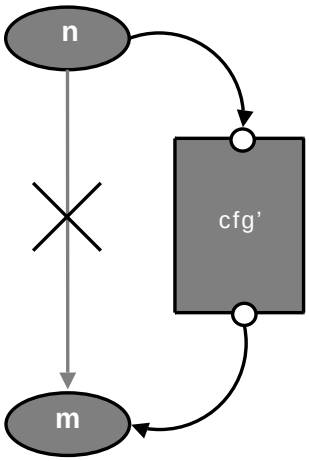

Figure 4.2: Inserting a subgraph into a (skeleton) control flow graph

always be a graph in which all nodes are reachable. This holds provided that there are no infinite repetitions in the source code that do not terminate by alternative means (i.e. break statement). It is assumed that every repetition that is declared as infinite will have at least one break statement.

A C program can have multiple function definitions that are placed above the main function. For convenience sake, prior to building the control flow graph, all function definitions are put in a list. Another list is constructed holding just the function names for quick reference when a function call is encountered.

It is assumed that the main function is the first function to be executed when running the program. By this assumption, the control flow graph shall also begin with the main function.

4.3 Basic Statements

With basic statements, a reference is made to the language constructions in C that are both the unique entry and exit node. Such statements are for example declarations and assignments.

4.3.1 Declarations

Declaration of a variable can take roughly 2 forms: Either with or without initialization.

```
Specifier+ Identifier ";"
Specifier+ Identifier "=" Initializer ";"
```

Figure 4.3: Declarations syntax

In figure 4.3, *Specifier+* denotes the type of variable (e.g. int, float, static long, etc.), *Identifier* denotes the variable name and *Initializer* can be a value of the specified type, the name of another variable or a function call. In figure 4.4, a basic statement is inserted into the control flow graph as illustration. Note that C_n and C_m is the context in which the subgraph is inserted.

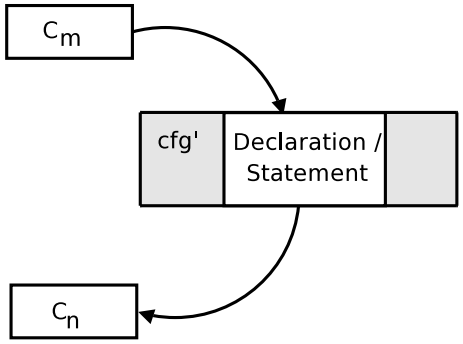

Figure 4.4: Inserting a Basic Statement

4.3.2 Assignments

Assignments have the following syntax structure:

```
Identifier "=" Expression ";"
```

Figure 4.5: Assignment syntax

Where *Identifier* is the name of the variable and *Expression* is the value that is assigned to. Note that this Expression need not be a value or variable name. It can also be a function call to a function that returns a value of the variable type.

4.4 Repetitions

Repetitions come in different flavors. In this project, the While-Do and Do-While repetitions will be used. Because of the normalization process, no for-repetitions are considered.

4.4.1 While-Do repetition

The syntax definition for a while-do repetition is illustrated in figure 4.6.

Expression denotes the guard condition and *Statements*$*$ is the repetition-body consisting of zero or more statements. This is a composite statement and gives a subgraph illustrated by figure 4.7.

```
while "(" Expression ")" "{" Statement* "}"
```

Figure 4.6: While-Do repetition syntax

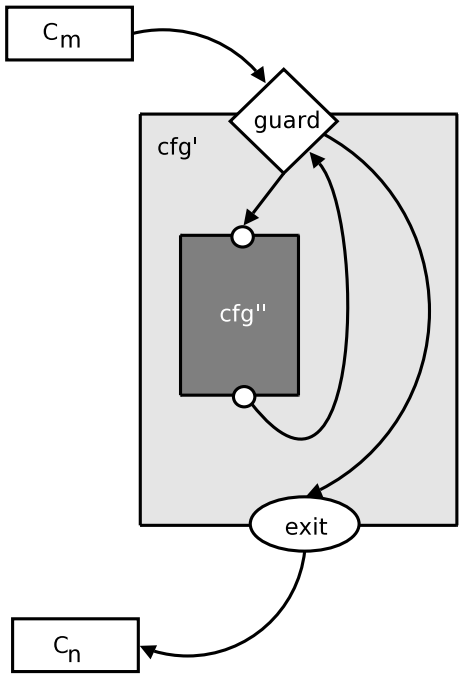

Figure 4.7: The While-Do repetition subgraph

The subgraph needs to have, if we want to insert it into the skeleton graph, an uniquely defined entry and exit point. In the case of a while-do construction, the guard condition will serve as the unique entry point and a separate exit point is created. The repetition body is recursively computed and inserted in the same way that the while-do block is inserted. This way nested statements can be handled.

4.4.2 Do-While repetition

The syntax definition for a do-while repetition is illustrated in figure 4.8.

```
do "{" Statement* "}" while "(" Expression ")";
```

Figure 4.8: Do-While repetition syntax

Expression again denotes the guard condition and *Statements*$*$ is the repetition-body. The do-while repetition differs from the while-do repetition in that the do-while repetition executes the body at least once where as the while-do can

skip the body entirely if the guard is false on the first check. This is also a composite statement and gives a subgraph illustrated by figure 4.9.

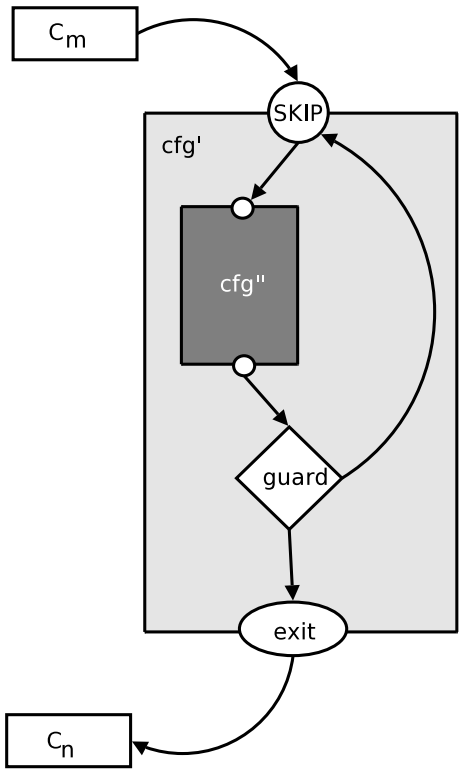

Figure 4.9: The Do-While repetition

In this subgraph the uniquely defined entry and exit point are two additional nodes.

4.5 If-then-else alternative

Due to the normal form transformations, the syntax definition for the If-Then-Else alternative is illustrated in figure 4.10.

```
if "(" Expression ")" "{" Statement* "}"
     else "{" Statement* "}"
```

Figure 4.10: If-the-else alternative syntax

In the if-then-else syntax, *Expression* is the guard condition and in both cases, *Statement*$*$ describes the statement body of the if-branch or else-branch which are again recursively computed and inserted. The corresponding subgraphs look

like figure 4.11.

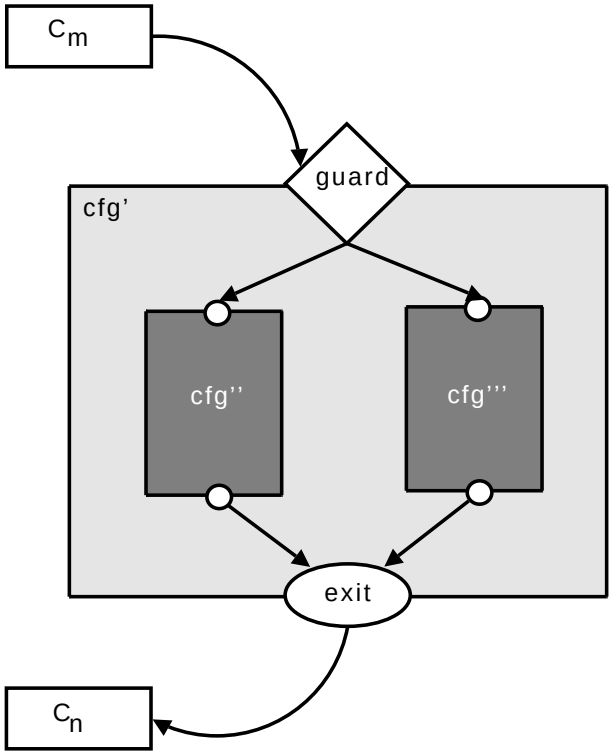

Figure 4.11: The If-Then-Else alternative subgraph

4.6 Global Definitions

As mentioned in the previous chapter, an ANSI-C program is made up out of global declarations and function definitions. Since both the include and define for constants pre-processor statements serve no purpose in the context of this project, they will be discarded. Since structs are reduced to global declarations when normalizing the source code, only the global declarations need to be transformed into the control flow graph.

ANSI-C dictates that declarations must be declared before statements and therefore, it can be assumed that putting all globally defined variables in front of the variables declared in the main function follows that rule. The global and local declarations will be separated by a separator edge labeled "SEPARATOR". This edge can later be used to determine which ATerms are defined locally and which globally. Since we are only interested in globally defined ATerms, a separation can be made when handling the global declarations. Only global ATerm declarations will be put in front of the variable declarations of the main function.

4.7 Flow-altering statements

As mentioned in the beginning of this section, the ANSI-C language provides the programmer with the means to alter the program flow with statements like "return", "break" and a "function call". The difference with regard to the non-flow-altering statements described above is, that is not a matter of sequentially linking together subgraphs. Sometimes an extra edge that spreads across blocks is needed to alter the flow in the desired way.

4.7.1 Function Calls

Throughout the program, functions will be called for specific computations or actions, thus breaking up the complex program into smaller, easier to compute, chunks. Many of these functions will be grouped together in a library which is then imported into the program (e.g. the function "printf" in the "stdio" library for in and output functionality).

In this proof-of-concept, only functions that are provided in the same file as the "main" function are considered. These function definitions are expected to be above the main function. What essentially happens when such function is called is that the flow of the current function is temporarily interrupted and the focus is shifted to another control flow (sub)graph, namely the graph of the function being called. Upon successful termination of that function, the focus is restored to the original control flow graph from which the function call originated.

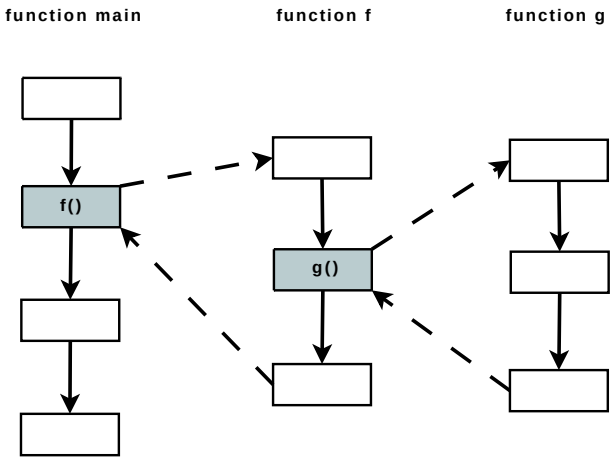

Figure 4.12: Flow and focus change in a function call

Figure 4.12 illustrates the shift of focus. In order to be able to use such a function and later check its path with a model checker, its control flow graph needs to pasted into the current CFG by means of subgraph insertion between the actual function call statement and an additional exit node. This is illustrated in figure 4.13.

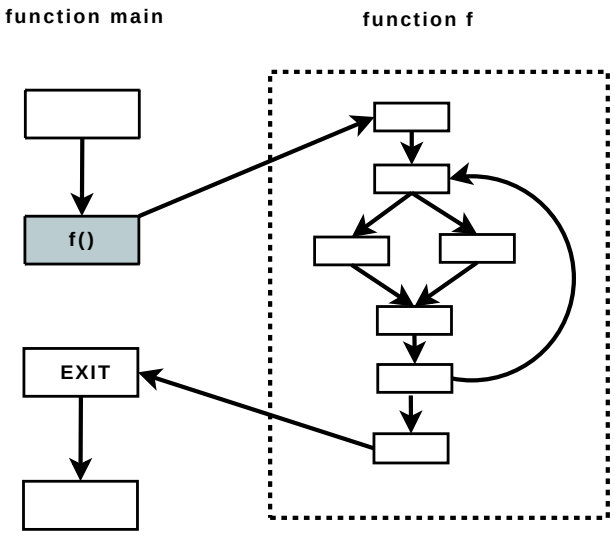

Figure 4.13: Pasting a function as a subgraph

Because a function call can be reduced to subgraph insertion with an unique entry and additional exit node, the control flow graph will remain to have a path from the start node to the stop node.

4.7.2 The problem of recursion

It is possible for functions to call themselves again and again until some exit condition is reached. This is called recursion and this can be done directly (e.g. function f calls function f) or indirectly (e.g. function f calls function g that calls function f). A very simple example in pseudo code is a function that computes x to the power n. Consider figure 4.14.

```
int power(int  x, int n) {
        if(n == 0) { return 1; }
        else { return x * power( x , n - 1 ); }
}
```

Figure 4.14: Example of direct recursion

The exit condition is reached when $n = 0$. If this is encountered, the recursive function will terminate and return the value of x to the power n.

If the recursive function were to be inserted in the control flow graph as if it were a non-recursive function, insertions would have to be repeated infinitely often since the condition of n is not evaluated. Also indirect recursion proves to be a problem since this also would mean infinitely many insertions, just on a

larger scale. Consider example 4.7.1:

Example 4.7.1. *The following code:*

```
f ( ) {                          g ( ) {
     ...;                             ...;
     g( );                            f( );
     ...;                             ...;
}                                }
```

Would result in:

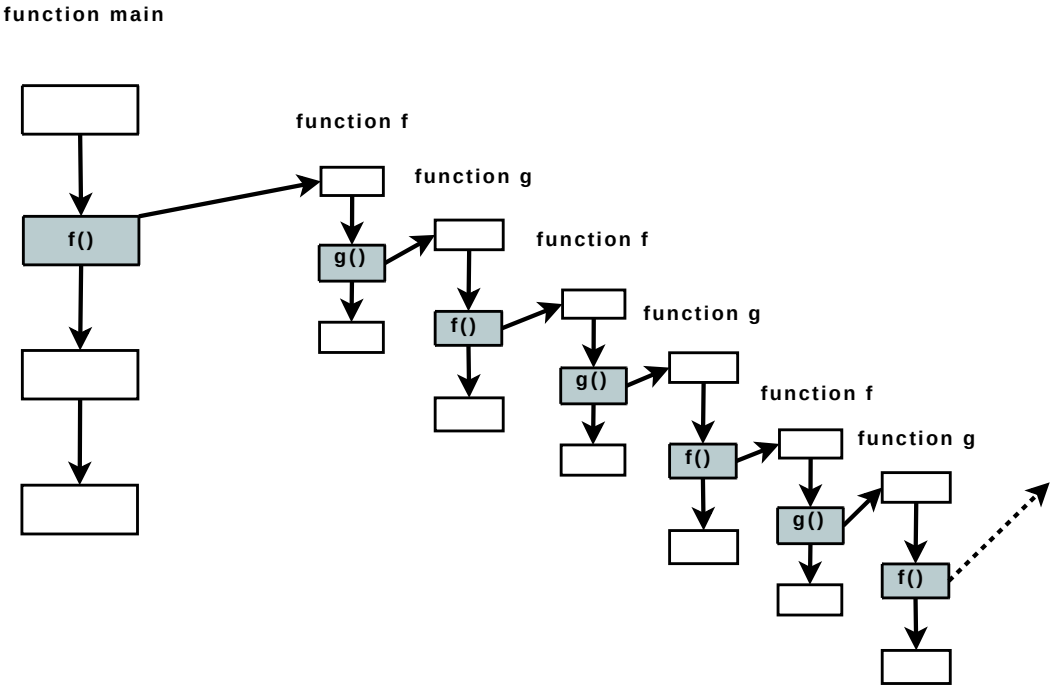

Figure 4.15: Indirect recursion causes infinite insertion

(End of Example)

Unfortunately, there is no simple solution to recursion. An overapproximation is suggested to handle direct recursion. Recursion in general (both direct and indirect) is not to be considered solved and thus are not supported in the subset of C.

Overapproximation for direct recursion

Using back edges, an overapproximation can be constructed. This means there are possibly more paths in the control flow graph then there are in the execution of the program. To support this, a queue of function calls is kept

during construction of the control flow graph. This queue consists of tuples $t = (func, start)$ where $func$ is the name of the function and $start$ is the node number of the function call to $func$. At any time, the queue holds the call history from the current function back to main. Upon termination of a function, the function will be pasted into the control flow graph and the head of the queue is popped and discarded. The new current function will again be the head element. For example, the queue can look like figure 4.16.

```
[ (f, n), (main, 0) ]
```

Figure 4.16: Example of a call history

In figure 4.16, another call to f would result in an direct recursive cycle and would yield infinite behavior. Upon occurrence of the function call, the name of the function is checked against the call history. If the function that is called is the same as the current function, a back edge is created instead of pasting the function between the call and an additional exit node. The back edge is created from the new function call in function f to the startnode of the previous call to f which is node n. To make sure that after m function calls, the remaining statements of f are also executed m times, another back edge is made from the last node of function f to all previous function calls to f. During the construction of the control flow graph of function f, a list of recursive calls is kept to be able to create these edges. Figure 4.7.1 illustrates the use of back edges in direct recursion.

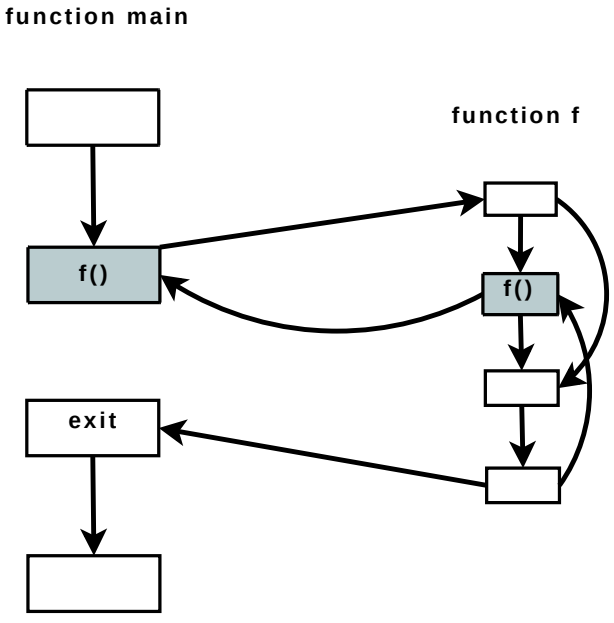

Figure 4.17: Using back edges to handle direct recursive functions

Note 4.7.1. *The function call creates a back edge to the actual function call instead of the first statement or declaration in the new function. In chapter 5, the control flow graph will be dualed (labels on edges instead of nodes). The transformation will correct the edges so that only one function call will be made in case of recursion.*

(End of Note)

A very important assumption is made for the overapproximation:

Assumption 4.7.1. *Recursively defined functions have a way of terminating.*

(End of Assumption)

This is illustrated in figure 4.17 by the arrow skipping the recursive call. The arrow from the recursive function call to the next statement of function f is needed to be able to execute the remainder of function f a number of times when the recursion is terminated. This is the case since only the node number of the call is kept.

The problems with indirect recursion

The back edges that work for direct recursion fail for indirect recursion. Since indirect recursion will be noticed when at least one function is completely inserted, additional edges must be created to ensure that all program executions are also possible in the control flow graph. Here the importance of the previously made assumption is evident. Upon inserting a non-recursive function, no edge is created between the call node and exit node. This is not needed because upon completion of the called function, the focus will be restored upon the calling function. However, when indirect recursion is encountered, and the previous functions all have been inserted, each of the functions in the recursion must be able to terminate the recursion. Each of the function therefore must have an edge skipping the function call in order to accomplish this otherwise the recursion will only be able to stop at the last of all functions in the recursion. This could for example be an else-branch that does not have the recursive function call. Consider figure 4.18.

The dotted arrow in figure 4.18 illustrate the extra edge that should be constructed to include all possible execution paths. Note that the function call to function f in h has no exit node. Similar to the case of direct recursion, this is because a back edge is made. An extra edge must be created from the end of the recursively called function f to all previous function calls to f to ensure that the remaining statements of h can be executed when the functions come out of recursion.

Unfortunately, the incremental nature of the control flow graph construction algorithm does not allow for a posteriori manipulation of edges in the control flow graph. After function h is inserted, the last node of function f is already inserted in the main control flow graph and is not easily retrievable. This is why indirect recursion an recursion in general can not easily be solved using the current incremental insertion algorithm.

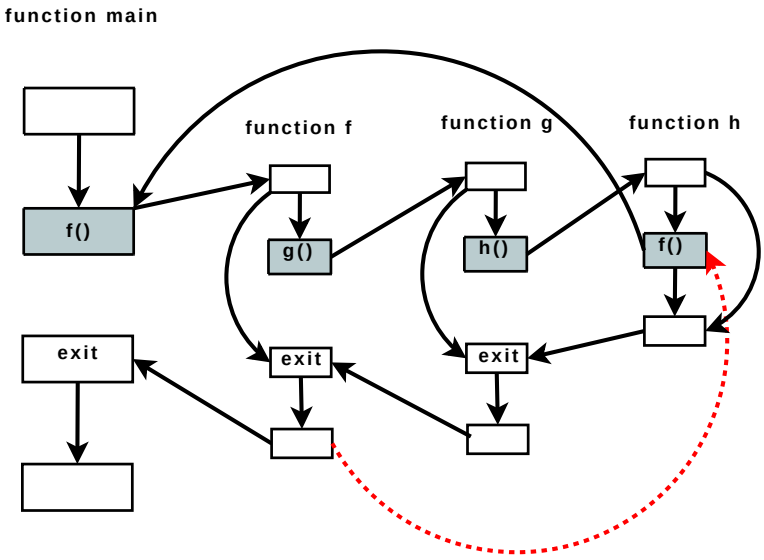

Figure 4.18: Need for an additional edge to cover possible paths

4.7.3 Return statements

A return statement will end a procedure. If the return statement is located in the "main" function of the program, the entire program will be terminated whereas if the return statement is present in any other function, that particular function ends and the flow continues a the point where the call to that function was made.

The call history proves valuable here. Since the head element of the queue represents the current function, we can check whether the return statement should link to the exit node of the corresponding function call (in case of a non-main function, the subgraph of the function is inserted between the function call and an additional corresponding exit node) or a link should be made to the stop node (i.e. node 1) of the control flow graph, terminating the entire program.

The edge from an return statement in the main function to the stop node of the control flow graph is not a local edge. As mentioned before, the return and break statements are the only ones that violate the uniquely defined exit point rule. The edge can always be made, because the stop node number is fixed to 1. Figure 4.19 illustrates the use of a return statement in an if-then-else alternative located in the main function.

