Monitor Inlining in ABS

Lei Mou
Abstract

Monitor Inlining in ABS

Lei Mou

Monitor inlining is a program rewriting technique to enforce security policy to an untrusted program. The program is modified so that the code for monitoring and policy enforcement is inserted into appropriate points. The embedded code will monitor the action of the program and alter its behavior in case of policy violation (security), otherwise interfere with it as little as possible (conservativity and transparency).

For sequential and multi threaded Java-like programs, inlining algorithms have been proposed, but for languages with different concurrency model, the problem remains to be solved. This master thesis is done in the context of the HATS (Highly Adaptable and Trustworthy Software) project. A framework for ConSpec policy enforcement for the Abstract Behavioral Specification (ABS) language is devised. ABS language is the central contribution of the HATS project, its concurrency model is based on Concurrent Object Groups, Asynchronous Method Calls and Futures. In the framework, an adapted version of ConSpec is used and implemented. The inliner is implemented in Java and is applied to the HATS case study trading system, and its correctness properties are discussed.
## Contents

1 Introduction .......................................................... 3
   1.1 Problem Statement .............................................. 4
   1.2 Related Works .................................................. 4
   1.3 Overview of the thesis .......................................... 5

2 Background .......................................................... 6
   2.1 The Core ABS Language ........................................ 6
      2.1.1 Structure of an ABS program ............................. 7
      2.1.2 Interface Subtyping ..................................... 8
      2.1.3 Method Invocation and Future Reference ............... 8
      2.1.4 Concurrent Object Group ................................. 9
      2.1.5 Statements Concerning Scheduling ....................... 9
      2.1.6 The ABS Compiler ....................................... 10
   2.2 ConSpec Language ............................................... 10
      2.2.1 ConSpec Syntax ........................................... 11
   2.3 Monitor Inlining ................................................ 11
      2.3.1 Correctness Properties .................................. 12
      2.3.2 Race Free Policies ...................................... 13
      2.3.3 Block and Non-blocking Inliner ......................... 13

3 The ABS Inliner .................................................... 15
   3.1 Overall Structure .............................................. 15
   3.2 Adaption to ConSpec rABS ..................................... 15
      3.2.1 Type Checking ........................................... 17
   3.3 Inlining Scheme ................................................ 18
      3.3.1 AST Accessor ............................................. 19
      3.3.2 Security Relevant Call Identification ................. 20
      3.3.3 Runtime Type Information ............................... 20
      3.3.4 ConSpec rABS Code Snippet Generation ............... 21
      3.3.5 Inlining Code Snippet ................................ 23
      3.3.6 Dealing with Ambiguity ................................ 25
      3.3.7 Example Inlining ....................................... 27

4 Correctness .......................................................... 30
   4.1 Security ........................................................ 30
   4.2 Transparency .................................................. 31
Chapter 1

Introduction

Program monitoring is a powerful technique to enforce security policies to programs that potentially perform malicious actions. The program monitor intercepts each untrusted action to determine whether it is allowed to proceed or not. The task of the monitor is to suppress actions that would lead to a violation of the policy, and otherwise interfere as little as possible.

Monitor inlining is one way to implement a program monitor. A monitor specification is compiled into snippets of code which are then inserted into appropriate places of the target program using program analysis and program rewriting. The functionality of the inserted code is to:

- Monitor security relevant events and provide procedures for deciding which action should be allowed, possibly by maintaining a set of security state variables.
- Take remedial actions in case of policy violation, such as terminating the program or recovering from the problematic action.

Thus the program monitor is contained in the program itself and guarantee that it is secure during runtime. Various dedicated languages are devised and implemented in the previous works in order to specify a security policy, such as PSLang [11], Polymer [4] and ConSpec [3].

The previous works on monitor inlining focused on both sequential and multithread Java-like programs [2,11,24,7,8]. However, for software systems implemented by a language with a concurrent model other than multithreading such as the Abstract Behavior Specification (ABS) language, no attempt has been made.

This thesis project is done in the context of the Highly Adaptable and Trustworthy Software (HATS) project [13], which introduces the application of formal methods to the life cycle of a software system. It aims at building software system with high adaptability and trustworthiness in a large-scale and cost-efficient manner. To formally specify and verify a component-based software system, the ABS language was designed and implemented in the HATS project.

ABS is a class-based object-oriented programming language. It is an executable modeling language which specifies the behavior properties of a software system, such as the concurrency model and the data types, but without mentioning the implementation details. Among all these properties, the concurrency model is of interest to us. The foundation of the ABS concurrency model is the Concurrent Object Group (COG), which is a combination of multithreading model of Java and the generalized concurrency model of Creol [18,9]. A COG
could be considered as a set of collaborating tasks that share the same scheduler. The communication between COGs is via \textit{asynchronous method calls} and \textit{futures}. Within a COG, a method could be called synchronously and asynchronously.

1.1 Problem Statement

Monitor inlining in ABS is not trivial and brings up the following questions. First, what policy specification language is needed in order to specify security policies to ABS programs. Second, methods to correctly inline security policies to the target program at source code level, since the concurrency model of ABS is radically different from the other programming languages. Furthermore, with the presence of COG, how does the local scheduling of a COG affects the inlining and how to deal with it. This master thesis answers these questions by addressing the following tasks:

- An adaptation of ConSpec language has been made in order to specify security policies in the context of ABS programs.
- An inlining scheme for ABS monitor is devised.
- A framework for monitor inlining in ABS with the adapted ConSpec language is implemented.
- The ABS inliner is applied to one of the case studies \textit{trading systems} of the HATS project.

The contributions of this thesis are:

- This is the first attempt to perform monitor inlining in a language with a concurrency model featured by concurrent object groups, asynchronous calls and futures.
- An inlining algorithm is proposed and an example inliner for ABS is implemented, which could be successfully enforcing race-free policies.

1.2 Related Works

Schneider characterized a class of security policies that could be formalized by \textit{security automata} \cite{23}. For specifying such policies, several dedicated specification languages have been devised. Erlingsson implemented monitor inlining for Java with the toolkit PoET/PSLang \cite{11}. PSLang expresses security policies in a small subset of Java code, which could provide powerful expressiveness. However, it is difficult to formally prove the correctness properties of the resulting program monitor by presenting policies in this way. In \cite{7, 8, 2}, a similar language \textit{Contract Specification Language (ConSpec)} \cite{3} is used. The security events specified in ConSpec are restricted to invocations of specific APIs, less expressive than PSLang, but with a simpler and cleaner semantics.

Most of the previous works on monitor inlining is done with policies given as security automata. Monitor inlining in sequential Java bytecode is done in \cite{2} and the correctness has been formally proved. Similar work has been done in .Net programs \cite{24}. For monitor inlining in multithreaded Java bytecode \cite{11, 7, 8}, mutually exclusive lock are introduced to the inlined program, protecting security state variables from concurrent access. In the work
of Dam and Lundblad [7], the transparency of the blocking inliner which locks the security relevant method calls is discussed. In their following work [8], a class of policies called race-free policies are identified and the non-blocking inliner is proved to be secure, transparent and conservative for such policies.

A variation of program monitors is called edit automata [20, 19]. With the same ability of event truncation, it could suppress an unexpected event or insert an additional event to the event stream of an application. An implementation of edit automata using the inlining approach is Polymer by Bauer, Jay and Walker [4]. Polymer supports policy combination by treating policies, suggestions and application events as first-class objects. However, mechanisms provided by the programming language are needed in order to implement event suppression and insertion by monitor inlining, such as exception handling.

1.3 Overview of the thesis

The thesis is structured as follows. Chapter 2 provides a general introduction to the core ABS language, including its syntax, concurrency model and compiler. The ConSpec specification language and correctness properties of the inline are presented, which are essential background knowledge of this thesis. Chapter 3 presents the ABS inliner, for specifying security policy for ABS programs. Chapter 4 discusses the correctness properties of the ABS inliner, security, transparency and conservativity. In Chapter 5, one case study Trading System from the HATS project is presented and Chapter 6 concludes the thesis.
Chapter 2

Background

To give a deeper insight into the problem, this section provides a brief introduction to all the essential background knowledge needed in this thesis. In Section 2.1, the structure and the concurrency model adopted by the core ABS language is explained in detail, then the structure and a general description of the ABS compiler is presented. In the following Section 2.2 a general introduction to ConSpec and to the syntax of ConSpec is provided. The last Section 2.3 provides the correctness properties of an inliner and briefly introduces race-free policies and non-blocking inliners, which are highly related to the inlining scheme in ABS.

