Analysis-Aware Design of Embedded Systems Software

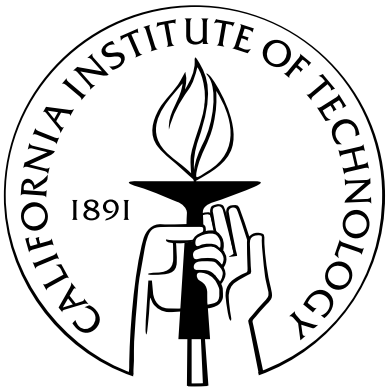
Thesis by

Mihai Florian

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2014

(Submitted November 6, 2013)

To my family and friends.

Acknowledgements

I would like to extend my gratitude to my advisor, Dr. Gerard J. Holzmann, for many insightful discussions that led to the ideas presented in this thesis and for his support throughout my time at Caltech and JPL. This work would not have been possible without his distinguished mentorship. I am extremely grateful to have had the privilege of working with and learning from Gerard.

I would like to thank my committee: Prof. K. Mani Chandy, Dr. Klaus Havelund, Dr. Rajeev Joshi, and Prof. Richard Murray for their valuable advice and support. I am thankful to Klaus for his mentorship and for his suggestions for improving this thesis. I also want to thank Rajeev for sharing some of his ideas with me.

My graduate student life at Caltech would not have been as pleasant without my close friends Alexandra, Annie, Dan, Ned, and Paula. I want to thank them all for our time together. I am thankful to my dear friend Andreea for her support and for keeping me motivated.

I would like to thank the CS department at Caltech. I am indebted to Maria Lopez for making my life in the department a lot simpler.

Last, but not least, I am thankful to my parents, Alexandru and Lodovica, for their love and continuous support.

The work was supported by NSF Grant CCF-0926190.

Abstract

In the past many different methodologies have been devised to support software development and different sets of methodologies have been developed to support the analysis of software artefacts. We have identified this mismatch as one of the causes of the poor reliability of embedded systems software. The issue with software development styles is that they are "analysis-agnostic." They do not try to structure the code in a way that lends itself to analysis. The analysis is usually applied post-mortem after the software was developed and it requires a large amount of effort. The issue with software analysis methodologies is that they do not exploit available information about the system being analyzed.

In this thesis we address the above issues by developing a new methodology, called "analysis-aware" design, that links software development styles with the capabilities of analysis tools. This methodology forms the basis of a framework for interactive software development. The framework consists of an executable specification language and a set of analysis tools based on static analysis, testing, and model checking. The language enforces an analysis-friendly code structure and offers primitives that allow users to implement their own testers and model checkers directly in the language. We introduce a new approach to static analysis that takes advantage of the capabilities of a rule-based engine. We have applied the analysis-aware methodology to the development of a smart home application.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Embedded systems are increasingly being integrated in many aspects of our lives. We interact with them many times a day, often without even noticing it. From our smart phones to our cars, we depend on the continuous reliable behavior of the software that runs on them. Unfortunately, although a lot of engineering effort goes into developing this software, cases in which it fails are frequent. A survey [20] projected that in 2012 "5.6 million smart-phone users experienced undesired behavior on their phones such as sending of unauthorized text messages or the accessing of accounts without their permission." The reliability of embedded systems software is going to become an even larger problem as we rely on more and more embedded systems and the software that runs on them becomes more and more complex.

For some embedded systems software, like the one that controls microwave ovens or washing machines, an error does not cause much more than user frustration and a simple reset usually restores the system to a functional state. But other embedded systems software, like flight software, automotive software, or medical software, is safety-critical. An error in such a system might have catastrophic consequences, including loss of human life [55]. Even if human lives are not at stake a software error might cause huge financial losses and compromise a company's public image. Developing reliable software for safety-critical systems is of utmost importance.

Current practices for increasing software reliability focus on testing and peer code reviews. Test-

ing is usually automated and includes unit testing, integration testing, regression testing. One issue with the common approach to testing is that test scenarios are manually written and thus are biased by the tester's expectations of common use cases. Often the bugs hide in uncommon use cases and thus would be missed by testing. Also, testing cannot control the interleaving of processes during execution. This interleaving is determined by the operating system's scheduler. Different interleavings might lead to different results and there is no guarantee that testing explores all relevant ones. According to [47]: "Most forms of testing average only about 30 to 35 percent in defect removal efficiency and seldom top 50 percent." Peer code reviews involve meetings in which developers go over the code line by line. They are expensive, but have proven to be quite efficient, while there is still room for improvement. Also according to [47], for peer code reviews: "The average for defect removal efficiency in the United States was only about 85 percent."

Formal methods have offered some hope in dealing with the software reliability problem, but currently there is insufficient support for integrating them in the development of reliable embedded systems software. For example, the use of automatic checkers to enforce coding rules is only slowly becoming part of general practice. These checkers usually find only "shallow errors" (e.g., null pointer dereferencing) and are unable to find "deeper" design errors. Other formal methods, like theorem proving and model checking, that have the potential of finding design errors are rarely integrated in the software development process.

We have identified two core issues that cause the current poor adoption of formal methods during embedded systems software development:

- First, embedded systems software is normally structured in a way that does not leverage available analysis tools. The software is developed in an "analysis-agnostic" style that often obfuscates the abstractions made and hampers analyses. For example, when trying to apply model checking to analyze a piece of software, it is not clear where all the relevant state is located. Parts of the state could be hidden inside imported libraries (e.g., a memory allocation library) and outside the reach of the model checker.

- Second, analysis tools, such as model checkers, do not leverage available knowledge of the

embedded systems' environment. For example, embedded systems software relies critically on specific schedulers that differ significantly from those used on common desktop and mainframe platforms. Virtually all real-time embedded systems software, for instance, uses priority-based, round robin, or rate-based monotonic schedulers. Until recently [33], existing model checking system could not directly model this type of scheduler. User-defined analysis rules that check project-specific properties have become common in static checkers [40, 29, 74], but something similar for model checking does not exist.

In this thesis we pursue the development of a different approach to the design and analysis of large complex and multi-threaded embedded systems software, which is based on a methodology that can be called "analysis-aware" [42] design. We believe that we can gain significant benefits by linking software development styles directly with the capabilities of software analysis tools.

We address the two core issues by designing and building a demonstration environment for safety-critical embedded systems software development.

1.2 Analysis-Aware Design

We address the issue of "analysis-agnostic" source code by developing a new specification language, called *Analysis-aware Language* (*AAL*), that can alternatively be seen as a systems programming language. The main goal of the new language, and the reason why we decided to define our own instead of using an existing one, is to enforce an analysis-friendly code structure. AAL is statically typed, has built-in support for concurrency with channel-based message passing, and avoids the use of global variables. Source code in AAL is organized in *modules* that implement functionality specified in *interfaces*.

The language has a formally defined semantics and allows us to write formal specifications as part of the source code in the form of *interface standards*, *function contracts*, and *assertion requirements*. AAL is richer than typical specification languages used by model checkers, but it avoids programming features from standard languages, like C, that produce poor structure (e.g., pointers).

Because scheduling disciplines can have an important influence on both execution and analysis, we also develop a specification language for describing schedulers, as part of the overall specification formalism for embedded system designs. AAL allows users to define their own *scheduler processes* which can be used to implement analyzers, testers, and model checkers within the language.

1.3 Design-Aware Analysis

We address the issue of uninformed analyses by building a set of analysis tools that allow direct execution of specifications and also offer verification support based on static analysis, testing, and model checking. The verification algorithms exploit the structure of the code and the schedulers introduced using the specification language. The analyses are tied into the source code development process and are automatically invoked by the Integrated Development Environment every time the source code is saved. Errors are reported by highlighting the lines of code that caused them.

We have implemented an extensible *static analysis framework* for AAL. The framework is built on top of the rule-based engine CLIPS [69]. Users can define their own static analysis rules as CLIPS programs that operate on the abstract syntax tree and the control flow graphs of the program under analysis. We believe this is the first time a rule-based engine has been used for static analysis, although, query languages, such as Datalog [1], were used before in [74]. CLIPS has the advantage of having a more expressive language. In general, static analysis rules written in CLIPS are short and their code closely resembles their mathematical definition.

As the analyses are invoked on every save, it is likely that they will be applied to partially-written code that is not yet executable. Dynamic analyses like testing and model checking require a closed system that can be executed and observed. We have implemented a method for *closing an open system* based on the interface specifications provided in the source code. The result is a set of executable test drivers which are then used by the *testing and model checking frameworks*.

The *testing framework* performs both concrete and symbolic executions and systematically generates input values that drive the code on different execution paths.

The *model checking* framework uses scheduler processes to implement the model checking algo-

rithms directly in AAL. The framework is extensible and allows users to define their own algorithms. Users can customize the analysis to better fit the specifics of the application they are developing. For example, the model checking algorithms can easily be designed to exploit available information such as the priorities of processes.

1.4 Outline

Chapter 2 provides background and preliminary material on programming languages design, static analysis, testing, and model checking that are necessary to understand the rest of the thesis. Chapter 3 describes the syntax and semantics of our executable specification. It also highlights the features that make the language "analysis-aware" like the memory model and the module and interface system. Chapter 4 presents a series of analysis tools that perform type checking, static analysis, testing, and model checking. These tools exploit the structure of the code imposed by the language and the extra knowledge about the environment (like scheduling information). Chapter 5 describes a case study: an appliance controller for a smart home. Chapter 6 concludes the thesis and provides possible directions for future work.

Chapter 2

Background

This chapter provides a brief description of the concepts used throughout the rest of the thesis. We touch on concepts from multiple fields of formal methods and we show how existing work in these fields relates to the contributions of this thesis.

2.1 Interactive Analysis

Software projects are commonly built using Integrated Development Environments (IDEs) (e.g, Visual Studio, Eclipse, etc.) that tie source code editing with compilers for the target programming languages and provide functionality for building projects. Each time a project is built, the IDE runs the compiler on the source code files of the project resulting in either a clean build or in a set of errors and warnings. Advances in compiler technology and currently available computing power have made the compilation process happen almost instantaneously. This allows the IDE to give rapid feedback to the developer by highlighting any lines that cause errors or warnings together with short descriptions of what went wrong. The development processes proceeds interactively with the developer building the project often and fixing all the errors and warnings immediately after they are introduced. The checks performed by a compiler are usually shallow and limited to syntax conformance and type checking. We would like to have a similar system that performs deeper analyses based on formal methods. This thesis presents our work towards this goal.

Many verification frameworks based on formal methods have been developed so far, but none appear to have all the features that are necessary to support a fully analysis-aware design process.

The closest framework to our approach is Eiffel [6, 58]. It provides an object-oriented programming language and an IDE in which applications can be coded, tested and verified. The language allows specifications in the form of contracts and encourages a design-by-contract software development process [57]. Contracts come in the form of method preconditions, postconditions, and class invariants and are closely related to Hoare logic [39]. Eiffel contracts are mostly checked dynamically at run time, but some efforts have been made to support proofs. The framework comes with AutoTest [54] tools that can generate test cases based on contracts. Contracts have been introduced in many other languages including the .NET languages from Microsoft (in the form of Code Contracts [30]) and Java (in the form of JML [51]). Our framework also uses contracts in the form of preconditions and postconditions at the function level. Function contracts are checked by the testing and model checking analyses.

Another framework for interactive development is Dafny [53] which can be integrated with the Visual Studio IDE. Dafny builds on previous work on ESC/Java2 [19] and Spec# [9] which all perform verification using a method called Verification Condition Generation, which is different than all the analysis methods used in this thesis. Unlike ESC/Java2 and Spec# which are targeted at verifying Java and C# code, Dafny comes with a new object-oriented language designed with verification in mind. Dafny translates the source code under analysis to the intermediate verification language Boogie 2 [8, 52]. Proof obligations that come from the semantics of the language and from user-specified contracts are encoded as assertions in Boogie 2. The resulting intermediate program is passed to a verifier for Boogie 2 which in turn generates verification conditions that are passed to the Z3 SMT solver [24]. Another framework that works similarly to Dafny is Why/Krakatoa/Caduceus [31] where the intermediate language is Why and Krakatoa and Caduceus are front-ends for annotated C and Java programs. These program verifiers work only for sequential code while our approach is targeted at concurrent code.

VCC [18] is a similar tool that works for concurrent C code. Like Dafny, it translates the source code and the annotations (contracts, invariants) to a Boogie 2 intermediate program. To deal with concurrency issues associated with shared memory, VCC introduces a restrictive memory model for

C and an ownership model for pointers. The memory model requires that each C pointer is well-typed. We also have a restricted memory model for AAL and an ownership model for references. As AAL does not have global variables that can be shared, passing the ownership of a reference from a process to another process is simplified and it amounts to passing the reference identifier on a channel.

KeY [10] is a software development tool that can be used to verify restricted Java programs. It takes a UML specification and a Java program and generates a set of proof obligations that are passed to an interactive verification system. Another program verifier is VeriFast [45, 46] that checks C programs using abstract symbolic execution.

2.2 Formal Semantics and Type Systems

The semantics of a language defines the meaning of a program written in that language. There are many ways in which the semantics of a language can be defined: operational semantics, denotational semantics, axiomatic semantics [63]. In this thesis we use operational semantics which defines what it means to execute a program written in the language.

The two common approaches to operational semantics are *big-step operational semantics* and *small-step operational semantics*. In this thesis we use both approaches. We use a big-step semantics to describe how expressions are evaluated and we use a small-step semantics to describe how statements are executed.

A big-step operational semantics defines how the final results of an execution are obtained.

A small-step operational semantics defines how the individual steps of the computation are performed.

In operational semantics (both big-step and small-step), the meaning of programs is specified using a transition system.

A transition system TS is a tuple $(S, Act, \rightarrow, s_0)$ where:

- S is a set of states;

- Act is a set of actions;
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation;
- s_0 is the initial system state.

An execution of the transition system is a (possibly infinite) sequence of states and actions: $s_0\ a_1\ s_1\ ...\ s_i\ a_{i+1}\ s_{i+1}\ ....$ where $(s_i, a_i, s_{i+1}) \in \rightarrow$.

Instead of $(s, a, s') \in \rightarrow$ we usually write $s \xrightarrow{a} s'$.

The difference between big-step and small-step operational semantics is how the transition relation $\rightarrow$ is defined.

The big-step operational semantics is concerned with the relation between the initial state and the final state of an execution. It usually simpler than the small-step operational semantics, but it fails to associate a meaning to executions that do not terminate (infinite executions) and cannot say anything about the intermediate steps of the execution. We use a big-step operational semantics to define the evaluation of expressions because this evaluation always terminates, and we are only interested in the final values.

The small-step operational semantics is concerned with the relation between a state and its immediate successors (i.e., a single step of computation). We use a small-step semantics to define how statements are executed because our language has built-in concurrency and we must be able to reason about how statements from different processes are interleaved.

In both cases we use a set of inference rules to define the transition relation.

A type system [66] consists of a set of well-typed judgments for all the syntactic constructs of a language. These judgments assign a type property to constructs such as variables, functions, and expressions that is a description of the kind of values these constructs can hold or compute. The main purpose of a type system is to prevent errors associated with operations expecting a certain kind of value being applied to values for which the operations do not make sense.

A language is called statically typed if the use of values according to their types is checked statically at compile-time, and respectively, a language is called dynamically typed if the type

checking happens at run time. Our language AAL is statically typed.

The type system is combined with the semantics of the language in a type safety theorem that states that a well-typed program is guaranteed to not get stuck according to the semantics. A program is said to be stuck when it reaches a state from where no rule of the semantics is applicable.

Most programming languages in use today do not come with a formal specification of their semantics and type systems. Our work on the type system of AAL was influenced by previous attempts made to formally specify the semantics and the type system of subsets of various languages: a Java-like language [49], C++ [76], Clight [12], OCaml light [65]. Also, [71, 72] provided a good reference for carrying out the type safety proof.

2.3 Static Analysis

Static analysis is the analysis of the source code of a program without actually running the program. Common checks include compliance with a coding style guide, detecting unreachable code, detecting some assertion failures, finding array index out of bounds errors, finding null pointer dereferences, and various other memory access errors. It has become common for static analysis tools to offer users the possibility of defining their own checks. Most tools use a domain specific language for this purpose.

There are many approaches to static analysis [62]. The traditional approach is based on data flow analysis that computes for each statement of the program some information that is relevant to the analysis (for example, the definition of what variables reach that statement). The information is introduced as a set of data flow equations and a fixed point computation is carried out to obtain the result.

Another approach is based on translating the program and the property being checked (what it means for the property to be violated) to a set of boolean constraints. A SAT solver is then used to check if this set of constraints is satisfiable. If yes, then a violation of the property was found and the resulting SAT model is used to construct a counter example. SATURN [77] is an example of a static analysis tool that takes this approach and which was used to check code bases of millions

of lines of code (the Linux kernel). Coverity is another static analysis tool that combines data flow analysis and constraint-based analysis and has proven to be successful at analyzing large projects [11].

Yet another approach is based on abstract interpretation [21]. It uses an abstract semantics of the language that keeps track of only the information relevant to the analysis (for example, the sign of the values of integer expressions or the intervals in which these values fall). A symbolic execution is then carried out over all the paths of the program to check if the property is satisfied. ASTREE [22] is an example of a tool that takes this approach.

Model checking the control flow graph of the program is the basis for another approach. The property being checked is encoded as a temporal property and checked against the control flow graph. Examples of tools taking this approach are UNO [40] and Coccinelle [14].

A more recent approach is to record all the information about the source code of the program in a database and encode the static analysis check as a set of queries over this database. This approach is taken by the SemmleCode tool [74, 23].

Our approach is a combination of data flow analysis and query-based analysis. It encodes the abstract syntax tree and the control flow graph of the program under analysis as a set of facts in a rule-based engine. These facts are added as the initial facts in the knowledge base of the engine and resembles populating a database with information about the program where each kind of fact corresponds to a table. The data flow equations are encoded as rules in the language of the engine, with the engine performing the fixed point computation leading to a set of inferred facts that represent the results of the analysis. We have several built-in rules, but users can write their own checks.

2.4 Concrete and Symbolic Testing

Testing is a dynamic analysis that consists of running the program and reporting any errors that are found at run time. Program components can also be tested in isolation (unit testing) which requires the creation of test cases, also called test drivers, which are closed systems that execute the code

under test.

The biggest challenge for testing is to exercise as many different behaviors of the program as possible. Traditionally, test cases are manually written by the developer. Code coverage measures are used to decide if more test cases are needed to exercise the code that is not covered by the existing test cases. Developers tend to write few test cases and even the ones that they write are biased towards their expectations of how the system will be used. But errors usually hide in unexpected scenarios, so it is likely that they will be missed.

Efforts have been made to automate the process of test case generation. Random testing [38], also called black-box fuzz testing (because it does not look at the source code), where input is generated randomly and then given to the program has proved to be very effective in finding bugs in UNIX and Windows NT utilities [59, 34].

Using symbolic execution for testing was introduced in [48]. Symbolic execution consists of running the program with symbolic values instead of concrete inputs and performing symbolic computations on these values instead of the concrete values. It has the advantage that in one symbolic execution it explores an entire class of concrete inputs. Symbolic execution was also used to generate concrete test cases. A concrete test case is generated for each symbolic run.

Using a combination of concrete execution and symbolic execution to dynamically generate test cases was introduced in CUTE [70] which coined the term *concolic testing*, also known as white-box fuzz testing (because it looks at the source code). A similar idea was explored in DART [36] and its successor SAGE [37]. They have added the use of static analysis results to compute function summaries and improve the scalability of the approach. The symbolic execution is embedded in the source code through code instrumentation. Another approach to concolic testing is to perform it at byte code level. This approach was taken in KLEE [15] which is a concolic tester for LLVM [50] byte code.

The basic idea of concolic testing is to perform a random initial concrete execution which is carried along a symbolic execution that gathers the conditions of all the branches taken in the execution. These conditions form a symbolic path that is used to generate new paths by negating

some of the conditions and using an SMT solver to obtain a new set of concrete inputs. If the SMT solver is not powerful enough to solve some of the constraints, then concrete values are as solutions instead. The new set of inputs is fed to the concolic tester and the process repeats until there are no more paths to explore or a target coverage was reached. The approaches in CUTE, DART, SAGE, and KLEE differ in the heuristics used to pick what paths to explore next.

We have implemented a concolic tester for our framework. Instead of using instrumented code to add the symbolic execution, we embed it directly in the execution engine for AAL.

2.5 Model Checking

Model checking [17, 7] is a methodology for solving the *model checking problem*: given a finite state model of a system and a property, automatically check if the model satisfies the property by exhaustively exploring all possible behaviors of the model. Properties are usually expressed in a temporal logic, with LTL [67] being the most popular one.

There are multiple approaches to model checking. Symbolic model checking keeps the explored state space and the property being checked as ordered binary decision diagrams. SMV [56] and its reimplementation NuSMV2 [16] are popular symbolic model checkers.

Explicit state model checking stores the state space explicitly and is usually based on automata-theoretic framework [73]. The model of the system is seen as a Büchi automaton, the language accepted by this automaton being all possible behaviors of the system. The negation of the property being checked is also converted to a Büchi automaton, the language accepted by this automaton being all behaviors that violate the property. The automa-theoretic approach to model checking constructs the intersection of the two automata and checks if the language accepted by the intersection is empty or not. If it is empty then the property is satisfied since there are no behaviors of the system that violate the property. If the intersection is not empty the model checking algorithm finds a behavior that violates the property.

When the model of the system is actually the source code of the system, then explicit state model checking can be considered a form of dynamic analysis.

SPIN [41] is a popular explicit state model checker based on the automata-theoretic framework. It verifies models written in ProMeLa which is a language for modeling asynchronous concurrent systems inspired by Dijksta's guarded commands language [25]. SPIN also supports formal models with embedded C code fragments and with some setup can be used to model check C source code. Recent versions of SPIN also support the specification of process priorities [33]. SPIN can be used with the swarm [44] tool to leverage multi-core CPUs and grid computing environments. SPIN can check omega-regular properties expressed as LTL formulas or directly as Büchi automata (in the form of never claims). Other examples of explicit state model checkers are Murphi[26] and LTSmin[13].

Java Path Finder (JPF) [75] is an explicit-state software model checker for verifying executable Java bytecode programs. JPF is built as a custom Java Virtual Machine that can interpret Java bytecode and it also integrates program analysis and testing.

Ideas from model checking have been used in systematic testing of multithreaded C code [35, 61, 32]. The non determinism associated with thread scheduling is systematically explored, but without storing concrete states.

Our approach to model checking AAL programs is based on the explicit state model checking methodology. The difference is that we use ideas from runtime verification to monitor LTL properties during the executions of the system instead of using the regular automata-theoretic based approach. We made this decision because we support parameterized LTL properties which leads to multiple LTL properties being verified at the same time, each associated with different values for the parameters. Also, the verification of a property does not necessarily start in the initial system state. Instead it starts from the state where a message appearing in the property was first seen.