4.7.4 Break statements

Break statements will "break" from the current repetition or switch statement and place the focus on the next statement after the repetition or switch. The break statement is often used as alternative termination for infinite repetitions. To be able to handle break statements, a break-destination is kept during the

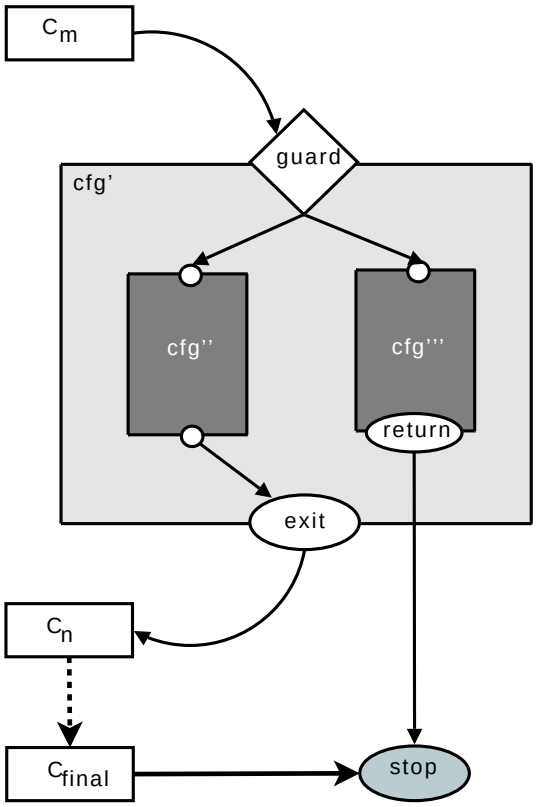

Figure 4.19: Return statement in the main function

building of the control flow graph. This destination is the node number of the unique exit node of the repetition and is updated everytime a new repetition is encountered. When a break statement is encountered, all remaining statements that still had to be computed (i.e. the statements following the break, if any, in the same branch/body) will be discarded and an edge will be created between the break statement and the exit node of the repetition thus ending it.

Figure 4.20 illustrates what happens when a break statement is encountered in a branch of the if-then-else alternative while in a While-Do repetition. Note that the break destination is the exit node of the repetition, since break does not have a purpose in an if-then-else alternative.

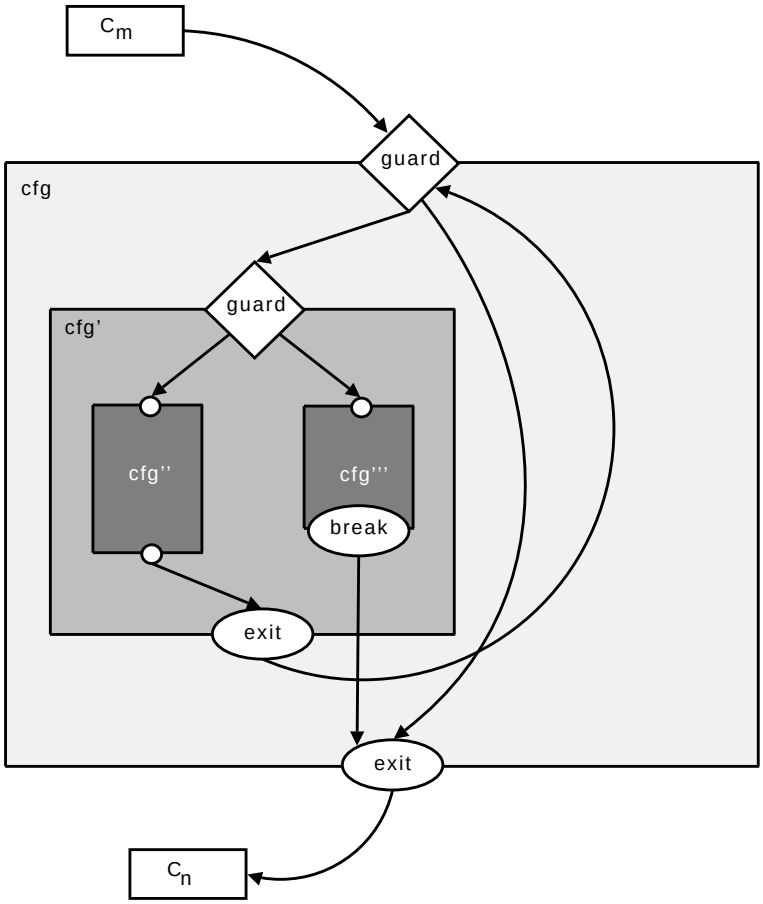

Figure 4.20: Break statement in a branch during a repetition

4.8 The nail warehouse

Consider the toy example of the nail warehouse mentioned in the introduction. The ANSI-C code for this warehouse is the following:

```
#include <stdio.h>
#include <aterm2.h>

ATermInt inv;

void sell(int number)
{
        ATermInt s;
        s = ATmakeInt(number);
        ATprintf("Nail number %t sold\n", s);
}

int order(void) { return 400000; }
```

```
int main(int argc, char* argv[])
{
        ATerm bottomOfStack;
        int k = 400000;
        int l;

        ATinit(argc, argv, &bottomOfStack);

        inv = ATmakeInt(400025-400000);

        while(k > 0) {
                sell(k);
                k--;
        }

        if(ATgetInt(inv) < 1000) {
                ATprotect(&inv);
                l = order();
                inv = ATmakeInt(order() + ATgetInt(inv));
                ATprintf("New nails ordered; New inventory = %t\n", inv);
        }
        else {
                ATprintf("Nothing ordered; Inventory = %t\n", inv);
        }

        ATprintf("Final Inventory = %t\n", inv);
        ATunprotect(&inv);

        return 0;
}
```

Using the building algorithm described in this chapter, the control flow graph can be built. Figure 4.21 shows the result.

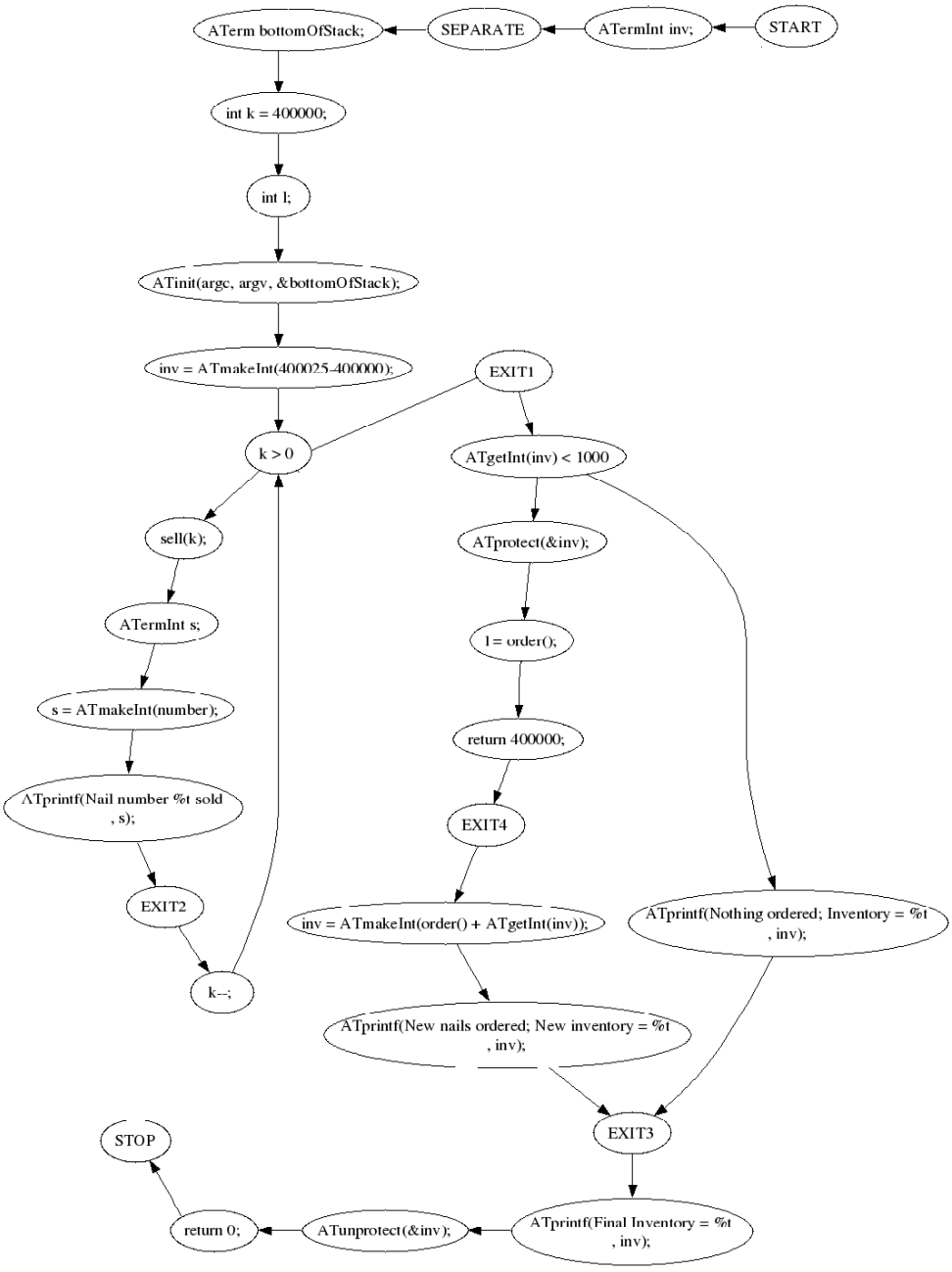

Figure 4.21: Control flow graph of the nail warehouse

Chapter 5

The Dual Graph

Some Model Checking systems require labels to be on the edges instead of on the nodes. The evaluator in the "Construction and Analysis of Distributed Processes" (CADP) toolkit [4], developed by the VASY team at INRIA Rhone-Alpes, is one of those checkers.

5.1 Structure of the dual graph

After the control flow graph, the "dual graph" is introduced as a second intermediate structure for capturing the control flow of an ANSI-C program. The dual graph can be defined as follows:

Definition 5.1.1. *A dual graph is an edge-labeled directed graph $DG = (T, s)$ with T being a set of labelled transitions and s being the start node of the graph. Each transition $t \in T$, is a triple $t = (i, l, j)$ consisting of an origin i, a transition label l and a target j.*

(End of Definition)

Note 5.1.1. *As was done in the control flow graph, also in the dual graph, the start symbol s is fixed to 0. A deadlock state denotes the end of the graph.*

(End of Note)

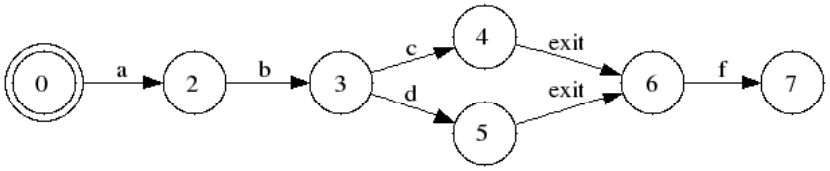

Figure 5.1: Example of an arbitrary Dual Graph

5.2 Building the dual graph

A given control flow graph $CGF = (N, E, s, t)$ is transformed into a dual graph $DT = (T, s)$ with $T = \{(e, label_{e'}, e') \mid (e, e') \in (E \setminus \{(x, 1)\}) \wedge (e', label_{e'}) \in N\}$

Figure 5.2 illustrates the step-by-step construction of a dual graph from a control flow graph. The same procedure is illustrated in figure 5.3. Note that the edge that is about to be transformed is made bold.

Control Flow Graph	Dual Graph
$\langle$ {(0,START),(1,STOP),(2,a),(3,b),(4,c), (5,d),(6,EXIT),(7,f)}, {**(0,2)**, (2,3), (3,4), (3,5), (4,6), (5,6), (6,7), (7,1)} $\rangle$	{ }
$\langle$ {(0,START),(1,STOP),(2,a),(3,b),(4,c), (5,d),(6,EXIT),(7,f)}, {**(2,3)**, (3,4), (3,5), (4,6), (5,6), (6,7), (7,1)} $\rangle$	{(0,a,2)}
$\langle$ {(0,START),(1,STOP),(2,a),(3,b),(4,c), (5,d),(6,EXIT),(7,f)}, {**(3,4)**, (3,5), (4,6), (5,6), (6,7), (7,1)} $\rangle$	{(0,a,2),(2,b,3)}
$\langle$ {(0,START),(1,STOP),(2,a),(3,b),(4,c), (5,d),(6,EXIT),(7,f)}, {**(3,5)**, (4,6), (5,6), (6,7), (7,1)} $\rangle$	{(0,a,2),(2,b,3),(3,c,4)}
...	...
$\langle$ {(0,START),(1,STOP),(2,a),(3,b),(4,c), (5,d),(6,EXIT),(7,f)}, { } $\rangle$	{(0,a,2),(2,b,3),(3,c,4),(3,d,5), (4,EXIT,6),(5,EXIT,6),(6,f,7)}

Figure 5.2: Building the dual graph illustrated

The control flow graph holds transitions going from a return node (or final statement) to the stop node 1. Since the label for the dual graph edge is stored in the target node (i.e. 1 with "STOP" as label) it does not serve any purpose in the dual graph (it is not a declaration or statement) and will be skipped entirely. This way, exiting a program with a transition to 1 in the control flow graph will be deadlock state in the dual graph.

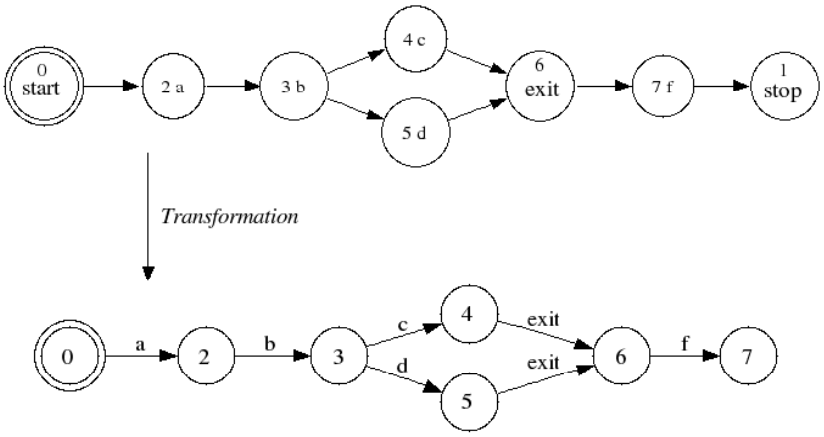

Figure 5.3: Example of control flow graph to dual graph transformation

Together with the control flow graph, the dual graph forms the intermediate representation of the program flow. Depending on the kind of input the model checker takes (i.e. labels on nodes or edges), one of these graphs can be used for further transformation towards that format. The dual graph contains the same information as the control flow graph. The entire declaration and statements are still used as action labels and the flow of declarations and statements is unaltered.

5.3 The nail warehouse revisited

Consider the control flow graph of the nail warehouse from the previous chapter. Figure 5.4 shows the result of transformation of the control flow graph into a dual graph.

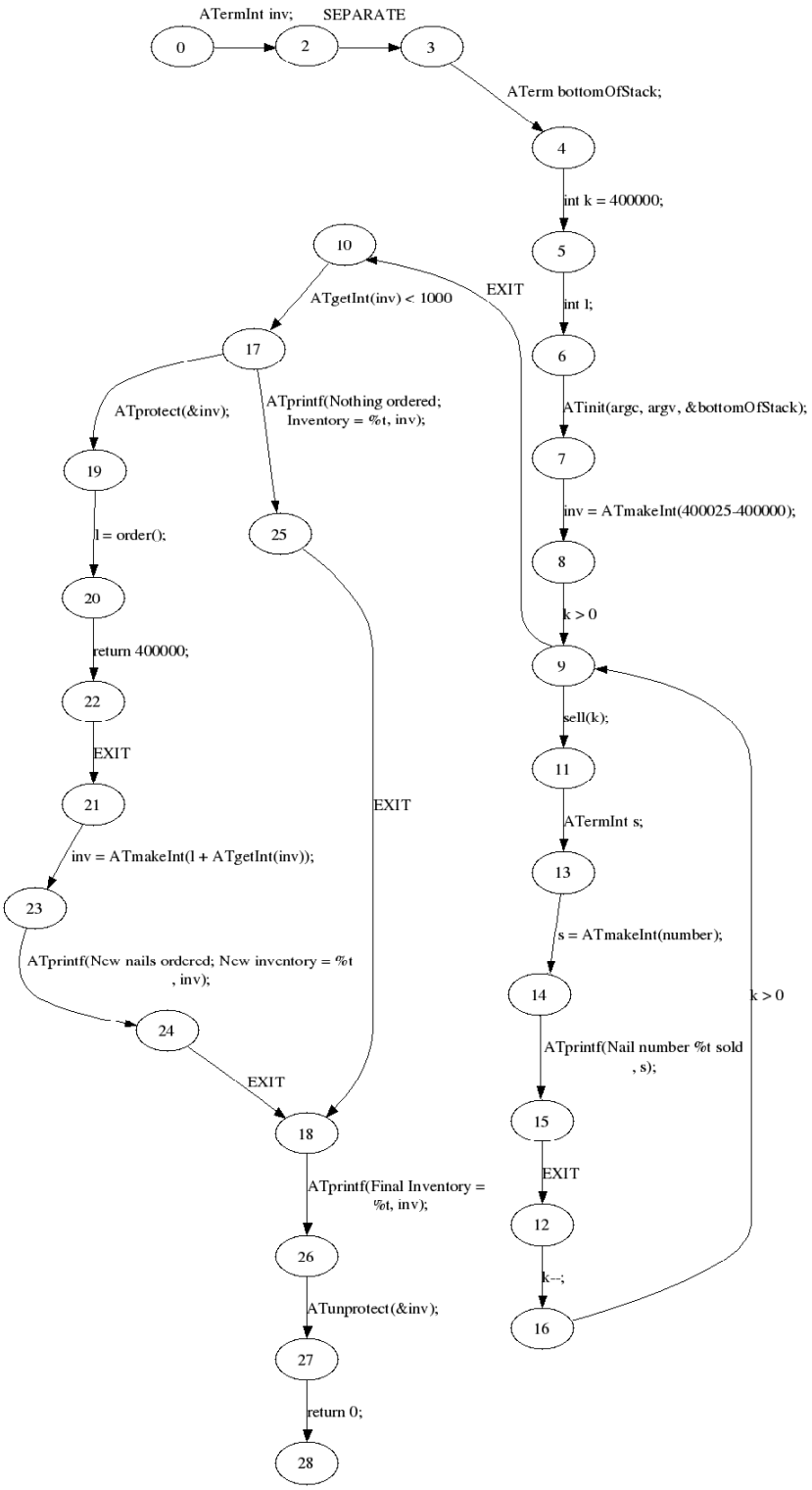

Figure 5.4: Dual Graph of the nail warehouse

Chapter 6

Abstraction

The main research question is to see if model checking techniques can be used to find certain patterns of unwanted behavior by checking temporal properties against some sort of model of the control flow of an ANSI-C program.

Since only a specific set of patterns needs to be checked against the model, not all declaration and statement information is needed when looking for global ATerm specific information. All possible paths in the model will be checked and hence, it is not so important to know whether an ordinary integer value is increased or what value is written to the screen, provided that it is not a value stored in a global ATerm.

A quick summary of the patterns that are interesting to check for:

- Any definition or use of an ATerm should be placed between a protect and an unprotect action.
- Every protect on a global ATerm is eventually followed by an unprotect on that ATerm and vice versa.

To that end, a rather aggressive abstraction technique, which looks a lot like slicing [29], is used to filter out everything that has nothing to do with global ATerms. This includes guard conditions if no global ATerms are involved. In order to do this, a new intermediate structure is needed to hold the abstracted data. The abstract graph is used for this.

6.1 The Abstract Graph

After creating a control flow graph and a dual graph, a third intermediate representation, the "abstract graph" is introduced. This graph is similar to the dual graph in structure with the difference that the action labels for all declarations and statements in the graph will be one of the following labels (where it name is the name of the global ATerm):

- *decl_name* for the declaration of a global ATerm
- *def_name* for the definition of a global ATerm

- *use_name* for the use of a global ATerm
- *prot_name* for the protection of a global ATerm
- *unprot_name* for the unprotection of a global ATerm
- *i* (internal action) for any action that it not related to a global ATerm.

6.2 Building the abstract graph

The build process consists of two phases. As mentioned shortly, in the first phase, the variable names of globally declared ATerms are collected in a list whereafter all declarations and statements in the labels of the dual graph edges are checked for the occurrence of these variable names and labeled accordingly.

6.2.1 Phase 1: Traversing the external declarations

A traversal over the transitions from the start of the dual graph to the *separator tag* constructs a list of variable names which are declared as global ATerms. All declarations done after the separator tag are local an thus automatically protected from removal. The traversal function looks into every declaration and breaks it down to the components listed in figure 6.1.

```
Specifier+ Identifier; -> Declaration
Specifier+ Identifier [ Expression ] ; -> Declaration
```

Figure 6.1: Declaration

All global ATerm names (identifiers) will be collected in a list of identifiers.

6.2.2 Phase 2: Abstracting declarations and statements

With the list of global ATerm names available, all transitions in the dual graph are "abstracted". Each declaration and statement (label on the dual graph transitions) will get one of the labels described in paragraph 6.1 according to their behavior. Consider the following behaviors and labels:

Declaration

If a global ATerm or array is declared, one of the types defined in the ATerm library would appear as a specifier followed by an identifier which holds the ATerm's name. Consider for example the declarations in figure 6.2.

The global ATerm of type integer called *foo* and an ATerm array called *bar* is declared. This edge in the abstract graph will receive the label: *decl_foo* and *decl_bar*

Note 6.2.1. *A declaration can be combined with initialization as is illustrated in figure 6.3.*

```
    Specifier+ Identifier ;          Specifier+ Identifier [ Expression ] ;
=>                                 =>
    ATermInt foo;                    ATermInt bar[10];
```

Figure 6.2: Basic declarations

```
    Specifier+ Identifier = Initializer;
=>
    ATermInt foo = ATmakeInt(42);

    Specifier+ Identifier [ Expression ] = Initializer;
=>
    ATermInt bar[10] = fillArray(...);
```

Figure 6.3: Declarations with initialization

This combined action will be split up into 2 separate edges in the abstract graph, namely the declaration first, followed by the definition. The two new edges will get the labels: decl_foo and def_foo respectively in the first example and: decl_ bar and def_ bar respectively in the second example.

(End of Note)

Definition

A global ATerm or an element of an array of ATerms is defined when it gets a value assigned to it. For example, in the following declaration in figure 6.4, a global ATerm of type integer called *foo* is assigned the value 42 and the element at index 0 in array *bar* is assigned value 371.

```
    Identifier = Expression;
=>
    foo = ATmakeInt(42);

    Identifier [ Expression ] = Expression;
=>
    bar[0] = ATmakeInt(371);
```

Figure 6.4: Basic definitions with function call

A definition is of interest since it is one of the actions that require the ATerm or array to be protected. In the definition, an Identifier containing the name of the ATerm is located at the left-hand side of the assignment operator. Definition of a global ATerm and an element in a global array of ATerms get the labels: *def_foo* and *def_ bar*.