2.1 The Core ABS Language

The ABS modeling language is a central contribution of the HATS project [13]. Traditional modeling languages such as UML represent a model at highly abstract level, without the ability to specify some detailed behavioral properties, such as concurrency model, data types, etc. Such models are not rigorous enough for formal verification. On the other hand, there are too many unnecessary details involved in programs implemented by commercial languages such as Java. The cost of formal specification and verification of such programs is too expensive. ABS is designed to fill the gap between these two. By providing abstractions such as functional data types which are not supported by traditional languages, an ABS program could be interpreted in an abstract manner. Besides, ABS also has a formal operational semantics to ensure its unambiguity and rigorousness.

As opposed to the full ABS language, which is designed to provide language constructs to support product line engineering according to the requirement specification in HATS deliverable D5.1 [13]. The core ABS language is a subset of the full ABS language and serves as a basis for this purpose.

The core ABS language is a combination of a functional sub-language and an imperative sub-language. The functional part is used to specify user defined data types with (non-high-order) functions and pattern matching [1]. Data types defined in this way abstracts away the implementation details at the same time facilitates the process of formal verification. The imperative part is an object oriented language, which adopts a Java-like syntax. ABS is class-based in the sense that an ABS program is organized as a set of classes but code reuse via class inheritance is ruled out from the core ABS for simplicity reason. ABS uses a nominal type system which differentiates the role of interfaces and classes. Unlike Java, ABS interfaces are types, while classes are not. A class is typed by the interfaces it implements, not by the
The concurrency model of core ABS generalized the concurrency model of Creol \[18, 9\]. The cornerstone of the concurrency model of core ABS is concurrent object groups (COG), which groups concurrent objects together according to the idea of coboxes \[21, 22\]. Communication between COGs is only via asynchronous calls and futures. Within a COG, a method could be invoked both synchronously and asynchronously. The following sections will briefly introduce the structure of a typical ABS program and elaborate on each concept involved in the concurrency model of the core ABS.

2.1.1 Structure of an ABS program

The syntax of core ABS is shown in Figure 2.1. A typical ABS program $P$ consists of a set of declarations and definitions $\text{Dd} \ F \ \text{In} \ \text{Cl}$ and an optional method body $[B]$ that specifies the initial activity of this module, which like the main method in Java. There are four types of declarations and definitions at the top level of an ABS program, data types $\text{Dd}$, functions $F$, interfaces $\text{In}$ and classes $\text{Cl}$. All of them have global scope. Data types and functions are defined using the functional sub-language while interfaces and classes are defined using the imperative sub-language.

Similar to Java, interface declaration $\text{In}$ declares a number of publicly available methods to the class that implements this interface. A set of method signatures $M_S$ are specified in the body of the interface $I$. Interfaces relations in ABS are hierarchical. An interface could extends zero or more interfaces. It should be noticed that, however, an ABS interface does not support method overloading, which is different from Java.

The body of an ABS class $\text{Cl}$ is composed of a number of field definitions $\overline{Tf}$ and method definitions $\overline{M}$. The name of an ABS class is also a constructor method. The parameters of the
constructor method are a subset of the fields, and they are initialized when the constructor is being called. The field definitions in the class body are ensured to be disjoint from the fields specified in the constructor by the type checker of the ABS compiler. All the other fields could be initialized in an optional explicit block $[B]$, which are dedicated for initializing fields of a class instance. Fields that are not initialized in either place are left undefined. The method definitions are implementations of methods in all the interfaces the class implements.

There are two kinds of expressions in ABS, pure expressions $e_p$ and expressions with side effects $e_e$. Pure expression $e_p$ are expressions with no side-effect, which includes state variables $v$, function expressions $e_f$, and the null expression. Function expressions $e_f$ are expression defined in the functional sub-language. Expressions such as `new [cog]`, constructors $C(e_p)$, asynchronous and synchronous call, and `get` operation have side effects, which are referred to as $e_e$.

2.1.2 Interface Subtyping

The ABS language adopts a nominal type system, which only considers interfaces as type for object references, instead of classes. In addition, subtyping and polymorphism is supported on interfaces. In the following text, the subtype relation is denoted as

$$\vdash T \preceq T'$$

which says "$T$ is a subtype of $T'$" and "$T'$ is supertype of $T$". The interface subtype relation is reflexive, transitive, and anti-symmetric, i.e. the $\preceq$-relation is a partial order on types. The definition of it is shown in Figure 2.2.

$$\begin{align*}
\text{(Sub-Refl)} & \quad \vdash T \preceq T \\
\text{(Sub-Trans)} & \quad \vdash T \preceq T' \quad \vdash T' \preceq T'' \quad \vdash T \preceq T'' \\
\text{interface } I \text{ extends } I^+ \ldots \quad I' \in I^+ & \quad \vdash I \preceq I' \\
\text{class } C(T) \text{ implements } I^+ \ldots \quad I' \in I^+ & \quad \vdash I_C \preceq I' \\
\text{(Sub-Fut)} & \quad \vdash \text{Fut}(T) \preceq \text{Fut}(T') \\
\text{(Sub-ClassDecl)} & \quad \vdash C(T) \preceq C(T')
\end{align*}$$

Figure 2.2: Subtype relation

The relationship of future reference `Fut<T>` is the same as their type parameter. For interfaces and classes, the subtyping relation is given by `extends` and `implements` declarations. An interface $I$ is subtype of interface $I'$ it extends. For a class $C$, $I_C$ is the interface that captures all the methods of $C$, then $I_C$ is subtype of interface $I'$ class $C$ implements.

2.1.3 Method Invocation and Future Reference

In ABS, A method activation is called a task. A method could be called synchronously, or asynchronously. The difference between these two types of method invocations lies in the fact of whether the calling task could proceed or not. If the calling task waits blockingly for the result of the callee task, the method call is synchronous. If the calling task could proceed immediately after the method invocation without waiting for the return of the callee task, the method call is asynchronous.

For asynchronous calls, the calling task continues its executing until the result of the call is claimed using the mechanism provided in ABS. For referring to the return value of an
asynchronous call, a place holder called future is used. The reference to a future is called a future reference. Future is the task identity of an asynchronous call. A variable referring to the future is called future reference. A future reference has the type of \texttt{Fut<T>}, where \( T \) is the actual return type of the asynchronous call. A \texttt{get} operation could be applied to the future reference. The task executing the \texttt{get} operation blocks immediately without releasing the lock of the group. The task will be able to proceed only if the asynchronous call has returned.

2.1.4 Concurrent Object Group

The concurrent object group is a combination of the generalized multithread concurrency model of Java and the generalized concurrency model of Creol.

Inside a COG, a generalized multi-thread concurrency model is used. A concurrent object group is a set of collaborating tasks which shares the same scheduling strategy and message queue. At one time, only one task is active within one COG, another task will be only scheduled after the execution of this task is finished, i.e. the scheduling strategy of a COG is non-preemptive. This mutual exclusion behavior of the tasks is achieved by a lock. A task is only eligible to run after it successfully grab the lock. Furthermore, the scheduling is also non-deterministic, it means that which task is scheduled can neither be foreseen nor controlled by the programmer. Inside a group, synchronous calls and (mutual) recursion, i.e., call-backs are allowed.

The situation of asynchronous calls are completely different. Asynchronous calls are the mechanism of communication between COGs. Different from synchronous calls, call-backs are not allowed for asynchronous calls for the following reasons. A task \( t_a \) in group \( A \) calling a method of an object which is from group \( B \) creates a task \( t_b \) in group \( B \). Although \( t_b \) could “call-back” objects from group \( A \), but in that case, the call is also asynchronous, being asynchronous means that the task created by the “call-back” could not be scheduled immediately, since the lock is acquired by some other task. What’s more, since the identity of a task is only available at run-time and there isn’t any support for identifying the identity of a task at the programming level, so communicating with the other tasks in a desired way is not possible.