Context-switch [43, 60] bounding is a method for reducing the number of executions that have to be explored by bounding the number of times the execution can switch from one thread to another. Delay-bounded scheduling [28] is another approach used to reduce the number of explored executions. Our scheduling formalism can model these reductions, but its purpose is to specify actual schedulers like priority based ones.

Chapter 3

Analysis-Aware Design

3.1 Overview

In this chapter we introduce an executable specification language called *Analysis-aware Language* (AAL) that is suitable for writing embedded systems software together with their formal specifications. Our premise is that the use of formal specifications can effectively reduce the number of residual software defects, and also empower and facilitate the use of formal software analyses methodologies.

The language is designed for *interactive development* (Figure 3.1) where the analysis drives the software development process. The analysis runs continuously in the background while the software is being written and flags errors as soon as possible. This is in contrast to the more common approach of "post-mortem analysis" where the software is written first and only after that, it is analyzed either by testing or by peer code reviews.

The specification language is close to a system programming language, but has a formally defined semantics and allows us to write formal specifications as part of the source code in the form of interface standards and assertion requirements. The language is also close to a typical specification language used in model checking, but is richer by providing functions and built-in abstract data types.

The language is statically typed with the type system consisting of base types (e.g., integers), built-in collections (e.g., sets, lists, and maps), and user defined types. Importantly, the language

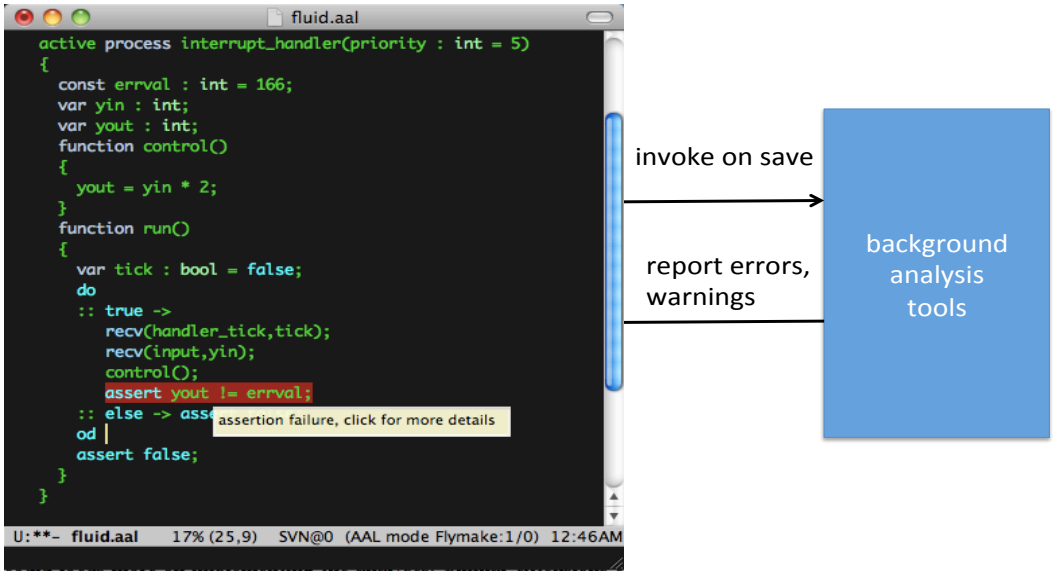


Figure 3.1: interactive development

also has built-in support for concurrency. A model of a system is defined as a set of modules interconnected by a set of interfaces. Each module consists of a set of concurrent processes that communicate by channel-based message passing. The system starts with a statically determined number of processes, but during its execution other processes can be dynamically created. Each module has exclusive read access to a set of input channels on which it can receive messages from processes that reside either in the same module or in other modules. The communication between processes from different modules must obey the specifications described in the interfaces.

Because concurrency related errors are among the hardest to analyze in embedded systems software, preventing such errors is a key part of the design. The memory model used relies on the notions of ownership and ownership transfer. Each process owns a set of memory regions that non-owning processes cannot access. Ownership of a memory region can only be transferred to another process by message passing. The memory model similarly avoids other features that can obstruct formal verification attempts. For instance, it excludes the use of explicit pointers and unsafe casts between incompatible types. The memory model is designed to prevent unwanted aliasing and to ensure that race conditions are prevented.

3.2 Language Features That Facilitate Analysis

The design of AAL was driven by the observation that often the easiest way to solve a problem is to remove the problem altogether. We have identified language concepts in other programming languages used for writing embedded systems software (e.g., C and C++) that unnecessarily complicate the analysis and we have removed them whenever it was possible. Where we could not remove a feature altogether we have tried to enforce coding disciplines that allow the analysis tools to catch common mistakes associated with these features as soon as possible, preferably during type checking and static analysis. The later stages of analysis, systematic testing and model checking, focus more on catching application-specific errors, the situations where the implementation given in modules deviates from the specification described in the interfaces.

We have decided not to have global variables that could be shared either between modules or between the processes inside the same module. Global variables (and shared memory regions in general) are the source of common concurrency errors known as race conditions. This happens when more than one process accesses (read or write) a variable at the same time and the outcome depends on the order in which these accesses occur. In general, since accesses to global variables look exactly like accesses to local variables, the use of global variables tends to obfuscate the places in the code where processes communicate.

For inter-process communication we have chosen channel-based message passing which is a safer way of sharing data between modules and processes than using global variables. Channels and messages are typed and it is impossible to send or receive a message of a different type than the one specified in the channel description. A process is expected to consist of a main event processing loop in which messages are received from all input channels. Each message is handled one at a time. Message handling usually consists of some computation followed by sending one or more messages to other processes. This code structure makes it more clear which are the points in the code where inter-process communication occurs. The developer can reason about the code in between the communications points in a sequential manner without having to worry about any interference from other proceses.

Another decision was to have a language without pointers. Pointers provide an unstructured way of accessing any piece of memory and it might be hard to understand what exactly is being accessed just from the code itself. Instead we have replaced pointers with a safer construct called references similar to the one used in languages like Java. References are typed and can only come from explicit memory allocation. The type of a reference specifies the type of values the reference can contain. In particular, a reference, if declared as such, can contain another reference. The actual memory layout of the program is not accessible to the developer. There are no constants that are references, for example, it is impossible to convert a memory address to a reference and then use it to access the contents. Also, it is impossible to convert a value of another type to a reference. No operations are allowed on references except accessing its contents and destroying the reference. Each reference is owned by at most one process. The ownership of a reference can be transfered from one process to another using message passing.

In our language expressions do not have side-effects. This is enforced by not allowing any function calls in expressions. To achieve the same effect as having function calls in an expression, the results of the function calls can be first saved in local variables, and then these local variables can be used in the expression. Evaluating an expression in the same context any number of times leads to the same value. This is important for analysis tools like model checkers that use the value of expressions to make decisions about what paths in the code to explore next. If evaluating expressions had side effects, then a model checker would have to reverse these side effects after each expression evaluation.

Another common source of errors is not handling all the possible cases of a branching statement. For example, in C this would amount to not having an else case for an if statement or a default case for a switch statement. We make this impossible in our language by requiring that each if statement must have an else branch. Also, each selection statement (that sends or receives messages) must have a timeout branch for handling the case where no messages can be received or sent.

The code is organized in modules that implement functionality specified in interfaces. An interface defines an abstraction of a module. This code organization allows testing against interface specifications as opposed to requiring all modules to be implemented. Without this feature, interac-

tive development would not be possible because the analysis would require all the code to be already written and thus would have to happen after the development stage instead of at the same time.

The language provides a constraint-based formalism for specifying schedulers that are used in the embedded systems. For example, almost all embedded systems use some form of priority-based scheduling. Knowing how the scheduler behaves can greatly simplify the model checking phase of the analysis. Traces that cannot occur in practice do not have to be explored at all.

Another important feature is the possibility of defining analyzers, testers, and model checking algorithms within the specification language. These analyzers are implemented as a special kind of process, called a scheduler process, that can introspect the rest of the system. This feature allows users to write analyzers that are aware of the particularities of the code being analyzed.

3.3 Concurrency Model

A running program (Figure 3.2) consists of a set of module instances. Each module instance encapsulates a set of currently active processes. These processes were either active in the initial system state or they were spawned during the program's execution. The set of module instances is specified in a configuration. The configuration also specifies how module instances are interconnected using interfaces and how the active processes of each module instance are scheduled for execution subject to user-defined scheduling constraints defined in schedulers.

The primary unit of computation is a process. A program execution starts from the initial system state described in the configuration. From there it evolves by interleaving actions from active processes according to the scheduling constraints. If no scheduling constraints are specified then processes are scheduled in a fully non deterministic manner.

Processes communicate with each other by passing messages over FIFO channels. Channels have a finite size buffer of at least one slot. Each slot can hold one message. Send operations block when the channel is full. Respectively, receive operations block when the channel is empty.

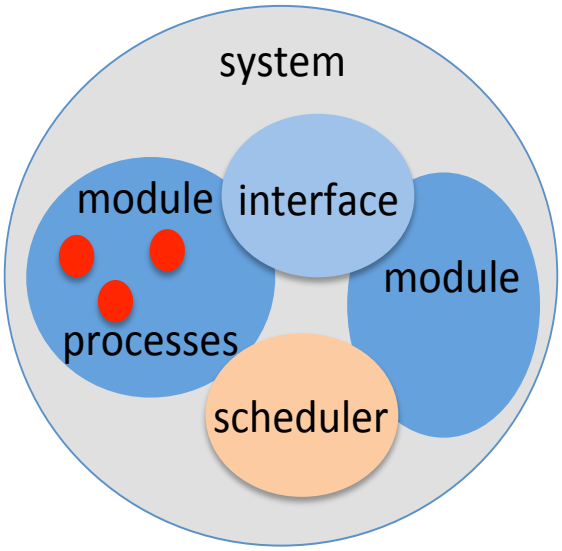


Figure 3.2: a running program in AAL

3.4 Memory Model and Ownership Model

The memory associated with the state of a running program is split into a set of non overlapping regions. Memory regions are of two kinds: the ones used for storing the values of program variables and the ones used for reference cells.

Because we do not have global variables, the memory regions associated with program variables are local to a process or a function call. They exist only as long as the process (or the function call) is active. These regions cannot be accessed from other processes. Also, since we do not have an address-of operator, reference identifiers cannot be used to access the contents of this kind of regions. The only way to access the contents of a region associated with variable v is to use the name of v. This rules out certain types of errors possible in other languages (e.g., C) such as returning the address of a local variable.

The memory regions associated with reference cells are explicitly allocated and deallocated in the program code. A program is expected to keep track of how much memory it uses so it does not run out of memory. These regions are uniquely identified by reference identifiers and are organized in a global heap. The heap is a map from reference identifiers to reference cells extended with ownership information (the identifier of the process that owns the region). The contents of a reference cell

can only be accessed by dereferencing the associated reference identifier. It is an error to access a reference cell from a process that does not own it.

Initially a reference cell is owned by the process that allocated it, but the ownership can be explicitly transfered to another process by passing the reference identifier over a channel. This amounts to updating the ownership information associated with the reference cell.

3.5 Syntax

In this section we introduce the main syntactic constructs of our analysis-aware executable specification language. All constructs are described in the commonly used Extended Backus-Naur Form [2] accompanied by short explanations in the main body of the text.

Example

To introduce the syntax of AAL, we start with the classic example of a program that prints "hello world" to the standard output (Figure 3.3).

```
module Hello
{
  active process hello()
  {
    function run()
    {
      printf("hello world\n");
    }
  }
}

config HelloCfg
{
  module h = Hello;
}
```

Figure 3.3: a small AAL program

At the highest level, an AAL program consists of a set of modules. In our example we have only one module called `Hello`. As processes are the execution units of AAL programs, we need at least one process (`hello` in this example) in order to be able to execute a piece of code. The process is

marked as being `active`, meaning that it can start executing from the initial state of the program. Processes are contained in modules. In our example, process `hello` is part of module `Hello`.

A process contains the definition of a set of functions, with each function being composed of statements that specify the operations to be executed. Each process must define a function named `run` which is the entry point of the process. The execution of a process begins with the first statement in its `run` function.

In our example, the `run` function contains only one statement: a call to a built-in function named `printf`. The `printf` function in AAL is similar to the `printf` function in C and it is used to output to the standard output. In our example it takes as an argument the string `"hello world"` that will be printed to standard output.

Every program must have a configuration that defines the initial state of the program. In our example, the configuration is called `HelloCfg` and it creates an instance `h` of module `Hello`. Instantiating a module in the configuration specifies that all the `active` processes of that module are part of the initial state and they can start executing immediately. In our case, the initial state contains an instance of the process `hello`. The execution of this process will print the string `"hello world"` to the standard output.

Systems

A system (Figure 3.4) is the highest level syntactic unit and is the collection of all the elements from all source files.

```
system ::= system_element*

system_element ::= module
                 |  interface
                 |  scheduler
                 |  config
                 |  schedproc
```

Figure 3.4: systems in AAL

Modules

Modules (Figure 3.5) are the primary way of organizing functionality. Each module implements one or more interfaces and can also import one or more interfaces.

```
module ::= 'module' identifier '{' module_element* '}'

module_element ::= 'import' identifier ';'
                 | 'export' identifier ';'
                 | 'sched' identifier ';'
                 | type_decl
                 | type_def
                 | constant_def
                 | function_decl
                 | function_def
                 | process
```

Figure 3.5: modules in AAL

A module must provide values for all the constants declared in the interfaces that it exports. In particular it must define all the channels used in the interface specifications.

A module consists of a set of processes which cooperate to implement the module's functionality. The actual organization into processes is not visible outside the module. For example, a module corresponding to a server could spawn a process for each client it talks to or have just one process handle requests from all the clients.

The code should be split into modules in such a way that each module could be analyzed in isolation given just the set of interfaces it imports.

In general, there can be more than one instance of a module active in the system at the same time. A module should avoid the assumption that it is the only active instance.

Figure 3.6 shows some examples of module definitions in AAL.

Interfaces

Interfaces (Figure 3.7) specify how modules interact. Each interface can contain type definitions (for example, the types of messages passed on channels), constant definitions, event definitions and linear temporal properties.

```
module Service
{
  export ServiceInterf;
  type msg_type = {req : int, reply : chan[int]};
  const get : int = 1;
  const c : chan[msg_type] = mkchan of msg_type[1];
  function f(x : int) : (r : int)
  {
    // ...
  }
  active process service()
  {
    // ...
  }
}

module Client
{
  import serv = ServiceInterface;
  active process client()
  {
    // ...
  }
}
```

Figure 3.6: examples of modules in AAL

The constant definitions can be used to specify the channels on which the inter-process communication between modules occurs. A channel defined as a constant forces any module implementing the interface to have a channel with the respective name and type. The channel is constant in the sense that it always refers to the same reference cell, not that its contents cannot change. This also guarantees that the channel will always be allocated and that messages can be sent to it.

An interface can declare a constant without providing its value. Each module that exports the interface must provide a specific value for the constant.

Temporal properties come in two flavors: linear temporal logic (LTL) formulas and regular expressions. They are expressed over parameterized definitions.

Definitions consist of sending and receiving messages and can be parameterized both by the channel on which the communication occurs and by the messages being passed.

The temporal properties are also parameterized by data. Each parameter is bound to a specific value on the first occurrence of an event with the same parameter. All further sequences of events

```
interface ::= 'interface' identifier '{' interface_element* '}'

interface_element ::= type_decl
                    | type_def
                    | constant_decl
                    | constant_def
                    | def
                    | regexp
                    | ltl

def ::= 'def' identifier '(' [identifier [',' identifier]*] ')' ':'
        ('send' | 'recv') '(' identifier ',' message ')' ';'

message ::= identifier
          | '{' identifier '=' identifier
                ',' identifier '=' identifier]* '}'

regexp ::= 'regexp' identifier '(' [identifier [',' identifier]*] ')' ':'
           regexp_body ';'

regexp_body ::= identifier '(' [identifier [',' identifier]*] ')'
              | '(' regexp_body ')'
              |  regexp_body '|' regexp_body
              |  regexp_body '*'
              |  regexp_body '?'
              |  regexp_body regexp_body+

ltl ::= 'ltl' identifier '(' [identifier [',' identifier]*] ')' ':' expr ';'
```

Figure 3.7: interfaces in AAL

with the same value of the parameter must satisfy the formula. Occurrences of events with different values for the parameters get bound to different instances of the formula.

Figure 3.8 shows an example of an interface definition in AAL. It corresponds to the interface exported by the `Service` module defined in Figure 3.6 and to the interface imported by the module `Client`. The interface defines a type `msg_type` of messages that can be received on a channel `c`. It contains definitions for receiving a request (`req(rc)`) and for sending a reply (`reply(rc)`). The interface also specifies an LTL property (`p0`) and a regular expression property (`p1`).

Schedulers

Schedulers (Figure 3.9) provide a constraint-based formalism for defining various scheduling policies. The purpose of this formalism is to describe what processes are allowed to execute next from the

```
interface ServiceInterf
{
  type msg_type = {req : int, reply : chan[int]};
  const get : int;
  const c : chan[msg_type];
  def req(rc): recv(c,{req = req, reply = rc});
  def reply(rc): send(rc,v);
  ltl p0(rc): [](req(rc) => <>reply(rc));
  regexp p1(rc): (req(rc) reply(rc))*;
}
```

Figure 3.8: an example of an interface in AAL

current system state.

The language offers support for specifying scheduling events that describe when the scheduling decisions should be taken. For example, if the scheduler is nonpreemptive then the scheduling should happen only when the currently running process blocks or yields. If the scheduler is preemptive then the scheduling could happen, in the worst case, after each step of a process.

```
sched ::= ('preemptive' | 'nonpreemptive') 'sched' identifier
          '{' sched_element* '}'
sched_element ::= var_def
                | constant_def
                | proc_set
                | update
proc_set ::= identifier '=' expr
update ::= 'update' identifier '{' stmt* '}'
```

Figure 3.9: schedulers in AAL

The scheduling language operates with sets of processes and process attributes. Predefined sets of processes are:

- `All`, the set of all processes in the system that were created and have not yet terminated,
- `Enabled`, the set of processes that can take a step in the current system state,
- `Blocked`, the set of processes that cannot take a step in the current state because they are waiting for an event to happen,
- `Running`, the set of processes currently running on the CPU, which on a single core system should consist of only one process.

We assume an infrastructure that defines and updates the process attributes used by the scheduler. Examples of such attributes would be priorities and various counters. These attributes are initialized when processes are created and are updated after each scheduling decision.

In the simplest form, a scheduler can be described as computing the set:

```
Next = {p in Enabled | constraint(p)}
```

where `constraint` is a first order logic formula on the current state of the system (including process attributes). If `Next` is empty, then the system halts and the analysis tool can report an error (unless the halt signifies successful termination). If `Next` is not empty then the process to execute next is nondeterministically picked from that set.

This formalism can be used to described priority-based scheduling:

```
Next = {p in Enabled | forall p' in Enabled.
                       priority(p) >= priority(p')}
```

Round Robin scheduling could be described as:

```
Next = {p in Enabled | not exists p' in Enabled.
                       p' in Between(current, p)}
```

where `current` is the currently running process and `Between` is a function that returns the set of all processes between two processes according to some apriori cyclic order.

Yet another example:

```
Next = {p in Enabled | forall p' in Enabled.
                       steps(p) - steps(p') < 10}
```

is a scheduler that picks a process `p` for execution only if `p` is enabled and it is at most 9 steps ahead of any other enabled process `p'`. Here `steps` is assumed to be updated by the execution environment. For example, if there are only two processes `p` and `p'`, both enabled, and `p` is 10 steps ahead of `p'`, then `p` will not be considered for execution until `p'` takes at least one step. If `p'` blocks (for example, waiting for a message), then `p` is considered for execution because the set `Enabled` contains only `p`.

Configurations

A configuration (Figure 3.10) is used to describe the initial state of the system. It allows the instantiation of modules and schedulers and specifies how all the components are connected. More than one instance of the same scheduler or module can be created. The instances are named so that they can be unambiguously connected.

```
config ::= 'config' identifier '{' config_element* '}'

config_element ::= 'module' identifier '=' identifier ';'
                 | 'sched'  identifier '=' identifier ';'
                 | 'connect_modules' '(' identifier '.' identifier ','
                                         identifier '.' identifier ')' ';'
                 | 'set_sched' '(' identifier '.' identifier ','
                                   identifier ')' ';'
```

Figure 3.10: configurations in AAL

`connect_modules` links the interface imported by a module to an interface exported by another module.

`set_sched` specifies the scheduler to be used to schedule the processes of a given module.

There can be more than one configuration, but only one of them (by default the first one) is used to generate the initial system state. Having multiple configurations is useful for testing under a variety of initial set-ups.

```
config TestConfig
{
  module service = Service;
  module client = Client;
  connect_modules(client.serv, service.ServiceInterf);
}
```

Figure 3.11: an example of a configuration in AAL

Figure 3.11 shows an example of a configuration definition in AAL. It defines two module instances: `service`, an instance of module `Service`, and `client`, an instance of module `Client`. Module `Service` exports an interface `ServiceInterf` that is imported by module `Client`. The module instances `service` and `client` are connected through the interface `ServiceInterf`.

Processes

A process (Figure 3.12) is the execution unit of the system.

```
process ::= ['active' ['[' number ']']] 'process' identifier
            '(' [var_decl [',' var_decl]*] ')' ['sched' 'by' identifier]
            '{' process_element* '}'

process_element ::= type_decl
                  | type_def
                  | var_def
                  | constant_def
                  | function_decl
                  | function_def
```

Figure 3.12: processes in AAL

Processes can be `active` meaning that they are present in the initial system state. More than one instance of the same process can be active at the same time. The qualifier `active[n]` in front of a process definition is used to specify that `n` instances of that process are active in the initial system state.

The arguments taken by a process correspond to process arguments (for example, the priority of the process) that are used for scheduling. Each process can be explicitly assigned to a scheduler. If no scheduler is specified, then the process is assigned to the default completely random scheduler. The process arguments can be changed by the process itself (like modifying a local variable) or by the scheduler that is associated with the process (this is the preferred way because it ensures a uniform handling of process attributes across all processes scheduled by the same scheduler).

Each process must contain a function called `run` which is the entry point of the process.

New processes can be created during execution using the statement

`spawn(process_name, arg1, ...) [sched by sched_name]`.