Note 6.2.2. *A definition can have an ATerm in the expression on the right-hand side. This is for example the case in the example in figure 6.5 (where foo*

is a global ATerm of type integer and bar is a global array of ATerms of type integer):

```
foo = ATmakeInt(ATgetInt(bar[1]));
bar[0] = ATmakeInt(ATgetInt(foo));
```

Figure 6.5: Definitions and Use

The simultaneous definition and use of multiple ATerms will be split up into as many use labels as there are ATerms used followed by a definition, since the value must first be retrieved before it can be assigned. In the examples, the corresponding labels for the new edges in the abstract graph are: use_bar and def_foo respectively in the first example and use_foo and def_bar respectively in the second example.

(End of Note)

Use

As described in the previous paragraph, an ATerm and array element will be used in an assignment if it is on the right-hand side of the assignment operator. However, it can also be used in another way.

Since expressions can again contain expressions, every expression needs to be checked internally to see if a match with a global ATerm name can be made at the Identifier level. Consider the case illustrated in figure 6.6 (where *foo* and *bar* are ATerms of type integer and *f* is an arbitrary function).

```
    Expression;
=>
    Expression "=" Expression;
=>
    Identifier "=" Identifier ( Expression );
=>
    Identifier "=" Identifier ( Identifier ( Identifier ) );
=>
    foo = f( ATgetInt( bar ) );
```

Figure 6.6: Examination of Expression provides new ATerm

Illustrated is the occurrence of an expression within an expression. If the expression on the right-hand side was not examined down to the identifier level, the use of the ATerm *bar* would not have been discovered. Every expression that is encountered must therefore be traversed in order to discover all occurrences of ATerms. This means that ATerms can also be used as argument in function calls or in guard conditions. A few more examples are given in figure 6.7.

References to a global ATerm *foo* get the label: *use_foo*.

```
if ( ATgetInt(foo) ) { ... }
f(foo);
ATprintf("value = %t\n", foo);
```

Figure 6.7: Examples of use

Note 6.2.3. *When more than one global ATerm is used in a statement, the edge will be split up into a number of new edges in the abstract graph. The number depends on how many ATerms are used. Each of them will get a use label assigned to it. For example, consider figure 6.8 The same holds for the use of an element in an array of ATerms.*

```
f(foo, bar, baz[1]);
```

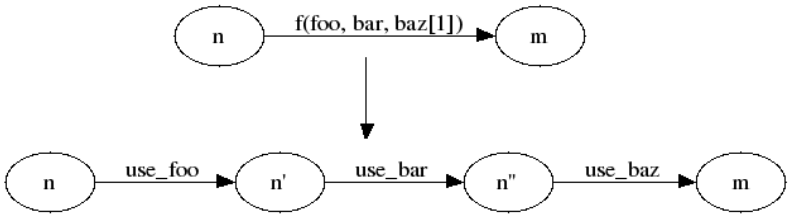

Figure 6.8: Multiple used ATerms

Function f uses 3 ATerms and will be split up into 3 separate edges in the abstract graph labeled: use_foo, use_bar and use_baz respectively.

(End of Note)

Protecting

An ATerm is protected when the *ATprotect( )*, *ATprotectArray( )* or *ATprotectAFun( )* function is called with a reference to the address of the ATerm that is to be protected. Consider for example figure 6.9 (where *foo* is an ATerm, *bar* an array of ATerms, *baz* a global function symbol and *size* is the size of the array).

```
ATprotect(&foo);
ATprotectArray(bar,size);
ATprotectAFun(baz);
```

Figure 6.9: Protection of ATerm structures

The protect actions in figure 6.9 get the labels: *prot_foo*, *prot_ bar* and *prot_ baz*.

Note 6.2.4. *Even though the actual value of the array index is not examined and it is not possible to inspect the use of specific elements in the array, this*

poses no problem. According to the ATerm user manual (see [15]), it is only possible to protect the entire array with ATprotectArray.

(End of Note)

Unprotecting

Similar to protection, an ATerm or array element is unprotected with the *ATunprotect( )*, *ATunprotectArray( )* or *ATunprotectAFun( )* function. Consider the example in figure 6.10 (where *foo* is an ATerm, *bar* an array of ATerms and *baz* a function application). The unprotect actions get the labels: *unprot_foo*, *unprot_bar* and *unprot_baz*.

```
ATunprotect(&foo);
ATunprotectArray(bar);
ATunprotectAFun(baz);
```

Figure 6.10: Unprotection of ATerm structures

6.3 Nail warehouse example

Consider the dual graph of the nail warehouse in figure 6.11. Given the set of labeling rules presented above, all edges in the dual graph can now be labeled constructing the abstract graph presented in figure 6.12.

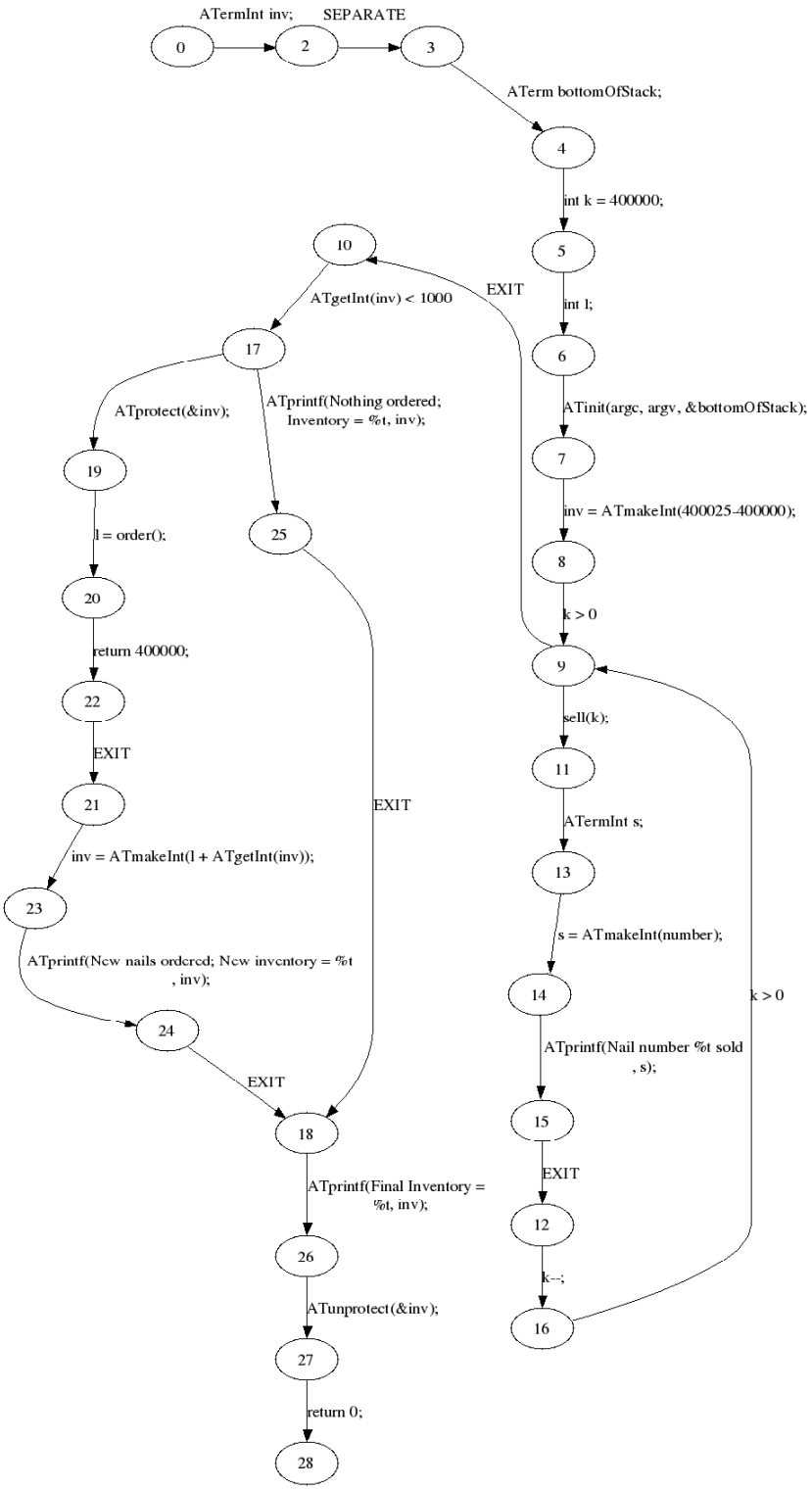

Figure 6.11: Dual Graph of the nail warehouse

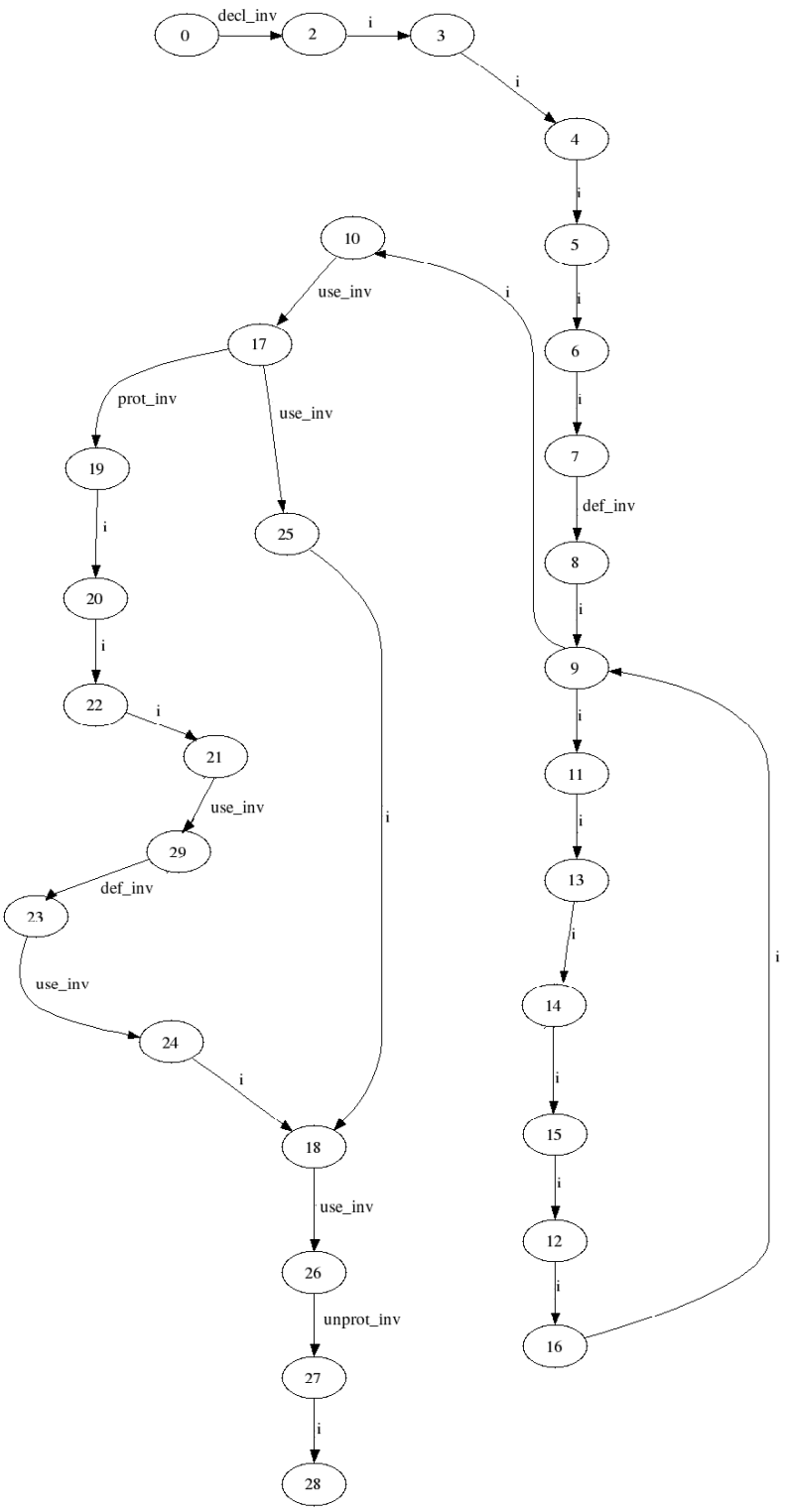

Figure 6.12: Abstract Graph of the nail warehouse

Chapter 7

Model Checking with CADP

Van de Pol gives a description of model checking in [31]:

> "Model Checking is an automatic verification method, to check that a requirement holds for a model of a system."

Given this definition, the abstract graph which holds an abstracted version of the flow of the program will be the model of the system and the requirements that need to be checked will be the patterns of unwanted behavior.

7.1 The CADP toolkit

The CADP (Construction and Analysis of Distributed Processes) toolkit [4] includes several tools that can be used for preparing and model checking the abstracted models of programs.

The CADP toolkit's on-the-fly model checker called evaluator, that is able to check safety properties such as patterns of proper use in regular alternation-free μ-calculus against a model of a program, works with binary coded graphs. This is a format for representing labeled transition systems in a very efficient way. Unfortunately neither the specification nor implementation of this format is public information.

To be able to use the evaluator, a transformation can be made from the Aldebaran graph format (AUT) to the BCG format using the *bcg_io* tool in the CADP toolkit. This tool works with a number of different formats (e.g. dot, bcg, aut) and can translate any of those formats into any other format it supports. Furthermore, the patterns of proper use that apply to the using the ATerm library need to be translated into regular alternation-free μ-calculus which is a calculus that describes the behavior of a system.

7.1.1 The Aldebaran Graph

An Aldebaran graph is very similar to the abstract graph. The Aldebaran graph needs a list of transitions as tuples of a start node number, action label and a

stop node number, each on a new line, and a header that contains a summary of the graph.

Abstract Graph	Aldebaran Graph
[	des (0,35,35)
(0,decl_inv,2) ,	(0, "decl_inv" ,2)
(2,i,3) ,	(2, "i" ,3)
(3,i,4) ,	(3, "i" ,4)
... ‘	...
(6,i,23) ,	(6, "i" ,23)
(23,use_inv,24) ,	(23, "def_inv" ,24)
... ‘	...
(34,use_inv,35) ,	(34, "use_inv" ,35)
(17,i,21)	(17, "i" ,21)
]	

The header of the Aldebaran graph is made up from the word “des” followed by a 3-tuple consisting of the graph's start node number, number of transitions and number of nodes. Like was done in all previous graphs, the start node of the Aldebaran graph is fixed to 0. The number of transitions is equal to the number of elements in the edgelist of the abstract graph and the number of nodes is computed to be the maximum of all node numbers + 1.

There are two more layout-based issues that are different from the abstract graph. In the Aldebaran graph, the transitions are separated by a newline instead of a comma and the action labels are placed between double quote symbols.

7.1.2 Reducing the models

After transforming the Aldebaran graph into a binary coded graph, reduction can be applied to minimize the size and complexity of the model. Because of the use of i transitions (also known as *tau-transitions*) in the model of the program, the technique of branching bisimulation reduction can be applied to reduce the state space of the graph. This reduction technique is implemented in the CADP toolkit's *bcg_min* tool.

Note 7.1.1. *For reduction, the bcg_min tool is used with options -normal and -branching*

(End of Note)

The result of reducing the state space of the nail warehouse problem is illustrated in figure 7.1.

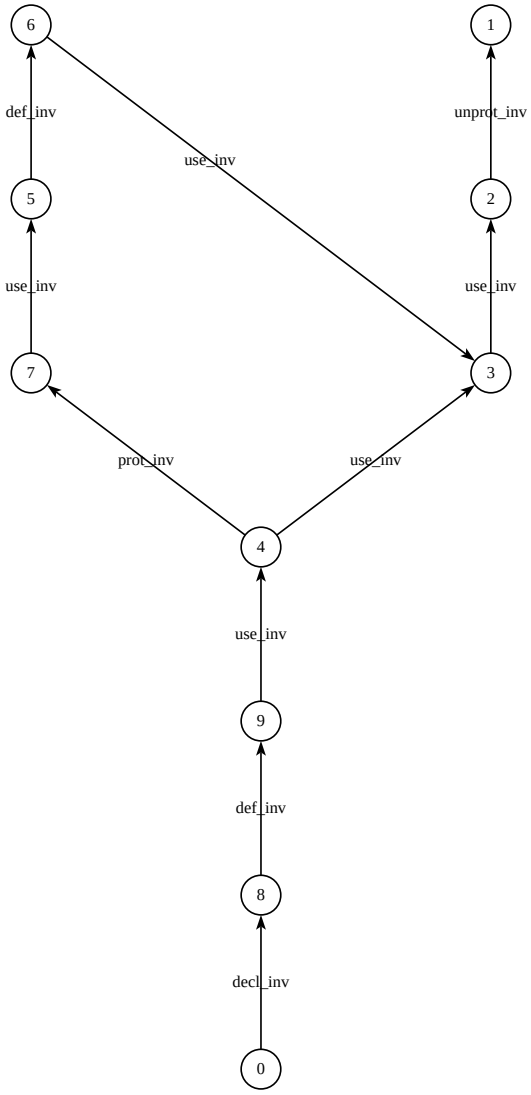

Figure 7.1: Reduced nail warehouse graph

7.1.3 Regular alternation-free μ-calculus

The requirements that can be checked on the model are described in regular alternation-free μ-calculus (described in [28] and [3] and figure 7.2). This calculus is an extension of modal-μ-calculus [14]. The CADP evaluator uses on-the-fly model checking for this which explores the state space on demand during the verification of the temporal formula.

By using regular alternation-free μ-calculus the systems behavior over time can be described. For example: using the [ ] and $\langle\ \rangle$ operators, a property can be described that needs to holds for all paths or at least one path. μ and ν denote fixed point operators and repetitions are denoted by * (zero or more times) and

```
Actions       A    ::=  string | 'regexp' | "true" | "false" | "not" A
                        | A1 "or" A2 | A1 "and" A2 | A1 "implies" A2
                        | A1 "equ" A2

Regular       R    ::=  A | "nil" | R1 "." R2 | R1 "|" R2 | R "*" | R "+"

State         F    ::=  "true" | "false" | "not" F | F1 "or" F2
                        | F1 "and" F2 | F1 "implies" F2 | F1 "equ" F2
                        | "<" R ">" F | "[" R "]" F | "@" "(" R ")"
                        | X | "mu" X "." F |"nu" X "." F

(Where X is a propositional variable and @ is an infinite loop operator)
```

Figure 7.2: μ-calculus

+ (at least one time). An example of a simple property that can be described by regular alternation-free μ-calculus:

Example 7.1.1. *Consider the following formula:*

```
[ true* . "OPEN !1" . (not "CLOSE !1")* . "OPEN !2" ] false
```

This temporal logic formula describes the following property: When process 1 executes the "open" action and process 2 executes "open" before process 1 had executed "close", the system is faulty.

(End of Example)

7.1.4 Evaluator

With the abstracted and reduced model and the safety properties in regular alternation-free μ-calculus, the CADP evaluator can be used to see if the properties hold for the models (i.e. if ATerms are used properly) or some property is violated (i.e. improper use of ATerms is detected). If a property is violated, the model checker generates an output trace from the start of the graph up to the point where the violation occurred. This may prove useful in repairing the error. All test programs in appendix B will be checked.

The evaluator is called in the following way:

```
bcg_open model.bcg evaluator -diag eval.bcg prop.mcl
```

This checks model *model.bcg* against property *prop.mcl* and provides trace *eval.bcg* in BCG format as well as on the screen. An example of such a trace is illustrated in figure 7.3.

```
FALSE
diagnostic sequence found at depth 3

<initial state>
"decl_mylist"
"i"
"def_mylist"
<goal state>
```

Figure 7.3: Trace of violation provided by the evaluator

7.2 Behavioral patterns

The patterns of unwanted behavior described in chapter 2 can be translated into regular alternation-free μ-calculus.

7.2.1 Protect before use

One of the patterns of unwanted behavior dictates the need to protect global ATerms prior to using or defining them. This pattern is split up into two formulas. Given in μ-calculus (where it a is the name of the global ATerm):

(1a) $[true* \text{ . "}decl\_a\text{" . } (not \text{ "}prot\_a\text{"})* \text{ . "}use\_a\text{" } or \text{ "}def\_a\text{"}]\ false$

Loosely translated: After the declaration of global ATerm a, no use or define action on a may occur when no protect action on a has preceded.

(1b) $[true* \text{ . "}unprot\_a\text{" . } (not \text{ "}protect\_a\text{"})* \text{ . "}def\_a\text{" } or \text{ "}use\_a\text{"}]\ false$

Loosely translated: After an unprotect action on a, no use of define action on a may occur without protecting a again.

When checking these two requirements against the model of the program that has improper ATerm use, a trace up to the violating statement is given.

7.2.2 Matching protect and unprotect

The other pattern that needs to be checked is the match between a protect and an unprotect action. If there is a protect but no unprotect, the memory will not be released creating a memory leak. If there is an unprotect but no protect action, the program will crash with a segmentation fault. Again translated in μ-calculus (where it a is the name of the global ATerm):

(2a) $[true* \text{ . "}prot\_a\text{" . } (not \text{ "}unprot\_a\text{"})*] \langle (not \text{ "}unprot\_a\text{"})* \text{ . "}unprot\_a\text{"} \rangle\ true$

Loosely translated: After every protect action, there eventually must be an unprotect action. This catches the case where a protected ATerm is not unprotected.

(2b) $[true* \ . \ "decl\_a" \ . \ (not \ "prot\_a")* \ . \ "unprot\_a"] \ false$

Loosely translated: After declaring a global ATerm, it needs to be protected before it can be unprotected. This catches the case where a global ATerm is unprotected but no preceding protect is present.

7.3 Checking the models

Consider the following table that gives an overview of the test programs in appendix B and how the ATerm library is used.

Program	Use of ATerms
Nail Warehouse 1	Global ATerm is defined without protection; Else-branch does not have protect (mismatch0match).
Nail Warehouse 2	Warehouse 1 fixed: Proper use of the ATerm library
Test program 1	Global ATerm is defined without protection.
Test program 2	Program 1 fixed: Proper use of ATerm library.
Test program 3	Global ATerm array is defined without protection.
Test program 4	Program 3 fixed: Proper use of ATerm library.
Test program 5	Global ATerm is protected but not unprotected.
Test program 6	Global ATerm array is protected but not unprotected.
Test program 7	Global ATerm is defined without protection; Unprotect without preceding protect (mismatch).
Test program 8	Global ATerm array is defined without protection; Unprotect without preceding protect (mismatch).

7.3.1 The Nail Warehouse

Nail Warehouse 1: Improper use

Consider again the reduced and abstracted graph of the nail warehouse problem in figure 7.1 and source code in appendix B.1. Using the safety properties in paragraph 7.2, the CADP evaluator can check for improper use of global ATerms.

The actual output of execution of this program is the following:

```
Nothing ordered; Inventory = 47961
Final Inventory = 47961
Segmentation fault (core dumped)
```

Checking the abstracted model against the safety properties, shows why the program crashed and gives a strange inventory number.

property (1a)	property (1b)	property (2a)	property (2b)
FALSE diagnostic sequence found at depth 2 <initial state> "decl_inv" "def_inv" <goal state>	TRUE	TRUE	FALSE diagnostic sequence found at depth 6 <initial state> "decl_inv" "def_inv" "use_inv" "use_inv" "use_inv" "unprot_inv" <goal state>

The output of the model checker now tells where improper use has been made. First of all, the trace given by property (1a) (i.e. declaration and use or define action must always have a protect action between them) shows the execution of actions done from <initial state> (start of the graph) to a state <goal state> which violates property (1a). As can be seen, after a declaration, a define action is done. In the code this is also visible as is illustrated in figure 7.4. After the global declaration of ATerm *inv* in (1), the first action on *inv* that occurs in the main function is the definition in (2). This violates property (1a) because there should have been a protect action between the declaration and assignment.