When creating a new object, the programmer could either attach it to an existing COG or place it into a newly created COG. In the first case, an object is created using statement \texttt{new}, the concurrency model corresponds to that of Java. In the second case, statement \texttt{new cog} is used instead, the concurrency model corresponds to that of Creol.

2.1.5 Statements Concerning Scheduling

There are two statements that affect the scheduling of a COG, \texttt{suspend} and \texttt{await}. \texttt{await} is the critical statement in handling the inlining of the asynchronous calls.

The task executing \texttt{suspend} statement introduces a scheduling point to the group. The task stops its execution temporarily, frees the lock it grabbed and lets the other tasks have the opportunity to be scheduled. Task execute statement \texttt{await g} will release the lock only if the guard \( g \) is evaluated to true, otherwise its execution will be continued. After being suspended, the task has to recheck the guard \( g \) in order to be scheduled again. The guard is a conjunction of a set of values, each one of them could be evaluated to Boolean. The basic
polling guard of the form \( n? \) needs to be mentioned. Here, \( n \) is a future reference, \( n? \) will be evaluated to true only if the corresponding task has finished its execution.

2.1.6 The ABS Compiler

The architecture of the ABS compiler could be illustrated by Figure 2.3 from the HATS Deliverable 1.1a [1]. The compiler consists of two parts, the frontend and the backend. The frontend is responsible for parsing the ABS source code, checking the ABS program for syntax errors and semantic errors. The output of the compiler frontend is an abstract syntax tree (AST), which is the internal representation of ABS models and the cornerstone of the monitor inlining.

Compared to traditional compiler backend, the backend of the ABS compiler is more like a code generator, only target code is generated and no optimization is performed.

The lexer and parser is implemented by JFlex [17] and Beaver [5]. For better integration, JastAdd [16] is adopted, which is the basis for AST representation, semantic analysis and code generation. Since the ABS inliner modifies the original program by modifying its AST, these tools are also used in the ABS inliner in order to keep consistency of the data representation between the inliner and the ABS compiler frontend.

2.2 ConSpec Language

ConSpec is a language designed for both specifying security requirements and representing security relevant behaviors of a target application written in Java-like languages. Automata is chosen as its formalism since automata could be applied in both purposes.

Inspired by PSLang [11], ConSpec adopts a similar but more constrained policy specifi-
cation structure, a set of security variables representing *security states*, and a set of *security relevant events* followed by a list of guarded-updates. The intention is to formalize a security automaton by the policy text. The security variables represent its state, and the security relevant events represent its transitions.

To be rigorous, ConSpec also limits the types of variables representing security state and the statements used in the guarded-updates structure. The type of security states must be one of the three: *int*, *string*, and *boolean*. In the guarded-update structure, the guard must be a side effect free expression and loop is not allowed inside the update block.

ConSpec also allows the programmer to specify a scope for the level at which a security policy to be applied. The following list shows each of them:

- **Multisession**: Multiple executions of the same application
- **Global**: Executions of all applications of a system.
- **Session**: Single execution of an application
- **Object**: Life time of an object of a specific class.

This work only considers **Session** as the scope.

### 2.2.1 ConSpec Syntax

Figure 2.4 shows the syntax of ConSpec. At the top of the security policy specification, the scope of the policy is specified, then follows a list of security variable declarations. The type of security variable is restricted to *PrimType* (*int*, *boolean* and *string*). Each of the security variable declarations must have an initial value.

Event clauses are constructs for specifying security relevant events and security updates, their syntax is shown in Figure 2.5. In ConSpec, a security relevant event is a method invocation, it could be a system call or an API from a third-party library. Each event is identified by its signature, which includes the class name to which the method belongs, the method name and the parameter list of the method. The signature of an event must be unique, i.e., events of the same type can not have the same signature. This is for ruling out policy ambiguity. The modifiers **BEFORE**, **AFTER** and **EXCEPTIONAL** specifies when the security state is updated, before, after the method call or at the time when an exception is thrown by the event. In the case of an **AFTER** event, the return value could be specified in the event clause, if the security update depends on the return value.

Each event clause has an associated guarded-updates structure. The guard should be a side-effect free boolean expression which can only mention the declared security variables, the variables in the parameter list and the return value of the **AFTER** event. The guards are evaluated from top to bottom, the update block will be performed if one of the guards evaluates to true. If no guards hold, the policy is violated unless an **ELSE** block is presented. In that case, the update in the **ELSE** block will be performed.

### 2.3 Monitor Inlining

Security policies given in terms of security automata can be enforced by *execution monitors* [23]. Execution monitor is an enforcement mechanism that monitors each execution
steps of a target system, and terminates the execution if it violates the given security policy. Such monitors can be implemented in various systems, such as the kernel of operating systems, firewalls, etc.

Monitor inlining is one way to implement the execution monitor [10], an *inlined reference monitor* (IRM) generated from a security policy is combined with the target program, so the monitor is contained within the program itself. Here *inlining* means the reference monitor is embedded into the program itself by program rewriting. The IRM enforces security policies on the target, interfere with the execution only in case of a policy violation, otherwise the IRM should not add or remove execution steps to the execution.

The traditional correctness properties of the monitor are *security*, *transparency* and *conservativity* [12, 19]. Before these properties are presented, a model of program execution is introduced, it will serve as a basis of the following discussion.

### 2.3.1 Correctness Properties

An execution \( E \) of a program could be considered as transitions from configuration to configuration. Each configuration contains the state of the program, an example is the stack. Each transition correspond to an execution step. In this thesis, the execution steps we are interested in are method calls. A method could be defined in the client program or as an API from a third party library. The *trace* of the execution is defined as all the transitions from the initial configuration to the final configuration (if the execution is finite). In [7], *observable traces* \( \omega(E) \) of an execution \( E \) are defined as invocations from client program to an API, or the returns from an API to the client program. Since all the implementations in ABS are all available, the observable traces \( \omega(E) \) are defined as method invocations except the invocations to permission checking methods, which is different from [7]. All the other traces are considered as non-observable in this thesis.

Having the model of program execution, the following text discuss the correctness properties of the IRM:

- Security means that all the traces of the inlined program are accepted by the security policy.
- Conservativity means that every trace of the inlined program is a trace of the original program.
• Transparency means that all the policy adherent traces of the original program are traces of the inlined program.

However, as shown in [7], for the IRM in a multithreaded setting, these traditional correctness properties are too strict due to the limitations of inlining itself, some policies can not be correctly enforced by a non-blocking inliner. In [7], a class of race free policies are identified and proved that a non-blocking inliner could successfully enforce such kind of policies.

2.3.2 Race Free Policies

From the inlining perspective, a policy is race free means that the policy could be successfully enforced without being affected by the scheduling regardless of the ordering of the call actions. Consider the example which is from [7]. The policy says method \texttt{b} defined in class \texttt{C} should be invoked after method \texttt{a} defined in the same class. However, the policy shown in Figure 2.6 can not be correctly enforced by inlining due to the fact that the ordering of the two method invocations depends on the scheduler. In the execution of the client program, either two observable traces \texttt{a, b, b, a} are both possible, or none of them is possible. The only way to correctly inline this policy is to introduce a lock in the body of method \texttt{a}, method \texttt{b} waits until the lock is released. In this case, the inliner needs to rewrite the API to cooperate with the client program. However, since the code of the API is not always accessible, such as operations for IO etc., the inliner is only restricted to rewrite the program itself.

Another alternative is to let \texttt{b} wait until \texttt{a} is returned. This actually enforce the security policy shown in Figure 2.7. This policy could be successfully enforced at the caller side, without interfering with the scheduler by modifying the body of the monitored API. Thus policy in Figure 2.6 is not race free, the policy in Figure 2.7 is race free.

Race free policies is the class of policies in which specified before events can only happen after the specified after events in the observable traces of the target program. Thus the policy could be enforced by modifying the client program. In addition, [7] also shows that a non-blocking inliner can correctly enforce the class of race free policies.