Figure 3.13 shows an example of a process definition in AAL. The process `service` is part of the module `Service`. It contains the definition of a local variable `value` and three functions: `next_value`, `handle_request`, and `run`. The execution of the process starts with the `run` function.

```
module Service
{
  export ServiceInterf;
  type msg_type = {req : int, reply : chan[int]};
  const get : int = 1;
  const c : chan[msg_type] = mkchan of msg_type[1];
  active process service() {
    var value : int = 1;
    function next_value(v : int) : (nv:int) {
      do
      :: true -> nv = v;
      :: true -> nv = 2 - v;
      :: else -> assert false;
      od
    }
    function handle_request(m : msg_type) {
      if
      :: m.req == get ->
         sel
         :: send(m.reply, value) ->
            value = next_value(value);
         :: timeout -> assert false;
         les
      :: else -> assert false;
      fi
    }
    function run() {
      var m : msg_type;
      do
      :: true ->
         sel
         :: recv(c,m) -> handle_request(m);
         :: timeout -> break;
         les
      :: else -> assert false;
      od
    }
  }
}
```

Figure 3.13: example of a process in AAL

Scheduler Processes

Scheduler processes (Figure 3.14) are used to define how processes are scheduled in the code itself. They serve a similar purpose as scheduler constraints, but they provide a lower level syntax that allows the developer to implement testing and model checking algorithms inside the language. Scheduler processes look like regular processes with the addition that they are allowed to use the

reflection API.

```
schedproc ::= 'schedproc' identifier '{' process_element* '}'
```

Figure 3.14: scheduler processes in AAL

The reflection API consists of a set of built-in primitives that allow direct manipulation to the current system state. The most important primitives are shown in Figure 3.15.

```
type action_type = { ptype : uint, pid : uint, ano : uint } ;
get_enabled_actions () : (a : array[action_type])
set_next_actions (next_actions : set[action_type]) : ()
get_current_state() : (state : array[char])
set_current_state (state : array[char]) : ()
```

Figure 3.15: reflection API in AAL

`get_enabled_actions` returns the actions that can be taken next from the current system state. An action is identified by a process name (`ptype`), a process identifier (`pid`), and an action number (`ano`). Action numbers differentiate between actions that are available at the same point in the same process due to the nondeterminism at process level introduced by the `if`, `do`, and `sel` statements.

`set_enabled_actions` is used to specify what actions are allowed to be taken next. The `next_actions` set serves a similar purpose to the `Next` set in the constraint-based scheduling formalism. If an action that is not enabled is passed to `set_enabled_actions` then, if the action is picked for execution, a run time error will be generated and the execution will stop.

`get_current_state` allows full access to the current state. The returned state can be inspected and it also can be saved such that it can later be restored with `set_current_state`. Together these primitives can be used to backtrack the current state which is necessary when implementing a model checking algorithm.

Types

Our language is statically typed, each expression is required to have a unique type that is determined at compile time. This is achieved by providing type annotations for all constant, variable,

and function declarations. Starting from these annotations a type checker statically enforces that expressions are used only in a context allowed by their types.

```
type_decl ::= 'type' identifier ';'
type_def  ::= 'type' identifier '=' type_use ';'
type_use  ::= identifier
            | 'int' | 'uint' | 'bool' | 'char' | 'string'
            | 'ref' '[' type_use ']'
            | 'array' '[' type_use ']'
            | 'chan' '[' type_use ']'
            | 'set' '[' type_use ']'
            | 'seq' '[' type_use ']'
            | 'map' '[' type_use ':' type_use ']'
            | '{' identifier ':' type_use [',' identifier ':' type_use]* '}'
```

Figure 3.16: types in AAL

Type declarations (Figure 3.16) introduce new types without specifying what the types stand for. Each declared type must eventually be defined. Type declarations are necessary for specifying (mutually) recursive types.

Type uses (Figure 3.16) specify the syntax used for type annotations when defining new constants, variables, functions, or types. A type use is either a previously declared type, one of the basic types, or one of the compound types.

There are four basic types: `int` for integer numbers, `uint` for unsigned integer numbers, `bool` for booleans, `char` for characters, and `string` for strings. The types `int`, `uint`, and `char` are bounded, but the specification of the bounds is left to the implementation. The type `string` stands for a sequence of characters.

The compound types come in two forms: *reference types* and *record types*. Reference types correspond to built-in collection types: references, arrays, channels, sets, sequences, and maps. A value of a reference type is a *reference* which uniquely identified a reference cell. References and reference cells are described in more detailed in Section 3.4. References, arrays, and channels can contain only a finite predefined number of values. Sets, sequences, and maps can contain an unbounded number of values, but the analysis is expected to impose finite bounds on their sizes. A record type stands for a tuple (the cartesian product) of some other types.

A *type definition* (Figure 3.16) associates a new name to an existing type.

```
type msg_type = {req : int, reply : chan[int]};
type tree;
type tree = ref {
              is_leaf : bool,
              data    : int,
              left    : tree,
              right   : tree
            };
type graph = ref {
               nb_nodes : int,
               edges : array[array[bool]]
             };
```

Figure 3.17: examples of types in AAL

Figure 3.17 shows some examples of type definitions in AAL. The type `msg_type` could stand for the type of messages sent and received over a channel. The recursive type `tree` is a definition of a binary tree. The type `graph` is a definition of a graph represented using an adjacency matrix.

Constants and Variables

Constants and variables are the primary way of storing and accessing values. Each variable (or constant) associates a value to its static name. Using the name of a variable (or a constant) is the only way the associated value can be accessed. Our language does not have an address-of operator and enforces a clear distinction between variables and references. The name of a variable does not stand for a reference identifier.

```
constant_decl ::= 'const' identifier ':' type_use ';'
constant_def  ::= 'const' identifier ':' type_use '=' expr ';'
var_decl ::= identifier ':' type_use
var_def  ::= 'var' var_decl ['=' expr] ';'
```

Figure 3.18: variables and constants in AAL

Figure 3.18 shows the syntax for constant and variable declarations and definitions. Variables are introduced using the keyword `var`, while constants are introduced using the keyword `const`.

Uninitialized constants and variables are set to a fixed value of their respective type. Integers are set to `0`, unsigned integers to `0u` , booleans to `false`, characters to `'\0'`, and strings to the empty

string "".

Functions

Functions (Figure 3.19) are the primary way of organizing code in our language and also the first level of abstraction. Functions can be declared and defined at two levels: inside processes where they are available only to that process, or inside modules where they are available to all processes inside that module. A declared function must be defined as some point at the same level.

```
function_decl ::= 'function' identifier '(' [var_decl [',' var_decl]*] ')'
                  [':' '(' [var_decl [',' var_decl]*] ')']
                  [contract*] ';'

function_def ::= 'function' identifier '(' [var_decl [',' var_decl]*] ')'
                 [':' '(' [var_decl [',' var_decl]*] ')']
                 [contract*]
                 '{' stmt* '}'

contract ::= 'pre' expr
           | 'post' expr
```

Figure 3.19: functions in AAL

Each function consists of a sequence of statements called the body of the function.

Functions use the call by value convention. Each argument has an associated variable that gets assigned to the value of the expression used for that argument when the function was called. It is an error to assign to an input argument in the function body.

Unlike in most main stream languages, but like in Dafny [53], in our language functions can have more than one return value. This is achieved by associating a (named) return variable with each return value. The return values are the values of the return variables when the function terminates either by reaching the closing curly brace or by executing a `return` statement. Note that, unlike in other languages, the `return` statement does not specify as arguments what values are returned.

Functions can specify contracts which consists of a set of preconditions and a set of postconditions. The preconditions specify what must be true at function entry and the postconditions specify what will hold at function exit (assuming that the preconditions were satisfied).

Our language has a set of predefined functions. The most important predefined functions are `send` and `recv` used for channel-based message passing.

`send` takes a channel as the first argument and a message as the second argument.

`recv` takes a channel as the first argument and an lval as the second argument. The message read from the channel will be placed in the lval. Note that this is an exception of how functions are called. It is the only case where we have call by reference (and only for the second argument).

Other useful predefined functions are `printf` and `length` that have their common syntax.

Expressions

Expressions (Figure 3.20) stand for values of various types. Each expression has a statically determined type and can only be used in places where values of that type are accepted. The process of computing the value of an expression in a context (state) is called *evaluating the expression*. Expressions are side-effect free, evaluating the same expression in the same context (state) leads to the same value.

```
expr ::= int_const | uint_const | bool_const
       |  char_const | string_const
       |  lval
       |  ('+' | '-' | '!' | '[]' | '<>' | 'X') expr
       |  expr ('+' | '-' | '*' | '/' | '%' | '&&' | '||' | '^' | '=>' | 'U' |
                '<' | '<=' | '>' | '>=' | '==' | '!=' | 'in') expr
       |  ('forall' | 'exists') identifier 'in' expr '::' expr
       |  ( expr )
       |  'mkarray' 'of' type_use '[' expr ']'
       |  'mkchan'  'of' type_use '[' expr ']'
       |  'ref' expr
       |  '{' identifier '=' expr [',' identifier '=' expr] '}'
       |  '{' [expr [',' expr]*] '}'
       |  '{' [expr ':' expr [',' expr ':' expr]*] '}'
       |  '{' expr '|' identifier 'in' expr '::' expr '}'
```

Figure 3.20: expressions in AAL

Lvals (Figure 3.21) describe how values stored directly in the program's memory can be accessed. They can be identifiers in which case the lval is the value associated with the name of a variable, fields in a record, indices in an array, sequence, or a map, or the contents of a reference cell.

```
lval ::= identifier
       |  lval '.' identifier
       |  lval '[' expr ']'
       |  lval '$'
```

Figure 3.21: lvals in AAL

Expressions are either constant values of various types (such as integers or characters), lvals, or various combinations of other expressions (such as unary or binary operations). Operators have their standard precedence (the one from languages like C).

Integer constants are introduced as sequences of digits. The sequence of digits can be preceded by the `-` sign. Some examples of integer constants are `-54` and `29`.

Unsigned integer constants are introduced as sequences of digits followed by the letter `u`. An example of an unsigned integer constant is `5429u`.

We use `false` and `true` for the two boolean constants.

Character constants are introduced by placing the character between `'` symbols. An example is `'a'`.

String constants are surrounded by `"` symbols. An example is "`hello world!`".

The more interesting cases of expressions are the ones used for introducing new references, arrays, channels, sets, sequences, maps, and set comprehensions. Unlike in other languages, by themselves, these expressions do not allocate memory, instead the corresponding memory allocation only happens when the expression is assigned to an lval either in a variable definition or in an assignment statement. If the expression is not immediately assigned to an lval then the corresponding value goes away and it is not stored anywhere (and there is no way to refer to it again).

Statements

Statements (Figure 3.22) are the basic units of computation in our language, they correspond to transition between states of the program.

We have two types of statements: *basic statements* and *compound statements*.

Basic statements correspond to assignments, variable definitions, function calls, assertions, and

```
stmt ::= lval [',' lval]* '=' expr [',' expr]* ';'
       |  var_def
       |  funcall ';'
       |  'break' ';' | 'continue' ';' | 'return' ';' | 'skip' ';'
       |  'assert' expr ';'
       |  '{' stmt* '}'
       |  'if' if_do_alt+ '::' 'else' '->' stmt+ 'fi'
       |  'do' if_do_alt+ '::' 'else' '->' stmt+ 'od'
       |  'sel' sel_alt+  '::' 'timeout' '->' stmt+ 'les'

funcall ::= [lval [',' lval]*] '=' identifier '(' [expr [',' expr]*] ')'

if_do_alt ::= '::' expr -> stmt+
sel_alt   ::= '::' identifier '(' [expr [',' expr]*] ')' -> stmt+
```

Figure 3.22: statements in AAL

various control changing statements (such as `break` and `return`).

Of the basic statements, the more interesting one is the assignment statement. It assigns a sequence of expressions to a sequence of lvals. The two sequences must have the same length and the type of an lval must match the type of the corresponding expression. Any occurrence of an lval from the right hand side in an expression on the left hand side evaluates to the value the lval had before the assignment statement. For example, the statement `a,b = b,a ;` effectively swaps the values of variables `a` and `b`.

The control changing statements stand for their usual counterparts in other languages.

Compound statements consist of blocks (sequences of statements) and branching statements (`if`, `do`, and `sel`). The `if` and `do` statements stand for nondeterministic choice and repetition and correspond to their usual counterparts in other languages (e.g., Promela) with some differences described below. As seen in Figure 3.22, these statements must always have an `else` branch.

Each branch in an `if` or `do` statement begins with an expression called the guard of the branch. This is different from Promela where each branch begins with a statement that could potentially change the state of the system.

Note that having guarded commands as statements would not make much sense in our language. Guarded commands are usually used for synchronization between processes. Each guard represents a condition on the global variables and the associated meaning is that execution stops until the

guard becomes true (for example, if some other process changes the value of some global variable that appears int the guard). The lack of global variables in our language rules out this use of guarded commands. If the intended meaning is that it is an error for the guard to ever be false, then a better way to express this is to use an `assert` statement. This is another reason for requiring an `else` branch for the `if` and `do` statements. It forces the developer to explicitly specify the intended meaning of the case when none of the guards are true: fall-through (`else -> skip;` or `else -> break;`) or error (`else -> assert false;`).

The more interesting compound statement is `sel`. It stands for nondeterministic selection of channel operations (send or receive). To some degree it resembles the `select` operation from the POSIX API. It also corresponds to an `if` statement from Promela that has only channel operations as first statements of its branches. As seen in Figure 3.22, the `sel` statement must always have a `timeout` branch.

3.6 Type System

The type system of AAL associates a type with each variable, each expression, and each function. There are two methods of defining a type system depending on whether references (also called locations) are considered to be values or not. The first method is to define a *dynamic type system* first that takes references into account and is parameterized by the contents of the heap and then specialize this type system into a *static type system* by considering the heap to be empty. The second approach would be to define the static type system directly. We have opted for defining a dynamic type system first because it also hints at what type-related checks have to be deferred to the runtime environment.

The type system is given as a set of well-typed judgments. These judgments provide the rules under which each syntactic form is well-typed. The goal is to define what it means for a program to be well-typed because only well-typed programs can be executed.

The easiest syntactic forms to type are constant values of primitive types. We define the *typeof* function that returns the type of a constant:

- $typeof(n) = int$, where n is an integer constant;
- $typeof(u) = uint$, where u is an unsigned integer constant;
- $typeof(b) = bool$, where b is a boolean constant;
- $typeof(chr) = char$, where chr is a character constant;
- $typeof(str) = string$, where str is a string constant.

For example, we have $typeof(\texttt{2}) = int$, and $typeof(\texttt{2u}) = uint$.

To type variables we need a type environment Γ which is a partial function that associates variable names to types:

$\Gamma : VarName \rightarrow Type^{?}$.

We use ϵ to denote the empty type environment: $\epsilon(x) = \perp$ for all $x \in VarName$.

We define some useful functions that operate on type environments.

Remove a variable from an environment:

$$(\Gamma - x)(y) = \perp \quad \text{if } x = y$$
$$= \Gamma(y) \quad \text{otherwise.}$$

Update an environment:

$$\Gamma[x = T](y) = T \quad \text{if } x = y$$
$$= \Gamma(y) \quad \text{otherwise.}$$

We also use the following alternative notations for updating a type environment: $\Gamma[x : T]$ and $\Gamma, x : T$.

The dynamic type system is parameterized by a heap H which is a partial function that associates locations to reference cells. A reference cell is a tuple $(owner, value)$ where $owner$ identifies the process that owns the reference cell and $value$ is the contents of the reference cell.

$H : Loc \rightarrow (ProcId \times Value)^{?}$.

Typing Judgement for Values

Values are not syntactic forms, but rather the result of expression evaluation. They can be thought of as a subset of expressions, those that cannot be evaluated further. To type expressions we must

first start with the type rules for values.

Because locations are values, the typing judgement is with respect to a heap H. The general form of the judgement is $H \vdash v : T$ meaning that in the context of heap H, value v has type T.

We define the judgment relation using a set of inference rules. More details on the notation are given in Appendix A.

We start by defining the rule for values that are constants of primitive types.

$$\frac{typeof(c) = T}{H \vdash c : T} \text{ value-constant}$$

The type of a record value is derived from the names and the types of its fields. Note that the names of the fields matter. Also, this rule implies that our type system has a structural typing rule. Two records that have the same names and types for the fields have equal types.

$$\frac{H \vdash v_i : T_i,\ i = 1, n}{H \vdash \{f_1 = v1, ..., v_n = v_n\} : f_1 : T1, ..., f_n : Tn} \text{ value-record}$$

A special value is used to denote the null reference. It is given the Null type to ensure that it cannot be used in expressions that involve accessing the contents of the reference.

$$\frac{}{H \vdash null : Null} \text{ value-null}$$

Next, we define the type of a reference. These rules are specific to the dynamic type system, and they are removed in the static type system since the static type system does not consider references to be values.

$$\frac{H(l) = (o, chan(cap, [v_1, ..., v_len])),\ cap > 0,\ len \leq cap \qquad H \vdash v_i : T,\ i = 1, len}{H \vdash l : chan[T]} \text{ value-chan}$$

$$\frac{H(l) = (o, ref\ v) \qquad H \vdash v : T}{H \vdash l : ref[T]} \text{ value-ref}$$

$$\frac{H(l) = (o, array[v_1, ..., v_n]) \qquad H \vdash v_i : T,\ i = 1, n}{H \vdash l : array[T]} \text{ value-array}$$

$$\frac{H(l) = (o, [])}{H \vdash l : seq[T]} \text{ value-empty-seq}$$

$$\frac{H(l) = (o, [v_1, ..., v_n]) \qquad H \vdash v_i : T,\ i = 1, n}{H \vdash l : seq[T]} \text{ value-seq}$$

$$\frac{H(l) = (o, \{\})}{H \vdash l : set[T]} \text{ value-empty-set}$$

$$\frac{H(l) = (o, \{v_1, ..., v_n\}) \qquad H \vdash v_i : T,\ i = 1, n}{H \vdash l : set[T]} \text{ value-set}$$

$$\frac{H(l) = (o, \{\})}{H \vdash l : map[T_1 : T_2]} \text{ value-empty-map}$$

$$\frac{H(l) = (o, \{k_1 : v_1, ..., k_n : v_n\}) \qquad H \vdash k_i : T1,\ i = 1, n \qquad H \vdash v_i : T2,\ i = 1, n}{H \vdash l : map[T_1 : T_2]} \text{ value-map}$$

Typing Judgement for Expressions

Next, we proceed to defining the typing rules for expressions. Since variables are expressions, the typing is done with respect to a type environment Γ. The general form of the judgment is $\Gamma, H \vdash e : T$ meaning that given the type environment Γ and the heap H the expression e has type T.

We start by defining two partial functions *DeltaUnop* and *DeltaBinop* that type the unary and binary operators of AAL.

We define the partial function $DeltaUnop : Unop \times Type \to Type?$, $DeltaUnop(unop, T_1) = T_2$ which given a unary operator *unop* and a type T_1 returns a type T_2. The meaning is that T_2 is the type of a unary operation expression that applies *unop* to a value of type T_1. Some examples:

$DeltaUnop(+, int) = int$

$DeltaUnop(-, int) = int$

$DeltaUnop(!, bool) = bool$

$DeltaUnop(+, bool) = \bot$

Similarly, we define the partial function

$DeltaBinop : Binop \times Type \times Type \to Type$, $DeltaBinop(binop, T_1, T_2) = T_3$ which given a binary operator *binop* and two types T_1 and T_2 returns a type T_3 that is the type of the binary operation expressions that applies *binop* to values of types T_1 and T_2. Some examples:

$DeltaBinop(+, int, int) = int$

$DeltaBinop(+, int, uint) = int$

$DeltaBinop(<, int, int) = bool$

$DeltaBinop(==, int, int) = bool$

$DeltaBinop(in, T, set[T]) = bool$

$DeltaBinop(in, T1, map[T1 : T2]) = bool$

Next we present the inference rules for the typing judgment for expressions.