Secondly, the trace of property (2b) shows that there is a possible path from start to a state that has an unprotect action, but no preceding protect action. This is the case when during execution of the program, the else branch is taken. Please refere to appendix B.1 for the complete source code of the nail warehouse.

Checking the models has provided two traces that explain the two problems with the nail warehouse addressed in the introduction. Neglecting to protect a global ATerms (1a) before using it can result in arbitrary meaningless values

```
ATermInt inv;                                          (1)

int main(int argc, char* argv[])
{
        ATerm bottomOfStack;
        int k = 400000;
        int l;

        ATinit(argc, argv, &bottomOfStack);

=>      inv = ATmakeInt(400025-400000);                (2)
```

Figure 7.4: Violation of safety property (1a)

and unprotecting without protecting first (2b) results in a segmentation fault. During the execution of the program, the garbage collected ATerm provided an arbitrary value that made the program follow the else path.

Nail Warehouse 2: Proper use

Checking the corrected version of the nail warehouse problem in appendix B.2., the output of the model checker speaks for itself.

property (1a)	property (1b)	property (2a)	property (2b)
TRUE	TRUE	TRUE	TRUE

7.3.2 Test programs

Appendix B.3 to B.10 hold some of the test programs written to trigger the unwanted behavior. Model checking these programs provides the following results.

Program 1

Program 1 declares a global ATerm of type integer, assigns value 42 to is forces the garbage collector to clean up unused ATerms and outputs the value of the global variable. Running the program shows:

```
Value of global ATerm = 352040
```

Checking the abstracted model of this program against the safety properties, results in the following traces.

property (1a)	property (1b)	property (2a)	property (2b)
FALSE diagnostic sequence found at depth 2 <initial state> "decl_global" "def_global" <goal state>	TRUE	TRUE	TRUE

This was to be expected since the globally defined ATerm *global* is not protected before it is defined. Further more, no traces are given with regard to mismatches because no protect or unprotects occur in the code.

Program 2

Program 2 is the corrected version of program 1. Here protect and unprotect actions are used to prevent the removal of the global ATerm *global*. Running the program shows:

```
Value of global ATerm = 42
```

Checking the abstracted model of this program against the safety properties, results in the following output.

property (1a)	property (1b)	property (2a)	property (2b)
TRUE	TRUE	TRUE	TRUE

In program 2, the ATerms are used properly, therefore the output of the model checker is as expected.

Program 3

Program 3 is very similar to program with the difference that instead of assigning a value to a global variable, a globally defined array of ATerms is used. The array is filled with the values of the array index. Running the program shows:

```
value of global at index 0 = 352038
value of global at index 1 = 352037
value of global at index 2 = 352036
value of global at index 3 = 352035
value of global at index 4 = 352034
value of global at index 5 = 352033
value of global at index 6 = 352032
value of global at index 7 = 352031
value of global at index 8 = 352030
value of global at index 9 = 352029
```

For global array *mylist*, the output of the model checker is as follows.

property (1a)	property (1b)	property (2a)	property (2b)
FALSE diagnostic sequence found at depth 2 <initial state> "decl_mylist" "def_mylist" <goal state>	TRUE	TRUE	TRUE

Since the global array of ATerms is filled without protection, property (1a) is violated, hence the arbitrary values.

Program 4

Program 4 is the corrected version of program 3. Here protect and unprotect actions are used to prevent the removal of the global array of ATerms *mylist*. Running the program shows:

```
value of global at index 0 = 0
value of global at index 1 = 1
value of global at index 2 = 2
value of global at index 3 = 3
value of global at index 4 = 4
value of global at index 5 = 5
value of global at index 6 = 6
value of global at index 7 = 7
value of global at index 8 = 8
value of global at index 9 = 9
```

Model checking the model against the safety properties results in the following output.

property (1a)	property (1b)	property (2a)	property (2b)
TRUE	TRUE	TRUE	TRUE

In this program, proper use has been made of the ATerm library. Therefore all safety properties hold.

Program 5

In program 5, the global ATerm *global* is protected before it is defined. However, no matching unprotect action is done, creating a memory leak. No visible evidence of this can be seen in the output when running the program as can be seen below.

```
Value of global ATerm = 42
```

However, when model checking the abstracted model of this program, the improper use of global ATerms is evident.

property (1a)	property (1b)	property (2a)	property (2b)
TRUE	TRUE	FALSE diagnostic sequence found at depth 4 <initial state> "decl_global" "prot_global" "def_global" "use_global" <goal state>	TRUE

The trace given by the model checker runs from the initial state to the end of the model. Since no unprotect action was found to match the protect action, the end of the model is considered to be the goal state. This mismatch in protect and unprotect actions cause property (2a) to be violated.

Program 6

Program 6 is similar to program 5 except that it uses a globally defined array of ATerms instead of a global ATerm. The array is protected and each array element is assigned the value of the corresponding array index. The array is never unprotected causing a memory leak. Running the program shows the following output.

```
value of global at index 0 = 0
value of global at index 1 = 1
value of global at index 2 = 2
value of global at index 3 = 3
value of global at index 4 = 4
value of global at index 5 = 5
value of global at index 6 = 6
value of global at index 7 = 7
value of global at index 8 = 8
value of global at index 9 = 9
```

Model checking the model against the safety properties results in the following output.

property (1a)	property (1b)	property (2a)	property (2b)
TRUE	TRUE	FALSE	TRUE

In this particular example, no output trace is given for the violation of property (2a). It is not clear why no textual trace is given, but the visual trace in figure

7.5 shows why the model violates property (2a).

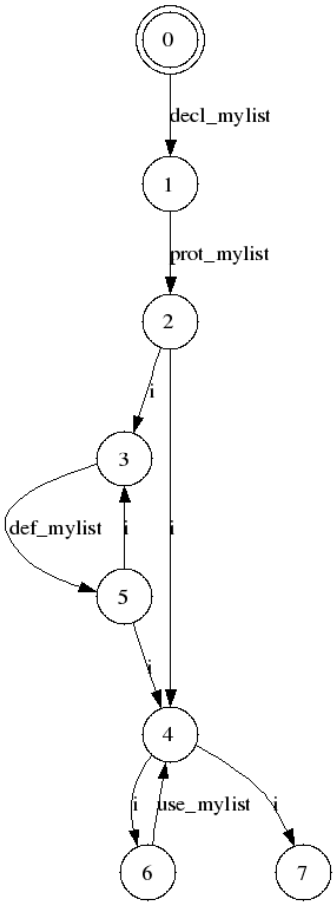

Figure 7.5: Visual trace of the violation of property (2a).

From the violation of property (2a), supported by the visual trace in figure 7.5, is is clear that no unprotect action matches the protect action in the model.

Program 7

Program 7 explores the other side of the protect and unprotect mismatch, namely a unprotect action without preceding protect action on the global ATerm *global*. This causes the program to crash with a segmentation fault as can be seen in the program output below.

```
Value of global ATerm = 352040
Segmentation fault (core dumped)
```

Besides the segmentation fault, the value of the global ATerm *global* should be 42. The output however shows a different value. Model checking the model

shows why this is the case.

property (1a)	property (1b)	property (2a)	property (2b)
FALSE diagnostic sequence found at depth 2 <initial state> "decl_global" "def_global" <goal state>	TRUE	TRUE	FALSE diagnostic sequence found at depth 4 <initial state> "decl_global" "def_global" "use_global" "unprot_global" <goal state>

Both property (1a) and (2b) were violated. This means that a globally defined ATerm was defined or used without protection and that an unprotect action was found without a preceding protect action.

Program 8

Program 8 is similar to program 7 with the exception that instead of a global ATerm, a globally defined array of ATerms is used. Again, both properties (1a) and (2a) are violated resulting in arbitrary values and a segmentation fault as can be seen in the output below.

```
value of global at index 0 = 352038
value of global at index 1 = 352037
value of global at index 2 = 352036
value of global at index 3 = 352035
value of global at index 4 = 352034
value of global at index 5 = 352033
value of global at index 6 = 352032
value of global at index 7 = 352031
value of global at index 8 = 352030
value of global at index 9 = 352029
Segmentation fault (core dumped)
```

Model checking this model against the safety properties confirms this.

property (1a)	property (1b)	property (2a)	property (2b)
FALSE diagnostic sequence found at depth 2 <initial state> "decl_mylist" "def_mylist" <goal state>	TRUE	TRUE	FALSE diagnostic sequence found at depth 4 <initial state> "decl_mylist" "i" "i" "unprot_mylist" <goal state>

Chapter 8

Results and Conclusions

8.1 Results

The master's project resulted in two products. Firstly, this thesis providing information on the method. Secondly, an implementation of the method described and a set of safety properties that can be used to check models of ANSI-C programs in the subset for improper use of the ATerm library.

The method of transforming ANSI-C programs – using language constructions from a subset of ANSI-C – via the control flow graph, dual graph and abstract graph to the Aldebaran graph described in this thesis is implemented in the ASF+SDF Meta Environment.

Not all ANSI-C language constructions are implemented in the solution. Constructions such as a "switch", "block definitions", "type casts", "pointers" and "recursion" are left as future work. The remaining subset is representative for the ANSI-C language in that it supports repetitions, alternatives, declarations, assignments, function calls and means of breaking the control flow with return and break statements. Given the current implementation and method of subgraph insertion, adding other language constructions such as "switch" and "continue" should not prove difficult. However, pointers and recursion will provide a bigger challenge.

The safety properties of proper use of globally defined ATerms are provided in regular alternation-free μ-calculus and can be used to check the models using the CADP toolkit. For now, a set of these properties is needed for every global ATerm, since the matching is done on the ATerms name. In the future this might be taken care of automatically.

8.2 Conclusions

The research done in this thesis needed to answer the following main question:

> "How can ANSI-C source code (using the ATerm library) be checked for patterns of unwanted behavior using Model Checking techniques."

And the following subquestions:

1. What are the problems with globally defined ATerms in combination with automatic garbage collection?
2. What kind of behavior is unwanted and how can it be recognized?
3. How can the program be represented in such a way, that it captures the flow of the program?
   (a) What parser for ANSI-C should be used?
   (b) What intermediate structures should be used?
4. What information is necessary and what information can be abstracted away?
5. How can we check this representation with model checking techniques to find the cases of unwanted behavior?
   (a) What model checker should be used?
   (b) How should the patterns of unwanted behavior be represented?

(1) Problems with global ATerms

The unwanted behavior of ANSI-C programs using the ATerm library comes from improper use of globally defined ATerms. These ATerms must be explicitly protected and unprotected by the programmer, if not, the automatic garbage collector may remove these ATerms and causes unwanted behavior. The problem with global ATerms is covered in detail in chapter 2.

(2) Kinds of unwanted behavior

Improper use of global ATerms can cause unwanted behavior. Three different behaviors are described in detail in chapter 2.2. These are:

- When a global ATerm is not protected before being used or defined, the automatic garbage collector may remove the ATerm. Referencing the removed ATerm will return an arbitrary value without warning of its removal.
- If a global ATerm is protected but not unprotected, the memory of the ATerm is protected from removal even when that is no longer necessary. This causes a memory leak in the programs.
- If a global ATerm is unprotected without a preceding protect, a segmentation fault will occur which causes the system to crash.

(3) Capturing the flow of the programs

The method described in this thesis provides a way of extracting the "flow" of the program from the source code into an intermediate structure called the control flow graph using language transformations. Building the control flow graph is done by means of transforming every declaration and statement in the program into a node and linking these together with directed edges. Chapter 4 describes this process. The ASF+SDF Meta Environment is used for the parsing the code and the transformations from ANSI-C code to the intermediate datastructures.

Transforming the control flow graph into a dual graph that has the action labels on the edges instead of on the nodes makes it better usable for model checking. Some model checkers (including the CADP evaluator used in this project) use edge-labeled graphs as input rather than node-labeled graphs. Chapter 5 describes the transformation from control flow graph into dual graph.

(4) Abstraction

The dual graph structure can be abstracted. The abstraction process (described in chapter 6) is again a set of language transformations that "abstract" away all information that is not related to global ATerms. All declarations and statements in the dual graph are checked for occurrence of the names of global ATerms and are given a label accordingly. Action labels describe when a global ATerm is declared, defined, used, protected and unprotected. An assignment with a global ATerm on the left-hand side will receive a *def_name* label. All other, non global ATerm related, declarations and statements will receive the label i which stands for internal action. The result of abstraction is a graph that denotes only the "behavior" of global ATerms.

(5) Checking the models

Many model checkers have their own input language. For example the SPIN model checker uses the Promela language. The CADP evaluator which is an on-the-fly model checker, provides the possibility to use a graph as input. A last set of transformations, transform the abstract graph into the Aldebaran graph format that serves as input for the CADP evaluator. The transformations are described in section 7.1.1.

The model of the program (i.e. the Aldebaran graph) is then transformed into a binary coded graph and reduced by applying branching bisimulation reduction, implemented in the CADP toolkit. A set of safety properties in regular alternation-free μ-calculus (section 7.2), together with the reduced model (section 7.2.1) can be checked by the model checker. If any violation of the safety properties is found, an error trace from start node to the violating action is given. This provides insight in where improper use has been made in using ATerms.

Checking the test programs that trigger the unwanted behavior result in error traces. These error traces show the improper behavior. Model Checking

the models of the programs actually finds the unwanted behavior as described in chapter 7.3.

This thesis does not include a case study in which third party source code is checked. The solution and implementation is not yet ready for full size programs. An attempt was made on the constelem.c program in the mCRL2 toolkit. This program uses 3 globally defined ATbool variables and a globally defined ATermTable that are not protected and unprotected. There we a couple of problems that made the current solution unable to check this program.

- Not all language constructions that were used in constelem.c were implemented in the solution. Examples are "continue", "exit" and pointers.
- The ANSI-C syntax definition that comes with the ASF+SDF Meta Environment was unable to parse the constelem.c code.
- Upon removal of the statements that caused parsing to fail, a lot of new ambiguities were introduced. a variable named "constructor" could also be parsed with"const" as specifier, making the variable constant. Also entire functions were considered ambiguous.

Pointer operations and aliasing will prove difficult as well as recursion and will provide a challenge in future work in this area of research.

Main research question

To answer the main research question more directly: By following the method described in this thesis, ANSI-C programs in the subset can be transformed into models. Checking these models against patterns of proper behavior by a model checker, provides error traces that detects improper use of global ATerms.

8.3 Related Work

8.3.1 Model Checking program source code

Copper

Verdaasdonk investigated in [39] how the Copper model checker can be used to verify architectural rules for software systems. The Copper model checker takes an ANSI-C program (normalized to pure ANSI-C if necessary) and a requirement in Linear Time Logic formula as input. The Copper tool has limited support for arrays and pointers and does not support floats or doubles. Considering these restrictions and the fact that a different kind of abstraction technique might be more efficient in checking for use of ATerms, the choice was made to use language transformations for a "custom-made" abstraction."

SPIN

Various attempts at model checking actual source code have been made using the SPIN model checker. Holzmann and Smith describe an extraction and verification method for C source code in [22]. Their method is based on verification

during code development. This means that at any time during the development of a program, a model may be extracted from it and checked by the SPIN model checker. The output is an annotated C trace that can be used for further development. The method in this thesis works with a posteriori verification of the programs.

Another attempt at model checking ANSI-C code was made by Holzmann in [21]. The automaton extraction tool AX is introduced which can abstract and verify models in combination with the SPIN model checker. It relies on a user-defined abstraction and annotations that determine whether certain paths are included in the model, changed or just printed as comment. The AX tool uses tabled abstraction that uses a table and preferences to determine how and what must be abstracted. The SPIN model checker is used to check the produced models against properties in temporal formulas. The model construction however is not fully automated. In some cases 25% to 50% of the model must be handwritten whereas the method described in this thesis produces the models fully automated and require no further annotations to the programs. Holzmann describes the same problems with pointers and aliases that were encountered in this project.

Cattel describes in [13], how a limited subset of sC++, an extension of C++ that adds support for concurrency, can be transformed into Promela. The article describes several semantic equations that have been implemented in prolog. Again the SPIN model checker is used to verify requirements on the Promela model.

Several attempts use Java as input language. DeMartini et. al. use the SPIN model checker in combination with the JCAT tool [16] to check for deadlocks in concurrent systems. The JCAT tool uses a transformation method that transforms Java code into Promela code that can be used by the SPIN model checker. Havelund and Pressburger describe a method of transforming Java code into Promela code using the Java PathFinder system in [19]. The Promela models are then checked by the SPIN model checker and in both cases, interesting results have been obtained.

Bandera

Corbett et.al. describe the Bandera toolkit in [17]. This toolkit can extract finite state models from Java source code. Bandera can transform Java code into a model in a number of model checker formats such as Promela and nuSMV for the SMV model checker and is able to translate the error traces back to source code again.

SLAM and Boolean programs

The SLAM model checker [10], used within Microsoft for checking third-party device drivers, uses boolean programs as models for C programs. SLAM is a tool for checking whether a given C program "obeys API usage rules". Boolean programs can be used as an intermediate datastructure to describe the structure of a program. These structures, described in [9], abstracts all variables and

guards to boolean expressions. This way, by a method of refinement and setting boolean values, a boolean program can be automatically checked for invariant violations and the feasibility of paths.

However, boolean programs concentrate on the question of reachability. This would only be a partial answer to the problem of checking the use of ATerms. Since the order and nature of certain types of actions on globally defined ATerms matter, reachability using boolean programs would not solve the problem. Another remark is that the boolean program abstraction rules out certain paths by setting the boolean guard to true or false. In this project, all possible paths are of interest.

Java Modeling Language

The Java Modelling Language (JML) described in [27], uses "annotation comments" to implement a formal behavioral interface specification language for Java. It uses a method called "design by contract" that creates contracts between classes and functions. These contracts contain pre and post conditions and invariants that can need to hold before the function can be executed. Because the desire was to check ANSI-C source code for the use of ATerms, regardless of whether a loop invariant holds or a precondition holds, a choice was made to transform the entire C program into a model rather than use similar annotations.

8.3.2 Fact extraction

CPP2XMI

The CPP2XMI tool described in [26] is able to reverse engineer UML models from C++ code. CPP2XMI is part of a toolset called SQuADT [6] that is used for analysis of C++ code.

Fact extraction using Rscript

Rscript provides a different approach to software analysis based on relational calculus described in [25]. Rscript is intended for the analysis of programming source code using fact extraction. Fact abstraction can provide usefull information and several program slicing and analysis applications have been made. For example the work of Vankov in [38] on a relational approach to program slicing. However, since the goal of this project is to use a model checking tool to check for unwanted behavior which needs to transform source code into a graph, the ASF+SDF Meta Environment was chosen because of its language transformation abilities and reputation.

8.4 Recommendations and future work

This project raised some interesting questions that may be investigated in the future. Also some optimizations and extentions might prove useful. Consider this non-exhausting list of future research topics and programming tasks:

- Solve ambiguity problems.
- Validate used transformations.
- Implement remaining ANSI-C language constructions.
- Investigate implementation of recursion (possibly by transformation to a process algebra language)
- Investigate implementation of pointers.
- Investigate possible problems that can occur from protecting a local ATerm.
- Investigate model checking other languages.
- Include guard conditions in flow graph constructions in extended finite state machines (eFSM). Perhaps the guard conditions can be used a some sort of heuristic in model checking. Somethings may or may not happen depending on the evaluation of the guard.
- Investigate other behavioral patterns to check for.
- Investigate possible optimizations on the model checking side. (This research focuses more on the language transformation side).
- Investigate possibility to extend the system to use model checking output to correct the errors in the source code.

Bibliography

[1] ASF+SDF meta environment homepage: http://www.asfsdf.org.

[2] CADP case studies: http://www.inrialpes.fr/vasy/cadp/case-studies/.

[3] CADP evaluator manual page: http://www.inrialpes.fr/vasy/cadp/man/evaluator.html.

[4] CADP website: http://www.inrialpes.fr/vasy/cadp/.

[5] SPIN and promela: http://www.spinroot.com.

[6] SQuADT homepage: http://www.laquso.com/research/repository.php.

[7] A.A. Aaby. Compiler construction using flex and bison. Technical report, Walla Walla University, 2003.

[8] F.E. Allen. Control flow analysis. *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[9] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, 2000.

[10] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.

[11] J.A. Bergstra and P. Klint. The discrete time toolbus - a software coordination architecture. *Science of Programming*, 31 (2-3):205–229, 1998.

[12] M. Braveboer, K.T. Kalleberg, and R. Vermaas ans E. Visser. Stratego/XT 0.16: Components for transformation systems. *PEPM06*, pages 95–99, 2006.

[13] T. Cattel. Modelling and verification of sc++ applications. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science*, 1384:232–248, 1998.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[15] H.A. de Jong and P.A. Olivier. *ATerm Library User Manual*. Centrum voor Wiskunde en Informatica.

[16] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software Practice and Experience*, 29(7):577–603, 1999.

[17] J.C. Corbett et.al. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[18] J.F. Groote, A.H.J. Mathijssen, B. Ploeger, M.A. Reniers, M.J. van Weerdenburg, and J. van der Wulp. Process algebra and mCRL2 - IPA basic course on formal methods 2006. Technical report, Technische Universiteit Eindhoven, 2006.

[19] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 1998.

[20] G.J. Holzmann. The model checker SPIN *IEEE Transactions on Software Engineering*. 23(5):279–295, 1997.

[21] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, Lecture Notes in Computer Science*, 1885:131–147, 2000.

[22] G.J. Holzmann and M.H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[23] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, second edition*. Prentice-Hall, 1978.

[24] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[25] P. Klint. A tutorial introduction to RScript. Technical report, Centrum voor Wiskunde en Informatica, 2005.

[26] E. Korshunova, M. Petković, M.G.J. van den Brand, and M.R. Mousavi. CPP2XMI: Reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In *IEEE Proceedings of the 13th working conference on reverse engineering (WCRE'06)*, pages 297–298, 2006.

[27] G. Leavens and Y. Cheon. Design by contract with JML. http://www.jmlspecs.org, 2003.

[28] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46:255–281, 2003.

[29] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[30] J.C. van de Pol. JITty: a rewiter with strategy annotations. *Rewriting Techniques nd Applications: 13th international conference, RTA 2002. Lecture Notes in Computer Science*, 2378:367–370, 2002.

[31] J.C. van de Pol. Algorithms for model checking - lecture notes. technische universiteit eindhoven. 2006.

[32] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[33] M.G.J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[34] M.G.J van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM transactions on Computational Logic*, V(N):1–38, 2004.

[35] M.G.J van den Brand, P. Klint, and J.J. Vinju. The language specification formalism ASF+SDF. Technical report, Centrum voor Wiskunde en Informatica, 2006.

[36] M.G.J van den Brand, P. Klint, and J.J. Vinju. The syntax definition formalism SDF. Technical report, Centrum voor Wiskunde en Informatica, 2006.