2.3.3 Block and Non-blocking Inliner

An inliner is considered as blocking if the security relevant call is protected by a global lock before the call is executed, the lock is not released until the method returns. Inliner implemented in this way could ensure the security property, but possibly introduce deadlock to the inlined program. Furthermore, the introduction of global locks may also result in severe
performance penalties, even if deadlock is avoided. A novel inlining algorithm is presented in [7], instead of lock across the entire security relevant call, the global lock is released before calling a security relevant method, inliner implemented in this way are called *non-blocking* inliners.
Chapter 3

The ABS Inliner

This chapter shows the solution of monitor inlining for an ABS program using the adapted version of ConSpec (referred as ConSpec_{ABS} in the following text) as the security policy specification language. In Section 3.1, the overall structure of the ABS inliner is presented. The ABS inliner consists of two parts, the ConSpec_{ABS} parser and the ABS inliner. These will be presented in Section 3.2 and Section 3.3 respectively.

3.1 Overall Structure

The ABS inliner is a standalone program which builds on top of the ABS compiler frontend, for the purpose of integrating the existing ABS parser and the ABS type checker. Since the AST is the bridge between the compiler frontend and backend, the modified AST generated by the inliner could be directly transferred to different compiler backends, thus the user can choose from all target languages, such as Java bytecode or Maude [6]. An ABS back-end is developed in this work, for generating the modified ABS source code. The overall structure of the ABS inliner is shown in Figure 3.1.

Before the inlining process starts, the ABS source program is parsed and type-checked by the ABS compiler frontend. The security policy is then parsed into a ConSpec_{ABS} AST by the ConSpec_{ABS} parser. Then the type checking of the ConSpec_{ABS} specification is performed along with the AST of the ABS program. The type checking is essential to the correctness of the inlined program, however, the ConSpec_{ABS} policy contains no information about the entities in the source program, such as interfaces, classes, method signatures. As a result, the ABS AST which provides these type information is an indispensable input for this step. Besides, the type checking is also responsible for getting rid of policy ambiguities. If the policy is type-correct and unambiguous, the inlining algorithm modifies the ABS AST according to the scheme mentioned in Section 3.3. The modified AST is then passed to the compiler back-end for code generation.

3.2 Adaption to ConSpec_{ABS}

ConSpec was originally designed to specify security policy for Java-like programs [3]. For applying ConSpec in the context of ABS, a few small but important changes have been made. The adapted version of ConSpec is from now on referred to as ConSpec_{ABS}.
Some syntactical changes have been made in order to use ABS syntax in the policy specification. The syntax of the expressions in security state declarations, the guards
and statements in the update clauses are changed from Java to ABS. For instance, we have added ABS-style parametric data types.

• The `EXCEPTIONAL` modifier has been removed from ConSpec since ABS lacks support for exception handling.

• Due to ambiguity, in ConSpecABS a security policy may not refer to inherited methods of an interface. This is elaborated in Section 3.3.6.

• The types of security state variables are changed from the original ConSpec Java types to the predefined ABS types: `Bool`, `Int` and `String` similar to [3].

• In addition to the usual arithmetic, relational and boolean operations, the applications of predefined functions in ABS standard library are also allowed in the security policy.

• ConSpecABS does not support any method calls or `get` operation as part of guards or update clauses. The intention is to rule out statements that have side-effect in the policy specification. This is in order to preserve transparency and will be discussed further in Chapter 4.

ConSpecABS parser is implemented using the same tool set as the ABS compiler front-end. Figure 3.2 shows the abstract grammar definition of ConSpecABS in terms of JastAdd, which is the framework of ConSpecABS. The concrete grammar is given in Beaver. The complete definition of ConSpecABS abstract syntax is attached in the appendix.

### 3.2.1 Type Checking

Since the policy specification is used to generate code snippets that will be inserted to the ABS program, the generated code must be correct in order to ensure the correctness of the generated program. The policy specification should be correct in three aspects.

• The entities in the policy specification itself is correct, for example, each security variable declaration is unique, each event signature is unique, etc. This is the basic responsibility of the type checker.

• The policy specification is type-corrected. There are entities from ABS program used in the policy, such as interfaces, classes, data types and functions. The declaration and definition of them is only available in the ABS program. The type checker has to get all the essential type information from the type corrected ABS AST in order to perform this checking.
• The policy is unambiguous. Since the policy specification is the guide to locate security relevant calls in the ABS program, the event specification must be clear and unambiguous. The reason of ambiguities in the policy specification and solutions is elaborated in Section 3.3.6.

3.3 Inlining Scheme

The ABS inliner is a program built on top of the ABS compiler frontend, it accepts a security policy specification given by ConSpec\textsubscript{ABS} and an ABS source program as input, producing an inlined ABS program with a non-blocking inlined reference monitor (IRM) as output.

The first step of the inlining is to generate an interface Monitor for the IRM according to the policy specification. Each method declared in Monitor corresponds to an event clause in the policy. The implementation of these methods are given in class MonitorImpl. The class MonitorImpl implements interface Monitor, in addition to the declared methods, a set of field declarations corresponding to the security state variables are also given in the class. Each defined method reacts according to the value of the security state variable, either by updating the security state, return to the caller if the call is permitted by the monitor, or terminating its execution in case of policy violation. It should be noted that there is no language construct provided in the ABS language for terminating a running ABS program, so a special \textit{exit} statement was added to the language. Interface Monitor and class MonitorImpl are then attached to the AST in the form of separate AST nodes.

Then an instance of the monitor is created and initialized in the beginning of the main method of the ABS program, for the purpose of asking the monitor for permission and updating the security state. The monitor runs in its own COG, in order to respond to requests from all the other COGs. An additional parameter \texttt{Monitor monitor} was added to the constructor of each class, so the instance could be known to all the classes in the source program.

The inliner then proceeds to locate all the possible security relevant calls by traversing the ABS AST. The inlining scheme varies according to the type of the security relevant call. For synchronous security call nodes in the AST, an asynchronous call to the permission checking method is inserted before or after the node in question. A \texttt{get} operation is also used together with the asynchronous call, to wait blockingly for the answer of the monitor. For the asynchronous call node, it is replaced with an asynchronous call node to the corresponding permission checking method in Monitor.

The modified ABS AST is then passed to the compiler backend, generating the desired target code. The ABS backend is used by default to generate modified ABS source code.

This section presents a thorough explanation of the inlining algorithm used in the ABS inliner. First of all, the AST accessor which acts as the bridge between the ABS inliner and the source program is presented in Section 3.3.1. Then the algorithm for identifying all the security relevant calls are presented in Section 3.3.2. To figure out the actual class of an object for correctly enforcing a security policy to a method call, a simple run-time type identification (RTTI) mechanism is implemented, and is presented in Section 3.3.3. The code snippet generation of ConSpec\textsubscript{ABS} is shown in Section 3.3.4. The way of inlining the generated code snippets are provided in Section 3.3.5. The following Section 3.3.6 shows how the ConSpec\textsubscript{ABS} type checker handles ambiguities in the policy specification, and at last, an inlining example is shown in the last Section 3.3.7.
3.3.1 AST Accessor

The AST visitor is the interface between the inliner and the ABS source program. The accessor builds an index of all the entities in the ABS source program by traversing the AST, which serves as basis for AST modification and security relevant call identification in the following steps. Apart from this, the accessor also performs the following tasks:

- Building indexes for all the statements containing `new (cog)`, `return` and synchronous and asynchronous method calls.
- Build the interface hierarchy in the ABS source program.

All the `new (cog)` statements needs to be located since a new parameter `Monitor monitor` is added to all the class constructor, for the reason that the monitor instance in the main block needs to be known to all the class of the ABS source program. As a consequence, an argument `monitor` needs to be inserted to the argument list of all the `new (cog)` statements.

For `return` statement directly returns the value of a method call, rewriting is needed in case that the method call is security relevant and an `AFTER` event with return value is applicable to this call. In this case, the inliner can not access the return value of the `return` statement, an example is given in Figure 3.3. A security policy declares `m` in interface `I` is security relevant, the `AFTER` event clause is applicable to the call `obj.m()`; in the implementation of the method in class `C`. The result is returned directly, which poses a problem to the inliner since the return value needs to be passed to the monitor asking for permission. In this case, such `return` statement is rewritten in the manner shown in Figure 3.4. A variable `ret` of type `T` is explicitly declared with an initial value, this variable is introduced for holding the return value of the method call, then the value is returned via this introduced variable.