If the expression is actually a value then we use the typing judgment for values.

$$\frac{H \vdash v : T}{\Gamma, H \vdash v : T} \text{ expr-val}$$

Variables are typed according to the type environment.

$$\frac{\Gamma(x) = T}{\Gamma, H \vdash x : T} \text{ expr-var}$$

$$\frac{\Gamma, H \vdash e_i : T_i,\ i = 1, n}{\Gamma, H \vdash \{f_1 = e_1, ..., f_n = e_n\} : \{f_1 : T_1, ..., f_n : T_n\}} \text{ expr-rec}$$

$$\frac{\Gamma, H \vdash e : f_1 : T_1, ..., f_i : T_i, ..., f_n : T_n}{\Gamma, H \vdash e.f_i : T_i} \text{ expr-field}$$

$$\frac{\Gamma, H \vdash e : T}{\Gamma, H \vdash ref\ e : ref[T]} \text{ expr-ref}$$

$$\frac{\Gamma, H \vdash e : ref[T]}{\Gamma, H \vdash e\$: T} \text{ expr-deref}$$

$$\frac{\Gamma, H \vdash e : \{f_1 : T_1, ..., f_i : T_i, ..., f_n : T_n\}}{e.f_i : T_i} \text{ expr-field}$$

$$\frac{\Gamma, H \vdash esz : int}{\Gamma, H \vdash mkchan \text{ of } T[esz] : chan[T]} \text{ expr-mkchan-int}$$

$$\frac{\Gamma, H \vdash esz : int}{\Gamma, H \vdash mkarray \text{ of } T[esz] : array[T]} \text{ expr-mkarray-int}$$

We have similar rules *expr-mkchan-uint* and *expr-mkarray-uint* for the case $\Gamma, H \vdash esz : uint$.

$$\frac{\Gamma, H \vdash: array[T] \qquad \Gamma, H \vdash idx : int}{\Gamma, H \vdash e[idx] : T} \text{ expr-array-index-int}$$

We have a similar rule*expr-array-index-uint*. We have similar rules *expr-seq-index-int* and *expr-seq-index-uint* for indexing in a sequence.

$$\frac{}{\Gamma, H \vdash [] : seq[T]} \text{ expr-empty-seq}$$

$$\frac{\Gamma, H \vdash e_i : T,\ i = 1, n}{\Gamma, H \vdash [e_1, ..., e_n] : seq[T]} \text{ expr-seq}$$

$$\frac{}{\Gamma, H \vdash \{\} : set[T]} \text{ expr-empty-set}$$

$$\frac{\Gamma, H \vdash e_i : T,\ i = 1, n}{\Gamma, H \vdash \{e_1, ..., e_n\} : set[T]} \text{ expr-set}$$

$$\frac{}{\Gamma, H \vdash \{\} : map[T]} \text{ expr-empty-map}$$

$$\frac{\Gamma, H \vdash ek_i : T_1,\ i = 1, n \qquad \Gamma, H \vdash ev_i : T_2,\ i = 1, n}{\{ek_1 : ev_1, ..., ek_n : ev_n\} : map[T1 : T2]} \text{ expr-map}$$

$$\frac{\Gamma, H \vdash e : T_1 \qquad DeltaUnop(binop, T_1) = T_2}{\Gamma, H \vdash unop(e) : T_2} \text{ expr-unop}$$

$$\frac{\Gamma, H \vdash e_1 : T_1 \qquad \Gamma, H \vdash e_2 : T_2 \qquad DeltaBinop(binop, T_1, T_2) = T_3}{\Gamma, H \vdash binop(e_1, e_2) : T_3} \text{ expr-binop}$$

Well-typed Statements

We define the inference rules for the judgements of well-typed statements and well-typed statement lists: $\Gamma, H, F \vdash wt\_stmt(s)$ and $\Gamma, H, F \vdash wt\_stmt\_list(sl)$. Where F is an environment matching function names to function types (the types of the arguments and the types of the return values).

$$\frac{}{\Gamma, H, F \vdash wt\_stmt\_list(\epsilon)} \text{ stmt-list-empty}$$

$$\frac{\Gamma, H, F \vdash wt\_stmt(s) \qquad \Gamma, H, F \vdash wt\_stmt\_list(sl)}{\Gamma, H, F \vdash wt\_stmt\_list(s\ sl)}\ \text{stmt-list-seq}$$

$$\frac{}{\Gamma[x = T], H, F \vdash wt\_stmt(var\ x : T;)}\ \text{stmt-var-def-uninit}$$

$$\frac{\Gamma, H \vdash e : T}{\Gamma[x = T], H, F \vdash wt\_stmt(var\ x : T = e;)}\ \text{stmt-var-def-init}$$

$$\frac{}{\Gamma, H, F \vdash wt\_stmt(skip;)}\ \text{stmt-skip}$$

$$\frac{}{\Gamma, H, F \vdash wt\_stmt(break;)}\ \text{stmt-break}$$

$$\frac{}{\Gamma, H, F \vdash wt\_stmt(continue;)}\ \text{stmt-continue}$$

$$\frac{}{\Gamma, H, F \vdash wt\_stmt(return;)}\ \text{stmt-return}$$

$$\frac{\Gamma, H \vdash e : bool}{\Gamma, H, F \vdash wt\_stmt(assert\ e;)}\ \text{stmt-assert}$$

$$\frac{\Gamma, H, F \vdash elhs_i : T_i,\ i = 1, n \qquad \Gamma, H, F \vdash erhs_i : T_i,\ lval(elhs_i),\ i = 1, n}{\Gamma, H, F \vdash wt\_stmt(elhs_1, ..., elhs_n = erhs_1, ..., erhs_n);}\ \text{stmt-assign}$$

$$\frac{\begin{array}{l}\Gamma, H, F \vdash ea_i : Ta_i,\ i = 1, n \\ \Gamma, H, F \vdash elhs_j : Tr_j,\ lval(elhs_j),\ j = 1, m \\ \Gamma, H, F \vdash f : (Ta_1, ..., Ta_n) \to (Tr_1, ..., Tr_m)\end{array}}{\Gamma, H, F \vdash wt\_stmt(elhs_1, ..., elhs_m = f(ea_1, ..., ea_n);}\ \text{stmt-fcall}$$

$$\frac{\Gamma, H \vdash e_i : bool \qquad \Gamma, H, F \vdash wt\_stmt\_list(sl_i) \qquad \Gamma, H, F \vdash wt\_stmt\_list(sl_e)\ i = 1, n}{\Gamma, H, F \vdash wt\_stmt(if :: e_1 \to sl_1 :: ... :: e_n \to sl_n :: else \to sl_e\ fi)}\ \text{stmt-if}$$

We have similar rules *stmt-do* and *stmt-sel*.

Other Well-typed Judgements

We have well-typed judgements for all higher level syntactic forms: functions, processes, modules, interfaces, schedulers, scheduler processes, and programs. Each of them is well-typed if all their subcomponents are well-typed. The goal is to have a definition of well-typed programs.

3.7 Semantics

3.7.1 Expression Evaluation

We define a big-step semantics for expression evaluation which is appropriate because in our language expressions are evaluated in a single step.

We define a partial function to_int that extracts an integer from a value.

$to\_int(n) = n$

$to\_int(x) = \bot$ for $x \notin constant$

We define similar functions for constants of other types (to_bool, etc.).

We define a partial function $delta\_unop$.

$delta\_unop(-, v) = -to\_int(v)$

$delta\_unop(!, v) = \neg to\_bool(v)$

We define a partial function $delta\_binop$.

$delta\_binop(+, v_1, v_2) = to\_int(v_1) + to\_int(v_2)$

For the cases where the binary operator is either `/` or `%` we have:

$delta\_binop(/, v_1, v_2) = to\_int(v_1)/to\_int(v_2)$ if $to\_int(v_2) \neq 0$

$delta\_binop(/, v_1, v_2) = \bot$ if $to\_int(v_2) = 0$

$delta\_binop(\%, v_1, v_2) = to\_int(v_1)$ % $to\_int(v_2)$ if $to\_int(v_2) \neq 0$

$delta\_binop(\%, v_1, v_2) = \bot$ if $to\_int(v_2) = 0$

Lemma 1. *If* $DeltaUnop(unop, T_1) = T_2$ *and* $H \vdash v_1 : T_1$ *then there exists a value* v *such that* $delta\_unop(unop, v_1) = v$ *and* $H \vdash v : T_2$.

Proof. By making sure that *delta_unop* and *DeltaUnop* match case by case. □

Lemma 2. *If* $DeltaBinop(binop, T_1, T_2) = T_3$ *and* $H \vdash v_1 : T_1$ *and* $H \vdash v_2 : T_2$*, then one of the following holds:*

1. *there exists a value* v *such that* $delta\_binop(binop, v_1, v_2) = v$ *and* $H \vdash v : T_3$*, or*
2. $binop = /$ *or* $binop = \%$*, and* $to\_int(v_2) = 0$*.*

Proof. By making sure that *delta_binop* and *DeltaBinop* match case by case.

The only cases where we have $H \vdash v_1 : T_1$ and $H \vdash v_2 : T_2$ such that $DeltaBinop(binop, T_1, T_2) = T_3$ and $delta\_binop(binop, v_1, v_2)$ is undefined are those where $binop = /$ or $binop = \%$, and $to\_int(v_2) = 0$. □

We define a variable environment $Rho : VarName \to Value^?$ that associates variable names to values.

We define the well-typed relation for variable environments with respect to type environments and heaps.

$$\frac{}{\epsilon, H \vdash \epsilon}\ \text{var-env-empty}$$

Note that ϵ is an empty environment that maps everything to $\bot$.

$$\frac{H \vdash v : T \qquad \Gamma, H \vdash Rho}{\Gamma[x = T], H \vdash Rho[x = v]}\ \text{var-env-var}$$

Next we prove the safety of variable environments.

Lemma 3. *If* $\Gamma, H \vdash Rho$ *and* $\Gamma(x) = T$ *then there exists a value* v *such that* $Rho(x) = v$ *and* $H \vdash v : T$*.*

Proof. Induction on the proof of $\Gamma, H \vdash Rho$. Details can be found in Appendix B.1. □

We define the semantics of expression evaluation using a partial function $Eval(e, Rho, H, Locs) = (v, Locs')$ where:

- e is the expression to be evaluated;
- *Rho* is an environment that maps variable names to values, *Rho* must associate a value for each variable used in e;

- H is a heap;
- $Locs$ and $Locs'$ are sets of pairs (l, v) that associate locations generated during expression evaluation to their corresponding values.

We define $Eval(e, Rho, H, Locs) = \perp$ for the cases where the expression e cannot be evaluated to produce a value (for example, in the case of a division by 0, or adding a string to an integer).

The execution of a statement evaluates each expression e in a single step by calling $Eval(e, Rho, H, \{\})$. The resulting $Locs'$ is a set of locations that are not part of the heap H. Expression evaluation does not modify the heap at all, it just generates a temporary set of locations. The locations in $Locs'$ will be added to the heap (and thus become permanent) only if the resulting value v is assigned to an l-value. This means that it is the assignment statement that modifies the heap. If v is not assigned to anything then the set of new locations is discarded.

For example, the statement `s = {1,2,3};` modifies the heap because the value of the expression `{1,2,3}` is assigned to `s`. The statement `if :: x in {1,2,3} -> ... fi` does not add any new locations to the heap for the value of the expression `{1,2,3}` because this value is not assigned to anything.

Some cases for expression evaluation are presented below, more details are given in Appendix B.2.

$Eval(c, Rho, H, Locs) = (c, Locs)$

$Eval(unop(e), Rho, H, Locs_1) = (delta\_unop(unop, v), Locs_2)$ where

$(v, Locs_2) = Eval(e, Rho, H, Locs_1)$

$Eval(ref\ e, Rho, H, Locs_1) = (l, Locs_2 \cup \{(l, ref\ v)\})$ where

$(v, Locs_2) = Eval(e, Rho, H, Locs_1)$

$l \notin H \cup Locs_2$

Next, we state a type safety theorem for expression evaluation. It states that if an expression can be typed then it evaluates to a value and moreover, the value has the same type as the expression. The only exception is when the expression contains a "division by zero" error. This error is not caught by the type system, but it can be caught by a dynamic analysis (testing or model checking).

Definition 1. An expression e in the context of a type environment Γ, a variable environment Rho, and a heap H, contains a *division by zero* if and only if e contains a sub-expression $binop(e_1, e_2)$ with $Eval(e_1, Rho, H, Locs_1) = (v_1, Locs_2)$ and $Eval(e_2, Rho, H, Locs_2) = (v_2, Locs_3)$ and $binop = /$ or $binop = \%$, and $to\_int(v_2) = 0$.

Theorem 1. *If* $\Gamma, H \vdash e : T$ *and* $\Gamma, H \vdash Rho$ *then one of the following holds:*

1. *there exists a value* v *such that* $Eval(e, Rho, H, Locs_1) = (v, Locs_2)$ *and* $H \cup Locs_2 \vdash v : T$*, or*
2. e *contains a division by zero in the context of* Γ*,* Rho*, and* H*.*

Proof. Structural induction on e. A sketch of the proof is given in Appendix B.3. □

3.7.2 Small-step Operational Semantics

In this section we present a small-step operational semantics for our language. A small-step semantics is required to capture how the execution can interleave statements from multiple processes.

3.7.3 Transition Systems

Every system that can be described as a transition system $TS(P) = (S, Act, \rightarrow, s_0)$ where:

- S is the set of states;
- Act is the set of actions;
- $\rightarrow$ is a subset of $S \times Act \times S$ and is the transition relation;
- s_0 is the initial system state.

An action a is a pair (p, i) where p is the process instance taking the transition, and i is 1 for most types of transitions, except for if, do, and sel where it is the number of the branch that is taken.

The small-step operational semantics associated with the transition system is that the execution starts in the initial system state s_0 and from there it moves to new states by following the transition relation $\rightarrow$.

3.7.4 State Representation

A state $s \in S$ is a tuple $(H, ls_1, ..., ls_p)$ where:

- H is the heap (as previously defined in the type system section);
- p is the number of process instances;
- ls_i is the local state of process instance i.

A local state ls is a tuple $(Rho, stack)$ where:

- Rho is a variable environment corresponding to the process-local variables;
- $stack$ is a list of frames corresponding to the call stack of the process.

A frame f corresponds to a function call and is a tuple $(Rho_f, sl, conts)$ where:

- Rho_f is a variable environment corresponding to the function-local variables;
- sl is a sequence of statements;
- $conts = ((break\_cont, continue\_cont)list)$ is a stack of pairs of sequences of statements;
- $break\_cont$ is the continuation for a break statement (intuitively: where to jump after a break statement);
- $continue\_cont$ is the continuation for a continue statement (intuitively: where to jump after a continue statement).

The Initial System State

The initial system state is obtained from the last configuration defined in the program.

For each module instance mi defined in the configuration, and for each active process of mi, there is a corresponding process instance in the initial system state. The set of active processes of a module instance mi that instantiates a module m is obtained from the code of m.

The local state ls_i of a process instance i is composed of Rho_i, the process-local variables of i, and $stack_i$, the call stack of i. The call stack $stack_i$ has only one frame that corresponds to the

`run` function of process i. The frame for the `run` function contains the body of the function as the sequence of statements and an empty stack of continuations.

The heap part of the initial system state contains, for each module instance mi, all the channels declared at module level. Also, the initial heap contains all the references created by the definitions of process-local variables of each process instance i.

3.7.5 The Transition Relation

We define the transition relation using the judgment $P \vdash s_1 \xrightarrow{(p,i)} s_2$ where P is the program, and $(s_1, (p,i)s_2) \in \rightarrow$ is a transition from state s_1 to state s_2 by taking the action (p,i). We show only a subset of the rules defining the transition relation. The ones not presented here follow a similar pattern.

$$\frac{Eval(e, Rho_f \cup Rho_i, H, \{\}) = (v, Locs),\ 1 \leq i \leq p}{\begin{array}{l} P \vdash (H, ..., (Rho_i, (Rho_f, var\ x : T = e;\ sl_i, conts)\ rest), ...) \xrightarrow{(i,1)} \\ \quad (H \cup (i, Locs), ..., (Rho_i, (Rho_f[x = v], sli, conts)\ rest), ...) \end{array}} \text{ var-def-init}$$

Defining a variable x, adds an entry in the current function's variable environment, that maps x to the value v to which e evaluates.

$$\frac{Eval(e, Rho_f \cup Rho_i, H, \{\}) = (v, Locs),\ 1 \leq i \leq p \qquad Rho_f(x)\neg = \perp}{\begin{array}{l} P \vdash (H, ..., (Rho_i, (Rho_f, x = e;\ sl_i, conts)\ rest), ...) \xrightarrow{(i,1)} \\ \quad (H \cup (i, Locs), ..., (Rho_i, (Rho_f[x = v], sli, conts)\ rest), ...) \end{array}} \text{ assign-fun-var}$$

Assigning a function-local variable modifies the current's function variable environment to reflect the assignment.

We have a similar rule for assigning a process-local variable.

$$\frac{\begin{array}{l} P \vdash fun\_def(g, (x_1, ..., x_n), (), body \\ Eval(e_j, Rho_f \cup Rho_i, H, Locs_{j-1}) = (v_j, Locs_j),\ j = 1, n \end{array}}{\begin{array}{l} P \vdash (H, ..., (Rho_i, (Rho_f, g(e_1, ..., e_n);\ sl_i, conts)\ rest), ...) \xrightarrow{(p,1)} \\ \quad (H \cup (i, Locs_n,), ..., (Rho_i, (\epsilon[x_j = v_1, j = 1, n], body, \epsilon)(Rho_f, g(e_1, ..., e_n);\ sli, conts)rest, ...) \end{array}} \text{ funcall-no-assign}$$

Calling a function g that does not return any values, creates a new frame for g in which the variable environment maps the input arguments to the values of the expressions passed to g. The

body of g is uses as the statements for this new frame, and the stack of continuations is empty. The frame for f (the function calling g) is left unmodified. The execution of a return statement from g will also modify the frame for f by advancing to the next statement.

Next we present a lemma that allows us to encode scheduling decisions as actions.

Lemma 4. *The transition system is* action-deterministic*: Given a state s_1 and an action (p, i) there exists at most one state s_2 such that $s_1 \xrightarrow{(p,i)} s_2$.*

Proof. Follows directly from the structure of the rules for the transition relation. □

3.7.6 Semantics in the Presence of a Scheduler Process

The above semantics changes slightly in the presence of scheduler processes.

The system state is split into two disjoint parts: the proper system state and the state of the scheduler process. The execution alternates between the scheduler process and the system being scheduled.

The execution starts in the initial state of the scheduler process and continues executing statements from the scheduler process until the control is explicitly passed to the system by calling the `set_next_actions` primitive. At this point the control switches to the system which can pick for execution any of the actions specified by the scheduler process. After an action is taken, the control passes back to the scheduler process, right after the call to `set_next_actions`. The above sequence repeats until the scheduler process terminates.

Chapter 4

Design-Aware Analysis

4.1 Interactive Analysis

We have implemented a framework that supports direct execution and analysis of specifications written in AAL. Developers interact with our framework through an Integrated Development Environment (IDE). Besides the common source code editing functionalities such as syntax highlighting, the IDE ties a series of analyses directly into the development process. The analyses include syntax checking, type checking, various static analyses, testing, and model checking.

Each time the source code is saved the IDE starts the analyses in the background, invisible to the user. Errors are flagged as soon as they are discovered. For each error detected by the analyses, the IDE highlights the line(s) of code that caused the error. Moving the mouse over a highlighted line provides a short description of what went wrong. Clicking on the line brings up a more detailed description of the error. We call this approach "interactive analysis."

Running the analyses on each save has the advantage of discovering errors as soon as they are introduced. The development cannot proceed without fixing all reported errors. It is common practice that developers save their source code often and thus the number of reported errors is usually small. This is in contrast to the traditional approach of "post-mortem analysis" where the analysis is triggered manually, usually in a different environment than the IDE, only after the software is fully developed. "Post-mortem analysis" usually floods the developer with lots of errors and warnings making it unclear when the errors where introduced and how to start fixing them.

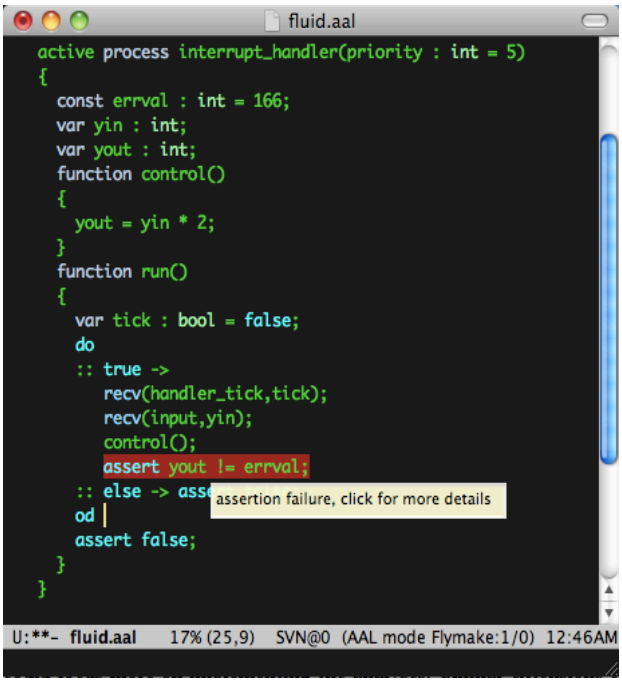


Figure 4.1: screen shot of the IDE

We have implemented the IDE as an Emacs mode by extending the existing flymake [3] mode which allows custom actions to be triggered on each save. We have picked Emacs because it is a popular editor and we are familiar with it. Moreover, since Emacs already provides the text editing functionality, we could focus more on implementing the analyses. Similar extensions could be made to other popular editors such as Vi. Figure 4.1 shows a screen shot of the IDE.

As developers save their source code before it is fully written, interactive analysis has to work on partially-written software. For some analyses, like syntax checking, type checking, and static analysis, this is commonly not a problem because they are static and do not require code execution which makes them applicable to partially-written software. For other analyses, like testing and model checking, interactive analysis is more challenging because they are dynamic and require code execution. We need a method to close partially-written systems and make them executable.

Our solution takes advantage of the modularity of the design, thus making the analyses design-aware. Often large systems are split into smaller components with the developer working on a single component at a time. To be able to analyze a component in isolation, because the other components that interact with it might not be implemented yet, our analyses automatically construct test drivers

for each component. A test driver and the corresponding component form a closed system that can be executed. We are also performing a symbolic execution that follows the same paths as the concrete run of a test driver. The analysis tools use the information gathered during the symbolic execution to guide future concrete runs in a way that exposes new behaviors of the analyzed component.

The following sections describe each of our main analyses (type checking, static analysis, testing, and model checking) in more detail. Syntax checking is not discussed since it is trivially done during source code parsing.

4.2 Type Checking

Type checking ensures that the language is used in a consistent way according to the semantics in Section 3.7. The analysis is done statically at compile-type without running the code. If the type checker does not detect any errors then the program is guaranteed to be well-typed according to the static type system described in 3.6. The code cannot execute if it does not pass the type checker. This allows us to remove most type checks at run time. Some type checks are still required at run time and they are required to ensure that the program is well-typed according to the dynamic type system (3.6). The checks that are left to the run-time have to do with inference rules that impose restrictions on values (bounds, ownership) because these values are not known statically.

The type checking algorithm tries to construct a proof that the program is well-typed. It is implemented as a recursive traversal of the program's Abstract Syntax Tree (AST). This maps directly to the tree structure of well-typed proofs. If the type checker is not able to construct a proof then it is guaranteed that no such proof exists and thus the program is not well-typed.

Type checking relies on a parsing phase that builds all the necessary symbol tables. A symbol table associates names to definitions and corresponds to an environment in the semantics. Instead of having just one global symbol table we have opted for multiple ones at the level of AST nodes where the definitions are introduced. For example, for a program P, $P.modules$ associates module names to the modules defined in program P, for a module M, $M.functions$ associates function names to functions defined in module M, and so on. We have opted for this approach because it simplifies

symbol lookup and makes the correspondence with the semantics more clear.

When type checking an AST node that can introduce type aliases (e.g., type definitions in modules and processes) the first step of the algorithm is to ensure that all declared type aliases are *well-defined*. A type alias is *well-defined* if it can be mapped to a unique *canonical type*. This is achieved by performing a depth first search exploration on the type alias directed graph $TAG = (TAV, TAE)$. The nodes in TAV consist of types (both type aliases and canonical types). There exists a directed edge $(t_1, t_2) \in TAE$ from type t_1 to type t_2 if the definition of t_1 uses t_2. At the end of this step types are replaced with their canonical form. An error is reported at the end of the exploration if any type aliases are left undefined. The exploration also finds bad cases of recursively defined types (recursive types with no base case). For example, the declarations and definitions shown in Figure 4.2 are allowed by the syntax, but are not correct. The problem is that no value of either type `t` or type `t1` can be constructed because there are no base cases for the recursion.

```
type t;
type t1 = t;
type t = t1;
```

Figure 4.2: incorrect recursive type declarations and definitions

An example of a correct recursive type definition is given in Figure 4.3. This happens to be the definition of a linked list of integers. This works because references can be used as base cases.

```
type t ;
type t1 =
{
  x : int,
  n : ref[t]
} ;
type t = t1 ;
```

Figure 4.3: correct recursive type declarations and definitions

The type checking algorithm recursively traverses the AST and tries to apply a typing rule at each node. There is a type checking function for each kind of node which makes the correspondence with the semantics more clear. The algorithm also infers the types of all expressions in the program and guarantees that on success each expression has a unique type.