[37] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen1, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. *Compiler Construction : 10th International Conference, CC 2001. Lecture Notes in Computer Science*, 2027:365–370, 2001.

[38] I. Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005.

[39] R. Verdaasdonk. Automatic checking of dynamic architectural rules. Master's thesis, Eindhoven University of Technology, 2007.

[40] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

List of Figures

2.1 Code fragment of improper use of global ATerms 10
2.2 Verbose ATerm output illustrates removal 10
2.3 Code fragment of proper use of global ATerms 11
2.4 Verbose ATerm output shows success 11
2.5 Code fragment of improper use of a global ATerm in a struct . . 12
2.6 Code fragment of proper use of a global ATerm in a struct 12
2.7 Code fragment with mismatch in the else branch. 13
2.8 Output shows mismatch and arbitrary value 13
2.9 Summary: Proper use of the ATerm library 14

3.1 SDF definition of a list 17
3.2 ASF equations for the elem and concat functions 18
3.3 Using the elem and concat function 18
3.4 Import graph of the C language in the ASF+SDF Meta Environment 19
3.5 Ambiguity in using parameters 20
3.6 Visualization of parameter ambiguity 20
3.7 Alternative use of parameters 21
3.8 Foo, declaration or statement? 21
3.9 Visualization of ambiguity with parenthesis 21
3.10 Source-to-source transformations of the if-then-else structure . . 24
3.11 Source-to-source transformations of the for structure 25
3.12 Transformations for multiple declarations 25
3.13 Transformations of a struct to global variables 26

4.1 Skeleton Control Flow Graph 28
4.2 Inserting a subgraph into a (skeleton) control flow graph 29
4.3 Declarations syntax 29
4.4 Inserting a Basic Statement 30
4.5 Assignment syntax 30
4.6 While-Do repetition syntax 31
4.7 The While-Do repetition subgraph 31
4.8 Do-While repetition syntax 31
4.9 The Do-While repetition 32
4.10 If-the-else alternative syntax 32
4.11 The If-Then-Else alternative subgraph 33
4.12 Flow and focus change in a function call 34
4.13 Pasting a function as a subgraph 35
4.14 Example of direct recursion 35

4.15 Indirect recursion causes infinite insertion 36
4.16 Example of a call history 37
4.17 Using back edges to handle direct recursive functions 37
4.18 Need for an additional edge to cover possible paths 39
4.19 Return statement in the main function 40
4.20 Break statement in a branch during a repetition 41
4.21 Control flow graph of the nail warehouse 43

5.1 Example of an arbitrary Dual Graph 44
5.2 Building the dual graph illustrated 45
5.3 Example of control flow graph to dual graph transformation . . . 46
5.4 Dual Graph of the nail warehouse 47

6.1 Declaration 49
6.2 Basic declarations 50
6.3 Declarations with initialization 50
6.4 Basic definitions with function call 50
6.5 Definitions and Use 51
6.6 Examination of Expression provides new ATerm 51
6.7 Examples of use 52
6.8 Multiple used ATerms 52
6.9 Protection of ATerm structures 52
6.10 Unprotection of ATerm structures 53
6.11 Dual Graph of the nail warehouse 54
6.12 Abstract Graph of the nail warehouse 55

7.1 Reduced nail warehouse graph 58
7.2 μ-calculus 59
7.3 Trace of violation provided by the evaluator 60
7.4 Violation of safety property (1a) 63
7.5 Visual trace of the violation of property (2a). 67

Appendix A

List of Tools

Operating System:	Xubuntu GNU/Linux 7.04 (Feisty Fawn) kernel: 2.6.20
Extra Software:	Java 1.6 GraphViz 2.8-2.6 Flex(old) 2.5.4a-7 Flex 2.5.33-10build1 Bison 1:2.3.dfsg-4build1 GCC 4:4.1.2-1ubuntu1 Curl 7.15.5-1ubuntu2
ASF+SDF Meta Environment:	asfsdf-meta-2.0pre.20865.28427
ATerm Library:	ATerm 1.4.2
CADP toolset	2006 "Edinburgh" (Stable)

Appendix B

Using the ATerm library

This appendix holds the various test programs that have been used to test the solution presented in this thesis. An overview of the test programs and their use of the ATerm library is given in the table below.

Program	Use of ATerms
Nail Warehouse 1	Global ATerm is defined without protection; Else-branch does not have protect (mismatch).
Nail Warehouse 2	Warehouse 1 fixed: Proper use of the ATerm library
Test program 1	Global ATerm is defined without protection.
Test program 2	Program 1 fiexed: Proper use of ATerm library.
Test program 3	Global ATerm array is defined without protection.
Test program 4	Program 3 fixed: Proper use of ATerm library.
Test program 5	Global ATerm is protected but not unprotected.
Test program 6	Global ATerm array is protected but not unprotected.
Test program 7	Global ATerm is defined without protection; Unprotect without preceding protect (mismatch).
Test program 8	Global ATerm array is defined without protection; Unprotect without preceding protect (mismatch).

B.1 Nail Warehouse 1

```
#include <stdio.h>
#include <aterm2.h>

ATermInt inv;

void sell(int number)
{
        ATermInt s;
        s = ATmakeInt(number);
        ATprintf("Nail number %t sold\n", s);
}

int order(void) {
        return 400000;
}

int main(int argc, char* argv[])
{
        ATerm bottomOfStack;
        int k = 400000;
        int l;

        ATinit(argc, argv, &bottomOfStack);

          inv = ATmakeInt(400025-400000);

        while(k > 0) {
                sell(k);
                k--;
        }

        if(ATgetInt(inv) < 1000) {
                ATprotect(&inv);
                l = order();
                inv = ATmakeInt(order() + ATgetInt(inv));
                ATprintf("New nails ordered; New inventory = %t\n", inv);
        }
        else {
                ATprintf("Nothing ordered; Inventory = %t\n", inv);
        }

        ATprintf("Final Inventory = %t\n", inv);
        ATunprotect(&inv);

        return 0;
}
```

B.2 Nail Warehouse 2

```
#include <stdio.h>
#include <aterm2.h>

ATermInt inv;

void sell(int number)
{
        ATermInt s;
        s = ATmakeInt(number);
        ATprintf("Nail number %t sold\n", s);
}

int order(void) {
        return 400000;
}

int main(int argc, char* argv[])
{
        ATerm bottomOfStack;
        int k = 400000;
        int l;

        ATinit(argc, argv, &bottomOfStack);

        ATprotect(&inv);

        inv = ATmakeInt(400025-400000);

        while(k > 0) {
                sell(k);
                k--;
        }

        if(ATgetInt(inv) < 1000) {
                l = order();
                inv = ATmakeInt(order() + ATgetInt(inv));
                ATprintf("New nails ordered; New inventory = %t\n", inv);
        }
        else {
                ATprintf("Nothing ordered; Inventory = %t\n", inv);
        }

        ATprintf("Final Inventory = %t\n", inv);
        ATunprotect(&inv);

        return 0;
}
```

B.3 Test program 1

```
#include <stdio.h>
#include <aterm2.h>

ATermInt global;

void foo(int counter)
{
        ATermInt s;
        s = ATmakeInt(counter);
        ATprintf("s = %t\n", s);
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;

        int counter = 0;
        int index = 400000;

        ATinit(argc, argv, &bottomOfStack);

        global = ATmakeInt(42);

        while(index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        ATprintf("Value of global ATerm = %t\n", global);

        return 0;
}
```

B.4 Test program 2

```
#include <stdio.h>
#include <aterm2.h>

ATermInt global;

void foo(int counter)
{
        ATermInt s;
        s = ATmakeInt(counter);
        ATprintf("s = %t\n", s);
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;

        int counter = 0;
        int index = 400000;

        ATinit(argc, argv, &bottomOfStack);

        ATprotect(&global);

        global = ATmakeInt(42);

        while(index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        ATprintf("Value of global ATerm = %t\n", global);

        ATunprotect(&global);

        return 0;
}
```

B.5 Test program 3

```
#include <stdio.h>
#include <aterm1.h>


ATerm mylist[10];

ATerm foo(int counter)
{
        ATerm s;
        s = ATmake ("<int>", counter);
        return s;
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;
        int counter = 0;
        int index = 400000;
        int i;

        ATinit(argc, argv, &bottomOfStack);

        for(i = 0; i < 10; i++) {
                mylist[i] = foo(i);
        }

        while (index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        for(i = 0; i < 10; i++) {
                ATprintf("value of global at index %d = %t\n", i, mylist[i]);
        }

        return 0;
}
```

B.6 Test program 4

```
#include <stdio.h>
#include <aterm1.h>

ATerm mylist[10];

ATerm foo(int counter)
{
        ATerm s;
        s = ATmake ("<int>", counter);
        return s;
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;
        int counter = 0;
        int index = 400000;
        int i;

        ATinit(argc, argv, &bottomOfStack);

        ATprotectArray(mylist, 10);

        for(i = 0; i < 10; i++) {
                mylist[i] = foo(i);
        }

        while (index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        for(i = 0; i < 10; i++) {
                ATprintf("value of global at index %d = %t\n", i, mylist[i]);
        }

        ATunprotectArray(mylist);

        return 0;
}
```

B.7 Test program 5

```
#include <stdio.h>
#include <aterm2.h>

ATermInt global;

void foo(int counter)
{
        ATermInt s;
        s = ATmakeInt(counter);
        ATprintf("s = %t\n", s);
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;

        int counter = 0;
        int index = 400000;

        ATinit(argc, argv, &bottomOfStack);

        ATprotect(&global);

        global = ATmakeInt(42);

        while(index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        ATprintf("Value of global ATerm = %t\n", global);

        return 0;
}
```

B.8 Test program 6

```
#include <stdio.h>
#include <aterm1.h>


ATerm mylist[10];

ATerm foo(int counter)
{
        ATerm s;
        s = ATmake ("<int>", counter);
        return s;
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;
        int counter = 0;
        int index = 400000;
        int i;

        ATinit(argc, argv, &bottomOfStack);

        ATprotectArray(mylist, 10);

        for(i = 0; i < 10; i++) {
                mylist[i] = foo(i);
        }

        while (index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        for(i = 0; i < 10; i++) {
                ATprintf("value of global at index %d = %t\n", i, mylist[i]);
        }

        return 0;
}
```

B.9 Test program 7

```
#include <stdio.h>
#include <aterm2.h>

ATermInt global;

void foo(int counter)
{
        ATermInt s;
        s = ATmakeInt(counter);
        ATprintf("s = %t\n", s);
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;

        int counter = 0;
        int index = 400000;

        ATinit(argc, argv, &bottomOfStack);

        global = ATmakeInt(42);

        while(index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        ATprintf("Value of global ATerm = %t\n", global);

        ATunprotect(&global);

        return 0;
}
```

B.10 Test program 8

```
#include <stdio.h>
#include <aterm1.h>


ATerm mylist[10];

ATerm foo(int counter)
{
        ATerm s;
        s = ATmake ("<int>", counter);
        return s;
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack;
        int counter = 0;
        int index = 400000;
        int i;

        ATinit(argc, argv, &bottomOfStack);

        for(i = 0; i < 10; i++) {
                mylist[i] = foo(i);
        }

        while (index > 0) {
                foo(counter);
                counter++;
                index--;
        }

        for(i = 0; i < 10; i++) {
                ATprintf("value of global at index %d = %t\n", i, mylist[i]);
        }

        ATunprotectArray(mylist);

        return 0;
}
```

Appendix C

Normalizing

C.1 Normalizing (SDF)

```
%% Normalization of C source code
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module NormalForm

imports c/Default-C-With-CPP
imports basic/Whitespace
imports basic/Comments

exports

  context-free start-symbols
        TranslationUnit
        Specifier
        Statement
        Expression
        { InitDeclarator ","}+

  context-free syntax
        normalize(TranslationUnit)                  -> TranslationUnit
        elimif(Statement)                           -> Statement
        elimfor(Statement)                          -> Statement+
        normalForm(TranslationUnit)                 ->
                TranslationUnit {traversal(trafo, bottom-up, continue)}
        normalForm(Statement*)                      ->
                Statement* {traversal(trafo, bottom-up, continue)}
        normalForm(Statement)                       ->
                Statement {traversal(trafo, bottom-up, continue)}
        structname(TranslationUnit)                 ->
                TranslationUnit {traversal(trafo, top-down, continue)}
        structname(Expression)                      ->
```

```
        Expression {traversal(trafo, top-down, continue)}
    nfexdecls( ExternalDeclaration+ )           -> ExternalDeclaration+
    nffuncs( FunctionDefinition )               -> FunctionDefinition
    nfstructs( ExternalDeclaration+ )           -> ExternalDeclaration+
    walkDecls(Declaration*)                     -> Declaration*
    walkStats(Statement*)                       -> Statement*
    splitUp( Specifier+, { InitDeclarator ","}+) -> ExternalDeclaration+
    splitUpLocal( Specifier+, { InitDeclarator ","}+) -> Declaration*
    splitUpCase( Expression, Statement) -> Statement
    splitUpStruct(Identifier, StructDeclaration+) ->  ExternalDeclaration+
    splitUpStructLocal(Specifier+, Identifier, {StructDeclarator ","}+)
        -> ExternalDeclaration+

hiddens
  variables
    "&TranslationUnit"[0-9]*          -> TranslationUnit
    "&FunctionDefinition"[0-9]*       -> FunctionDefinition
    "&Statement"[0-9]*                -> Statement
    "&Statement*"[0-9]*               -> Statement*
    "&Expression"[0-9]*               -> Expression
    "&Statement*"[0-9]*               -> Statement*
    "&Statement"[0-9]*                -> Statement
    "&Statement+"[0-9]*               -> Statement+
    "&Declaration"[0-9]*              -> Declaration
    "&Declaration+"[0-9]*             -> Declaration+
    "&Declaration*"[0-9]*             -> Declaration*
    "&Specifier+"[0-9]*               -> Specifier+
    "&Specifier"[0-9]*                -> Specifier
    "&Specifier*"[0-9]*               -> Specifier*
    "&InitDeclarators"[0-9]*          -> { InitDeclarator ","}+
    "&InitDeclarator"[0-9]*           -> InitDeclarator
    "&Declarator"[0-9]*               -> Declarator
    "&Initializer"[0-9]*              -> Initializer
    "&Identifier"[0-9]*               -> Identifier
    "&ExternalDeclaration"[0-9]*      -> ExternalDeclaration
    "&ExternalDeclaration+"[0-9]*     -> ExternalDeclaration+
    "&StructDeclarations"[0-9]*       -> StructDeclaration+
    "&StructDeclaration"[0-9]*        -> StructDeclaration
    "&StructDeclarator"[0-9]*         -> StructDeclarator
    "&StructDeclarators"[0-9]*        -> {StructDeclarator ","}+
```

C.2 Normalizing (ASF)

```
equations

%% Normalization of if-then-else

[elimif-0]
        &Statement*3 := walkStats(&Statement1)
```

```
        ====>
        elimif( if (&Expression) &Statement1 ) =
                if (&Expression) { &Statement*3 } else { skip(); }

[elimif-1]
        &Statement*3 := walkStats(&Statement*)
        ====>
        elimif( if (&Expression) { &Statement* } ) =
                if (&Expression) { &Statement*3 } else { skip(); }

[elimif-2]
        &Statement*3 := walkStats(&Statement1),
        &Statement*4 := walkStats(&Statement2)
        ====>
        elimif( if (&Expression) &Statement1 else &Statement2 ) =
                if (&Expression) { &Statement*3 } else { &Statement*4 }

[elimif-3]
        &Statement*3 := walkStats(&Statement*),
        &Statement*4 := walkStats(&Statement)
        ====>
        elimif( if (&Expression) { &Statement* } else &Statement ) =
                if (&Expression) { &Statement*3 } else { &Statement*4 }

[elimif-4]
        &Statement*3 := walkStats(&Statement),
        &Statement*4 := walkStats(&Statement*)
        ====>
        elimif( if (&Expression) &Statement else { &Statement* } ) =
                if (&Expression) { &Statement*3 } else { &Statement*4 }

[elimif-5]
        &Statement*3 := walkStats(&Statement*1),
        &Statement*4 := walkStats(&Statement*2)
        ====>
        elimif( if (&Expression) { &Statement*1 } else { &Statement*2 } ) =
                if (&Expression) { &Statement*3 } else { &Statement*4 }

[default-elimif]
        elimif(&Statement) = &Statement

%% Top-functions for Normalization of External Declarations

[normalize-0]
        &TranslationUnit1 := normalForm(&TranslationUnit),
        &TranslationUnit2 := structname(&TranslationUnit1)
        ====>
        normalize(&TranslationUnit) = &TranslationUnit2

[normalForm-extdecl-0]
```

```
        &ExternalDeclaration+1 := nfexdecls(&ExternalDeclaration+),
        &ExternalDeclaration+2 := nfstructs(&ExternalDeclaration+1),
        &ExternalDeclaration+3 := nfexdecls(&ExternalDeclaration+2)
        ====>
        normalForm( &ExternalDeclaration+ ) = &ExternalDeclaration+3

%% Normalization of structs. Structs -> Globals and structname transformation

[nfstructs-0]
        &ExternalDeclaration := &ExternalDeclaration+,
        struct &Identifier1 { &StructDeclarations } &Identifier2 ;
                := &ExternalDeclaration,
        &ExternalDeclaration+2 := splitUpStruct(&Identifier2, &StructDeclarations)
        ====>
        nfstructs( &ExternalDeclaration+ ) = &ExternalDeclaration+2

[nfstructs-1]
        &ExternalDeclaration &ExternalDeclaration+2 := &ExternalDeclaration+,
        struct &Identifier1 { &StructDeclarations } &Identifier2 ;
                := &ExternalDeclaration,
        &ExternalDeclaration+3 := splitUpStruct(&Identifier2, &StructDeclarations)
        ====>
        nfstructs( &ExternalDeclaration+ ) = &ExternalDeclaration+3
                nfstructs( &ExternalDeclaration+2 )

[default-nfstructs]
        nfstructs(&ExternalDeclaration+) = &ExternalDeclaration+

[structname-0]
        structname(&Identifier1.&Identifier2) = &Identifier1_&Identifier2

%% Normalization of Declarations

[nfexdecls-0]
        &ExternalDeclaration := &ExternalDeclaration+,
        &Specifier+ &InitDeclarators ; := &ExternalDeclaration,
        &ExternalDeclaration+2 := splitUp(&Specifier+, &InitDeclarators)
        ====>
        nfexdecls( &ExternalDeclaration+ ) = &ExternalDeclaration+2

[nfexdecls-1]
        &ExternalDeclaration &ExternalDeclaration+2 := &ExternalDeclaration+,
        &Specifier+ &InitDeclarators ; := &ExternalDeclaration,
        &ExternalDeclaration+3 := splitUp(&Specifier+, &InitDeclarators)
        ====>
        nfexdecls( &ExternalDeclaration+ ) = &ExternalDeclaration+3
                nfexdecls( &ExternalDeclaration+2 )

[nfexdecls-2]
        &ExternalDeclaration := &ExternalDeclaration+,
```

```
        &FunctionDefinition := &ExternalDeclaration
        ====>
        nfexdecls( &ExternalDeclaration+ ) = nffuncs(&FunctionDefinition)

[nfexdecls-3]
        &ExternalDeclaration &ExternalDeclaration+2 := &ExternalDeclaration+,
        &FunctionDefinition := &ExternalDeclaration
        ====>
        nfexdecls( &ExternalDeclaration+ ) = nffuncs(&FunctionDefinition)
                nfexdecls( &ExternalDeclaration+2 )

[default-nfexdecls]
        nfexdecls(&ExternalDeclaration+) = &ExternalDeclaration+

%% Normalization of Function Definitions

[nffuncs-0]
        &Specifier* &Declarator &Declaration* { &Declaration*2 &Statement* }
                := &FunctionDefinition,
        &Declaration*3 := walkDecls(&Declaration*2),
        &Statement*3 := walkStats(&Statement*),
        &FunctionDefinition2 := &Specifier* &Declarator &Declaration*
                { &Declaration*3 &Statement*3 }
        ====>
        nffuncs(&FunctionDefinition) = &FunctionDefinition2

%% Traverse Declarations and Statements

[walkDecls-1]
        &Declaration &Declaration*3 := &Declaration+,
        &Specifier+ &InitDeclarators ; := &Declaration,
        &Declaration*2 := splitUpLocal(&Specifier+, &InitDeclarators)
        ====>
        walkDecls(&Declaration+) = &Declaration*2 walkDecls(&Declaration*3)

[walkDecls-2]
        walkDecls() =

[walkStats-1]
        &Statement &Statement* := &Statement+,
        &Statement1 := elimif(&Statement),
        &Statement+1 := elimfor(&Statement1)
        ====>
        walkStats(&Statement+) = &Statement+1 walkStats(&Statement*)

[walkStats-2]
        walkStats() =

%% Normalization of for-repetition
```

```
[elimfor-0]
        for ( &Expression1 ; &Expression2 ; &Expression3 ) { &Statement* }
                := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := &Expression1 ; while (&Expression2) { &Statement*1
                &Expression3 ; }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-1]
        for ( ; &Expression2 ; &Expression3 ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := while (&Expression2) { &Statement*1 &Expression3 ; }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-2]
        for ( &Expression1 ; ; &Expression3 ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := &Expression1 ; while (1) { &Statement*1 &Expression3 ; }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-3]
        for ( &Expression1 ; &Expression2 ; ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := &Expression1 ; while (&Expression2) { &Statement*1 }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-4]
        for ( ; ; &Expression3 ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := while (1) { &Statement*1 &Expression3 ; }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-5]
        for ( &Expression1 ; ; ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := &Expression1 ; while (1) { &Statement*1 }
        ====>
        elimfor(&Statement) = &Statement+1

[elimfor-6]
        for (  ; &Expression2 ; ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := while (&Expression2) { &Statement*1 }
        ====>
        elimfor(&Statement) = &Statement+1
```

```
[elimfor-7]
        for ( ; ; ) { &Statement* } := &Statement,
        &Statement*1 := walkStats(&Statement*),
        &Statement+1 := while (1) { &Statement*1 }
        ====>
        elimfor(&Statement) = &Statement+1

[default-elimfor]
        elimfor(&Statement) = &Statement

%% Normalization of multiple declarations (e.g. int i,j,k;)

[splitUp-0]
        splitUp(&Specifier+, &InitDeclarator) = &Specifier+ &InitDeclarator ;

[splitUp-1]
        splitUp(&Specifier+, &InitDeclarator, &InitDeclarators) =
                &Specifier+ &InitDeclarator ; splitUp(&Specifier+,
                &InitDeclarators)

[splitUpLocal-0]
        splitUpLocal(&Specifier+, &InitDeclarator) = &Specifier+
                &InitDeclarator ;

[splitUpLocal-1]
        splitUpLocal(&Specifier+, &InitDeclarator, &InitDeclarators) =
                &Specifier+ &InitDeclarator ; splitUpLocal(&Specifier+,
                &InitDeclarators)

[splitUpStruct-0]
        &Specifier+ &StructDeclarators ; := &StructDeclaration,
        &ExternalDeclaration+ := splitUpStructLocal(&Specifier+,
                &Identifier2, &StructDeclarators)
        ====>
        splitUpStruct(&Identifier2, &StructDeclaration) = &ExternalDeclaration+

[splitUpStruct-1]
        &Specifier+ &StructDeclarators1 ; := &StructDeclaration,
        &ExternalDeclaration+ := splitUpStructLocal(&Specifier+, &Identifier2,
                &StructDeclarators1)
        ====>
        splitUpStruct(&Identifier2, &StructDeclaration &StructDeclarations ) =
                &ExternalDeclaration+ splitUpStruct(&Identifier2,
                &StructDeclarations )

[splitUpStructLocal-0]
        &Identifier1 := &StructDeclarator
        ====>
        splitUpStructLocal(&Specifier+, &Identifier2, &StructDeclarator) =
```

```
                &Specifier+ &Identifier2_&Identifier1 ;

[splitUpStructLocal-0]
        &Identifier1 := &StructDeclarator
        ====>
        splitUpStructLocal(&Specifier+, &Identifier2, &StructDeclarator,
                &StructDeclarators) = &Specifier+ &Identifier2_&Identifier1 ;
                splitUpStructLocal(&Specifier+, &Identifier2, &StructDeclarators)
```