The accessor also collects all the entries of synchronous and asynchronous method calls in the AST. This information is used to locate all the possible security relevant calls and make preparation for the inlining phase.

Another essential information for the inliner is the interface hierarchy. It is important to the inliner because it needs to know the relationship between any two interfaces. The accessor gathers this information by traversing the AST. In the implementation, each interface has an
associated HashMap to store all its sub-interfaces. The associated HashMap of all interfaces in
the source program actually forms a subtype matrix. Since the sub-type relation is transitive,
the interface hierarchy could be obtained by computing the transitive closure of this matrix.

3.3.2 Security Relevant Call Identification

It is essential for the inliner to correctly identify all the security relevant calls in order to
enforce the security policy to the ABS program. In ConSpec_{ABS}, an event clause could be
given in terms of interface or class name. In the following text, the first is referred to as
interface event clause and the latter is referred to as class event clause, the corresponding
event is referred to as interface event and class event, respectively.

For each synchronous and asynchronous call, all the applicable event clauses specified in
the policy are collected according to the following rules. Since in ABS there is no method
overloading, it is enough to see if an event clause is applicable to a method call by examining
the method name and the type of the method call.

- For an interface event clause specified by interface I, it is applicable to a call o.m if
  they both refer to the same method m and if the type of o is I or sub-interface of I.
  Since the interface hierarchy is obtained by the AST accessor, the matching of interface
  event clause could be done during compile time.

- For a class event clause specified by interface C, it is possibly applicable to a call if they
  both refer to the same method m. To see if it is applicable, an implemented runtime
type identification introduced in the following Section 3.3.3 is used, so the matching of
class event clause is postponed to runtime.

- In the implementation, HashMap is used to store the list of applicable event clauses
  associated with a security relevant call.

- This HashMap is then used in the inlining stage in Section 3.3.5.

3.3.3 Runtime Type Information

Identifying which class an object belongs to is essential to enforce security policy if a class
event is presented in the specification. However, an object reference is typed by interface in
ABS, and class is not type. Furthermore, the ABS language doesn’t provide any language
constructs for performing this task. This poses a problem for the inliner, since ABS is run in
terms of Java bytecode, at run-time the type available is a Java class, not an ABS interface.

To solve this problem, a simple RTTI mechanism is implemented in the ABS inliner. An
interface RuntimeType shown in Figure 3.5 is introduced to the source program. Since the
ABS program is organized by a set of classes, and the name of each class is unique, so the
class name is used as the identity of each class. The inliner modifies all the interfaces in
the original ABS program to extend the RuntimeType interface. Each class implements the
getRuntimeType() method by returning its name as a string literal. The return value is then
used in an if statement to figure out if the object is an instance of the desired class. If so,
the corresponding permission checking method is called with its body. The idea is shown in
Figure 3.6.
interface RuntimeType {
    String getRuntimeType();
}

String runtimeType = "";
runtimeType = obj.getRuntimeType();
if (runtimeType == "ClassName") {
    // Call to the corresponding
    // permission checking method
}

3.3.4 ConSpec<sub>ABS</sub> Code Snippet Generation

ConSpec<sub>ABS</sub> specification is the basis for generating interface Monitor and class MonitorImpl, and the permission checking method declared and defined within each of them.

Difference between synchronous and asynchronous calls

Due to the different semantics of synchronous and asynchronous method calls, the inlining scheme is different and the code snippet generated is different. Since synchronous method call waits until the result is returned, the generated code snippet could be inserted right before or after the call, thus BEFORE event could be processed immediately before the call is performed, and AFTER event clause could be processed immediately after the call is returned. For asynchronous calls, it is not the case, since the caller can continue its execution without waiting for the return of the call and the task created may possibly scheduled after the call is initiated. So it is possible that the AFTER event clause is processed before the call in question is returned, if using the same inlining scheme as for synchronous calls.

The key issue to solve this problem is to ensure that the AFTER event clause will always be processed after the asynchronous call. For this purpose, a wrapper method is created for the asynchronous case along with the permission checking method of the synchronous case. The wrapper method wraps the original call and waits for the result using the await statement, allowing the other tasks in the monitor COG to be scheduled before the result is returned. BEFORE and AFTER events are handled immediately before the call is initiated and after the call is returned, respectively. During the inlining phase, the asynchronous security relevant call could be replaced directly by the wrapper method. The following example further illustrates this idea.

class MonitorImpl {
    Int Imethod(I o, Int v) {  
        Fut<Int> res;  
        Int val = 0;  
        res = o!method(v);  
        await res?;  
        [AFTER I.m]  
        val = res.get;  
        return val;  
    }
}

...  
I o;  
o = new cog SomeClass();  
Fut<Int> res;  
// A call to security relevant  
// method I.method(Int)  
res = o!method(1);  
...  
Int val = 0;  
val = res.get;  
...

...  
I o;  
o = new cog SomeClass();  
Fut<Int> res;  
// The call is replaced by a  
// wrapper method.  
res = monitor!Imethod(o, 1);  
...  
Int val = 0;  
val = res.get;  
...

Figure 3.5: RuntimeType Interface Figure 3.6: Run-time type identification.

Figure 3.7: Wrapper method. Figure 3.8: Original call. Figure 3.9: Replaced call.
In the source program shown in Figure 3.8, an asynchronous security relevant call `o!method(1);` is performed via object reference `o` with an argument `1`. The inlined program is shown in Figure 3.9 with the original call replaced with the call to the wrapper method. The wrapper method created in `MonitorImpl` is shown in Figure 3.7: `Int Imethod(I o, Int v)`. Apart from the original parameters, the callee object reference is also added in the signature since the actual method call `o!method(v)` is performed within the wrapper method. In the method body, the original method is called with an `await` statement waiting for the result, then the `AFTER` event is processed (if there is one). At last, the result is returned.

**Interface `Monitor`**

Each interface event clause in the policy specification corresponds to two permission checking methods in the interface `Monitor`. The two permission checking methods correspond to both synchronous and asynchronous security relevant calls, respectively. For the synchronous call, the method will be called asynchronously by the client program asking the monitor for permission. For the asynchronous call, the actual method call is replaced with an asynchronous call to the permission checking method.

Each method declaration is generated according to the following rules:

**method name**

For the synchronous case, it is a concatenation of the event type string ("before"/"after"), interface/class name and the method name. For the asynchronous case, the method name is the concatenation of interface name and the method name.

**return type**

For the synchronous case, the return type of the method is `Unit`, for noticing the caller that the permission checking method has returned. For the asynchronous case, the return type is the same as the original method, since the original call will be replaced by the wrapper method.

**signature**

For the synchronous case, the generated signature is the same as the original one for `BEFORE` events and `AFTER` events without return value. An additional parameter corresponding to the return value will be appended to the original signature if the `AFTER` event has a return value. For the asynchronous case, in addition to the aforementioned way, another parameter corresponding to the callee is also needed in the signature, since in the wrapper method, the reference of the callee is needed to perform the original call.

For an interface event clause, the type of the callee is the same as the interface used in the event clause. For a class event clause, all the interfaces that the class implements and all their super interfaces are traversed, in order to find the interface which declares the given method.

**MonitorImpl**

The class `MonitorImpl` implements interface `Monitor`, it consists of two parts, field declarations and method definitions. Each field declaration corresponds to one security state declaration in the `MonitorImpl`.
For the method corresponding to synchronous method calls, the body of each method is generated by converting the associated guarded-updates in the manner shown in Figure 3.10 and Figure 3.11.

Guard1 -> { UpdateBlock1; }
...
GuardM -> { UpdateBlockM; }
[ ELSE -> { ElseBlock; } ]

if (Guard1) {
    UpdateBlock1;
} else if ...
} else if (GuardM) {
    UpdateBlockM;
} else {
    exit(1)/[ElseBlock];
}

Figure 3.10: Guarded Updates

Figure 3.11: Converted ABS code

The guarded-update structure in Figure 3.10 is directly translated into an if...else if...else... statement shown in Figure 3.11 in order to be consistent with ConSpec semantics which says that the guards are evaluated from top to bottom. In the else branch of the generated if statement, the ELSE block is placed inside if it is presented. Otherwise the exit(1) statement is placed inside, corresponding to the case of policy violation.