It is important to notice that for most AST nodes, because of their syntactic form, there is only one typing rule from the semantics that is applicable. The exceptions are the constants "[]" and "{}" which correspond to the empty sequence, empty set, and empty map. Because the type of the elements are unknown the type of the sequence, set, or map cannot be inferred without extra information. In the semantics this corresponds to the typing rules for empty sequences, sets, or maps which do not impose any restrictions on the type of the elements (the type of the elements does not appear in a premise of the typing rule). The type checking algorithm introduces a temporary type for them: *empty_seq* or *empty_set_map*. The temporary types of these constants are replaced with actual types based on the context in which the constants are use (i.e., in the definition of a variable, in an assignment or in a function call). This is possible because in our language all variable definitions have type annotations, the type of each variable is explicitly mentioned in the code and does not have to be inferred. For example, from the definition `var x:set[int] = {};` it can be inferred that in this context "{}" has type "$set[int]$".

Type Checking Programs

The type checking algorithm for programs is shown in Figure 4.4. The functions for type checking interfaces, schedulers, scheduler processes, and configurations are not presented here because they follow a similar structure to the one for type checking modules: the result is a conjunction of the type checking results for each of their components. This part shows how the algorithm reaches the more interesting part of type checking statements.

Some type checking functions take as an extra argument a stack of type symbol tables. This is required because, for example, the types in a process could be defined in terms of the types in the parent module and the well-defined check has to include them in the graph it constructs.

Type Checking Statements

Figure 4.5 show how statements are type checked based on their syntactic form. Trivial statements that have no components are handled immediately. The checking of other statements is handed to

```
function type_check_program(P:Program) : (result : bool)
{
  result = and (type_check_interface(I) for I in P.interfaces)
           and (type_check_module(M) for M in P.modules)
           and (type_check_sched(S) for S in P.scheds)
           and (type_check_sched_proc(SP) for SP in P.schedprocs)
           and (type_check_config(C) for C in P.configs);
}
function type_check_module(M : Module) : (result : bool)
{
  result =     check_defined_types([], M.types)
           and (type_check_constant(C) for C in M.constants)
           and (type_check_function(F) for F in M.functions)
           and (type_check_process([M.Types],P) for P in M.processes);
}
function type_check_process(envs : seq[TypeSymTbl], P : Process) : (result : bool)
{
  result = and (type_check_var(V) for V in P.parameters)
           and (type_check_var(V) for V in P.locals)
           and (type_check_function(F) for F in P.functions)
}
function type_check_function(F : Function) : (result : bool)
{
  result = and (type_check_var(V) for V in F.arguments)
           and (type_check_var(V) for V in F.returns)
           and (type_check_stmt(S) for S in F.body);
}
```

Figure 4.4: the type checking algorithm for programs

```
function type_check_stmt(S : Statement) : (result : bool)
{
  switch(S)
  {
    case Skip:            result = true;
    case Break:           result = true;
    case Continue:        result = true;
    case Return:          result = true;
    case VarDef v:        result = type_check_var_def(v);
    case Assign a:        result = type_check_assign(a);
    case UserFcall fc:    result = type_check_user_fcall(fc);
    case BuiltinFcall fc: result = type_check_builtin_fcall(fc);
    case If i:            result = type_check_if(i);
    case Do d:            result = type_check_do(d);
    case Sel s:           result = type_check_sel(s);
  }
}
```

Figure 4.5: the type checking algorithm for statements

specialized functions. Note how each of these cases corresponds to a well-typed statement rule in the type system (Section 3.6).

Figure 4.6 shows how assignment statements are type checked. First it is checked that the number of expressions on the left hand side of the assignment is equal to the number of expressions on the right hand side. Next, for each pair of left hand side and right hand side expressions it is checked if they have the same types. This is implemented in the function $unify\_types$ which takes a temporarily typed expression e and a type t and returns true if the expression can be typed to t. It structurally checks if $e.type$ is equal to t with special cases for empty sequences, set, and maps because, as described earlier, their types could not have been inferred until the context in which they are used becomes clear. The assignment statement is one of such contexts. The $unify\_types$ also sets $e.type$ to t in case of success.

```
function type_check_assign(a:Assign) : (result : bool)
{
  // a has the syntactic form el1,...,eln = er1,...,erm;
  var partial : bool = false;
  result = true;
  if n != m { result = false; return;}
  for i in 1,n
  {
    partial = type_check_expr(eli) and
              type_check_expr(eri);
    if not unify_types(eri,eli.type) { partial = false; }
    if not partial { result = false; }
  }
}
```

Figure 4.6: the type checking algorithm for assignments

Figure 4.7 shows how variable definitions are typed checked. It is similar to how assignments are checked with the exception that the type for the left hand side does not have to be inferred because it is already given as a type annotation for the variable.

Figure 4.8 shows how function call statements are type checked. First it ensures that the right number of arguments were passed to the function and that the result is captured in the right number of l-value expressions. Next, it checks that the arguments passed to the function have the same type as the as the formal parameters of the function. Like for assignment, type unification can happen

```
function type_check_var_def(v : Var) : (result : bool)
{
  // v has the syntactic form: var v : T = e;
  v.type = T;
  result = type_check_expr(e);
  if not unify_types(e,v.type) { result = false; }
}
```

Figure 4.7: the type checking algorithm for variable definitions

here for the expressions used as arguments. The final step is to ensure that the expressions used to capture the results have the appropriate types.

```
function type_check_user_fcall(fc : UserFcall) : (result : bool)
{
  // fc has the syntactic form: el1,...,eln = f(e1,...,em) where
  // f is a function f(va1:Ta1,...,vap:Tap) : (vr1:Tr1,...,vrq:Trq)
  // assumes that the declaration of f was already type checked
  if not n == q { result = false; return; }
  if not m == p { result  = false; return; }
  var partial : bool = false;
  result = true;
  for i in 1,m
  {
    partial = type_check_expr(ei) ;
    if not unify_types(ei,Tri) { partial = false; }
    if not partial { result = false; }
  }
  for i in 1,n
  {
    partial = type_check_expr(eli);
    if eli.type != Tri { partial = false; }
    if not partial { result = false; }
  }
}
```

Figure 4.8: the type checking algorithm for user defined function calls

Figure 4.9 shows how `do` statements are typed checked. Each subcomponent is type checked and also it is checked that each guard has type bool. The type checking functions for `if` and `sel` statements have a similar structure and are not presented here.

Type Checking Expressions

Figure 4.10 shows how expressions are type checked based on their syntactic form. Note how these cases correspond to typing rules in the type system (Section 3.6). This is the point in the algorithm

```
function type_check_do(d : Do) : (result : bool)
{
  // d has the syntactic form:
  // do
  // :: guard1 -> stmt1
  // ...
  // :: ... guardn -> stmtn
  // :: else -> stmte
  // od
  result = and (type_check_expr(guardi) and guardi.type == bool
                and type_check_stmt(stmti) for i in 1,n)
           and type_check_stmt(stmte);
}
```

Figure 4.9: the type checking algorithm for if statements

where the types of expressions are inferred.

Figure 4.11 shows how binary operations are type checked. $DeltaBinop(binop, T_1, T_2)$ is an implementation of the function with the same name from the type system. Given a binary operator and the types of the arguments it computes the type of the result if the binary operator is defined for values of T_1 and T_2 and returns $undefined$ otherwise.

Correctness of The Type Checking Algorithm

Lemma 5. *For any expression e that does not contain "[]" or "{}", $type\_check\_expr(e)$ with $e.type = T$ if and only if $\Gamma| - e : e.type$ where $e.type$ is the type inferred and where Γ is the environment that maps variable names used in the expression e to types.*

Proof. We proceed by structural induction on the expression e. We give a sketch of the proof and show only some of the cases. The other cases follow the same structure.

Case 1: e is a constant c.

The function $type\_check\_expr(e)$ (Figure 4.10) executes the case for a constant. The result is always be $true$ and $e.type = typeof(c)$.

Since constants are values, we identify the following candidate rule from the type system (where e is v):

```
function type_check_expr(e:Expr) : (result : bool)
{
  switch(e)
  {
    case Constant c:     result = true; e.type = typeof(c);
    case EmptySeq s:     result = true; e.type = empty_seq;
    case EmptySetMap sm: result = true; e.type = empty_set_map;
    case Record r:       result = type_check_expr_rec(r);
                         if result { e.type = r.type; }
    case Chan c:         result = type_check_expr_chan(c);
                         if result { e.type = c.type; }
    case Ref r:          result = type_check_expr_ref(r);
                         if result { e.type = r.type; }
    case Array a:        result = type_check_expr_array(a);
                         if result { e.type = a.type; }
    case Seq s:          result = type_check_expr_seq(s);
                         if result { e.type = s.type; }
    case Set s:          result = type_check_expr_set(s);
                         if result { e.type = s.type; }
    case Map m:          result = type_check_expr_map(m);
                         if result { e.type = m.type;}
    case Unop u:         result = type_check_expr_unop(u);
                         if result { e.type = u.type;}
    case Binop b:        result = type_check_expr_binop(b);
                         if result { e.type = b.type; }
    case Quant q:        result = type_check_expr_quant(q);
                         if result { e.type = q.type; }
    case SetComp sc:     result = type_check_expr_set_comp(sc);
                         if result { e.type = sc.type; }
    case Var v:          e.type = v.type;
    case Deref d:        result = type_check_expr_deref(d);
                         if result { e.type = d.type; }
    case Field f:        result = type_check_expr_field(f);
                         if result { e.type = f.type; }
    case Index idx:      result = type_check_expr_index(idx);
                         if result { e.type = i.type; }
  }
}
```

Figure 4.10: the type checking algorithm for expressions

$$\frac{\vdash v : T}{\Gamma \vdash v : T}\ \text{expr-val}$$

The rule has the premise $\vdash v : T$, and we know that in this case v is a constant c, so the candidate rule for typing values is:

$$\frac{typeof(c) = T}{H \vdash c : T}\ \text{value-constant}$$

```
function type_check_expr_binop(b:Binop) : (result : bool)
{
  // b has the syntactic form binop(e1,e2)
  var partial : bool = false;
  result = type_check_expr(e1);
  partial = type_check_expr(e2);
  if not partial { result = false; }
  var t : Type = DeltaBinop(binop,e1.type,e2.type);
  if t is undefined
  { result = false; }
  else
  { b.type = t; }
}
```

Figure 4.11: the type checking algorithm for expressions

"⇒": $(type\_check\_expr(e) \wedge e.type = T) \Rightarrow \Gamma \vdash e : T$

Because $e.type = T$ and $e.type = typeof(c)$ we have $typeof(c) = T$ which is exactly the premise of the rule. This makes the rule applicable and thus we can construct a proof for $\Gamma| - e : T$.

"⇐": $\Gamma \vdash e : T \Rightarrow (type\_check\_expr(e) \wedge e.type = T)$

The only proof tree that can be constructed for a constant c is the one shown above which imposes the restriction $T = typeof(c)$. The $type\_check\_expr$ function always evaluates to $true$ for constants and infers $e.type = typeof(c)$. This implies $e.type = T$.

Case 2: e is a binary operation b having the syntactic form $binop(e_1, e_2)$.

The function $type\_check\_expr(e)$ executes the case for a binary operation. Its result is the same as the result of $type\_check\_expr\_binop(b)$ and in case of success $e.type$ is $b.type$.

The only candidate rule for typing binary operations is:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad DeltaBinop(binop, T_1, T_2) = T}{\Gamma \vdash binop(e_1, e_2) : T}\ \text{expr-binop}$$

"⇒": $(type\_check\_expr(e) \wedge e.type = T) \Rightarrow \Gamma \vdash e : T$

The expressions e_1 and e_2 are smaller than e and by the inductive hypothesis we have the proofs for:

$(type\_check\_expr(e_1) \wedge (e_1.type = T_1)) \equiv \Gamma| - e_1 : T_1$ and

$(type\_check\_expr(e_2) \wedge (e_2.type = T_2)) \equiv \Gamma| - e_2 : T_2$.

The premise states that $type\_check\_expr(e)$ is true which happens only if $type\_check\_expr(e_1)$ and $type\_check\_expr(e_2)$ are both true and moreover $DeltaBinop(binop, e_1.type, e_2.type)$ has to be defined which implies $e_1.type = T_1$ and $e_2.type = T_2$ ($e_1.type$ and $e_2.type$ have to be defined).

$e.type$ ends up being $DeltaBinop(binop, e_1.type, e_2.type)$. The premise also states that $e.type = T$ so we have $DeltaBinop(binop, e_1.type, e_2.type) = T$.

Because $type\_check\_expr(e_1)$ is true and $e_1.type = T_1$ we have $\Gamma \vdash e_1 : T_1$ and $\Gamma \vdash e_2 : T_2$.

All the premises for the typing rule are true which makes the rule applicable and thus we can infer $\Gamma \vdash e : T$.

"$\Leftarrow$": $\Gamma \vdash e : T \Rightarrow (type\_check\_expr(e) \wedge e.type = T)$

The only typing rule that can be applied to type binary operations is the *expr-binop* rule presented above. For the rule to be applicable all its premises must hold. This means we have $\Gamma \vdash e_1 : T_1$, $\Gamma \vdash e_2 : T_2$, and $DeltaBinop(binop, T_1, T_2) = T$.

Using the induction hypothesis we get:

$type\_check\_expr(e_1) \wedge (e_1.type = T_1)$ and $type\_check\_expr(e_2) \wedge (e_2.type = T_2)$.

Looking at the type checking function for binary operations we can conclude that

$type\_check\_expr(e)$ is true and $e.type = T$.

The other cases of the proof can be carried out in a similar fashion.

□

Lemma 6. *If type checking of a statement s succeeds then all the expressions that are part of the statement are well-typed.*

Proof. The proof is by structural induction on the statement s. The only interesting cases are those for variable definition, assignment and function call where type unification can happen. In all these cases the expressions that had temporary types are given well-defined types. □

Lemma 7. *For a statement s $type\_check\_stmt(s)$ if and only if $\Gamma \vdash wt\_stmt(s)$ where Γ associates variable names to types for all variables that appear in s.*

Proof. The proof is by structural induction on s and has a similar structure as the one for expressions.

□

Similar lemmas are introduced for processes, modules, interfaces, configurations, schedulers, and scheduler processes. They make use of the fact that for each of these syntactic forms, we have only one well-typed inference rule.

Theorem 2. *Correctness of the type checking algorithm: For any program P* $type\_check\_program(P)$ *if and only if* $wt\_program(P)$.

Proof. The theorem can be proved from the lemmas for modules, interfaces, configurations, schedulers, and scheduler processes and from the fact that we have only one inference rule for $wt\_program(P)$.

□

4.3 Static Analysis

We have built a static analysis framework for AAL that allows us to implement a series of checks that detect code use patterns likely to indicate the presence of errors. Some examples of such code use patterns are using an uninitialized variable, defining a variable that is never used, assigning to an input variable, etc. Static analysis checks are meant to be run on well-typed programs and the IDE runs them only if the type checking algorithm presented in the previous section succeeds.

The framework is extensible and allows users to define their own checks. Each check is implemented as a static analysis rule in a separate file. These files containing the rules are placed in a common folder from where the static analysis framework applies them in sequence.

Traditionally, static analyzers start by building a *control flow graph* for each function. Next, the analysis proceeds function by function, repeatedly exploring its control flow graph to annotate each node with information relevant to the analysis. The computation of the annotations stops when no new information can be inferred (it is a fixed point computation). Finally, the static analysis results are computed from the annotations. The fixed-point computation is usually explicitly implemented

in an imperative language and the information to be inferred is defined using a domain-specific language particular to the analyzer.

We take a similar overall approach, but deviate from the common practice by mapping the static analysis problem to an existing rule-based engine. Instead of defining our own domain-specific language for writing the static analysis rules, each such rule is implemented as a program in the language of the rule-based engine. A program written for a rule-based engine consists of the definition of a set of initial facts ($Facts$) and the definition of a set of rules ($rules$) used to infer new facts ($Facts'$) based on the existing ones. The engine then repeatedly applies the rules until no new facts can be inferred ($Facts' == Facts$). It can easily be seen that this is a fixed-point computation $Facts' = rules(Facts)$ which makes the rule-based engine an attractive framework for implementing static analyzers.

4.3.1 The Control Flow Graph

The *control flow graph* (CFG) $G = (V, E)$ encodes the order in which statements from a function are executed and is a key concept used in many static analyses. The CFG of a function f is built from the AST of function f. The nodes in this graph consist of the statements from the body of the function with the addition of two extra nodes representing the entry and exit points of the function. The entry node stands for the statement in the parent function that called the current function, and the exit node stands for the statements in the parent function that immediately follow the function call. There exists an edge $(n_1, n_2) \in E$ whenever n_2 can immediately follow n_1 in some execution of the function.

The control flow graph is required for writing *flow-sensitive* analyses in which the order of the statements matters. An analysis is called *flow-insensitive* if it is not flow sensitive.

4.3.2 Static Analysis Framework as a Rule-based Engine

The static analysis framework is built around the CLIPS (C Integrated Production Language) rule-based engine [69]. There are other candidate rule-based engines such as Drooles [68] and ILOG JRules

[5], but we have selected CLIPS because it can be easily integrated in our code base. Moreover, it can be extended to add new built-in functionality.

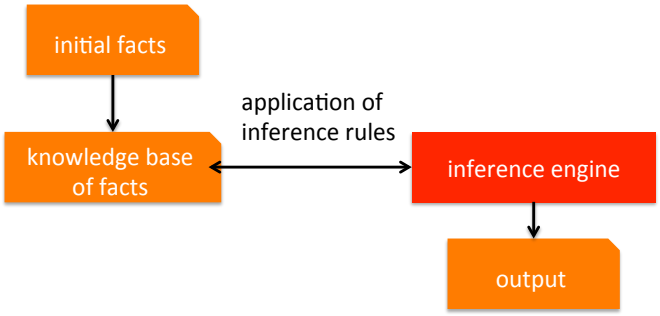


Figure 4.12: the CLIPS rule-based engine

Figure 4.12 shows how the CLIPS rule-based engine works. It maintains a knowledge base of facts that is first populated with the initial facts. During execution, CLIPS repeatedly applies the set of inference rules that can add new facts to the knowledge base. The execution stops when no new rules can be applied to further change the knowledge base. The rules can query the knowledge base of facts to produce output by writing either to the standard output, or to a file.

This mode of operation is called *forward chaining* because it infers all new facts from existing facts and rules. This is in contrast to *backward chaining* that answers queries by trying to prove them. Examples of systems that perform backward chaining include Datalog [1] and Prolog [4]. It can be argued that forward chaining can produce facts that are irrelevant to answering the user's queries. This effect is minimised in our case because, in general, static analysis rules query information about all the statements in a function and thus all the generated facts tend to be relevant. To avoid duplicate work, it is useful to infer all the facts upfront and then ask the queries instead of constructing a proof that answers queries one at a time.

The architecture of our static analysis framework is shown in Figure 4.13. The first step is to encode the AST and the CFGs in the language of CLIPS. This encoding provides CLIPS with the set of initial facts which have to fit in predefined templates. Next, the framework picks a file defining a static analysis rule and feeds it to the CLIPS engine. CLIPS starts running the program associated with the static analysis rule and produces as output the set of detected errors. When the

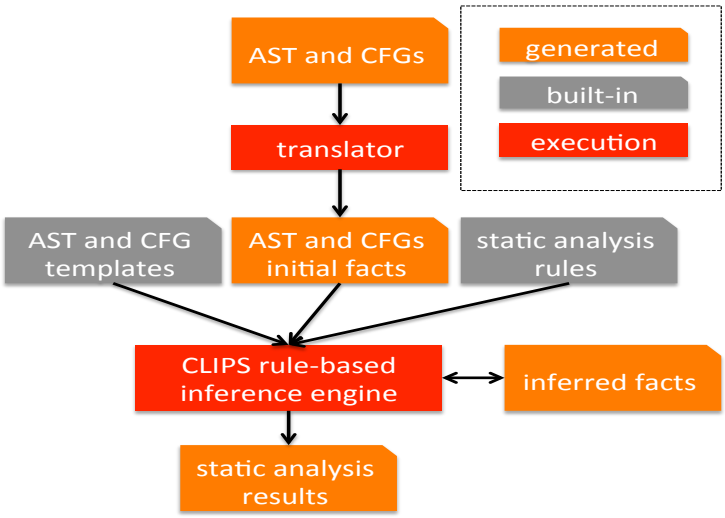


Figure 4.13: the architecture of the static analysis framework

computation finishes, the output is collected and passed to the IDE so that it can be presented to the user. Then the process is repeated for the next static analysis rule, and so on, until all rules have been applied. An optimization is to apply this process to a function at a time. This keeps CLIPS's knowledge base small because it is populated only with facts relevant to the current function.

To extend the framework with a new static analysis rule, the user has to implement it in the language of CLIPS, save it to a file, and place this file in the folder from where the framework picks the rules.

One advantage of using CLIPS for defining static analysis rules is that the language is Turing-complete which allows us to define complex checks. This means that it is possible for a check to not terminate. If the execution of a static analysis rule takes longer than a preset amount of time, then the analysis is stopped and an error is flagged.

4.3.3 The CLIPS Language

CLIPS is a multi-paradigm language with a LISP-like syntax that includes procedural, object oriented, and rule-based features. Here we focus on the rule-based subset because it is particularly useful for writing static analysis rules.

Values include symbols, numbers, and strings.

Relations are encoded as facts. Each fact has a name and a content. For example, `(language AAL)` encodes that the symbol *AAL* is part of the relation *language*. The contents of a fact does not have to be a single value, it can be a sequence of values. For example, `(language AAL 0.1)` could be used to state that the language *AAL* has version 0.1.

It is useful to name the values contained in a fact so that their meaning is more clear. This is achieved by defining templates. A template encodes the structure of a fact. It has a name and a list of named slots, with each slot containing a value. For example,

```
(deftemplate language
    (slot name)
    (slot version))
(language (name AAL) (version 0.1))
```

can be used to encode the same relation. Templates resemble the definitions of structures in other languages (like C) with facts being instances of these structures. Another way of thinking of templates is as defining a schema in a database. The name of a template represents the name of a table, and each slot represents the header of a column.

A group of initial facts is defined with the `deffacts` construct, like in the following example:

```
(deffacts languages
    (language (name AAL) (version 0.1))
    (language (name python) (version 3.0))
    (language (name perl) (version 6.0)))
```

The general format of an inference rule is:

```
(defrule name
   <patterns>
      =>
   <actions>)
```

The semantics of a rule is that when the patterns on the left-hand side of the rule match in the

current state of the knowledge base, then the actions on the right hand side are executed. The patterns are implicitly connected by "and" operators. Other connectors ("not", "or") are available, but they have to be explicit.

A pattern consists of a fact, possibly with variables instead of the values. A pattern matches the current state of the knowledge base if the knowledge base contains a fact that matches the values used in the pattern. If variables are used then the first occurrence of the variable binds it to the corresponding value in the matched fact. All patterns that mention the same variable must match the same value.