Appendix D

Control Flow Graph

D.1 Triple

```
%% Generic Parameterized Triple
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Triple[X Y Z]

  imports basic/Whitespace
  imports basic/Booleans

exports

  context-free start-symbols
        Triple[[X,Y,Z]]

  sorts Triple[[X,Y,Z]]

  context-free syntax
        "(" X "," Y "," Z ")"                       -> Triple[[X,Y,Z]]

        makeTriple(X, Y, Z)                         -> Triple[[X,Y,Z]]
        fst(Triple[[X,Y,Z]])                        -> X
        snd(Triple[[X,Y,Z]])                        -> Y
        trd(Triple[[X,Y,Z]])                        -> Z
        eqTriple(Triple[[X,Y,Z]],Triple[[X,Y,Z]])   -> Boolean

hiddens
  sorts X Y Z

  variables
        "X"[0-9]*            -> X
        "Y"[0-9]*            -> Y
        "Z"[0-9]*            -> Z
```

```
        "Triple"[0-9]*        -> Triple[[X,Y,Z]]

equations

[makeTriple-1]
        makeTriple(X,Y,Z) = (X,Y,Z)

[fst-1]
        fst((X,Y,Z)) = X

[snd-1]
        snd((X,Y,Z)) = Y

[trd-1]
        trd((X,Y,Z)) = Z

[eqTriple-1]
        eqTriple(Triple, Triple) = true

[default-eqTriple]
        eqTriple(Triple1, Triple2) = false
```

D.2 Tuple

```
%% Generic Parameterized Tuple
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Tuple[X Y]

  imports basic/Whitespace

exports

  context-free start-symbols
        Tuple[[X,Y]]

  sorts
        Tuple[[X,Y]]

  context-free syntax
        "(" X "," Y ")"                         -> Tuple[[X,Y]]

        makeTuple(X, Y)                         -> Tuple[[X,Y]]
        fst(Tuple[[X,Y]])                       -> X
        snd(Tuple[[X,Y]])                       -> Y

hiddens
  sorts X Y
```

```
  variables
        "X"[0-9]*            -> X
        "Y"[0-9]*            -> Y
        "Tuple"[0-9]*        -> Tuple[[X,Y]]

equations

[makeTuple-1]
        makeTuple(X,Y) = (X,Y)

[fst-1]
        fst((X,Y)) = X

[snd-1]
        snd((X,Y)) = Y
```

D.3 DeclStat

```
%% DeclStat - (Declaration/Statement/Expression datatype)
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module DeclStat

  imports basic/Whitespace
  imports c/Declarations
  imports c/Statements
  imports c/Expressions
  imports basic/Booleans

  exports
        sorts DeclStat

        context-free syntax
                Declaration              -> DeclStat
                Declarator               -> DeclStat
                Statement                -> DeclStat
                Expression               -> DeclStat
                "STOP"                   -> DeclStat
                "START"                  -> DeclStat
                "EXIT"                   -> DeclStat
                "SKIP"                   -> DeclStat
                "SEPARATE"               -> DeclStat

                isDecl(DeclStat)         -> Boolean
                isStat(DeclStat)         -> Boolean
                isExpr(DeclStat)         -> Boolean
```

```
hiddens
  context-free start-symbols
        DeclStat Boolean

  variables
        "&Declaration"[0-9]*              -> Declaration
        "&Declarator"[0-9]*               -> Declarator
        "&Statement"[0-9]*                -> Statement
        "&Expression"[0-9]*               -> Expression

equations

[isDecl-1]
        isDecl(&Declaration) = true

[default-isDecl]
        isDecl(&DeclStat) = false

[isStat-1]
        isStat(&Statement) = true

[default-isStat]
        isStat(&DeclStat) = false

[isExpr-1]
        isExpr(&Expression) = true

[default-isExpr]
        isExpr(&DeclStat) = false
```

D.4 Control Flow Graph structure (SDF)

```
%% Control Flow Graph datastructure
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module CFGraph

  imports basic/Whitespace

  imports basic/Integers
  imports basic/Booleans
  imports DeclStat

  %% import Edgeslist

  imports Tuple[Integer Integer]
  imports containers/List[Tuple[[Integer,Integer]]]
```

```
  %% import Nodelist

  imports Tuple[Integer DeclStat]
  imports containers/List[Tuple[[Integer,DeclStat]]]

exports

  %% Set aliases for shorter names
  %% Lists are used instead of sets for easier handling

  aliases
        List[[Tuple[[Integer,DeclStat]]]]        -> NodeList
        List[[Tuple[[Integer,Integer]]]]         -> EdgeList

  context-free start-symbols
        Tuple[[Integer,Integer]]
        Tuple[[Integer,DeclStat]]
        NodeList
        EdgeList
        CFGraph
        Integer
        Boolean

  sorts CFGraph

  context-free syntax

        %% Control Flow Graph structure and constructors

        <NodeList,EdgeList>                            -> CFGraph

        emptyCFGraph()                                 -> CFGraph
        initialCFGraph()                               -> CFGraph
        makeCFGraph(NodeList,EdgeList)                 -> CFGraph

        %% Adding a Node or Edge to the CFG

        addNodeToCFGraph(Integer, DeclStat, CFGraph)   -> CFGraph
        addEdgeToCFGraph(Integer, Integer, CFGraph)    -> CFGraph

        %% Auxiliary functions

        getNodes(CFGraph)                              -> NodeList
        getEdges(CFGraph)                              -> EdgeList

        isEmptyCFGraph(CFGraph)                        -> Boolean

        mergeCFGraphs(CFGraph, CFGraph)                -> CFGraph

        getLastNodeId(CFGraph)                         -> Integer
```

```
        loopNodes(NodeList)                          -> Tuple[[Integer,DeclStat]]

        loopEdges(EdgeList)                          -> Tuple[[Integer,Integer]]

        findNode(NodeList, Integer)                  -> Tuple[[Integer,DeclStat]]
hiddens

  context-free syntax

        %% Internal functions

        addNode(Integer, DeclStat, NodeList)         -> NodeList
        addEdge(Integer, Integer, EdgeList)          -> EdgeList

  variables
        "&Integer"[0-9]*                             -> Integer
        "&DeclStat"[0-9]*                            -> DeclStat
        "&Boolean"[0-9]*                             -> Boolean

        "&Edge"[0-9]*                                -> Tuple[[Integer,Integer]]
        "&Node"[0-9]*                                -> Tuple[[Integer,DeclStat]]

        "&NodeList"[0-9]*                            -> NodeList
        "&EdgeList"[0-9]*                            -> EdgeList
        "&CFGraph"[0-9]*                             -> CFGraph
```

D.5 Control Flow Graph structure equations (ASF)

```
equations

%% Add functions for nodes and edges

[addNode-1]
        &Node := makeTuple(&Integer, &DeclStat)
        ====>
        addNode(&Integer, &DeclStat, &NodeList) =
                concat([&Node], &NodeList)

[addEdge-1]
        &Edge1 := makeTuple(&Integer1, &Integer2)
        ====>
        addEdge(&Integer1, &Integer2, &EdgeList) =
                concat([&Edge1], &EdgeList)

%% Get functions for nodes and edges

[getNodes-1]
        getNodes(<&NodeList,&EdgeList>) = &NodeList

[getEdges-1]
        getEdges(<&NodeList,&EdgeList>) = &EdgeList

%% Constructor (empty and initial)

[emptyCFGraph-1]
        emptyCFGraph() = <[],[]>

[initialCFGraph-1]
        initialCFGraph() = <[(1,STOP) , (0,START)],[(0,1)]>

%% Make actual graph with <>

[makeCFGraph-1]
        makeCFGraph(&NodeList,&EdgeList) = <&NodeList,&EdgeList>

%% Add a node and edge to the graph

[addNodeToCFGraph-1]
        &NodeList := getNodes(&CFGraph),
        &EdgeList := getEdges(&CFGraph),
        &NodeList1 := addNode(&Integer, &DeclStat, &NodeList)
        ====>
        addNodeToCFGraph(&Integer, &DeclStat, &CFGraph) =
                makeCFGraph(&NodeList1,&EdgeList)

[addEdgeToCFGraph-1]
```

```
        &NodeList := getNodes(&CFGraph),
        &EdgeList := getEdges(&CFGraph),
        &EdgeList1 := addEdge(&Integer1, &Integer2, &EdgeList)
        ====>
        addEdgeToCFGraph(&Integer1, &Integer2, &CFGraph) =
                makeCFGraph(&NodeList,&EdgeList1)

%% Merge two CFGraphs

[mergeCFGraphs-1]
        &NodeList3 := concat(getNodes(&CFGraph1), getNodes(&CFGraph2)),
        &EdgeList3 := concat(getEdges(&CFGraph1), getEdges(&CFGraph2))
        ====>
        mergeCFGraphs(&CFGraph1, &CFGraph2) =
                makeCFGraph(&NodeList3, &EdgeList3)

%% Get highest node number (e.g. from loopbody) to continue numbering

[getLastNodeId-1]
        &NodeList := getNodes(&CFGraph),
        &Node := loopNodes(&NodeList)
        ====>
        getLastNodeId(&CFGraph) = fst(&Node)

%% Traverse the nodes and edges

[loopNodes-1]
        empty(tail(&NodeList)) == true,
        &Node := head(&NodeList)
        ====>
        loopNodes(&NodeList) = &Node

[default-loopNodes]
        empty(tail(&NodeList)) == false
        ====>
        loopNodes(&NodeList) = loopNodes(tail(&NodeList))

[loopEdges-1]
        empty(tail(&EdgeList)) == true,
        &Edge := head(&EdgeList)
        ====>
        loopEdges(&EdgeList) = &Edge

[default-loopEdges]
        empty(tail(&EdgeList)) == false
        ====>
        loopEdges(&EdgeList) = loopEdges(tail(&EdgeList))

%% Find a certain node
```

```
[findNode-1]
        &Node := head(&NodeList),
        fst(&Node) == &Integer
        ====>
        findNode(&NodeList, &Integer) = &Node

[findNode-2]
        &Node := head(&NodeList),
        &Integer1 := fst(&Node),
        &Integer1 != &Integer,
        &Integer1 != -1
        ====>
        findNode(&NodeList, &Integer) = findNode(tail(&NodeList), &Integer)
```

D.6 Building the control flow graph (SDF)

```
%% Module for transformation form Normalized C to control flow graph
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module BuildCFG

imports CFGraph
imports DeclStat
imports NormalForm

imports basic/Integers
imports basic/Whitespace
imports basic/Booleans

%% import for graph-integer tuple. Integer holds node numbering hint
imports Triple[Integer Integer Integer]
imports Tuple[CFGraph Triple[[Integer,Integer,Integer]]]
imports Tuple[Identifier Integer]

%% imports for listifications
imports containers/List[FunctionDefinition]
imports containers/List[DeclStat]
imports containers/List[Identifier]
imports containers/List[Tuple[[Identifier,Integer]]]


exports

  aliases
        Triple[[Integer,Integer,Integer]]                  -> Info
        Tuple[[CFGraph,Triple[[Integer,Integer,Integer]]]] -> CFGtuple

  context-free start-symbols
        TranslationUnit
        CFGraph
        FunctionDefinition
        List[[DeclStat]]
        List[[FunctionDefinition]]
        List[[Identifier]]
        List[[Tuple[[Identifier,Integer]]]]
        Info
        CFGtuple
        Boolean
        Statement
        Declaration
        Identifier

  context-free syntax
```

```
%% Main function and Constructor

parse(TranslationUnit)                              -> CFGraph

emptyCFGtuple(Integer)                              -> CFGtuple

%% Listify the Function Definitions

getFunc(TranslationUnit, List[[FunctionDefinition]]) ->
        List[[FunctionDefinition]] {traversal(accu, top-down, break)}

getFunc(FunctionDefinition, List[[FunctionDefinition]]) ->
        List[[FunctionDefinition]] {traversal(accu, top-down, break)}

%% Listify the Global Declarations

getGlobals(TranslationUnit, List[[DeclStat]])       ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

getGlobals(ExternalDeclaration, List[[DeclStat]])   ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

%% Listify function names and get function name

getFunctionName(FunctionDefinition) -> Identifier

funcNameListify(List[[FunctionDefinition]], List[[Identifier]]) ->
        List[[Identifier]]

%% Get the Main function out of all function declared

getMainFunction(List[[FunctionDefinition]])        -> FunctionDefinition

returnFunc(List[[FunctionDefinition]], Identifier) -> FunctionDefinition

%% Create Lists of Declarations and Statements for easy handling

declListify(FunctionDefinition, List[[DeclStat]])  ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

declListify(Declaration*, List[[DeclStat]])        ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

declListify(Declaration, List[[DeclStat]])         ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

statListify(FunctionDefinition, List[[DeclStat]])  ->
        List[[DeclStat]] {traversal(accu, top-down, break)}
```

```
statListify(Statement*, List[[DeclStat]])          ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

statListify(Statement, List[[DeclStat]])           ->
        List[[DeclStat]] {traversal(accu, top-down, break)}

%% Auxiliary functions for disecting Graph tuple

getCFGraph(CFGtuple)                               -> CFGraph

getInfo(CFGtuple)                                  -> Info

%% Constructor

emptyCFGtuple()                                    -> CFGtuple

%% Makes a subgraph from a DeclStat with arguments:
%% DeclStat, Hint for node#, FunctionList, List funcnames,
%% current func + hist, current func start, break dest.

makeBlock(DeclStat, Integer, List[[FunctionDefinition]],
        List[[Identifier]], List[[Tuple[[Identifier,Integer]]]],
        Integer)                                   -> CFGtuple

%% Language construction checks

isBreak(DeclStat)                                  -> Boolean

checkReturn(DeclStat)                              -> Boolean

checkRecursion(DeclStat, Identifier, List[[Identifier]])-> Boolean

checkCall(Identifier, List[[Tuple[[Identifier,Integer]]]]) -> Integer

%% If cuntion is called, add call to call queue

addCall(Identifier, Integer, List[[Tuple[[Identifier,Integer]]]])
        -> List[[Tuple[[Identifier,Integer]]]]

%% Remove old edge when insering a subgraph and subgraph insertion

removeOldEdge(CFGraph)                               -> CFGraph

insertBlock(CFGtuple, CFGtuple, Integer, Integer)    -> CFGtuple

%% Main function for building the CFGraph

buildCFGraph(List[[DeclStat]], List[[FunctionDefinition]],
        List[[Identifier]])                                -> CFGtuple
```

```
        %% ConstructCFG builds a CFG using a list of DeclStats
        %% (function body) with arguments:
        %% Begin, End, Statement* tuple, Functionlist, function names,
        %%current function, current function start, breakdest.

        constructCFG(Integer, Integer, List[[DeclStat]], CFGtuple,
                List[[FunctionDefinition]], List[[Identifier]],
                List[[Tuple[[Identifier,Integer]]]], Integer)   -> CFGtuple


hiddens
  variables
        "&TranslationUnit"                -> TranslationUnit
        "&FunctionDefinition"[0-9]*       -> FunctionDefinition
        "&FunctionList"                   -> List[[FunctionDefinition]]
        "&CFGraph"[0-9]*                  -> CFGraph
        "&Specifier*"[0-9]*               -> Specifier*
        "&Declarator"[0-9]*               -> Declarator
        "&Declaration*"[0-9]*             -> Declaration*
        "&Declaration"[0-9]*              -> Declaration
        "&DeclStat"[0-9]*                 -> DeclStat
        "&DSList"[0-9]*                   -> List[[DeclStat]]
        "&IdList"[0-9]*                   -> List[[Identifier]]
        "&Statement*"[0-9]*               -> Statement*
        "&Statement"[0-9]*                -> Statement
        "&Loopbody"[0-9]*                 -> Statement*
        "&InitDeclarators"[0-9]*          -> { InitDeclarator ","}+
        "&CFGtuple"[0-9]*                 -> CFGtuple
        "&Info"[0-9]*                     -> Info
        "&Hint"[0-9]*                     -> Integer
        "&Integer"[0-9]*                  -> Integer
        "&Boolean"[0-9]*                  -> Boolean
        "&NodeList"[0-9]*                 -> NodeList
        "&EdgeList"[0-9]*                 -> EdgeList
        "&Expression"[0-9]*               -> Expression
        "&Expression*"[0-9]*              -> {Expression ","}*
        "&Specifier+"[0-9]*               -> Specifier+
        "&Specifier*"[0-9]*               -> Specifier*
        "&Identifier"[0-9]*               -> Identifier
        "&Edge"[0-9]*                     -> Tuple[[Integer,Integer]]
        "&Begin"[0-9]*                    -> Integer
        "&End"[0-9]*                      -> Integer
        "&Guard"[0-9]*                    -> Expression
        "&ExternalDeclaration"[0-9]*      -> ExternalDeclaration
        "&Parameters?"[0-9]*              -> Parameters?
        "&Parameters"[0-9]*               -> Parameters
        "&Calls"[0-9]*                    -> List[[Tuple[[Identifier,Integer]]]]
        "&Call"[0-9]*                     -> Tuple[[Identifier,Integer]]
```