For methods corresponding to asynchronous method calls, the body of the wrapper method is generated in the manner shown in Figure 3.12 in order to process the associated BEFORE and AFTER clauses immediately before and after the asynchronous security relevant call.

T IMethod(I callee, ...) {
    [BEFORE EVENT CLAUSE]
    Fut<T> res;
    T val = init_value;
    res = callee!method(...);
    await callee?
    [AFTER EVENT CLAUSE]
    val = res.get;
    return val;
}

Figure 3.12: Wrapper Methods Generation

Firstly, the translated guarded-update structure of the corresponding BEFORE event clause, if there is one, is placed within the body of the method. Then follows a future reference res, with the return type T as the type parameter. A variable of type T is also declared for holding the actual return value of the actual method call. In the middle, the actual method is called asynchronously with the callee argument declared in the signature, and the result is waited using the await statement, in order to let the other tasks in the monitor COG to have the opportunity to get scheduled. The corresponding AFTER event clause, if there is one, is processed immediately after the method has returned. At last, the return value is retrieved using the get operation and returned to the calling method.

3.3.5 Inlining Code Snippet

This is the final stage of the inlining. Since all the applicable event clauses of a call is available in Section 3.3.2 all these applicable event clauses are converted to ABS code snippets and
then are ready to be inlined into the program. For a method call, the set of applicable event clauses are then further divided into two sets, interface events and class events. Different inlining schemes are then performed to each set, accordingly.

**Inlining Interface Events**

The number of interface events that are applicable to a security relevant call should not be larger than 2. The following shows each possible case.

- **number = 1**
  - The interface event is either **BEFORE** or **AFTER** event.

- **number = 2**
  - It is legal only if the two events are **BEFORE** and **AFTER** events of the same method in the same interface. Any other cases are due to ambiguities in the policy specification.

- **number > 3**
  - This corresponds to the case of interface multiple inheritance, which is another kind of ambiguity in the policy specification. This topic is elaborated in Section 3.3.6.

If the security relevant call is synchronous, the following inlining scheme is adopted. For **BEFORE** and **AFTER** event with no return value, the asynchronous permission checking method performed by the monitor instance is inserted right before or after the call. For **AFTER** event with a return value, the statement of call may need to be rewritten in order to introduce a variable for holding the return value.

If the security relevant call is asynchronous, the original call is directly replaced with a call to the wrapper method in the monitor.

**Inlining Class Events**

All the class events applicable to the call are further divided into two sets, the **BEFORE** event set and the **AFTER** event set. Each set corresponds to a code snippet for choosing the appropriate permission checking method according to the actual class of the callee object. Since there is no language construct for figuring out the exact class of an object reference, the inlining of class events totally depends on the simple RTTI mechanism we implemented. The inlining scheme differs according to the type of the security relevant call:

**Synchronous**

The inlining scheme of synchronous security relevant call for a set of $M$ class event clauses is shown in Figure 3.13. An **if** statement is introduced for finding the appropriate permission checking method to execute. For **BEFORE** or **AFTER** event set, the code snippet is inserted right before or after the synchronous security relevant call, respectively.

**Asynchronous**

For asynchronous security relevant calls, the same structure is also adopted with the difference that the original call is moved from the original program to the body of the **else** block of the introduced **if** statement. The original call in the client program is replaced with the code snippets shown in Figure 3.14.
String runtimeType = "";
runtimeType = obj.getRuntimeType();

if (runtimeType == "Class1") {
    // Call to the corresponding
    // permission checking method
} else if (runtimeType == "Class2") {
    ...
} else if ...
else if (runtimeType == "ClassM") {
    ...
}

String runtimeType = "";
runtimeType = obj.getRuntimeType();

if (runtimeType == "Class1") {
    // Call to the corresponding
    // wrapper method
} else if (runtimeType == "Class2") {
    ...
} else if ...
else if (runtimeType == "ClassM") {
    ...
} else {
    // Original asynchronous call.
}

Figure 3.13: Code snippet of M class events for synchronous SRC
Figure 3.14: Code snippet of M class events for asynchronous SRC

Asynchronous Call to the Permission Checking Method

An asynchronous call to the monitor is performed before or after each synchronous security relevant method call, via the monitor instance created in the beginning of the program. Then a get operation is introduced to wait blockingly for the return of the asynchronous call, thus the permission for a synchronous security relevant call is examined by the monitor. An example is shown in Figure [3.13]. Since the return value of the permission checking method is Unit, a future reference returned of type Fut<Unit> is declared for holding the return value, then follows the asynchronous call to the method. Here the get operation is introduced for waiting for the monitor permission.

3.3.6 Dealing with Ambiguity

Under certain circumstances the policy may be ambiguous due to the fact that both method declared in an interface or defined in a class could be security relevant. The situation is even complicated if taking into consideration that a class could implement more than one interface. This section shows how these ambiguities are formed, and the solution to each of them.

Ambiguity in Interface Event

Since ABS allows interface inheritance, the policy is problematic if I.m is declared as security relevant, where method m is not declared in interface I, but it is inherited from its super interface. An event specified in this way means that method m in all the super-interfaces of I are not security relevant, while at the same time, method m in all the sub-interfaces of I are security relevant. Then there needs to be a way to identify the type of the object reference, since all the possible security relevant calls are identified by method name.

Due to interface inheritance and an object reference is typed by interface in ABS program, polymorphic behavior is involved and run-time type identification (RTTI) is needed in order to figure out the actual type of an object reference. Unfortunately, there is no RTTI support in ABS, so this can not depend on the ABS language constructs. On the other hand, although we could rely on the RTTI support in the Java backend, for example, instanceof, but that doesn’t works as expected. Because the type we could get during run-time is always a Java
In order to resolve this ambiguity, a restriction is introduced to the policy specification. For an interface event, if method $m$ in interface $I$ is specified as security relevant, method $m$ must be declared in interface $I$, which means method $m$ in all the sub-interfaces of $I$ are all security relevant. This approach is not as flexible as the problematic one, but it is the best way we could use since the language support of ABS is not powerful enough to solve this problem.

**Ambiguity in both Interface and Class Event**

In a policy specification, if an interface event clause and a class event clause are both referring to the same method, then the policy specification is possibly ambiguous. For example, the policy is ambiguous if an interface event $I.m()$ and a class event $C.m()$ both are declared as security relevant, while in the ABS program class $C$ implements interface $I$. In this case, the two event clauses are both applicable to a call $o.m$, if $o$ is of type or subtype of $I$.

This can be resolved by the ConSpec$_{ABS}$ type checker. The type checker collects all the security event clauses with the same method name, and sees if both interface and class events are involved. If so, the type checker continues to check if the aforementioned situation occurs. So this kind of ambiguity can be avoided at compile time.

The situation is even complicated in the case of multiple interface inheritance. Suppose we have a target ABS program that declares two interfaces $I_a$ and $I_b$ which both declare a method $m$. If $I_a$ and $I_b$ have a common sub-interface $I_c$, i.e., interface $I_c$ extends both $I_a$ and $I_b$, the policy specification shown in Figure 3.16 is ambiguous for all calls to $o.m$, if $o$ is of type or subtype of $I_c$.

This ambiguity can not be handled by the ConSpec$_{ABS}$ type checker, since the policy has no information about the hierarchy of interfaces in the target program. So the handling of such ambiguity can only be deferred to the inlining stage, and there are four possible ways to handle such ambiguity:

1. Report the ambiguity to the programmer as an error and abort inlining.

Since there are multiple event clauses applicable to a single call, multiple updates to the security state will be performed. The security state will be corrupted, and the inlined
program is not correct. By showing the ambiguity, the programmer will know exactly why there is ambiguity and modify the policy accordingly.

2. Require that G1 and G2 be satisfied at the same time and perform U1 and U2 in some predefined order.
   Since an interface only declared a method, a call to I_c.m can be considered as a call to either I_a.m or I_b.m as well. For the security reason, both event clauses should be applied, however, this is not consistent with the semantics of ConSpec. Besides, this solution is not right if the two update blocks U1 and U2 have dependency on the execution order of each other.

3. Require either G1 or G2 be satisfied. Perform the updates of the one which is satisfied.
   This solution is the worst because it allows the programmer to find a way around the monitor. Besides, the order of evaluation of G1 and G2 also affects the inlining.