Actions can consist of printing a message, and asserting or retracting facts. Facts are asserted with `(assert fact)` and retracted with `(retract fact-index)`, where `fact-index` is the index the knowledge base associates to the fact. The index can be retrieved on the left-hand side of the rule in a variable like this: `?fact_idx <- pattern`.

An example of a simple rule is:

```
(defrule print_version
   (language (name ?n) (version ?v))
      =>
   (printout t "Language " ?n " has version " ?v crlf))
```

More than one rule can be defined in the same file and rules can be prioritized by giving each of them a "salience."

4.3.4 Encoding the Abstract Syntax Tree and the Control Flow Graph

For every application of a static analysis rule the initial facts are precomputed by the static analysis framework. We have defined a set of templates that encodes the relevant relations between the elements of the AST and the CFG. Figure 4.14 shows some of the templates. We have tried to have templates with a low number of slots. This increases the number of generated facts, but helps when the patterns of a rule are matched against the knowledge base. For example, instead of listing the

statements that form the body of a function in the `fun` template we have opted for a new template `stmt` that associates statements with the function in which they appear. Also, instead of recording for each statement its kind (e.g., assignment, if, do, function call, etc.) we have opted to create a template for each kind of statement. For example, `stmt_funcall` records that a statement is a function call to a function named `fcallname`.

```
(deftemplate fun (slot id) (slot name) (slot module) (slot proc))
(deftemplate stmt (slot id) (slot fun) (slot file) (slot line))
(deftemplate var (slot id) (slot name) (slot kind))
(deftemplate fun_arg (slot fun) (slot var))
(deftemplate fun_ret (slot fun) (slot var))
(deftemplate stmt_entry (slot fun) (slot stmt))
(deftemplate stmt_exit (slot fun) (slot stmt))
(deftemplate stmt_funcall (slot fun) (slot stmt) (slot fcallname))
(deftemplate defines (slot fun) (slot stmt) (slot var))
(deftemplate uses (slot fun) (slot stmt)(slot var))
(deftemplate succ (slot fun) (slot src) (slot dst))
```

Figure 4.14: examples of templates

The more interesting templates are `defines` and `uses` that record that a statement defines or uses a variable. They encode the use-def information computed by most static analyzers. Again, instead of listing all the variables that are used in a statement, we have opted to define the `uses` template to have just one variable for a statement. Multiple instantiations of this template (multiple facts) can be used to encode that more than one variable is used in a statement.

The encoding of the AST and the CFG are implemented as graph traversals, generating for each node all the relevant facts. These facts are grouped in a `deffacts` construct and written to a file from where they can be loaded by the CLIPS engine.

Most of the facts come from the AST. The only facts coming from the CFG are instances of the `succ` template which record that the statement `dst` is reachable in one step from the statement `src`. This is a direct encoding of the edges of the CFG and this information is required for writing *flow-sensitive* analyses.

4.3.5 Computing the Reachability Relation

Some static analysis rules might make use of the reachability relation that encodes that a statement `dst` is reachable from a statement `src` in zero or more steps.

```
(deftemplate reachable (slot fun) (slot src) (slot dst))

(defrule reachable-base
  (fun (id ?f))
  (stmt (id ?s) (fun ?f))
=>
  (assert (reachable (fun ?f) (src ?s) (dst ?s))))

(defrule reachable-step
  (fun (id ?f))
  (stmt (id ?s1) (fun ?f))
  (stmt (id ?s2) (fun ?f))
  (stmt (id ?s3) (fun ?f))
  (reachable (fun ?f) (src ?s1) (dst ?s2))
  (succ (fun ?f) (src ?s2) (dst ?s3))
=>
  (assert (reachable (fun ?f) (src ?s1) (dst ?s3))))
```

Figure 4.15: computing the reachability relation

Figure 4.15 shows the rules for computing the reachability relation. The first rule is the base case stating that each statement is reachable from itself. The second rule is the induction step stating that if a statement s_2 is reachable from a statement s_1 and statement s_3 is a successor of s_2 then s_3 is reachable from s_1.

Because the number of statements is finite, the reachable relation has a finite number of elements, so only a finite number of `reachable` facts can be generated. Note that CLIPS will not retry rules on facts it has already executed on, so it will not generate the same fact more than once. This guarantees that the execution of the above rules always terminates.

4.3.6 Examples of Static Analysis Rules

In this section we present some examples of built-in static analysis rules. They provide a reference for users willing to write their own checks. It can be seen from these examples that, in general, rules are short and their code closely resembles their mathematical definition.

Assigning to an input variable

Although assigning to an input is allowed by the semantics, it is not recommended. Users might be confused how arguments are passed and might think that changing an argument in the body of the function reflects in a change of the variable passed as an argument when the function was called. AAL uses a call-by-value calling convention and any changes to input variables are lost after the function returns.

```
(deftemplate assigns_input (slot fun) (slot stmt) (slot var))

(defrule arg_assign
  (fun (id ?f))
  (fun_arg (fun ?f) (var ?v))
  (var (id ?v) (name ?varname))
  (stmt (id ?s) (fun ?f) (file ?file) (line ?line))
  (defines (fun ?f) (stmt ?s) (var ?v))
=>
 (printout err ?file ":" ?line ": "
           "assigning to input variable " ?varname crlf))
```

Figure 4.16: the static analysis rule for detecting assignments to input variables

Figure 4.16 shows a static analysis rule that detects all places in the code where an input variable is assigned. The rule is a direct translation of the following definition. For a function f, an input variable v is assigned if there exists a statement s in the body of f that defines v. This rule is an example of a *flow-insensitive* analysis.

Forgetting to assign to an output variable

If there exists an execution path that reaches the exit point of the function such that an output variable is never assigned on this path is an indication that the function might not behave as thought. Called without the right arguments the function might take this path leading to unexpected results. The function will return the default value for the output variable, and the user might think that it is actually the result of some computation that just happens to be the default value.

Figure 4.17 shows a static analysis rule that detects return variables that are not necessarily assigned by the end of the function. This rule is an example of a *flow-sensitive* analysis because the

```
(deftemplate reach_not_assign
  (slot fun) (slot stmt) (slot var))

; the function entry point can be reached without assigning
; to output variable v
(defrule reach_not_assign_base
  (fun (id ?f))
  (fun_ret (fun ?f) (var ?v))
  (stmt_entry (fun ?f) (stmt ?s))
=>
  (assert (reach_not_assign (fun ?f) (stmt ?s) (var ?v))))

; a statement s2 can be reached without assigning to output variable v
; if there exists a pred statement s1 that can be reached without
; assigning to v and s2 does not assign to v
(defrule reach_not_assign_step
  (fun (id ?f))
  (fun_ret (fun ?f) (var ?v))
  (stmt (id ?s1) (fun ?f))
  (stmt (id ?s2) (fun ?f))
  (succ (fun ?f) (src ?s1) (dst ?s2))
  (reach_not_assign (fun ?f) (stmt ?s1) (var ?v))
  (not (defines (fun ?f) (stmt ?s2) (var ?v)))
=>
  (assert (reach_not_assign (fun ?f) (stmt ?s2) (var ?v))))

(defrule print_results
  (stmt_exit (fun ?f) (stmt ?s))
  (reach_not_assign (fun ?f) (stmt ?s) (var ?v))
  (stmt (id ?s) (fun ?f) (file ?file) (line ?line))
  (var (id ?v) (name ?varname))
=>
  (printout err ?file ":" ?line ": "
                "path to function exit without assigning "
                "return variable " ?varname crlf))
```

Figure 4.17: the static analysis rule for detecting output variables that are not assigned

information at one statement is computed based on the information at its predecessors.

The check consists of three inference rules, two of them used to compute the required information, and one of them used to print the result. The information computed is captured in facts that instantiate the `reach_not_assign` template. The presence of such a fact means that there exists a path starting at the function entry point that reaches statement *stmt* without assigning variable *var*.

The first rule is the base case and states that the function entry reachable via a path that does not assign to output variable v. This is trivially true since the entry point is the first statement in

each path.

The second rule is the induction step and states that if a statement s_2 not defining an output variable v is reachable from a statement s_1 and there exists a path that reaches s_1 without assigning to v, then there exists a path that reaches s_2 without assigning v.

The third rule, the one that prints the result of the analysis, queries the knowledge base for an output variable v such that the exit point of the function is reachable via a path that does not assign v.

Use after free

It is an error to use a reference after it was freed. Such an access is not allowed by the dynamic type system in Section 3.6, but cannot be ruled out by the static type system. The static type system ignores the heap and has no way of knowing that a reference is no longer valid.

Figure 4.18 shows a static analysis rule that detects references that are used after they have been freed without having been reassigned in between. The description of how the rule works is presented in the comments.

Unused variables

Defining a variable and then never using it is a potential error because the user might assume that the value of the variable influences in some way the behavior of the program. It is at least an indication of unused code which in general should be avoided.

Figure 4.19 shows a static analysis rule that detects variables that are declared, but never used. The rule is a direct translation of the following definition. A variable v declared in a function f at statement s is unused if no other statement s_1 uses it. This is another example of a flow-insensitive analysis.

```
(deftemplate use_after_free
  (slot fun) (slot stmtfree) (slot stmtuse) (slot var))

(defrule find_use_after_free
  ; we have a function f
  (fun (id ?f))
  (var (id ?v) (name ?varname))
  ; that contains a statement s0
  (stmt (id ?s0) (fun ?f))
  ; which happens to be a function call of 'free'
  (stmt_funcall (fun ?f) (stmt ?s0) (fcallname free))
  ; and uses a variable v
  (uses (fun ?f) (stmt ?s0) (var ?v))
  ; we have another statement s1 (different than s0)
  (stmt (id ?s1&~?s0) (fun ?f) (file ?file) (line ?line))
  ; that uses the same variable v
  (uses (fun ?f) (stmt ?s1) (var ?v))
  ; and s1 is reachable from s0
  (reachable (fun ?f) (src ?s0) (dst ?s1))
  ; there is no statement between s0 and s1 that defines v
  (not (and
        (defines (fun ?f) (stmt ?s2) (var ?v))
        (reachable (fun ?f) (src ?s0) (dst ?s2))
        (reachable (fun ?f) (src ?s2) (dst ?s1))))
  (not (use_after_free (fun ?f) (stmtfree ?s0) (stmtuse ?s1) (var ?v)))
=>
  (assert (use_after_free (fun ?f) (stmtfree ?s0) (stmtuse ?s1) (var ?v)))
  (printout err ?file ":" ?line ": "
                "use after free of variable " ?varname crlf))
```

Figure 4.18: the static analysis rule for detecting references that are used after they were freed

```
(deftemplate unused_var (slot fun) (slot stmt) (slot var))

(defrule unused
  (fun (id ?f))
  (stmt_var (fun ?f) (stmt ?s) (var ?v))
  (var (id ?v) (name ?varname))
  (not (uses (fun ?f) (stmt ?s1) (var ?v)))
  (stmt (id ?s) (fun ?f) (file ?file) (line ?line))
=>
 (assert (unused_var (fun ?f) (stmt ?s) (var ?v)))
 (printout err ?file ":" ?line ": "
                "unused variable " ?varname crlf))
```

Figure 4.19: the static analysis rule for detecting unused variables

4.4 Dynamic Analyses

All the analyses presented so far were static, they do not require the execution of the program. This allowed them to work on partial specifications. The two analyses that are left, testing and model checking, are dynamic and require the execution of the program.

4.4.1 Closing an Open System

A partial specification means that the program is not complete and it is unlikely that it will be executable. Without taking any measures analyses like testing and model checking would not be applicable. To solve this problem we have devised a method for closing an open system.

The result of closing an open system is a set of test drivers for the parts of the code that are already written. More specific, the result is a set of test driver templates that can be used to generate concrete test cases.

A test driver template consists of an AST for the test driver in which some nodes are marked as being symbolic. The testing analysis can then instantiate the symbolic inputs with specific concrete values resulting in a concrete test case. These values could be random if they come from a random testing tool, or specifically picked values if they come from a concolic testing tool.

Note that we do not have to generate source code files since the analyses that use the test cases operate on abstract syntax trees. It would be redundant to generate the AST, write the source file and then parse it only to get back the same AST. It might be useful though to pretty-print the AST such that the user can inspect it and see what exactly was tested.

An open system can be closed at different levels: at function level, at process level, and at module level. If the system is closed at the level of a function f then the generated test drivers are suppose to test only the functionality of f. Similar considerations apply for closing the system at process and module levels.

Some of the test drivers generated might leave out parts of the available code that are not relevant for the level at which the system is closed. For example, if we close the system at function level, and we have two functions f_1 and f_2 that do not interact, then the test drives for f_1 will leave f_2 out.

Assumptions About the Available Code

For the test drivers to do anything meaningful we have to make some assumptions about the available code that they are supposed to test.

We have to assume that the available code contains at least one function, otherwise there is nothing to execute yet.

Another assumption is that the code can be parsed, otherwise the parser would have generated syntax errors and the developer would have to fix those first. Test drivers for code that is not syntactically correct do not make much sense. This assumption implies that if the module that is currently being developed imports some interfaces, then those interfaces are already written, otherwise the parser would have complained about unknown symbols every time something from an interface is used. It does not necessarily require that there are any modules implementing those interfaces yet. This forces the developer to specify the interfaces before attempting to use them. Forcing the developer to write the pieces of the code in the right order is another example of the development process being analysis-aware.

The assumption that the available code is syntactically correct also implies that whenever a function is declared at a process level, the process is declared and it contains definitions for all process-level variables that are used in the available code. This does not require that the process has a run function implemented, or that there exists any piece of implemented code that calls the available code. Similarly, the module containing the function must be defined, although it does not have to be complete, for example, it might not have any processes defined yet.

We have to assume that the code passes the type checker, otherwise the type checker would have generated some errors and the developer would have to fix those first. We cannot generate test drivers for code that does not use the language according to its semantics.

The code is not required to have a configuration yet, and even if one exists it will be replaced with a new configuration generated for the test driver.

Generating Stub Modules From Interfaces

Closing the system at any level might require generating modules that implement specified interfaces. If the code being analyzed sends a message to a channel declared in an interface then there must exist a module that implements the interface and that can receive the message.

A generated module must be able to receive any messages on its input channels. Otherwise the analysis tools would find deadlock errors when a message is sent to a channel from which the generated module never reads. This would flood the user with error messages that most likely indicate an problem in the generated module. This imposes the following structure on the generated code presented in Figure 4.20.

```
module TestDriver
{
  active process test()
  {
    function run()
    {
      var m1 : t1 ;
      ...
      var mn : tn ;
      do
      :: true ->
         sel
         :: recv(c1,m1) ->
            // handler code that does not contain any recv operations,
            // but that can contain any number of send operations
         :: recv(c2,m2) ->
         ...
         :: recv(cn,mn) ->
         :: timeout -> break ; // nobody is sending any more messages
         les
      od
    }
  }
}
```

Figure 4.20: the high level structure of a generated stub module

Channels c_i, $i = 1, n$ are the ones declared in the interface, and t_i is the type of messages for channel i. Variables m_i are used to store the received messages. We impose the restriction that the handler for each received message does not contain any `recv` operations. This mean that all `recv` operations must happen at the top level. We also impose the restriction that the top level `sel` does

not contain any `send` operations. All `send` operations occur in the message handlers.

This is a design pattern that is common in embedded systems software design and usually considered as good practice. If possible, AAL developers should try to follow it. A static analysis rule can be written to detect violations of this pattern.

The restrictions above imply that the generated module can only receive messages on the channels declared in the interface. For example, it is not possible to receive a message on a channel that was received from another module. It is possible, and quite common, to send a message to a channel received from another module.

The expected pattern is to receive request messages on the channels declared in the interfaces. A request message usually contains a channel on which the reply should be sent.

The stub module is generated to behave according to the interface it implements. To achieve this we use the LTL formulas and the regular expressions in the interface specification to generate the code for the stub module.

A behavior of a module with respect to a formula is a set of possibly infinite sequence of send and receive operations. Each operation carries as data a channel c (the one on which the operation is performed) and a message m (that one that is being sent or received). Each sequence must start with a receive operation on a channel that is defined in the interface.

If the formula is parameterized then the first receive operation in the sequence binds all the parameters to actual values. All further operations in the sequence match these values. Also, a behavior contains only sequences that all bound the parameters to the same values. There exists a different behavior for a formula for each different combination of parameters.

Each LTL formula and each regular expression from the interface specification is converted to a Büchi automaton. The edges of the generated automaton are labelled with parameterized send and receive operations. Each label consists of exactly one such operation.

We use the `ltl2ba` tool [64] to convert an LTL formula to a Büchi automaton. The resulting automaton is slightly different than the one we need. The difference comes from the labels of the transitions. We need exactly one send or receive operation per label, while `ltl2ba` generates a

boolean combination of send and receive operation. We take each transition generated by `ltl2ba` and convert it to the form we need. We use the function *convert_trans* to perform the conversion. The function *convert_trans* ensures that the generated automaton does not have labels that are conjunctions of two or more send or receive operations. Such labels would require two operations to happen at the same time, which is not possible.

$convert\_trans : S \times BooLExpr(L) \times S \rightarrow (S \times L \times S)set$ where:

- S is the set of states of the Büchi automaton;
- $L = \{recv, send\}$ is the set of labels in automaton we need;
- $BoolExpr(L)$ is the set of all possible boolean combinations of elements from L and represents the set of labels of the automaton generated by `ltl2ba`.

Below is the definition of the *convert_trans* function:

$convert\_trans(s_1, true, s_2) = \{(s_1, recv, s_2), (s_1, send, s_2)\}$

$convert\_trans(s_1, false, s_2) = \{\}$

$convert\_trans(s_1, recv, s_2) = \{(s_1, recv, s_2)\}$

$convert\_trans(s_1, send, s_2) = \{(s_1, send, s_2)\}$

$convert\_trans(s_1, \neg recv, s_2) = \{(s_1, send, s_2)\}$

$convert\_trans(s_1, \neg send, s_2) = \{(s_1, recv, s_2)\}$

$convert\_trans(s_1, recv \wedge send, s_2) =$ conversion error

$convert\_trans(s_1, recv \wedge \neg send, s_2) = \{(s_1, recv, s_2)\}$

$convert\_trans(s_1, \neg recv \wedge send, s_2) = \{(s_1, send, s_2)\}$

$convert\_trans(s_1, \neg recv \wedge \neg send, s_2) = \{\}$

$convert\_trans(s_1, recv \vee send, s_2) = \{(s_1, recv, s_2), (s_1, send, s_2)\}$

$convert\_trans(s_1, recv \vee \neg send, s_2) = \{(s_1, recv, s_2)\}$

$convert\_trans(s_1, \neg recv \vee send, s_2) = \{(s_1, send, s_2)\}$

$convert\_trans(s_1, \neg recv \vee \neg send, s_2) = \{(s_1, recv, s_2), (s_1, send, s_2)\}$

We have a conversion error for the case of $recv \wedge send$ because each position in a sequence is exactly one operation. We cannot have two operations at the same time. If a conversion error is encountered it is reported to the user.

Another approach to ensure that the automaton generated does not contain conjunctions of send or receive operations is to extend the LTL formula with the restriction that at least one operation has to happen at any point of the sequence and that the operations are always mutually exclusive: `[](send || recv) && [](!(send && recv))`.

As a result of this conversion we obtain the following Büchi automata $B = (S, L, \rightarrow, s_0, F)$ where:

- S is the set of states, the same set as generated by `ltl2ba`;
- $L = \{recv, send\}$ is the set of labels;
- $\rightarrow \subseteq S \times L \times S$ is the transition relation obtained by translating the transition relation of the automaton generated by `ltl2ba`;
- s_0 is the initial state, the same as the one in the automaton generated by `ltl2ba`;
- $F \subseteq S$ is the set of final states, the same as the one in the automaton generated by `ltl2ba`.

We define the function $reach\_acc\_cycle : S \rightarrow bool$ that is $true$ for a state s if and only if there exists at least one execution of the Büchi automaton B starting at s that reaches an acceptance cycle. A model checking algorithm can be used to compute this function for each state in S.

A similar construction can be devised for regular expressions.

The above construction is carried out for each formula defined in the interface. Let $B(f)$ be the Büchi automaton for formula f.

Next we describe how we use the Büchi automata to generate a module that has the same behaviors as the formulas defined in the interface.

Remember that we have to fit the generated code to the previously described pattern where all the receive operations happen at the top level (Figure 4.20).

0. For each formula we store a map that associates bound values of the parameters to automata. For each combination of parameters we have to keep a different automaton.

1. On each `recv` branch of the top level `sel` we have a channel c and a message m. For each formula we apply the following construction. If the formula f is applicable to channel c and message m then we look up the corresponding automaton in the automata map associated with f. If no such automaton exists then we create a new one and add it to the map (this is where the binding happens).

2. Now for each applicable formula f we have an automaton a that is in some current state s. We compute the set of successors states of s reachable via a recv transition such that an acceptance cycle is still reachable from them: $Next = \{s' | (s, recv, s') \in a.transitions \wedge reach\_acc\_cycle(s')\}$. If $Next$ is empty that we generate an error because an unexpected message was just received. We need to filter out the successor states that cannot reach an acceptance cycle in order to avoid generating a behavior that violates the formula. We pick (nondeterministically) a state s' from $Next$ and we set the current state s of a to s'.

3. We enter a do loop that has cases comparing the current state s to each s_1 state in a. On each branch of the do loop we have an if selection that has the following branches.

3.1 A true branch that breaks out of the do loop to the outer most loop.

3.2. A branch for each outgoing transition from s, $(s, send, s_1)$ with $reach\_acc\_cycles(s_1)$ being true. On each such branch we generate a symbolic input for a message that will be sent, we send the message, we set the current state of the automaton s to s_1, and we return at the beginning of the do loop.

We do not include cases for when $reach\_acc\_cycle(s_1)$ is false because we do not want to generate behavior that does not satisfy the formula.

The do loop has the invariant that the current state can reach an acceptance cycle: $reach\_acc\_cycle(s)$. The invariant is true at the beginning of the loop because the current state was picked from the set $Next$. It is maintained by the loop body because each if branch that does not exit the loop has $reach\_acc\_cycles(s_1)$ as part of the guard and sets the current state to s_1.

This invariant implies that there exists at least one successor state that can reach an acceptance cycle so the execution would not block unless an unexpected message is received in the outer-most

loop.

The algorithm presented above has the effect of splitting each state of an automaton in two parts: a part for handling receive operations (the outer loop with the selection) and a part for handling send operations (the inner do loop). This approach was necessary in order to ensure that the generated code fits the pattern where all receive operations occur at the top level of a main event-processing loop and all send operations occur in the event-handlers (Figure 4.20).