D.7 Building the control flow graph (ASF)

```
equations

%% Main function

[parse-1]
        &DSList := concat(getGlobals(&TranslationUnit, []), [SEPARATE]),
        &FunctionList := getFunc(&TranslationUnit, []),
        &IdList := funcNameListify(&FunctionList, []),
        &FunctionDefinition := getMainFunction(&FunctionList),
        &DSList1 := declListify(&FunctionDefinition, []),
        &DSList2 := statListify(&FunctionDefinition, &DSList1),
        %% Include global ATerm declarations
        &DSList3 := concat(&DSList, &DSList2),
        &CFGtuple := buildCFGraph(&DSList3, &FunctionList, &IdList)
        ====>
        parse(&TranslationUnit) = getCFGraph(&CFGtuple)

%% getFunc - Listify all Function Definitions in source code

[getFunc-1]
        getFunc(&FunctionDefinition, &FunctionList) =
                concat(&FunctionList, [&FunctionDefinition])

%% getGlobals - Listify all ATerms that are globally
%% defined (not in a function definition)

[getGlobals-1]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATerm &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-2]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermInt &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-3]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATbool &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-4]
```

```
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermReal &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-5]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermAppl &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-6]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermList &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-7]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermBlob &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-8]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermPlaceholder &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-9]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermTable &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-10]
        &Declaration := &ExternalDeclaration,
        &Specifier* ATermIndexedSet &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[getGlobals-11]
        &Declaration := &ExternalDeclaration,
```

```
        &Specifier* AFun &InitDeclarators ; := &Declaration
        ====>
        getGlobals(&ExternalDeclaration, &DSList) =
                concat(&DSList, [&Declaration])

[default-getGlobals]
        getGlobals(&Declaration, &DSList) = &DSList

%% getMainFunction - get the Main function out of all functions declared

[getMainFunction-1]
        &FunctionDefinition := head(&FunctionList),
        getFunctionName(&FunctionDefinition) == main
        ====>
        getMainFunction(&FunctionList) = &FunctionDefinition

[default-getMainFunction]
        getMainFunction(&FunctionList) = getMainFunction(tail(&FunctionList))

%% getFunctionName - get name of given function

[getFunctionName-1]
        &Specifier* &Declarator &Declaration*1
                { &Declaration*2 &Statement* } := &FunctionDefinition,
        &Declarator2 ( &Parameters ) := &Declarator,
        &Identifier := &Declarator2
        ====>
        getFunctionName(&FunctionDefinition) = &Identifier

%% returnFunc - Returns the function definition matching the Identifier

[returnFunc-1]
        &FunctionDefinition := head(&FunctionList),
        &Identifier1 := getFunctionName(&FunctionDefinition),
        &Identifier1 == &Identifier
        ====>
        returnFunc(&FunctionList, &Identifier) = &FunctionDefinition

[default-returnFunc]
        returnFunc(&FunctionList, &Identifier) =
                returnFunc(tail(&FunctionList), &Identifier)


%% funcListify - Listify the functionnames in the file for matching

[funcNameListify-1]
        empty(&FunctionList) == true
        ====>
        funcNameListify(&FunctionList, &IdList) = &IdList
```

```
[default-funcNameListify]
        &Identifier := getFunctionName(head(&FunctionList)),
        &IdList2 := cons(&Identifier, &IdList)
        ====>
        funcNameListify(&FunctionList, &IdList) =
                funcNameListify(tail(&FunctionList), &IdList2)

%% declListify - Listify all Declarations

[declListify-1]
        declListify(&Declaration, &DSList) =
                concat(&DSList, [&Declaration])

%% statListify - Listify all Statements

[statListify-1]
        statListify(&Statement, &DSList) =
                concat(&DSList, [&Statement])

%% Start the build process with initial graph

[buildCFGraph-1]
        &CFGraph := initialCFGraph(),
        &Info := makeTriple(0, 1, 1),
        &CFGtuple := makeTuple(&CFGraph, &Info),
        &Calls := addCall(main, 0, [])
        ====>
        buildCFGraph(&DSList, &FunctionList, &IdList) =
                constructCFG(0, 1, &DSList, &CFGtuple, &FunctionList,
                &IdList, &Calls, 0)

%% Get functions

[getCFGraph-1]
        getCFGraph(&CFGtuple) = fst(&CFGtuple)

[getInfo-1]
        getInfo(&CFGtuple) = snd(&CFGtuple)

%% Constructor for empty graph

[emptyCFGtuple-1]
        emptyCFGtuple(&Hint) = makeTuple(emptyCFGraph(), (0,&Hint,0))

%% Construct CFG builder with eye for break and return statements

[constructCFG-empty]
        empty(&DSList) == true
        ====>
        constructCFG(&Integer1, &Integer2, &DSList, &CFGtuple,
```

```
                &FunctionList, &IdList, &Calls, &Integer10) = &CFGtuple

[constructCFG-nonempty]
        empty(&DSList) == false,
                %% Construct function arguments
        &DeclStat := head(&DSList),
        checkReturn(&DeclStat) == false,
        isBreak(&DeclStat) == false,
        &Hint := snd(getInfo(&CFGtuple)),
                %% Create a block to insert
        &CFGtuple1 := makeBlock(&DeclStat, &Hint, &FunctionList,
                &IdList, &Calls, &Integer10),
                %% Insert 1st into 2nd
        &CFGtuple2 := insertBlock(&CFGtuple1, &CFGtuple, &Integer1, &Integer2),
        &End := trd(getInfo(&CFGtuple2))
        ====>
        constructCFG(&Integer1, &Integer2, &DSList, &CFGtuple, &FunctionList,
                &IdList, &Calls, &Integer10) = constructCFG(&End, &Integer2,
                tail(&DSList), &CFGtuple2, &FunctionList, &IdList, &Calls, &Integer10)

[constructCFG-break]
        %% Upon break, empty DSlist. Break ends loop
        empty(&DSList) == false,
                %% Construct function arguments
        &DeclStat := head(&DSList),
        checkReturn(&DeclStat) == false,
        isBreak(&DeclStat) == true,
        &Hint := snd(getInfo(&CFGtuple)),
                %% Create a block to insert
        &CFGtuple1 := makeBlock(&DeclStat, &Hint, &FunctionList, &IdList,
                &Calls, &Integer10),
                %% Insert 1st into 2nd
        &CFGtuple2 := insertBlock(&CFGtuple1, &CFGtuple, &Integer1, &Integer10),
        &End := trd(getInfo(&CFGtuple2))
        ====>
        constructCFG(&Integer1, &Integer2, &DSList, &CFGtuple, &FunctionList,
                &IdList, &Calls, &Integer10) = constructCFG(&End, &Integer2, [],
                &CFGtuple2, &FunctionList, &IdList, &Calls, &Integer10)

[constructCFG-return-main]
        empty(&DSList) == false,
                %% Construct function arguments
        &DeclStat := head(&DSList),
        checkReturn(&DeclStat) == true,
        fst(head(&Calls)) == main,
        &Hint := snd(getInfo(&CFGtuple)),
                %% Create a block to insert
        &CFGtuple1 := makeBlock(&DeclStat, &Hint, &FunctionList,
                &IdList, &Calls, &Integer10),
                %% Insert 1st into 2nd
```

```
        &CFGtuple2 := insertBlock(&CFGtuple1, &CFGtuple, &Integer1, 1)
        ====>
        constructCFG(&Integer1, &Integer2, &DSList, &CFGtuple, &FunctionList,
                &IdList, &Calls, &Integer10) = &CFGtuple2

[constructCFG-return-nonmain]
        empty(&DSList) == false,
                %% Construct function arguments
        &DeclStat := head(&DSList),
        checkReturn(&DeclStat) == true,
        fst(head(&Calls)) != main,
        &Hint := snd(getInfo(&CFGtuple)),
                %% Create a block to insert
        &CFGtuple1 := makeBlock(&DeclStat, &Hint, &FunctionList, &IdList,
                &Calls, &Integer10),
                %% Insert 1st into 2nd
        &CFGtuple2 := insertBlock(&CFGtuple1, &CFGtuple, &Integer1, &Integer2)
        ====>
        constructCFG(&Integer1, &Integer2, &DSList, &CFGtuple, &FunctionList,
                &IdList, &Calls, &Integer10) = &CFGtuple2

%% Check is DeclStat is return, break or recursive

[checkReturn-1]
        return &Expression ; := &DeclStat
        ====>
        checkReturn(&DeclStat) = true

[default-checkReturn]
        checkReturn(&DeclStat) = false

[isBreak-1]
        break ; == &DeclStat
        ====>
        isBreak(&DeclStat) = true

[isBreak-2]
        break ; != &DeclStat
        ====>
        isBreak(&DeclStat) = false

[checkRecursion-1]
        &Identifier ( &Expression* ) ; := &DeclStat,
        elem(&Identifier, &IdList) == true
        ====>
        checkRecursion(&DeclStat, &Identifier, &IdList) = true

%% Remove old ege for insertion of a subgraph

[removeOldEdge-1]
```

```
        &NodeList := getNodes(&CFGraph),
        &EdgeList := getEdges(&CFGraph)
        ====>
        removeOldEdge(&CFGraph) = makeCFGraph(&NodeList,tail(&EdgeList))

%% Insert a subgraph into the while CFG

[insertBlock-1]
        &CFGraph1 := getCFGraph(&CFGtuple1),
                %% Info of graph to be added
        &Begin1 := fst(getInfo(&CFGtuple1)),
        &Hint1 := snd(getInfo(&CFGtuple1)),
        &End1 := trd(getInfo(&CFGtuple1)),
                %% Get reveiving Graph
        &CFGraph := removeOldEdge(getCFGraph(&CFGtuple)),
        &CFGraph3 := mergeCFGraphs(&CFGraph1, &CFGraph),
        &CFGraph4 := addEdgeToCFGraph(&Integer1, &Begin1, &CFGraph3),
        &CFGraph5 := addEdgeToCFGraph(&End1, &Integer2, &CFGraph4),
        &Info := makeTriple(&Integer1, &Hint1, &End1),
        &CFGtuple5 := makeTuple(&CFGraph5, &Info)
        ====>
        insertBlock(&CFGtuple1, &CFGtuple, &Integer1, &Integer2) = &CFGtuple5

%% Construct subgraph per language construction

[makeBlock-functioncall-not-in-file]
        &Identifier1 = &Identifier2 ( &Expression* ) ; := &DeclStat,
        elem(&Identifier2, &IdList) == false,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph1, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-functioncall-not-in-file-1]
        &Identifier2 ( &Expression* ) ; := &DeclStat,
        elem(&Identifier2, &IdList) == false,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph1, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-functioncall-recursion]
        &Identifier9 ( &Expression* ) ; := &DeclStat,
        &Integer11 := checkCall(&Identifier9, &Calls),
```

```
        &Integer11 != -1,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &CFGraph2 := addEdgeToCFGraph((&Hint + 1), &Integer11, &CFGraph1),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph2, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-functioncall-recursion-1]
        &Identifier8 = &Identifier9 ( &Expression* ) ; := &DeclStat,
        &Integer11 := checkCall(&Identifier9, &Calls),
        &Integer11 != -1,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &CFGraph2 := addEdgeToCFGraph((&Hint + 1), &Integer11, &CFGraph1),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph2, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-functioncall-paste]
        &Identifier2 ( &Expression* ) ; := &DeclStat,
        elem(&Identifier2, &IdList) == true,
        &Integer11 := checkCall(&Identifier2, &Calls),
        &Integer11 == -1,
        &CFGraph := emptyCFGraph(),
        &Begin := &Hint + 1,
        &End := &Hint + 2,
        &Calls2 := addCall(&Identifier2, &Begin, &Calls),
        &CFGraph1 := addNodeToCFGraph(&Begin, &DeclStat, &CFGraph),
        &CFGraph2 := addNodeToCFGraph(&End, EXIT, &CFGraph1),
        &CFGraph4 := addEdgeToCFGraph(&Begin, &End, &CFGraph2),
        &CFGtuple := makeTuple(&CFGraph4, makeTriple(&Begin, (&Hint + 2), &End)),
        &FunctionDefinition := returnFunc(&FunctionList, &Identifier2),
        &DSList1 := declListify(&FunctionDefinition, []),
        &DSList2 := statListify(&FunctionDefinition, &DSList1),
        &CFGtuple1 := constructCFG(&Begin, &End, &DSList2, &CFGtuple, &FunctionList,
                &IdList, &Calls2, &Begin),
        &CFGraph5 := getCFGraph(&CFGtuple1),
        &Hint1 := snd(getInfo(&CFGtuple1))
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                makeTuple(&CFGraph5, makeTriple(&Begin, &Hint1, &End))

[makeBlock-functioncall-paste-1]
        &Identifier1 = &Identifier2 ( &Expression* ) ; := &DeclStat,
        elem(&Identifier2, &IdList) == true,
```

```
        &Integer11 := checkCall(&Identifier2, &Calls),
        &Integer11 == -1,
        &CFGraph := emptyCFGraph(),
        &Begin := &Hint + 1,
        &End := &Hint + 2,
        &Calls2 := addCall(&Identifier2, &Begin, &Calls),
        &CFGraph1 := addNodeToCFGraph(&Begin, &DeclStat, &CFGraph),
        &CFGraph2 := addNodeToCFGraph(&End, EXIT, &CFGraph1),
        &CFGraph4 := addEdgeToCFGraph(&Begin, &End, &CFGraph2),
        &CFGtuple := makeTuple(&CFGraph4, makeTriple(&Begin, (&Hint + 2), &End)),
        &FunctionDefinition := returnFunc(&FunctionList, &Identifier2),
        &DSList1 := declListify(&FunctionDefinition, []),
        &DSList2 := statListify(&FunctionDefinition, &DSList1),
        &CFGtuple1 := constructCFG(&Begin, &End, &DSList2, &CFGtuple, &FunctionList,
                &IdList, &Calls2, &Begin),
        &CFGraph5 := getCFGraph(&CFGtuple1),
        &Hint1 := snd(getInfo(&CFGtuple1))
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                makeTuple(&CFGraph5, makeTriple(&Begin, &Hint1, &End))

[makeBlock-return]
        return &Expression ; := &DeclStat,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph1, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-break]
        isBreak(&DeclStat) == true,
        &CFGraph := emptyCFGraph(),
        &CFGraph2 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph2, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

[makeBlock-declaration]
        &Declaration := &DeclStat,
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph1, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple
```

```
[makeBlock-while]
        while ( &Guard ) { &Statement* } := &DeclStat,
        &CFGraph := emptyCFGraph(),
        &Begin := &Hint + 1,
        &End := &Hint + 2,
        &CFGraph1 := addNodeToCFGraph(&Begin, &Guard, &CFGraph),
        &CFGraph2 := addNodeToCFGraph(&End, EXIT, &CFGraph1),
        &CFGraph3 := addEdgeToCFGraph(&Begin, &End, &CFGraph2),
        &CFGraph4 := addEdgeToCFGraph(&Begin, &Begin, &CFGraph3),
        &CFGtuple := makeTuple(&CFGraph4, makeTriple(&Begin, (&Hint + 2), &End)),
        &DSList1 := statListify(&Statement*, []),
        &CFGtuple1 := constructCFG(&Begin, &Begin, &DSList1, &CFGtuple,
                &FunctionList, &IdList, &Calls, &End),
        &CFGraph5 := getCFGraph(&CFGtuple1),
        &Hint1 := snd(getInfo(&CFGtuple1))
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                makeTuple(&CFGraph5, makeTriple(&Begin, &Hint1, &End))

[makeBlock-ifthenelse]
        if ( &Guard ) { &Statement*1 } else { &Statement*2 } := &DeclStat,
        &CFGraph := emptyCFGraph(),
        &Begin := &Hint + 1,
        &End := &Hint + 2,
        &CFGraph1 := addNodeToCFGraph(&Begin, &Guard, &CFGraph),
        &CFGraph2 := addNodeToCFGraph(&End, EXIT, &CFGraph1),
        &CFGraph3 := addEdgeToCFGraph(&Begin, &End, &CFGraph2),
        &CFGtuple := makeTuple(&CFGraph3, makeTriple(&Begin, (&Hint + 2), &End)),
        &DSList1 := statListify(&Statement*1, []),
        &DSList2 := statListify(&Statement*2, []),
        &CFGtuple1 := constructCFG(&Begin, &End, &DSList1, &CFGtuple, &FunctionList,
                &IdList, &Calls, &Integer10),
        &CFGraph4 := getCFGraph(&CFGtuple1),
        &Info := getInfo(&CFGtuple1),
        &CFGraph5 := addEdgeToCFGraph(&Begin, &End, &CFGraph4),
        &CFGtuple2 := makeTuple(&CFGraph5, &Info),
        &CFGtuple3 := constructCFG(&Begin, &End, &DSList2, &CFGtuple2, &FunctionList,
                &IdList, &Calls, &Integer10),
        &CFGraph6 := getCFGraph(&CFGtuple3),
        &Hint1 := snd(getInfo(&CFGtuple3))
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                makeTuple(&CFGraph6, makeTriple(&Begin, &Hint1, &End))

[makeBlock-do]
        do { &Statement* } while ( &Guard ) ; := &DeclStat,
        &CFGraph := emptyCFGraph(),
                %% Skipnode
        &Begin := &Hint + 1,
```

```
                %% Skipnode
        &End1 := &Hint + 2,
                %% Guard
        &Begin1 := &Hint + 3,
                %% Exit
        &End := &Hint + 4,
        &CFGraph1 := addNodeToCFGraph(&Begin, SKIP , &CFGraph),
        &CFGraph2 := addNodeToCFGraph(&End1, SKIP, &CFGraph1),
        &CFGraph3 := addNodeToCFGraph(&Begin1, &Guard, &CFGraph2),
        &CFGraph4 := addNodeToCFGraph(&End, EXIT, &CFGraph3),
        &CFGraph5 := addEdgeToCFGraph(&End1, &Begin1, &CFGraph4),
        &CFGraph6 := addEdgeToCFGraph(&Begin1, &Begin, &CFGraph5),
        &CFGraph7 := addEdgeToCFGraph(&Begin1, &End, &CFGraph6),
        &CFGraph8 := addEdgeToCFGraph(&Begin, &End1, &CFGraph7),
        &DSList1 := statListify(&Statement*, []),
        &CFGtuple := makeTuple(&CFGraph8, makeTriple(&Begin, (&Hint + 4), &End)),
        &CFGtuple1 := constructCFG(&Begin, &End1, &DSList1, &CFGtuple,
                &FunctionList, &IdList, &Calls, &End),
        &CFGraph9 := getCFGraph(&CFGtuple1),
        &Hint1 := snd(getInfo(&CFGtuple1))
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                makeTuple(&CFGraph9, makeTriple(&Begin, &Hint1, &End))

[default-makeBlock]
        &CFGraph := emptyCFGraph(),
        &CFGraph1 := addNodeToCFGraph((&Hint + 1), &DeclStat, &CFGraph),
        &Info := makeTriple((&Hint + 1), (&Hint + 1), (&Hint + 1)),
        &CFGtuple := makeTuple(&CFGraph1, &Info)
        ====>
        makeBlock(&DeclStat, &Hint, &FunctionList, &IdList, &Calls, &Integer10) =
                &CFGtuple

%% Add function call to call queue

[addCall-1]
        addCall(&Identifier, &Integer, &Calls) =
                concat([makeTuple(&Identifier, &Integer)], &Calls)

%% Lookup function for call queue

[checkCall-1]
        fst(head(&Calls)) != &Identifier
        ====>
        checkCall(&Identifier, &Calls) = checkCall(&Identifier, tail(&Calls))

[checkCall-2]
        fst(head(&Calls)) == &Identifier
        ====>
        checkCall(&Identifier, &Calls) = snd(head(&Calls))
```

```
[checkCall-3]
        checkCall(&Identifier, []) = -1
```

Appendix E

Dual Graph

E.1 Dual Graph structure (SDF)

```
%% Dual graph structure for program flow
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Dualgraph

  imports basic/Whitespace
  imports basic/Integers
  imports DeclStat

  imports Triple[Integer DeclStat Integer]
  imports containers/List[Triple[[Integer,DeclStat,Integer]]]

exports
  aliases

        List[[Triple[[Integer,DeclStat,Integer]]]]        -> DualGraph
        Triple[[Integer,DeclStat,Integer]]                -> DualEdge

  context-free start-symbols
        Triple[[Integer,DeclStat,Integer]]
        DualGraph
        Integer
        DeclStat
        Boolean

sorts DualGraph

  context-free syntax
        getHighestIndex(DualGraph, Integer)                -> Integer
        addEdge(Integer, DeclStat, Integer, DualGraph)     -> DualGraph
```

```
hiddens
  variables
        "&Integer"[0-9]*              -> Integer
        "&DeclStat"[0-9]*             -> DeclStat
        "&Boolean"[0-9]*              -> Boolean

        "&DualEdge"[0-9]*             -> Triple[[Integer,DeclStat,Integer]]

        "&DualGraph"[0-9]*            -> DualGraph
```

E.2 Dual Graph structure equations (ASF)

```
equations

[addEdge-1]
        &DualEdge := makeTriple(&Integer1, &DeclStat, &Integer2)
        ====>
        addEdge(&Integer1, &DeclStat, &Integer2, &DualGraph) =
                concat(&DualGraph, [&DualEdge])

[getHighestIndex-1]
        empty(&DualGraph) == true
        ====>
        getHighestIndex(&DualGraph, &Integer) = &Integer

[getHighestIndex-2]
        empty(&DualGraph) == false,
        &DualEdge := head(&DualGraph),
        &Integer1 := trd(&DualEdge),
        &Integer2 := max(&Integer, &Integer1)
        ====>
        getHighestIndex(&DualGraph, &Integer) =
                getHighestIndex(tail(&DualGraph), &Integer2)
```

E.3 Building the dual graph (SDF)

```
%% Module for transforming a CFG into a dual graph
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Dual

imports Dualgraph
imports BuildCFG
imports DeclStat

exports

  context-free start-symbols

  context-free syntax

        makeDualGraph(CFGraph)                              -> DualGraph

        translate(Tuple[[Integer,DeclStat]])
                                -> Triple[[Integer,DeclStat,Integer]]

        recurseEdges(EdgeList, NodeList, DualGraph)      -> DualGraph

        reverseEdges(EdgeList, EdgeList)                 -> EdgeList

        reverseNodes(NodeList, NodeList)                 -> NodeList

hiddens
  variables
        "&CFGraph"[0-9]*                                -> CFGraph
        "&Edge"[0-9]*                                   -> Tuple[[Integer,Integer]]
        "&Node"[0-9]*                                   -> Tuple[[Integer,DeclStat]]
        "&NodeList"[0-9]*                               -> NodeList
        "&EdgeList"[0-9]*                               -> EdgeList

        "&DualGraph"[0-9]*                              -> DualGraph
        "&DualEdge"[0-9]*                               -> DualEdge

        "&Integer"[0-9]*                                -> Integer
        "&DeclStat"[0-9]*                               -> DeclStat
        "&Specifier+"[0-9]*                             -> Specifier+
        "&Identifier"[0-9]*                             -> Identifier
```

E.4 Building the dual graph (ADF)

```
equations

%% Reversal function for more intuitive look

[reverseEdges-1]
        reverseEdges([], &EdgeList2) = &EdgeList2

[reverseEdges-2]
        reverseEdges(&EdgeList1, &EdgeList2) = reverseEdges(tail(&EdgeList1),
                cons(head(&EdgeList1), &EdgeList2))

[reverseNodes-1]
        reverseNodes([], &NodeList2) = &NodeList2

[reverseNodes-2]
        reverseNodes(&NodeList1, &NodeList2) = reverseNodes(tail(&NodeList1),
                cons(head(&NodeList1), &NodeList2))

%% Construction of Dual Graph

[makeDualGraph-1]
        &NodeList := reverseNodes(getNodes(&CFGraph),[]),
        &EdgeList := reverseEdges(getEdges(&CFGraph),[])
        ====>
        makeDualGraph(&CFGraph) = recurseEdges(&EdgeList, &NodeList, [])

%% traverse all edges of CFG and transformation into dual graph

[recurseEdges-1]
        empty(&EdgeList) == true
        ====>
        recurseEdges(&EdgeList, &NodeList, &DualGraph) = &DualGraph

[recurseEdges-2]
        empty(&EdgeList) == false,
        &Edge1 := head(&EdgeList),
        &Integer1 := fst(&Edge1),
        &Integer2 := snd(&Edge1),
        &Integer2 != 1,
        &Node := findNode(&NodeList, &Integer2),
        &DeclStat := snd(&Node),
        &DualGraph1 := addEdge(&Integer1, &DeclStat , &Integer2, &DualGraph)
        ====>
        recurseEdges(&EdgeList, &NodeList, &DualGraph) =
                recurseEdges(tail(&EdgeList), &NodeList, &DualGraph1)


[recurseEdges-3]
```

```
        empty(&EdgeList) == false,
        &Edge1 := head(&EdgeList),
        &Integer1 := fst(&Edge1),
        &Integer2 := snd(&Edge1),
        &Integer2 != 1,
        &Node := findNode(&NodeList, &Integer2),
        &DeclStat := snd(&Node),
        &DualGraph1 := addEdge(&Integer1, &DeclStat, &Integer2, &DualGraph)
        ====>
        recurseEdges(&EdgeList, &NodeList, &DualGraph) =
                recurseEdges(tail(&EdgeList), &NodeList, &DualGraph1)


[deafult-recurseEdges]
        empty(&EdgeList) == false,
        &Edge1 := head(&EdgeList),
        &Integer1 := fst(&Edge1),
        &Integer2 := snd(&Edge1),
        &Integer2 == 1
        ====>
        recurseEdges(&EdgeList, &NodeList, &DualGraph) =
                recurseEdges(tail(&EdgeList), &NodeList, &DualGraph)
```

Appendix F

Abstract Graph

F.1 Label

```
%% Label type for abstraction labels
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Label

  imports basic/Strings
  imports languages/c/syntax/Identifiers

exports
  sorts Label

        context-free start-symbols
                Label

        context-free syntax
                "use_" Identifier         -> Label
                "decl_" Identifier        -> Label
                "def_" Identifier         -> Label
                "prot_" Identifier        -> Label
                "unprot_" Identifier      -> Label
                "i"                       -> Label

hiddens
  variables
        "&Label"[0-9]*                    -> Label
```

F.2 Abstract Graph structure (SDF)

```
%% Module for Abstract graph datastructure
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)
```

```
module AbstractGraph

  imports basic/Whitespace
  imports basic/Integers
  imports Label

  imports Triple[Integer Label Integer]
  imports containers/List[Triple[[Integer,Label,Integer]]]

exports
  aliases

        List[[Triple[[Integer,Label,Integer]]]] -> AbstGraph
        Triple[[Integer,Label,Integer]]         -> AbstEdge

  context-free start-symbols
        Triple[[Integer,Label,Integer]]
        AbstGraph
        Label
        Integer
        Boolean

sorts AbstGraph

  context-free syntax

        addEdge(Integer, Label, Integer, AbstGraph)   -> AbstGraph
        makeAbstEdge(Integer, Label, Integer)         -> AbstEdge
        mergeAbstGraphs(AbstGraph, AbstGraph)         -> AbstGraph
        emptyAbstGraph()                              -> AbstGraph
        getIndex(AbstGraph, Integer)                  -> Integer

hiddens
  variables
        "&Integer"[0-9]*               -> Integer
        "&Label"[0-9]*                 -> Label
        "&AbstEdge"[0-9]*              -> Triple[[Integer,Label,Integer]]
        "&AbstGraph"[0-9]*             -> AbstGraph
```

F.3 Abstract Graph structure equations (ASF)

```
equations

[addEdge-1]
        &AbstEdge := makeTriple(&Integer1, &Label, &Integer2)
        ====>
        addEdge(&Integer1, &Label, &Integer2, &AbstGraph) =
                concat(&AbstGraph, [&AbstEdge])

[makeAbstEdge-1]
        makeAbstEdge(&Integer1, &Label, &Integer2) =
                makeTriple(&Integer1, &Label, &Integer2)

[mergeAbstGraphs-1]
        mergeAbstGraphs(&AbstGraph1, &AbstGraph2) =
                concat(&AbstGraph1, &AbstGraph2)

[emptyAbstGraph-1]
        emptyAbstGraph() = []

[getIndex-1]
        empty(&AbstGraph) == true
        ====>
        getIndex(&AbstGraph, &Integer) = &Integer

[getIndex-2]
        empty(&AbstGraph) == false,
        &AbstEdge := head(&AbstGraph),
        &Integer1 := trd(&AbstEdge)
        ====>
        getIndex(&AbstGraph, &Integer) =
                getIndex(tail(&AbstGraph), max(&Integer, &Integer1))
```