4. Introduce a modifier to ConSpec_ABS and explicitly specify which one should apply in this case.
   This solution is also bad since the introduced modifier will bring unnecessary construct to ConSpec_ABS and resulting ugly security policy specification, although it could solve this problem as well.

The first solution is the most reasonable one, since the inliner shouldn’t decide which one should apply if ambiguity occurs. Besides, the first option is the one that could avoid introducing erroneous behavior to the inlined program, thus it is adopted in the implementation.

### 3.3.7 Example Inlining

Here we show an example of the inlining algorithm. The target program is shown in Figure 3.17. In the target program, the security relevant method `testClass.method()` executes repeatedly within the loop, which violates the security policy specified in Figure 3.18. The policy says that the method could not be executed more than three times.

```
module Count;
  interface Relevant {
    Unit method();
  }
  class TestClass implements Relevant {
    Unit method() {}
  }
{ Relevant testClass; testClass = new TestClass(); while (True) {
    testClass.method();
  }
}
```

```
SCOPE Session
SECURITY STATE
    Int count = 0;
    BEFORE Relevant.method() PERFORM
      count < 3 -> {
        count = count + 1;
      }
```

Figure 3.17: A sample ABS program

Figure 3.18: A ConSpec_ABS policy
After compiling, the interface Monitor and MonitorImpl are generated from the above policy specification, which is shown in Figure 3.19. The resulting program after inlining is shown in Figure 3.20.

The permission checking method beforeRelevantMethod() declared in interface Monitor is generated from the BEFORE event clause in the policy. The body is translated directly from the guarded-update structure in this event clause in the manner presented in Section 3.3.4. If the security policy is violated, i.e., the execution of the method has exceeded three times, the program will be terminated by exit(1) statement.

In the inlined program, an instance monitor is created at the beginning of the optional block, running in its own COG. The original interface Relevant is modified to extend the interface RuntimeType, so as to provide RTTI support in class TestClass. In addition, the monitor instance is appended to the signature of the TestClass constructor. In the loop, the inliner correctly identifies the security relevant call, and the code for calling the permission checking method is then inserted before the call.

```plaintext
module ABSMonitor;
export *;
import * from ABS.StdLib;

interface Monitor {
    Unit beforeRelevantMethod();
}

class MonitorImpl implements Monitor {
    Int count = 0;
    Unit beforeRelevantMethod() {
        if (count < 3) {
            count = count + 1;
        } else {
            exit(1);
        }
        return Unit;
    }
}
```

Figure 3.19: The monitor interface and implementation.
module Count;

import * from ABS.StdLib;
import * from ABSMonitor;
import * from ABSRTTI;

interface Relevant extends RuntimeType {
    Unit method();
}

class TestClass(Monitor monitor) implements Relevant {
    Unit method();
    String getRuntimeType() {
        return "TestClass";
    }
}

Monitor monitor;
monitor = new cog MonitorImpl();
SecurityRelevant testClass;
testClass = new TestClass(monitor);
while (True) {
    Fut<Unit> returned0;
    returned0 = monitor!beforeRelevantMethod();
    returned0.get;
    testClass.method();
}

Figure 3.20: ABS code resulting from inlining the program shown in Figure 3.17 with policy in Figure 3.18
Chapter 4

Correctness

This chapter examines the correctness properties of the ABS inliner, which are security, conservativity and transparency. As shown in [7] the traditional correctness properties are too strict for the IRM in a multithreaded setting. Furthermore, unlike [7], the monitor inlining of ABS is done in the source code level, essential method calls are introduced to the program due to the concurrency model of ABS. The following sections briefly discuss each of the correctness properties.

4.1 Security

The ABS inliner is in essence a non-blocking inliner [7], since the updates to the security state are synchronized due to the fact that the security state is only accessible with the monitor class. In addition, the control is given back to the client program from the IRM before each security relevant call is performed, and other concurrent security relevant methods are not prevented of being called. Due to the fact that the ABS inliner is non-blocking, it is insecure for non race-free security policies [7]. In the following text, an execution $E$ of the inlined program is picked and it shows that the IRM accepts the trace of the execution $\omega(E)$.

Firstly by examining the basic case, only one COG in the entire program, we could notice the following fact. All the security relevant calls are identified either at compile time or at run-time with the help of `getRuntimeType()`, and corresponding calls to the permission checking methods in the `MonitorImpl` are inserted into appropriate places,

Since the body of each permission-checking method is directly translated from the guarded-updates structure of the corresponding event clause in the security policy, it is guaranteed that such methods will always give control to the caller, if and only if the corresponding security relevant call adheres to the security policy. Thus for this case, each security relevant method invocation in $E$ is examined and permitted (if not terminated) by the monitor.

For the case of multiple COGs in a program, $E$ is composed of arbitrarily interleaved executions of each COG. This makes the situation possible that the monitor allows one COG to continue with a method call $m_1$ and then allows another COG to proceed with a method call $m_2$, while the COGs get scheduled so that $m_2$ executes before $m_1$. Since the security policy is race-free, these kind of situations are guaranteed to be safe.
4.2 Transparency

Transparency is the property that every policy adherent traces of the original program is also a possible trace of the inlined program.

The monitoring code in ConSpec\textsubscript{ABS} is side-effect free as mentioned in Section 3.2. This is achieved by ruling out all the statements with side-effect in the update blocks. The allowed statements are restricted to pure expressions and assignments. Statements such as \texttt{await} and \texttt{get} operations which would introduce new scheduling points or deadlocks to the monitor are excluded from the policy specification.

For any policy adherent execution of the original program, a corresponding execution of the inlined program could be obtained by inserting the security state updates immediately before and after each security relevant call. Since the trace of the original program is policy-adherent, the inserted code snippet would not interfere with the following execution. That means the traces of the original program are traces of the inlined one, the transparency property of the inliner thus holds.

4.3 Conservativity

Conservativity is the inverse of transparency, i.e., every trace of the inlined program is also a trace of the original program.

All the statements that have possibly side-effect are excluded from ConSpec\textsubscript{ABS} at syntactical level. As a result, the guarded update block of each event clause will not perform any method calls. Thus no observable events will be added to the trace by the IRM. Although the call to the permission checking methods are also added to the observable traces, they can never be considered as security relevant given any security policy. Moreover, since the return value of the monitor methods is \texttt{Unit}, and the following execution of the client program is guaranteed not to be affected. For an arbitrary execution of the inlined program, by removing all the configurations corresponding to computational steps performed by the inlined code snippets, we can get a valid execution of the original program. Thus, all the traces of the inlined program are valid traces of the original program.
Chapter 5

Case Studies

In this chapter, the inliner is applied to the HATS case study trading systems. The trading system case study describes a typical model in a supermarket handling sales, which includes the processes that the products are scanned using bar code scanner and payed by cash or credit card at a single cash desk. The following section briefly introduces the trading system model, and presents and applies a simple security policy to the case study.

5.1 Trading System

The basic element of the trading system, the cash desk, where the payments of goods (either by cash or credit card) is handled. Each cash desk can be in an express checkout mode, serving customers with a few commodities to speed up the clearing. All devices needed are wired to achieve this purpose as illustrated in Figure 5.1.

![Figure 5.1: Hardware components of a single cash desk](image)

A device called cash box plays the central role in a cash desk. A sale is started and finished with cash box, and it is used to process cash payment. For payment with credit card, a dedicate device called card reader is used. After the sale is finished, the bill is printed by a printer. To inform the customers whether a cash desk is in express checkout mode, a light display is adopted. All these devices and components are wired by a central PC called cash.
desk PC. The software installed in the PC is also responsible for handling the sale process as well as communicating with the bank running in that machine.

![Figure 5.2: Entities in a store](image)

A set of cash desks is called *cash desk line*. Several cash desk lines together formed a *store*, which is shown in Figure 5.2, each of them connects to the *store server*, which itself connects to a *store client*. The manager of the store could use the store client to manage inventories as well as performing administrative tasks.

![Figure 5.3: Entities in an enterprise](image)

A set of stores is organized into *enterprise*, a server called *enterprise server* connects each of the stores. An *enterprise client* is also connected to the enterprise server, helping the enterprise manager performing tasks in every aspect of the enterprise management. Figure 5.3 shows the entities in an enterprise.