Note that the generated code is not yet executable because it is parameterized by the symbolic inputs used to construct the messages being sent (step 3.2 above). It is up to the testing framework to replace these symbolic inputs with concrete values. Also, the generated stub module uses non-determinism extensively. A single concrete run of the module would explore only one of the many possible executions. It is up to the model checking framework to explore all possible executions.

Theorem 3. *The stub module generated by the above algorithm for an interface I satisfies all the formulas defined in I assuming that the environment does not send unexpected messages. If an unexpected message is sent then an error is generated.*

Proof. It follows from the invariant of the do loop that if no unexpected messages are received, then the current state of each automaton maintained by the stub module can reach an acceptance cycle. This means that the execution up to the current state is a prefix of an accepting execution. □

Closing an Open System at Function Level

Closing a system at the level of a function f requires generating a template for test drivers that exercise only the code of f.

The first step is to identify all the values used by f that come from the outside and replace them with symbolic inputs. There are four sources of such values:

- all the input arguments of f;
- any process-level variables used by f if f is defined inside a process p;
- all the return values of user-defined functions called by f;

- any messages received by f.

The next step is to identify all the interfaces used by f and generate stub modules for them.

Next we modify the body of f to remove all calls to user-defined functions. We replace the results of each function call with symbolic inputs. If f is declared in a process p and calls a function g that is also declared in p then after a call to g all the local variables used by f are reassigned to new symbolic inputs. This accounts for the possibility that g modifies process-local variables.

Next we remove all the definitions from the module m containing f except the definition of f, the definitions of types used by f, the declaration of imported interfaces used by f. If f is declared inside a process p then we proceed in a similar way, but we keep the definition of p.

Next, if f is not already inside a process then we define a new process that has a function *run* that calls f only once. Before the call to f we define a variable for each input of f and we assign it to a symbolic input. If f is defined inside a process p then we do not define a new process, instead we use f and proceed as before. In this case we also have to define the process-level variables used by f and assign them to symbolic inputs.

Finally, we generate a configuration that creates an instance of the module containing f and connects this module to instances of the stub modules that were generated.

Closing an Open System at Process Level

There are two sources of values that come from outside a process p:

- the return values of user-defined functions that were not defined in the body of p (functions defined at module level);
- any messages received by p.

The system is closed by generating stub modules for any interfaces used by p and replacing the return values of functions defined at module level with symbolic inputs.

Similar to closing a system at a function level, all irrelevant module-level definitions are removed (e.g., other processes, functions).

Finally, we generate a configuration that creates an instance of the module containing p and connects this module to instances of the stub modules that were generated.

Closing an Open System at Module Level

There is only one source of values that come from outside a module m: any messages received by m. The only thing needed is to generate stubs for all the interfaces imported by m and to link everything together in a configuration.

4.4.2 Testing

We have implemented a testing framework that takes as input a partially-written program and reports a set of errors that could be encountered at run-time.

Testing is a dynamic analysis and as a partial program is not executable, we first have to close it. We use the method described in the previous section to generate a set of test driver templates that close the partial program. We generate test driver templates for closing at the level of each defined function, process, or module.

The Concolic Testing Algorithm

For each test driver template, we apply the concolic testing algorithm which first instantiates the symbolic inputs of the test driver template to some random concrete values. Then runs a concrete execution collecting along the way a set of constraints on the symbolic inputs that must be satisfied for the execution to take the same path. Then it negates some of these constraints such that other branches of the code can be reached. The set of mutated constraints is solved using an SMT solver which results in a new set of concrete values for the symbolic inputs. The process is repeated until an error is found, all paths are explored, or the time allocated for the test driver expires. The operation of the concolic algorithm is depicted in Figure 4.21.

The testing framework relies on an *executive* that can interpret AAL programs. The executive is extended with symbolic execution functionalities.

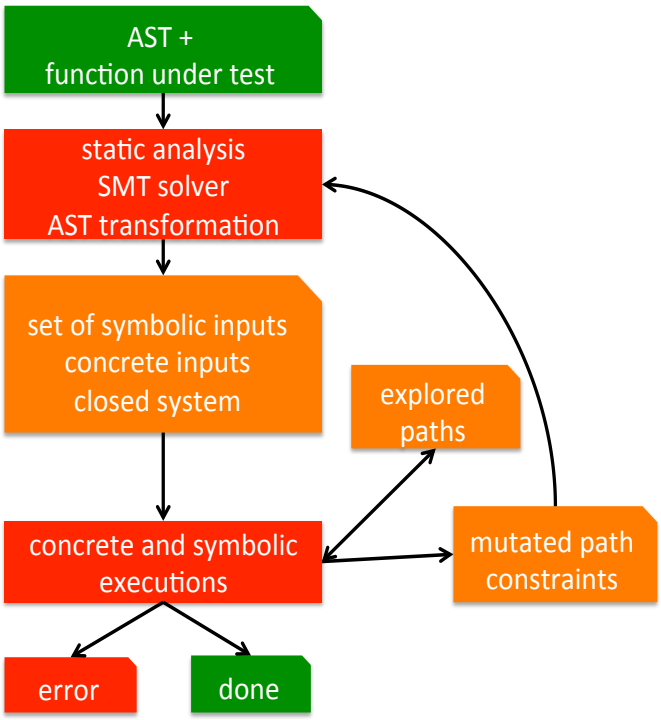


Figure 4.21: a high level view of the concolic testing algorithm

The Executive

The executive implements a machine capable of running programs according to the semantics presented in Section 3.7. It maintains the current state of the program and provides a function *next* that given a state s_1 and an action a computes the state s_2 which is the successor of s_1 that is reached by taking a in s_1. If according to the semantics, no such successor state exists then *next* returns an error.

The implementation uses a lower level semantics that is a refinement of the one presented in Section 3.7. The difference consists in how the control flow of the execution is captured. In 3.7 we have used a stack of frames with each frame being a tuple $(Rho_f, stmts, conts)$. The control flow inside a function was encoded using *continue* and *break* continuations. It is not feasible for the executive to store sequences of statements as part of the state (*stmts* and *conts*). Instead it stores the identifiers of nodes in the control flow graph. Now, a frame associated with a call to a function f is stored as a tuple (Rho_f, pc) where, just as before, Rho_f is the variable environment of f, but now the control flow is encoded using a program counter *pc*. The program counter is the identifier of a

node in the control flow graph of f and corresponds to the last executed statement (if one existed) or to the function entry point if no statement was executed yet.

The rules for the lower level semantics are almost the same, but instead of executing the first statement in the statement list at the top of the stack frame, we have to execute a statement that corresponds to a successor of pc. The requirements $(pc, pc') \in CFG(f).E$, $stmt(pc') = s$ are added to the premises of the rules where s stands for the statement to be executed next and $CFG(f).E$ is the set of edges of the control flow graph of f. Below we show an example of a modified rule:

$$\frac{\begin{array}{l} Eval(e, Rho_f \cup Rho_i, H, \{\}) = (v, Locs),\ 1 \leq i \leq p \\ (pc, pc') \in CFG(f).E \\ stmt(pc') = (var\ x : T = e;) \end{array}}{\begin{array}{l} P \vdash (H, ..., (Rho_i, (Rho_f, pc)\ rest), ...) \xrightarrow{(i,1)} \\ \quad (H \cup (i, Locs), ..., (Rho_i, (Rho_f[x = v], pc')\ rest), ...) \end{array}}\ \text{var-def-init}$$

The lower level semantics used by the executive would not have to change if we decide to add new branching constructs to AAL. For example, if we were to add a *goto* statement then we would not have to change anything because the effect of the statement would already be reflected in the control flow graph.

Representation of Concrete States

Figure 4.22 shows how concrete states are represented by the executive. Each state has a tree structure and can dynamically grow or shrink depending on the number of allocated references, the number of active processes, and the number of frames on the call stacks.

Representation of Symbolic States

Symbolic states have a similar structure to the concrete states. With the following changes or additions:

- a set of symbolic inputs is added;

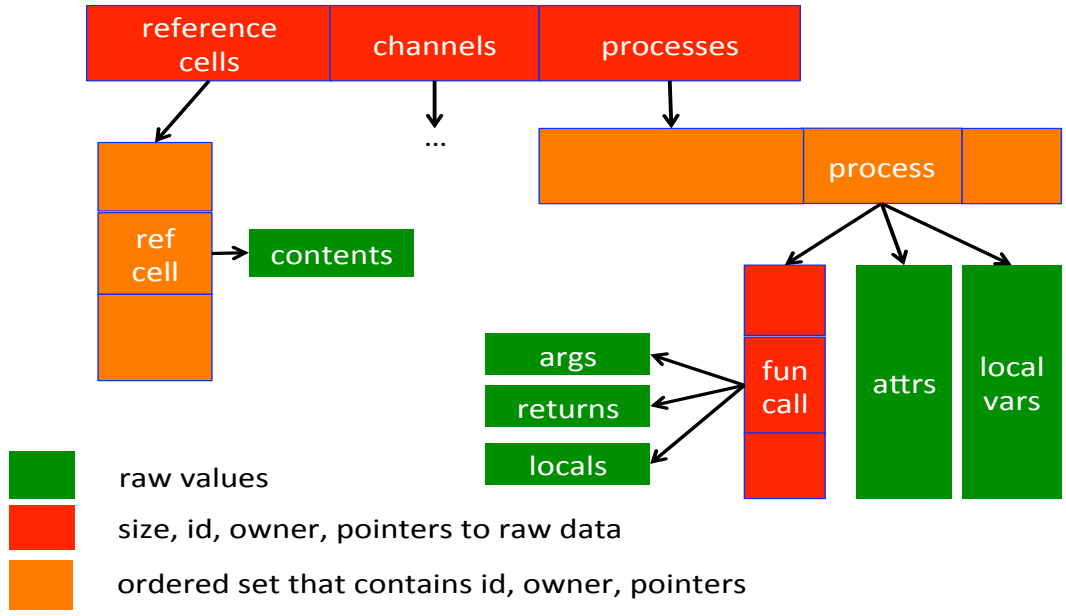


Figure 4.22: the representation of a concrete state

- each state also stores a list of linear constraints on the symbolic inputs;
- the heap is replaced with a symbolic heap;
- variable environments are replaced with symbolic variable environments.

A *symbolic value* sv is a linear expression on the symbolic inputs. The linearity restrictions comes from the limitations of the `yices` [27] SMT solver we use.

Concrete and Symbolic Execution

The symbolic execution follows the concrete execution, the only change is the way expressions are evaluated.

We define a new function:

$SymEval(e, Rho, H, Locs, SymRho, SymH, SymLocs) = (v, Locs', sv, SymLocs')$ which takes an expression, a concrete state $(Rho, H, Locs)$ and a symbolic state $(SymRho, SymH, SymLocs)$ and returns a concrete value v, a symbolic value sv, and the new values for $Locs$ and $SymLocs$. $SymLocs$ maps locations allocated during evaluation to symbolic values and it is used to update the symbolic heap whenever the expression is assigned to an l-value. The concrete evaluation part is needed

to handle non-linear expressions when the symbolic evaluation defaults to the concrete evaluation. Below we present some of the cases for $SymEval$.

Just as before, evaluating a constant returns the constant:

$SymEval(c, Rho, H, Locs, SymRho, SymH, SymLocs) = (c, Locs, c, SymLocs)$

Evaluating a symbolic input $symin$:

$SymEval(symin, Rho, H, Locs, SymRho, SymH, SymLocs) = (Rho[symin], Locs, symin, SymLocs)$

In this case the concrete evaluation looks-up the symbolic input in the variable environment to find the concrete value associated to it in the concrete state.

Evaluating a variable looks-up the variable in the environment:

$SymEval(x, Rho, H, Locs, SymRho, SymH, SymLocs) =$

$(Rho[x], Locs, SymRho[x], SymLocs)$

$SymEval(unop(e), Rho, H, Locs, SymRho, SymH, SymLocs) =$

$(delta\_unop(unop, v), Locs', delta\_sym\_unop(unop, sv), SymLocs')$

where

$(v, Locs', sv, SymLocs') = SymEval(e, Rho, H, Locs, SymRho, SymH, SymLocs)$

$SymEval(binop(e_1, e_2), Rho, H, Locs_1, SymRho, SymH, SymLocs_1) =$

$(delta\_binop(binop, v_1, v_2), Locs_3, delta\_sym\_binop(binop, sv_1, sv_2), SymLocs_3)$

where

$binop$ is a linear binary operator

$(v_1, Locs_2, sv_1, SymLocs_2) = SymEval(e_1, Rho, H, Locs_1, SymRho, SymH, SymLocs_1)$

$(v_2, Locs_3, sv_1, SymLocs_3) = SymEval(e_2, Rho, H, Locs_2, SymRho, SymH, SymLocs_2)$

Evaluating a non-linear binary operation defaults to the concrete evaluation:

$SymEval(binop(e_1, e_2), Rho, H, Locs_1, SymRho, SymH, SymLocs_1) =$

$(delta\_binop(binop, v_1, v_2), Locs_3, delta\_binop(binop, v_1, v_2), Locs_3)$

where

binop is a non-linear binary operator

$(v_1, Locs_2, sv_1, SymLocs_2) = SymEval(e_1, Rho, H, Locs_1, SymRho, SymH, SymLocs_1)$

$(v_2, Locs_3, sv_1, SymLocs_3) = SymEval(e_2, Rho, H, Locs_2, SymRho, SymH, SymLocs_2)$

$SymEval(ref\ e, Rho, H, Locs_1, SymRho, SymH, SymLocs_1) =$

$(l, Locs_2 \cup \{(l, ref\ v)\}, l, SymLocs_2 \cup \{l, ref\ sv\})$

where

$(v, Locs_2, sv, SymLocs_2) = SymEval(e, Rho, H, Locs_1, SymRho, SymH, SymLocs_1)$

$l \notin H \cup Locs_2$

$l \notin SymH \cup SymLocs_2$

Note that the symbolic evaluation generates the same set of locations. The difference is that they are mapped to symbolic values instead of concrete values.

$SymEval(e\$, Rho, H, Locs_1, SymRho, SymH, SymLocs_2) = (v, Locs_2, sv, SymLocs_2)$

where

$(l, Locs_2, l, SymLocs_2) = SymEval(e, Rho, H, Locs_1, SymRho, SymH, SymLocs_1)$

$(l, ref\ v) in H \cup Locs_2$

$(l, ref\ sv) in SymH \cup SymLocs_2$

The other cases of *SymEval* are similar and are not presented here.

4.4.3 Model Checking

The last analysis implemented in our framework is *explicit state model checking*. It is a dynamic analysis and thus requires the system to be closed. We use the method presented in Section 4.4.1 to create test drivers templates. The framework instantiates these templates to concrete test drivers and then applies the model checking algorithms presented in this section.

We have considered two approaches to implementing an explicit state model checker for AAL. The

first approach is to implement a model checker on top of the executive described in Section 4.4.2. This is the more standard approach and it is harder to customize. The second approach is to implement the model checker directly in AAL as a scheduler process. We have settled on the second approach as it is new, more customizable and allows more experimentation. Also, it allows users to implement their own model checking algorithms.

In general, a model checker takes as input a model M of a system and a property P about M, and exhaustively explores all possible behaviors of M checking if each of them satisfies P. If there exists a behavior of M violates P then a counter example is produced.

In our case, the model is the transition system $TS(P)$ where P is the program under analysis (a concrete test driver).

Safety properties consist of:

- access to an invalid or null reference;
- access to a reference owned by a different process;
- division by zero;
- running out of memory;
- assertion failures;
- function contract violations;
- deadlock.

Liveness properties can be specified as parameterized LTL or regular expression formulas inside interface specifications. A module M exporting an interface I must satisfy the properties specified in I.

Implementing a model checker for a language in the same language raises some questions about self referencing. For example, how can the model checker capture the current state when the fact that it is currently trying to do this is also part of the state. We have dealt with this problem by splitting the current state of the whole system in two parts: a part that corresponds to the actual

system being model checked, and a part that corresponds to the scheduler process running the model checking algorithm. Another issue is how states are captured and stored as efficiently as possible. In some cases this leads to different implementations of a data structure (for example, a map) being used in the actual application than in the scheduler process.

State Serialization

State serialization is the process of saving a state as a sequence of bytes. There are two requirements that serialization must satisfy:

- it must be deterministic: serializing a state always produces the same sequence of bytes, $s_1 = s_2 \Rightarrow serialize(s_1) = serialize(s_2)$;
- it must be reversible: there exists a deserialization method for restoring a sequence of bytes back to a state, $restore(serialize(s)) = s$.

State serialization is essential for implementing model checking algorithms. States are stored and compared in their serialized version. Restoring a state from its serialized version is used to backtrack to a previous visited state.

We have introduced the concept of a scheduler process in AAL to give users the possibility to write their own model checking algorithms. To this end, we have added two built-in functions that serialize and restore the current state:

```
get_current_state() : (state : array[char])
set_current_state(state : array[char]) : ()
```

These primitives are only accessible from scheduler processes and are not available for the main program.

State serialization introduces some performance penalties, but cannot be avoided because states have a tree structure and not all states have the same size. For example, calling a function adds a new frame on the call stack which means that the current state has to grow.

Methods for serializing and restoring tree structures are well known and will not be discussed here with the exception of serializing maps.

A map is serialized as an array of $(key, value)$ pairs ordered by keys. If maps are stored as tree structures (for example, as hash tables) then serialization is an expensive process because it requires the traversal of the tree. If maps are present in the code they have to be serialized after every step of the program. The cost can add up quickly and slow down model checking process.

An alternative approach is to always store the maps in their serialized form. This slows down map accesses, but map accesses are considerably less frequent than state serialization, state serialization happens at each step, map accesses happen only at steps that specifically read or write to the map.

We have implemented both approaches in the executive. The hash table version of maps is always used in the scheduler process because the state of the scheduler process does not have to be serialized. Moreover, it is likely that a scheduler process implementing a model checker will use a map to store the state space, so map accesses are very frequent.

If the system does not have a scheduler process, or if the scheduler process does not call the state serialization built-in functions then the executive will use hash tables to represent maps in the main program. Otherwise, it will use the array of (key,value) pairs representation.

Similar considerations apply to serializing sets.

A Model Checker As a Scheduler Process

We show here how a model checker for finding safety violations can easily be implemented as a scheduler process in AAL.

We start by defining the data structures used to store the set of visited states (the state space) and the current execution stack (Figure 4.23).

A frame on the stack stores the corresponding state and some extra information about the state of the exploration: the actions that can be taken from that state and which of these actions are already explored. We give each state an identifier, but this is not required for the algorithm to work. It is used for printing information about the exploration.

Next, we show the model checking algorithm. It consists of a DFS exploration of the state space. We have written an iterative version of DFS that explicitly maintains the stack of states

```
type frame = {
  state     : array[char],
  nbactions : int,
  actions   : array[action_type],
  taken     : array[bool],
  id        : int
};
type state_extra = {id : int, on_stack : bool};

const MAX_STACK_SZ : int = 10000;
var last_id : int = 1;
var state_space : map[array[char] : state_extra] = {};
var state_stack : array[frame] = mkarray of frame[MAX_STACK_SZ];
var top : int = -1;
```

Figure 4.23: the data structures used for storing the state space and the execution stack

(Figure 4.24).

```
function dfs()
{
  var next_action : int = -1;
  push_new_state();
  do
  :: top < 0 -> break;
  :: else ->
     next_action = find_next_action();
     if
     :: next_action >= 0 && next_action < state_stack[top].nbactions ->
        state_stack[top].taken[next_action] = true;
        set_next_actions({state_stack[top].actions[next_action]});
        push_new_state();
     :: else -> pop_state_and_restore();
     fi
  od
  printf("%d states explored\n", last_id);
}
```

Figure 4.24: DFS state space exploration

The algorithm starts by pushing the initial state on the stack. It then enters a loop in which it picks an unexplored action to be taken from the current state.

`set_next_actions` tells the executive to take this action and switches control to the program being analyzed. After the program takes a step, the control returns immediately after the call of `set_next_actions`. The new current state is pushed on the stack and the process is repeated.

If a state has no untaken actions it is poped from the stack and the previous state is restored.

The algorithm terminates when the stack of states becomes empty.

Note that there is no code for error reporting. Error messages are generated by the executive when it reaches an error state.

A state is pushed on the stack (Figure 4.25) only if it was not previously visited. If it was previously visited (`state in state_space`), we backtrack and restore the parent state. When a state is pushed on the stack we also have to get the set of actions that can be taken from it. This is done by calling the `get_enabled_actions` built-in function.

```
function push_new_state()
{
  var state : array[char];
  var taken : array[bool];
  state = get_current_state();
  if
  :: state in state_space ->
     if
      :: top >= 0 -> set_current_state(state_stack[top].state);
      :: else -> skip;
     fi
     return;
  :: else -> skip;
  fi
  top = top + 1;
  assert top >= 0 && top < MAX_STACK_SZ;
  state_stack[top].state = state;
  state_stack[top].actions = get_enabled_actions();
  state_stack[top].nbactions = length (state_stack[top].actions);
  state_stack[top].id = last_id;
  last_id = last_id + 1;
  if
  :: state_stack[top].nbactions > 0 ->
     state_stack[top].taken = mkarray of bool[state_stack[top].nbactions];
  :: else -> skip;
  fi
  state_space[state] = {id = state_stack[top].id, on_stack = true};
}
```

Figure 4.25: saving the current state

Finding what action to take next is presented in Figure 4.26. In this implementation actions are picked from left to right. A different implementation of this function could pick them in some other order (right to left, random, etc.) resulting in states being explored in a different order. This is useful for implementing swarm verification where we run multiple instances of the model checking

algorithm each of them exploring states in a different order. The idea is that each instance does not have the resources necessary (time and memory) to explore the entire state space so an error can be missed, but many instances exploring different parts of the state space have a higher chance of finding the error.

```
function find_next_action() : (result : int)
{
  var len : int = 0;
  var i : int = 0;
  var priority : int = 0;
  assert top >= 0 && top < MAX_STACK_SZ;
  result = -1;
  if
  :: state_stack[top].nbactions <= 0 -> return;
  :: else -> skip;
  fi
  len = length(state_stack[top].taken);
  do
  :: i >= state_stack[top].nbactions -> break;
  :: else ->
     if
     :: !state_stack[top].taken[i] -> result = i; break;
     :: else -> skip;
     fi
     i = i + 1;
  od
}
```

Figure 4.26: deciding what action to take next

Removing a state from the top of the stack (Figure 4.27) restores the parent state (if one exists). Note that we have to free the arrays that held the extra information associated with the state being removed (`actions` and `taken`).