F.4 Building the abstract graph (SDF)

```
module Abstraction

imports Dual
imports AbstractGraph
imports containers/List[Identifier]

exports
  context-free start-symbols
        List[[Identifier]]
        AbstGraph
        AbstEdge
        Label
        Boolean

  context-free syntax

        %% Collect all defined ATerms

        collectATerms(DualGraph, List[[Identifier]]) ->
                List[[Identifier]] {traversal(accu, top-down, break)}
        collectATerms(DualEdge, List[[Identifier]]) ->
                List[[Identifier]] {traversal(accu, top-down, break)}
        collectATerms(DeclStat, List[[Identifier]]) ->
                List[[Identifier]] {traversal(accu, top-down, break)}

        %% Purge list: remove local ATerms

        purgeATermList(List[[Identifier]], List[[Identifier]])
                -> List[[Identifier]]

        %% Main function + Auxilliary functions for transformations

        transform(DualGraph)                    -> AbstGraph

        transformGraph(DualGraph, AbstGraph, List[[Identifier]], Integer)
                                                -> AbstGraph

        transformEdge(DualEdge, List[[Identifier]], Integer)
                                                -> AbstGraph

        %% Check functions for protect/unprotect and assignment

        isProtect(Expression)                   -> Boolean
        isAssign(Expression)                    -> Boolean
        isFcall(Expression)                     -> Boolean

        %% Functiopns fow unfolding Expressions to find hidden ATerms uses
```

```
        useATerms(List[[Identifier]], Integer, Integer, Identifier, Integer,
                List[[Identifier]], AbstGraph) -> AbstGraph
        useATerms2(List[[Identifier]], Integer, Integer, Integer,
                List[[Identifier]], AbstGraph) -> AbstGraph

        unfoldExpression(Expression, List[[Identifier]], List[[Identifier]])
                -> List[[Identifier]] {traversal(accu, top-down, break)}
        unfoldExpression(Identifier, List[[Identifier]], List[[Identifier]])
                -> List[[Identifier]] {traversal(accu, top-down, break)}

        %% Construct appropriate label

        constructLabel(List[[Identifier]], List[[Identifier]]) -> Label

        labelDefs(List[[Identifier]], Label)                -> Label
        labelUses(List[[Identifier]], Label)                -> Label

hiddens
  variables
        "&AbstGraph"[0-9]*                                  -> AbstGraph
        "&AbstEdge"[0-9]*                                   -> AbstEdge
        "&ATerms"[0-9]*                                     -> List[[Identifier]]
        "&IdList"[0-9]*                                     -> List[[Identifier]]
        "&Specifier+"[0-9]*                                 -> Specifier+
        "&Specifier*"[0-9]*                                 -> Specifier*
        "&Identifier"[0-9]*                                 -> Identifier
        "&DeclStat"[0-9]*                                   -> DeclStat
        "&DualGraph"[0-9]*                                  -> DualGraph
        "&DualEdge"[0-9]*                                   -> DualEdge
        "&Integer"[0-9]*                                    -> Integer
        "&Index"[0-9]*                                      -> Integer
        "&Label"[0-9]*                                      -> Label
        "&Expression"[0-9]*                                 -> Expression
        "&Expression*"[0-9]*                                -> {Expression ","}*
```

F.5 Building the abstract graph (ASF)

```
equations

%% ------------- ATerm level 1 interface declarations -------------%%
[collectATerms-aterm-1]
        &Specifier* ATerm &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-aterm-2]
        &Specifier* ATerm &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-aterm-3]
        &Specifier* ATerm &Identifier[ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-atbool-1]
        &Specifier* ATbool &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-atbool-2]
        &Specifier* ATbool &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

%% ------------ ATerm level 2 interface declarations -------------------%%

[collectATerms-int-1]
        &Specifier* ATermInt &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-int-2]
        &Specifier* ATermInt &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-int-3]
        &Specifier* ATermInt &Identifier[ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-real-1]
        &Specifier* ATermReal &Identifier ; := &DeclStat
        ====>
```

```
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-real-2]
        &Specifier* ATermReal &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-real-3]
        &Specifier* ATermReal &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-appl-1]
        &Specifier* ATermAppl &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-appl-2]
        &Specifier* ATermAppl &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-appl-3]
        &Specifier* ATermAppl &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-afun-1]
        &Specifier* AFun &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-afun-2]
        &Specifier* AFun &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-afun-3]
        &Specifier* AFun &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-list-1]
        &Specifier* ATermList &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-list-2]
        &Specifier* ATermList &Identifier = &Expression ; := &DeclStat
        ====>
```

```
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-list-3]
        &Specifier* ATermList &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-placeholder-1]
        &Specifier* ATermPlaceholder &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-placeholder-2]
        &Specifier* ATermPlaceholder &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-placeholder-3]
        &Specifier* ATermPlaceholder &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-blob-1]
        &Specifier* ATermBlob &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-blob-2]
        &Specifier* ATermBlob &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-blob-1]
        &Specifier* ATermBlob &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-table-1]
        &Specifier* ATermTable &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-table-2]
        &Specifier* ATermTable &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-table-3]
        &Specifier* ATermTable &Identifier [ &Expression ] ; := &DeclStat
        ====>
```

```
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-indexedset-1]
        &Specifier* ATermIndexedSet &Identifier ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-indexedset-2]
        &Specifier* ATermIndexedSet &Identifier = &Expression ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-indexedset-3]
        &Specifier* ATermIndexedSet &Identifier [ &Expression ] ; := &DeclStat
        ====>
        collectATerms(&DeclStat, &ATerms) = concat(&ATerms, [&Identifier])

[collectATerms-separator-0]
        collectATerms(SEPARATE, &ATerms) = concat(&ATerms, [separator])

[purgeATermList-0]
        &Identifier := head(&ATerms),
        &Identifier == separator
        ====>
        purgeATermList(&ATerms, &ATerms2) = &ATerms2

[purgeATermList-0]
        &Identifier := head(&ATerms),
        &Identifier != separator
        ====>
        purgeATermList(&ATerms, &ATerms2) = purgeATermList(tail(&ATerms),
                concat(&ATerms2, [&Identifier]))

%% -------------------- Transformation init ----------------------------%%

[transform-1]
        &ATerms := collectATerms(&DualGraph, []),
        &ATerms2 := purgeATermList(&ATerms, []),
        &AbstGraph1 := emptyAbstGraph(),
        &Index := getHighestIndex(&DualGraph, 0),
        &AbstGraph := transformGraph(&DualGraph, &AbstGraph1, &ATerms2, &Index)
        ====>
        transform(&DualGraph) = &AbstGraph

[transformGraph-1]
        empty(&DualGraph) == true
        ====>
        transformGraph(&DualGraph, &AbstGraph, &ATerms, &Index) = &AbstGraph

[transformGraph-2]
```

```
        empty(&DualGraph) == false,
        &DualEdge := head(&DualGraph),
        &AbstGraph1 := transformEdge(&DualEdge, &ATerms, &Index),
        &Index1 := getIndex(&AbstGraph1, 0),
        &Index2 := max(&Index, &Index1),
        &AbstGraph2 := mergeAbstGraphs(&AbstGraph, &AbstGraph1)
        ====>
        transformGraph(&DualGraph, &AbstGraph, &ATerms, &Index) =
                transformGraph(tail(&DualGraph), &AbstGraph2, &ATerms, &Index2)

%%------------------------ Edge handling ---------------------------------%%

%%------------------------ Declarations ----------------------------------%%
[transformEdge-decl]
        &DeclStat := snd(&DualEdge),
        &Specifier+ &Identifier ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, decl_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-array]
        &DeclStat := snd(&DualEdge),
        &Specifier+ &Identifier [ &Expression ]; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, decl_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-declinit]
        &DeclStat := snd(&DualEdge),
        &Specifier+ &Identifier = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &AbstGraph1 := addEdge(&Integer1, decl_&Identifier, &Index1,
                emptyAbstGraph()),
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &AbstGraph2 := useATerms(&IdList1, &Index1, &Index1, &Identifier,
                &Integer2, &ATerms, &AbstGraph1)
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2
```

```
[transformEdge-arrayinit]
        &DeclStat := snd(&DualEdge),
        &Specifier+ &Identifier [ &Expression1 ] = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &AbstGraph1 := addEdge(&Integer1, decl_&Identifier, &Index1,
                emptyAbstGraph()),
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &IdList2 := unfoldExpression(&Expression1, &IdList1, &ATerms),
        &AbstGraph2 := useATerms(&IdList1, &Index1, &Index1, &Identifier,
                &Integer2, &ATerms, &AbstGraph1)
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

%%------------------------- Assignments -----------------------------%%

[transformEdge-3]
        &DeclStat := snd(&DualEdge),
        &Identifier = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &AbstGraph2 := useATerms(&IdList1, &Integer1, &Index1, &Identifier,
                &Integer2, &ATerms, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

[transformEdge-array-9]
        &DeclStat := snd(&DualEdge),
        &Identifier [ &Expression1 ] = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &IdList2 := unfoldExpression(&Expression1, &IdList1, &ATerms),
        &AbstGraph2 := useATerms(&IdList2, &Integer1, &Index1, &Identifier,
                &Integer2, &ATerms, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

[transformEdge-8]
        &DeclStat := snd(&DualEdge),
        &Identifier = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == false,
        &Integer1 := fst(&DualEdge),
```

```
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &AbstGraph2 := useATerms2(&IdList1, &Integer1, &Index1, &Integer2,
                &ATerms, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

[transformEdge-array-10]
        &DeclStat := snd(&DualEdge),
        &Identifier [ &Expression1 ] = &Expression ; := &DeclStat,
        elem(&Identifier, &ATerms) == false,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &IdList2 := unfoldExpression(&Expression1, &IdList1, &ATerms),
        &AbstGraph2 := useATerms2(&IdList2,&Integer1, &Index1, &Integer2,
                &ATerms, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

%%------------------ Function call void + misc ---------------------%%

[transformEdge-4]
        &DeclStat := snd(&DualEdge),
        &Expression ; := &DeclStat,
        isAssign(&Expression) == false,
        isProtect(&Expression) == false,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &AbstGraph2 := useATerms2(&IdList1, &Integer1, &Index1, &Integer2,
                &ATerms, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

[transformEdge-7]
        &DeclStat := snd(&DualEdge),
        &Expression := &DeclStat,
        isAssign(&Expression) == false,
        isProtect(&Expression) == false,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &Index1 := &Index + 1,
        &IdList1 := unfoldExpression(&Expression, [], &ATerms),
        &AbstGraph2 := useATerms2(&IdList1, &Integer1, &Index1,
                &Integer2, &ATerms,emptyAbstGraph())
        ====>
```

```
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph2

%%----------------- Protect and unprotect --------------------------%%

[transformEdge-prot]
        &DeclStat := snd(&DualEdge),
        ATprotect ( & &Identifier ) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, prot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-unprot]
        &DeclStat := snd(&DualEdge),
        ATunprotect ( & &Identifier ) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, unprot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-prot-afun-1]
        &DeclStat := snd(&DualEdge),
        ATprotectAFun ( &Identifier ) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, prot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-unprot-afun-2]
        &DeclStat := snd(&DualEdge),
        ATunprotectAFun ( &Identifier ) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, unprot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-prot-array-1]
        &DeclStat := snd(&DualEdge),
```

```
        ATprotectArray ( &Identifier , &Expression1) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, prot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

[transformEdge-unprot-array-2]
        &DeclStat := snd(&DualEdge),
        ATunprotectArray ( &Identifier ) ; := &DeclStat,
        elem(&Identifier, &ATerms) == true,
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, unprot_&Identifier, &Integer2,
                emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

%%-------------------------- Default case, no ATerms -----------------%%

[default-transformEdge]
        &DeclStat := snd(&DualEdge),
        &Integer1 := fst(&DualEdge),
        &Integer2 := trd(&DualEdge),
        &AbstGraph1 := addEdge(&Integer1, i, &Integer2, emptyAbstGraph())
        ====>
        transformEdge(&DualEdge, &ATerms, &Index) = &AbstGraph1

%%------- Get the use of zero or more ATerms out of an Expressoion ------%%

[useATerms-1]
        empty(&IdList) == true,
        &AbstGraph1 := addEdge(&Integer1, def_&Identifier, &Integer2,
                &AbstGraph)
        ====>
        useATerms(&IdList, &Integer1, &Index, &Identifier, &Integer2,
                &ATerms, &AbstGraph) = &AbstGraph1

[useATerms-2]
        empty(&IdList) == false,
        &Identifier1 := head(&IdList),
        &AbstGraph1 := addEdge(&Integer1, use_&Identifier1,
                &Index, &AbstGraph)
        ====>
        useATerms(&IdList, &Integer1, &Index, &Identifier, &Integer2,
                &ATerms, &AbstGraph) = useATerms(tail(&IdList), &Index,
                ( &Index + 1), &Identifier, &Integer2, &ATerms, &AbstGraph1)
```

```
[useATerms2-1]
        empty(&IdList) == true,
        &AbstGraph1 := addEdge(&Integer1, i, &Integer2, &AbstGraph)
        ====>
        useATerms2(&IdList, &Integer1, &Index, &Integer2, &ATerms, &AbstGraph) =
                &AbstGraph1

[useATerms2-2]
        empty(tail(&IdList)) == false,
        &Identifier1 := head(&IdList),
        &AbstGraph1 := addEdge(&Integer1, use_&Identifier1, &Index, &AbstGraph)
        ====>
        useATerms2(&IdList, &Integer1, &Index, &Integer2, &ATerms, &AbstGraph) =
                useATerms2(tail(&IdList), &Index, ( &Index + 1), &Integer2,
                &ATerms, &AbstGraph1)

[useATerms2-3]
        empty(tail(&IdList)) == true,
        &Identifier1 := head(&IdList),
        &AbstGraph1 := addEdge(&Integer1, use_&Identifier1,
                &Integer2, &AbstGraph)
        ====>
        useATerms2(&IdList, &Integer1, &Index, &Integer2,
                &ATerms, &AbstGraph) = &AbstGraph1

[unfoldExpression-1]
        elem(&Identifier, &ATerms) == true,
        &ATerms2 := concat(&ATerms1, [&Identifier])
        ====>
        unfoldExpression(&Identifier, &ATerms1, &ATerms) = &ATerms2

[isProtect-1]
        ATprotect ( & &Identifier) := &Expression
        ====>
        isProtect(&Expression) = true

[isProtect-2]
        ATunprotect ( & &Identifier) := &Expression
        ====>
        isProtect(&Expression) = true

[isProtect-3]
        ATprotectArray ( &Identifier, &Expression1 ) := &Expression
        ====>
        isProtect(&Expression) = true

[isProtect-4]
        ATunprotectArray ( &Identifier) := &Expression
        ====>
        isProtect(&Expression) = true
```

```
[isProtect-5]
        ATprotectAFun ( &Identifier) := &Expression
        ====>
        isProtect(&Expression) = true

[isProtect-6]
        ATunprotectAFun ( &Identifier) := &Expression
        ====>
        isProtect(&Expression) = true

[default-isProtect]
        isProtect(&Expression) = false

[isAssign-1]
        &Identifier = &Expression2 := &Expression
        ====>
        isAssign(&Expression) = true

[isAssign-2]
        &Identifier [&Expression1 ] = &Expression2 := &Expression
        ====>
        isAssign(&Expression) = true

[default-isAssign]
        isAssign(&Expression) = false

[isFcall-1]
        &Identifier ( &Expression1 ) := &Expression
        ====>
        isFcall(&Expression) = true

[default-isFcall]
        isFcall(&Expression) = false
```

Appendix G

Aldebaran graph

G.1 Quote

```
%% Quote module for creating " symbols
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Quote
  imports basic/Whitespace
  imports basic/Strings

exports

  sorts Quote

  context-free start-symbols Quote

  context-free syntax
      "\""                    -> Quote

      makeQuote()             -> Quote

hiddens
  variables
      "&Quote"[0-9]*      -> Quote

equations

[makeQuote-1]
      makeQuote() = "
```

G.2 AutEdge

```
%% Module for Aldebaran Edge (including Quote)
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module AutEdge
  imports basic/Whitespace
  imports basic/Integers
  imports Label
  imports Quote

exports

  context-free start-symbols
      AutEdge
      Integer
      Label

  sorts AutEdge

  context-free syntax
        "(" Integer "," Quote Label Quote "," Integer ")"
            -> AutEdge

        makeAutEdge(Integer, Label, Integer)      -> AutEdge

        %% Access functions

        fst(AutEdge)                              -> Integer
        snd(AutEdge)                              -> Label
        trd(AutEdge)                              -> Integer

hiddens
  variables
        "&Integer"[0-9]*          -> Integer
        "&Label"[0-9]*            -> Label
        "&AutEdge"[0-9]*          -> AutEdge
        "&Quote"[0-9]*            -> Quote

equations

[makeAutEdge-1]
      makeAutEdge(&Integer1, &Label, &Integer2) =
            (&Integer1, makeQuote()&LabelmakeQuote() ,&Integer2)

[fst-1]
      fst((&Integer1, makeQuote()&LabelmakeQuote() ,&Integer2)) =
            &Integer1
```

```
[snd-1]
      snd((&Integer1, makeQuote()&LabelmakeQuote() ,&Integer2)) =
            &Label

[trd-1]
      trd((&Integer1, makeQuote()&LabelmakeQuote() ,&Integer2)) =
            &Integer2
```

G.3 Aldebaran graph structure (SDF)

```
%% Module for Aldebaran graph datastructure
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module Aldebaran

imports basic/Whitespace
imports basic/Integers
imports Label
imports AutEdge
imports Triple[Integer Integer Integer]
imports AbstractGraph

exports

aliases
      Triple[[Integer,Integer,Integer]]          -> AutHeader

  context-free start-symbols
      AutEdge
      AutEdge*
      AutHeader
      Triple[[Integer,Integer,Integer]]
      AutGraph

  context-free syntax

      "des" AutHeader AutEdge*                  -> AutGraph

      makeHeader(Integer, Integer, Integer)     -> AutHeader

      makeEdge(Integer, Label, Integer)         -> AutEdge

      getAutEdgeList(AutGraph)                  -> AutEdge*

  sorts AutGraph

hiddens
  variables
```

```
      "&AutEdge"[0-9]*            -> AutEdge
      "&AutEdge*"[0-9]*           -> AutEdge*
      "&Label"[0-9]*              -> Label
      "&Integer"[0-9]*            -> Integer
      "&AutHeader"[0-9]*          -> AutHeader
      "&AutGraph"[0-9]*           -> AutGraph
      "&DualGraph"[0-9]*          -> DualGraph
```

G.4 Aldebaran graph structure equations (ASF)

```
equations

[makeAutHeader-1]
      &AutHeader := makeTriple(&Integer1, &Integer2, &Integer3)
      ====>
      makeHeader(&Integer1, &Integer2, &Integer3) = &AutHeader

[makeAutEdge-1]
      &AutEdge := makeAutEdge(&Integer1, &Label, &Integer2)
      ====>
      makeEdge(&Integer1, &Label, &Integer2) = &AutEdge
```

G.5 Building the Aldebaran graph (SDF)

```
%% Module for transformation of the abstract graph
%% into Aldebaran format
%% Model Checking the ATerm Library
%% Joost Gabriels (2007)

module BuildAut

imports Abstraction
imports AbstractGraph
imports Aldebaran

exports
  context-free start-symbols
      AutEdge*
      Triple[[Integer,DeclStat,Integer]]
      AutEdge
      Integer
      AutGraph

  context-free syntax

      %% Build function and conversion function from abstract
      %% edge to Aldebaran Edge
```

```
      buildAut(AbstGraph, AutEdge*)            -> AutEdge*
      convert(Triple[[Integer,Label,Integer]]) -> AutEdge

      %% Get functions for abstract graph

      getNodesNumber(AbstGraph, Integer)       -> Integer
      getEdgesNumber(AbstGraph)                -> Integer

      %%  Main function

      buildAutGraph(AbstGraph)                 -> AutGraph

hiddens
  variables
      "&AutEdge*"[0-9]*                        -> AutEdge*
      "&AutEdge+"[0-9]*                        -> AutEdge+
      "&AutEdge"[0-9]*                         -> AutEdge
      "&Integer"[0-9]*                         -> Integer
      "&Label"[0-9]*                           -> Label
      "&AutGraph"[0-9]*                        -> AutGraph
      "&AutHeader"[0-9]*                       -> AutHeader
      "&AbstGraph"[0-9]*                       -> AbstGraph
      "&AbstEdge"[0-9]*                        -> AbstEdge
```

G.6 Building the Aldebaran graph (ASF)

```
equations

%% Convert edge

[convert-1]
      &Integer1 := fst(&AbstEdge),
      &Label := snd(&AbstEdge),
      &Integer2 := trd(&AbstEdge),
      &AutEdge1 := makeEdge(&Integer1, &Label, &Integer2)
      ====>
      convert(&AbstEdge) = &AutEdge1

%% Build Aldebaran graph (not newline separation in indentation)

[buildAut-1]
      empty(tail(&AbstGraph)) == false,
      &AbstEdge := head(&AbstGraph),
      &AutEdge := convert(&AbstEdge),
      &AutEdge*1 := &AutEdge*
&AutEdge
      ====>
```

```
      buildAut(&AbstGraph, &AutEdge*) =
            buildAut(tail(&AbstGraph),&AutEdge*1)

[buildAut-2]
      empty(tail(&AbstGraph)) == true,
      &AbstEdge := head(&AbstGraph),
      &AutEdge := convert(&AbstEdge),
      &AutEdge*1 := &AutEdge*
&AutEdge
      ====>
      buildAut(&AbstGraph, &AutEdge*) = &AutEdge*1

%% Get node/edge number from abstract graph edge

[getNodesNumber-1]
      empty(&AbstGraph) == true,
      &Integer1 := &Integer + 1
      ====>
      getNodesNumber(&AbstGraph, &Integer) = &Integer1

[getNodesNumber-2]
      empty(&AbstGraph) == false,
      &AbstEdge1 := head(&AbstGraph),
      &Integer1 := trd(&AbstEdge1),
      &Integer2 := max(&Integer, &Integer1)
      ====>
      getNodesNumber(&AbstGraph, &Integer) =
            getNodesNumber(tail(&AbstGraph),&Integer2)

[getEdgesNumber-1]
      getEdgesNumber(&AbstGraph) = length(&AbstGraph)

%% Main function

[buildAutGraph-1]
      &AutEdge* := buildAut(&AbstGraph,),
      &Integer1 := getNodesNumber(&AbstGraph, 0),
      &Integer2 := getEdgesNumber(&AbstGraph),
      &AutHeader := makeHeader(0, &Integer2, &Integer1),
      &AutGraph := des &AutHeader
&AutEdge*
        ====>
        buildAutGraph(&AbstGraph) = &AutGraph
```