In [15], a more detailed description of the Trading System is provided.

### 5.2 Security Policy

The security policy asserts that the number of active sales will always be larger than or equal to 0. The policy is specified in ConSpecABS in Figure 5.4. The two method invocations *newSaleStarted* and *saleFinished* are declared as security relevant. The security state variable *saleInProgress* representing the number of active sales, is increased and decreased after and before each method, respectively. The two methods are defined in the interface *CashBoxEventReceiver*, shown in Figure 5.5.
SCOPE Session

SECURITY STATE
  Int salesInProgress = 0;

AFTER CashBoxEventReceiver.newSaleStarted()
    PERFORM
      True -> {
        salesInProgress = salesInProgress + 1;
      }

BEFORE CashBoxEventReceiver.saleFinished()
    PERFORM
      salesInProgress > 0 -> {
        salesInProgress = salesInProgress - 1;
      }

Figure 5.4: Security policy

interface CashBoxEventReceiver {
  Unit newSaleStarted();
  Unit saleFinished();
  Unit keypadSend(Int numCode);
  Unit paymentModeSelected(PaymentMode paymentMode);
  Unit moneyReceived();
  Unit changeToNormalMode();
}

Figure 5.5: Interface CashBoxEventReceiver

The inliner could successfully identify the two security relevant calls in the program, and performs the inlining. After the inlining process, the resulting program behaves the same as before, no policy violation is detected.
Chapter 6

Conclusions

6.1 Summary

Monitor inlining is a powerful program rewriting technique to enforce security to a target program. Most of the previous work has been done in sequential and traditional multithreaded Java-like programs. No attempt has been made for monitor inlining in the context of a language with a novel concurrency model, such as ABS.

This master thesis devised and implemented a framework for monitor inlining in ABS program with policy specification language ConSpec\textsubscript{ABS}, which is adapted from ConSpec. The inlining scheme for synchronous and asynchronous calls are presented, the implementation is briefly introduced and the correctness properties of the ABS inliner are discussed. Finally, the inliner is applied to one of the case studies of the HATS project, trading systems.

The inlined reference monitor is represented in the ABS program by an interface \texttt{Monitor} and a class \texttt{MonitorImpl}. The method declarations and definitions within them are generated according to the ConSpec\textsubscript{ABS} policy specification. ConSpec\textsubscript{ABS} is adapted from ConSpec by modifying the syntactical structure and setting limitations of the legitimate statements, so as to avoid side effects in the policy specification.

Same as ConSpec, in ConSpec\textsubscript{ABS}, security relevant events are restricted to method calls. Event clauses are given via interface or class, each type of event clause is treated differently. Interface event clauses could be applied to both synchronous and asynchronous calls, while class event clauses could only be applied to asynchronous calls.

Due to the different semantics of synchronous and asynchronous method calls in ABS, the inlining scheme of the two types of calls are different. For synchronous method calls, the inlining scheme is quite intuitive, the permission checking method is directly inserted before or after the call in question, corresponding to \texttt{BEFORE} and \texttt{AFTER} event clauses, respectively. For asynchronous method call, it is replaced by a call to the corresponding wrapper method defined in the monitor. The wrapper method wraps the \texttt{BEFORE} and \texttt{AFTER} event clauses within the method body, and performs the original call between them. This is for handling the two event clauses right before the initiation of the call and immediately after the return of the method due to the behavior of asynchronous calls.

Because of the restrictions in ConSpec\textsubscript{ABS}, the policy that can be expressed is quite limited. In the case study of the trading system, a simple security policy asserting the active sales are always larger than 0 is enforced, and the resulting program completely adheres to the security policy.
6.2 Future Works

Since the type of security state variables in ConSpec\textsubscript{ABS} are only limited to \texttt{Int}, \texttt{String} and \texttt{Boolean}, the expressiveness of the current ConSpec\textsubscript{ABS} is restricted to a small set of security policies. One of the possible extensions to the current ConSpec\textsubscript{ABS} is to support data structures defined in the ABS standard library such as \texttt{List}, \texttt{Map} in the policy specification, such that ConSpec\textsubscript{ABS} is much expressive than the current version.

Since the ABS language is not mature enough, further extensions of the implementation depends on the constructs provided in the ABS language. For example, with the help of built-in RTTI support, the implemented pseudo code is no longer needed. This will result in a less error prone and more concise version of the inlined program. Furthermore, in ConSpec\textsubscript{ABS} interface events clause would be able to support inherited methods, since with RTTI, the identification of the runtime type is straight forward.
Chapter 7

Acknowledgement

First and foremost, I would like to thank my supervisor professor Mads Dam and Ph.d Andreas Lundblad, I appreciated your guidance to my thesis and the comments to my report. I very much appreciated the opportunity to do my master thesis at KTH and to get involved in the HATS project.

In addition, I very much appreciated the help of my reviewer Ahmed Rezine, thank you for being my reviewer and your advice on the extension of my thesis, although I don’t have enough time to do that.

I would like to thank my family, it is because of their financial and psychological support that I was able to do my thesis in Sweden.

I would like to thank all my friends in Sweden, Lu Liu and Xiaolin Bai, and in China as well, your support and encouragement helped me a lot during my time of loneliness and depression.

I would like to thank all the people who are involved in the Sino-Swedish programme, Anders Berglund and all the others both in Sweden and in China, for providing us such an opportunity to be able to see the world, to experience something different.
Bibliography


Appendix A

ConSpec_{ABS} Abstract Grammar

\[
\text{Policy ::= SecVarDecl* EventClause*;}
\]
\[
\text{EventClause ::= Event GuardedUpdate* [ElseBlock:Block];}
\]
\[
\text{abstract Event ::= Interface:TypeUse <Method> Param:VarDecl*;}
\]
\[
\text{BeforeEvent : Event;}
\]
\[
\text{AfterEvent : Event ::= [RetVal:VarDecl];}
\]
\[
\text{GuardedUpdate ::= Guard:Exp Block;}
\]

Figure A.1: ConSpec_{ABS} Overall Structure

\[
\text{abstract Decl ::= <Name>;}
\]
\[
\text{VarDecl : Decl ::= TypeUse;}
\]
\[
\text{SecVarDecl : Decl ::= TypeUse InitVal:Exp;}
\]
\[
\text{UnknownDecl : Decl;}
\]

Figure A.2: Declarations

\[
\text{abstract Stmt;}
\]
\[
\text{Block : Stmt ::= Stmt*;}
\]
\[
\text{AssignStmt : Stmt ::= Variable:VarUse Value:Exp;}
\]

Figure A.3: Statements
abstract Exp;
VarUse : Exp ::= <Name>;

TypeUse ::= <Name>;
ParametricTypeUse : TypeUse ::= Param:TypeUse*;

<table>
<thead>
<tr>
<th>Figure A.4: Type Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>NullExp : Exp;</td>
</tr>
<tr>
<td>AbsFnApp : Exp ::= &lt;Function&gt; Param:Exp*;</td>
</tr>
<tr>
<td>AbsConstructorExp : Exp ::= &lt;Constructor&gt; Param:Exp*;</td>
</tr>
</tbody>
</table>

| abstract Unary : Exp ::= Operand:Exp; |
| abstract Binary : Exp ::= Left:Exp Right:Exp; |

| abstract BoolExp : Binary; |
| NegExp : Unary; |
| OrExp : BoolExp; |
| AndExp : BoolExp; |

| abstract RelationalExp : Binary; |
| LTExp : RelationalExp; |
| GTEQExp : RelationalExp; |

| abstract EqualityExp : RelationalExp; |
| EQExp : EqualityExp; |
| NEQExp : EqualityExp; |

| abstract ArithmeticExp : Binary; |
| MinusExp : Unary; |
| MulExp : ArithmeticExp; |
| DivExp : ArithmeticExp; |
| ModExp : ArithmeticExp; |
| AddExp : ArithmeticExp; |
| SubExp : ArithmeticExp; |

| abstract Literal : Exp; |
| IntLiteral : Literal ::= <Value>; |
| StringLiteral : Literal ::= <Value>; |

<table>
<thead>
<tr>
<th>Figure A.5: Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
</tr>
</tbody>
</table>