We have presented a simple implementation of a model checking algorithm. Many interesting versions can be implemented. For example, if all processes have a `priority` parameter then the algorithm above can be modified to pick the actions to be executed next in a way that generates only schedules that are allowed by the priorities. The built-in function `get_value` can be used to get the current value of a variable in the context of a process. For example

```
priority = get_value(state_stack[top].actions[i], "priority");
```

can be used to get the priority of the process taking action `i`.

```
function pop_state_and_restore()
{
  assert top >= 0 && top < MAX_STACK_SZ;
  state_space[state_stack[top].state].on_stack = false;
  free(state_stack[top].actions);
  if
  :: state_stack[top].nbactions > 0 -> free(state_stack[top].taken);
  :: else -> skip;
  fi
  top = top-1;
  if
  :: top >= 0 -> set_current_state(state_stack[top].state);
  :: else -> skip;
  fi
}
```

Figure 4.27: restoring the previous state

So far we have presented a model checker that can detect only violations of safety properties.

To detect violations of LTL and regular expression properties we take an approach based on property monitors.

A property monitor for a formula f parameterized by data d is a state machine that takes a step whenever a send or receive operation relevant to f occurs. At each step an action can be executed, for example, report an error.

It is unlikely that developers would write such monitors by hand so we auto generate monitor templates for each formula and then we present them to the developers. The developers can then include the generated monitors in their model checking algorithms.

To generate a monitor template for a formula f we take a similar approach to the one in Section 4.4.1. We generate the automaton corresponding to formula f and use it as the state machine of the monitor. For each state of the monitor we also define the function $reach\_acc\_cycle$ with the same meaning as the one in Section 4.4.1.

These monitors can be used to implement a model checking algorithm that checks if no accept cycles are reachable from the current state. If this is the case, then we know we have an error because no matter how we extend the execution, the property cannot be satisfied.

Chapter 5

Case Study: A Smart Home

We have applied our framework to the development of a *Smart Home* application. It controls various home appliances such as lights, audio and video systems, heating, etc.

We show how our specification for the application is analysis-aware because we had to provide interfaces for the various components and these interfaces had to contain enough specifications (in the form of LTL formulas) such that they could be used to generate the stub modules required for testing and model checking. For example, a first attempt at specifying some of the LTL formulas turned out to be too general because it allowed undesired behaviors. Testing against the stub modules revealed the unexpected scenarios and forced us to refine the specification using more precise LTL formulas.

Next we present a short overview of the functionality of the smart home application. Users can switch between different modes of operation according to their needs.

- *The At Home - Daytime* mode which controls the appliances for the case when people are at home during the day. In this case interior lights are turned on and off based on motion and room brightness, the blinds are open, exterior lights are off, climate control is turned on, and the audio-video system is off.

- *The At Home - Nighttime* mode which controls the appliances for the case when people are at home during the night. The difference between the daytime mode is that interior lights are off, exterior lights are on, and the blinds are closed.

- *The At Home - Movie* mode which controls the appliances for the case when people are at

home and watching a movie. The difference between the daytime mode is that the audio-video system is on, the interior lights are off and the blinds are closed.

- *The Away* mode which controls the appliances for the case when people are away from home. In this case all appliances are off or closed, with the exception of the security system.

5.1 Architecture

The architecture of the smart home application shown in Figure 5.1 consists of a set of modules that implement the control part, module that models user input and a set of modules that model input from various sensors.

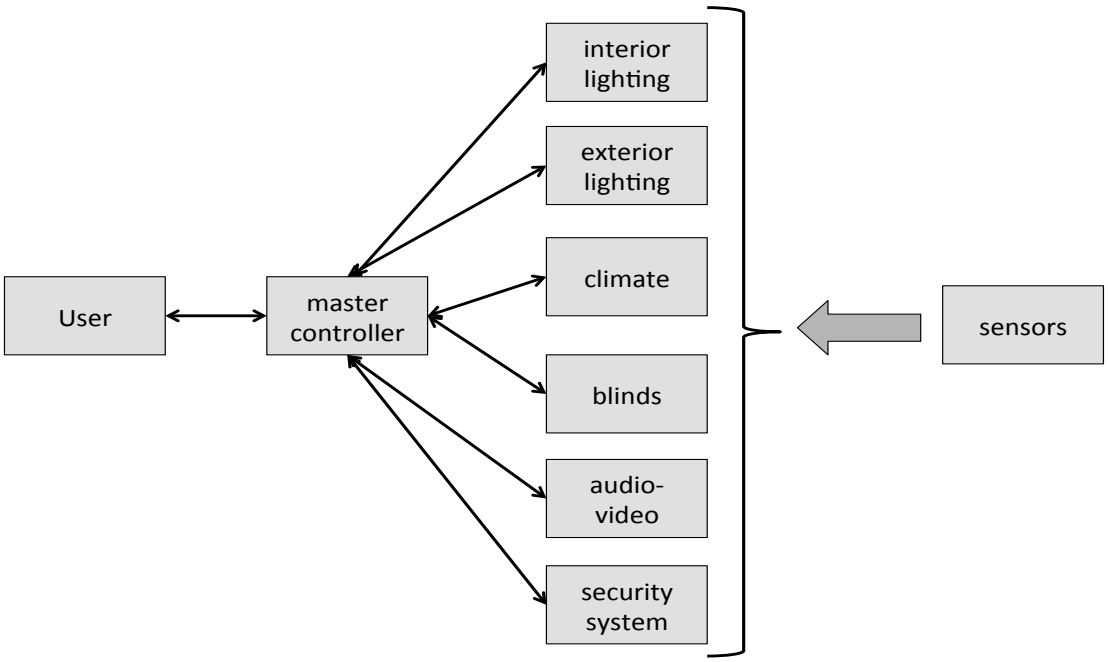


Figure 5.1: the architecture of the smart home application

User input is modeled as a module that can send commands to a master controller. These commands encode setting the current mode and getting the status of the various appliances.

The master controller is responsible for receiving user input and coordinating the various con-

trollers for the appliances.

For each appliance there is a controller module. They are turned on or off by the master controller. When on, they repeatedly poll a set of sensors and act accordingly to the readings.

The user input module and the sensors module model the environment, the other modules implement the smart home application itself.

5.2 Interfaces

According to our methodology, the development has to start with defining a set of interfaces. We define an AAL interface for each individual component. Each interface defines channels on which the component can receive messages from other components. The type of the messages is also defined along with constants that encode various kinds of messages.

The interface for the master controller is shown in Figure 5.2. It defines the fact that the master controller can receive commands from a user on the `usr_cmds` channel. Commands are encoded as records with the first field (`cmd`) specifying the kind of command. We name the valid commands (e.g., `at_home_day`, `get_temperature`, etc.) by defining constants for them. These constants are used as the `cmd` field of messages received by a user. This encoding of messages resembles a variant type, where values are grouped into classes according to the constructor used to create them. The various constants used for the `cmd` field correspond to the constructors.

Each message from the user contains a reply channel on which results are sent back. The specification also defines the type of the reply messages (the `usr_cmd_reply` type).

The master controller's interface also shows that it is connected to the controllers for each individual appliance by importing the appropriate interfaces. The interfaces are given shorter names for ease of reference.

Note that we do not show the LTL formulas defined in the interfaces here. They will be described in detail in the next section.

Figure 5.3 shows the interface for the climate controller. It follows the same pattern as the master controller: It defines a channel (`cmds`) on which it can receive commands from the master

```
interface MasterCtrlInterf
{
  import ic = IntLightCtrlInterf;
  import ec = ExtLightCtrlInterf;
  import bc = BlindsCtrlInterf;
  import ac = AudioVideoCtrlIntef;
  import cc = ClimateCtrlInterf;
  import sc = SecuritySystemCtrlInterf;

  type usr_cmd_reply = {
    seq_no : int,
    status : int,
    value  : int
  };
  const success : int;

  type usr_cmd_msg = {
    cmd     : int,
    seq_no  : int,
    param   : int,
    reply_c : chan[usr_cmd_reply]
  };
  const at_home_day : int;
  const at_home_night : int;
  const at_home_movie : int;
  const away : int;
  const get_temp : int;
  const set_temp : int;

  const usr_cmds : chan[usr_cmd_msg];
}
```

Figure 5.2: the interface for the master controller

controller and the set of valid commands (`on`, `off`, etc.). Again, each message received from the master controller includes a channel on which a reply message should be sent. The interface also defines the structure of the reply messages.

Interfaces for the other appliance controllers are defined similarly (except that they do not have the `get_temp` and `set_temp` commands).

A motion sensor (Figure 5.4) can receive poll requests on a channel. The current value read by the sensor is sent back on the reply channel provided in the request message. There are two possible replies `no_motion_detected` and `motion_detected`.

The interface for the temperature sensor (Figure 5.5) is similar to the one for the motion sensor, but it does not provide a set of allowed replies. Instead a reply message will contain an integer

```
interface ClimateCtrlInterf
{
  import ts = TemperatureSensorInterf;

  type cmd_reply =
  {
    seq_no : int,
    status : int,
    value  : int
  };
  const success : int;

  type cmd_msg =
  {
    seq_no : int,
    cmd : int,
    temp : int,
    reply_c : chan[cmd_reply]
  };
  const off      : int;
  const on       : int;
  const set_temp : int;
  const get_temp : int;

  const cmds : chan[cmd_msg];
}
```

Figure 5.3: the interface for the climate controller

```
module MotionSensorInterf
{
  const no_motion_detected : int;
  const motion_detected : int;

  type sensor_msg = {
    req     : int,
    reply_c : chan[int]
  };
  const get : int;

  const poll : chan[sensor_msg];
}
```

Figure 5.4: the interface for the motion sensor

representing the current temperature.

Interfaces for the other kinds of sensors are defined in a similar way.

```
interface TemperatureSensorInterf
{
  type sensor_msg = {
    req     : int,
    reply_c : chan[int]
  };
  const get : int;

  const poll : chan[sensor_msg];
}
```

Figure 5.5: the interface for the temperature sensor

5.3 Specifications

We now define the specification for the smart home application. Specifications in AAL are introduced at interface level and refer to messages sent and received on the channels declared in the interfaces. All the definitions and LTL formulas presented here are added to the corresponding interfaces.

These specifications are what make the development of the system analysis-aware. They must specify what is the allowed behavior of the modules implementing the interfaces. The LTL formulas presented here are used at analysis time to generate stub modules to close the system so that it can be tested and model checked.

We describe the specification for each component.

LTL Formulas for The Master Controller

We give the specifications for each mode, focusing more on the *At Home - Daytime* mode. The specifications for the other modes follow a similar pattern.

The master controller functions correctly if for every message received from the user it instructs the individual appliance controllers to take the corresponding actions to implement the user's intent.

The At Home - Daytime Mode

The *At Home - Daytime* mode starts with the user sending an `at_home_day` message to the master controller.

In response to this message, the master controller has to instruct the interior lighting controller

to turn on. This does not necessarily mean that the lights will be turned on. The decision to turn lights on and off is made by the lighting controller based on input from the brightness and motion sensors. The master controller turns on the lighting controller by sending and `on` message. Also, the blinds controller is told to open the blinds.

Similarly, the master controller turns off the exterior lighting controller by sending an `off` message. Turing off the exterior lighting controller does not mean it cannot receive other messages from the master controller. It only means that the light will turn off and no sensors will be polled until the controller is turned back on.

The master controller also turns on the climate controller which will decide to turn on the heater or the AC based on the temperature read from the temperature sensor.

All other controllers are also explicitly turned off.

We can encode these requirements in LTL and place them in the master controller's interface.

We make the following definition for readability purposes (these are also part of the interface for the master controller):

```
def at_home_day(rcu,seq0): recv(usr_cmds, {cmd=at_home_day,seq_no=seq0,reply_c=rcu});
```

which encodes receiving an `at_home_day` message from the user. The definition is parameterized by a reply channel (`rcu`) on which a reply will be sent back to the user and a sequence number (`seq0`).

Similarly we define `at_home_night`, `at_home_movie`, `away`, `set_temp`, and `get_temp` to be used in the specifications for the other modes.

`def int_lights_on(rcmc,seq0): send(ic.cmds,{seq_no=seq0,cmd=on,reply_c=rcmc});` which encodes sending an `on` message to the interior light controller (where `ic = IntLightCtrlInterf;`).

Similarly we define `int_lights_off`, `ext_lighs_on`, `ext_lights_off`, and so on for every message than can be sent to an appliance controller.

```
def int_lights_reply(rcmc,seq0): recv(rcmc,{seq_no=seq0,status=success});
```

which encodes receiving a successful reply from the interior lights controller.

We also define a successful reply to the user:

```
def success(rcu,seq0): send(rcu,{seq_no=seq0,status=ok,value=v});.
```

The requirement that the interior lighting controller is turned on as a consequence of receiving an `at_home_day` message from the user is formalized as follows:

```
ltl at_home_day_int_lights_on(rcu):

       ((!int_lights_on(rcmc,seq0)) U at_home_day(rcu,seq1))

       ((!int_lights_reply(rcmc,seq2)) U int_lights_on(rcmc,seq3))

    && ((!success(rcu,seq4)) U int_lights_reply(rcmc,seq5))

    && [](at_home_day(rcu,seq6) =>

          X(int_lights_on(rcmc,seq6) &&

          X(int_lights_reply(rcmc,seq6)

          X(success(rcu,seq6) &&

          X((!(   int_lights_on(seq7)

               || int_lights_reply(rcmc,seq8)

               || success(rcu,seq9)))

             U at_home_day(rcu,seq10))))));
```

The formula is parameterized by `rcu`, the channel on which the reply must be sent back to the user. The sequence number is not a parameter of the LTL formula, but the same sequence number (`seq6`) that is the parameter for `at_home_day` is expected for turning on the lights controller and sending a reply to the user. The other variables for sequence numbers (all except `seq6`) are used because we are interested in also observing messages with other sequence numbers which in the context of the formula mean that we do not want to see a send success event with any sequence number, not just the one received in the `at_home_day` message. Here the `rcmc` is the reply channel provided by the master controller to the interior lights controller.

Definitions not mentioned in the formula are not observed (for example, turning off the exterior lights) so the above formula does not impose any restriction on them. We use other similar LTL formulas to specify how those cases are handled. Also, it is implicit that definitions are mutually exclusive and exactly one must occur at each time.

The formula states that every time the master controller receives a `at_home_day` message, it must next turn on the interior light controller and then send a success reply to the user. Turning on the interior lights controller is encoded as a request to the controller and a reply from the controller.

The `((!int_lights_on(rcmc,seq0)) U at_home_day(rcu,seq1))`,

`((!int_lights_reply(rcmc,seq2)) U int_lights_on(rcmc,seq3))`

`((!success(rcu,seq4)) U int_lights_reply(rcmc,seq5))`

parts of the formula state that the associated behavior must start with receiving an `at_home_day()` message (it cannot start with sending `int_lights_on()` and it cannot send a `success()` reply before turning on the interior lights controller. Also, an `int_lights_reply()` is not received before a request was sent to the controller.

Initial attempts at writing this formula did not include this restriction leading to unexpected behavior. We talk more about a similar situation for the specification of the sensors in the analysis section.

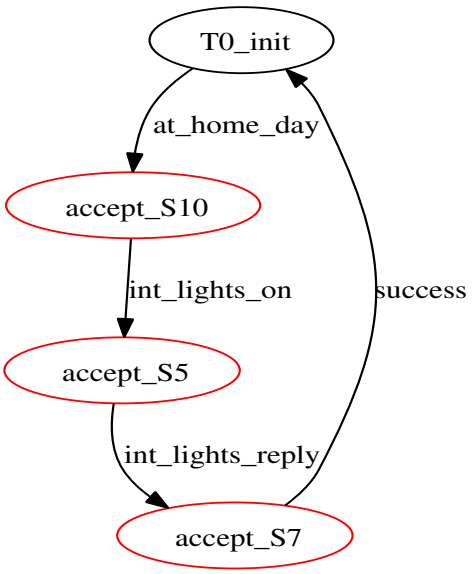


Figure 5.6: the Büchi automaton generated for the int_lights_on LTL formula

Figure 5.6 shows the automaton generated for this formula to be used in stubbing the master controller. From this automaton it can be seen that the LTL formula captures our intent.

The At Home - Nighttime Mode

The *At Home - Nighttime* mode starts with the user sending an `at_home_night` message to the master controller.

In response to this message, the master controller has to turn off the interior lighting controller and turn on the exterior one. The climate controller is turned on, and the blinds are changed to closed. All other controllers are turned off.

The specification for this case looks similar to the one for the daytime mode. It it just updated with sending the appropriate `on` and `off` messages.

For example, the following formula specifies how a message is sent to turn off the interior lighting controller. It looks very similar to the one for turning on the interior lights controller and the same comments apply.

```
ltl at_home_night_int_lights_off(rcu):
        ((!int_lights_off(rcmc,seq0)) U at_home_night(rcu,seq1))
        ((!int_lights_reply(rcmc,seq2)) U int_lights_off(rcmc,seq3))
    && ((!success(rcu,seq4)) U int_lights_reply(rcmc,seq5))
    && [](at_home_night(rcu,seq6) =>
           X(int_lights_off(rcmc,seq6) &&
           X(int_lights_reply(rcmc,seq6)
           X(success(rcu,seq6) &&
           X((!(   int_lights_off(seq7)
                || int_lights_reply(rcmc,seq8)
                || success(rcu,seq9)))
              U at_home_night(rcu,seq10))))));
```

The At Home - Movie Mode

The *At Home - Movie* mode starts with the user sending an `at_home_movie` message to the master controller. The master controller must turn off the interior lighting controller, close the blinds and start the audio-video system. The climate controller must also be turned on. All other controllers are turned off.

The specification for this mode is similar to the one for the daytime mode.

The Away Mode

The *Away* mode starts with the user sending an `away` message to the master controller. In this mode all appliances are turned off except the security system.

The specification for this mode looks similar to the one for the daytime mode.

LTL Formulas for Appliance Controllers

The appliance controllers function correctly if they set their mode accordingly to the commands received from the master controller and if they take appropriate actions based on input from the sensors.

We have the following definitions for turning on or off the appliance controller:

```
def on(rcmc,seq0) : recv(cmds, {seq_no=seq0, cmd=on, reply_c=rcmc});
def off(rcmc,seqo) : recv(cmds, {seq_no=seq0, cmd=off, reply_c=rcmc});
```

where `rcmc` is the channel used to reply to the master controller.

We define a successful reply to the master controller as follows:

```
def success(rcmc,seq0): send(rcmc,{seq_no=seq0, status=success, value=v});
```

The communication with the master controller is specified in the following LTL formula:

```
ltl mc_success(rcmc):   ((!success(rcmc,seq0)) U (on(rcmc,seq1) || off(rcmc,seq2)))
                  && [](on(rcmc,seq3) || off(rcmc,seq3)) => X(success(rcmc,seq3)))
                  && [](success(rcmc,seq4) =>
                        X((!success(rcmc,seq5)) U (on(rcmc,seq6) || off(rcmc,seq7))));
```

The formula expresses that exactly one `success` reply has to be sent back to the master controller for each `on` or `off` commands received. This behavior can be seen in the automaton generated (Figure 5.7).

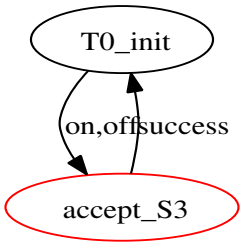


Figure 5.7: the Büchi automaton generated for the mc_success LTL formula

We define polling a sensor and receiving a reply from the sensor:

```
def poll(rcs) : send(ts.poll, {req=get, reply_c=rcs});
 def sensor_reply(rcs): recv(rcs,v);
```

where `rcs` is the channel provided by the appliance controller in the message to the sensor and on which it receives replies, and `v` is the value returned by the sensor.

The fact that the controller has to poll the sensor while it is turned on and not to poll it while it is turned off is specified using the following LTL formula:

```
ltl poll_occurence():    ((!poll()) U on())
                      && ((!sensor_reply()) U poll())
                      && [](off() => X((!poll()) U on()))
                      && [](on() => X(poll()))
                      && [](poll() => X(sensor_reply()))
                      && [](sensor_reply() => X((!sensor_reply()) U poll()));
```

For convenience we have omitted the parameters of the definitions. The formula also states that a reply from the sensor must be received after each `poll`.

The automata that is generated for the above formula is shown in Figure 5.8. We can see that each `on` command that is received is followed by a non empty sequence of `poll` and `sensor_reply`

until it receives an `off` command.

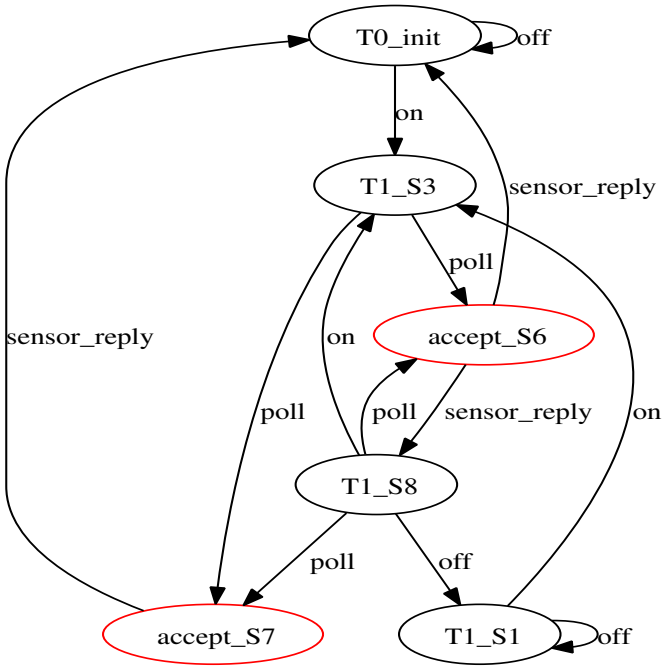


Figure 5.8: the Büchi automaton generated for the poll_occurence LTL formula

LTL Formulas for Sensors

When a sensor is polled by an appliance controller it has to send its current reading. Here we use `poll(rc)` for the receiving of a poll message by a sensor that specifies the reply channel `rc` to where the current reading must be sent. Also, we use `value(rc,v)` to encode sending a message with the reading of the sensor to the polling controller.

```
   def poll(rc)  : recv(poll,{req=get,reply_c=rc});
def value(rc) : send(rc,v);
```

Our initial attempts at specifying the behavior of the sensor used the common LTL formula that is used to specify response properties:

```
ltl poll_sensor(rc): [](poll(rc) => X(<>value(rc,v)));
```

It turned out during analysis that the above formula was too general at it allowed unexpected behavior. This observation also influenced the specifications for the other components. We provide more details on this in the analysis section.

Instead, we use the following more precise LTL formula:

```
ltl poll_sensor(rc) :    ((!value(rc)) U poll(rc))
                      && [](poll(rc) => X(value(rc) &&
                                          X((!value(rc)) U poll(rc))));
```

Another equivalent LTL formulation of the same property is:

```
ltl poll_sensor(rc) :    ((!value(rc)) U poll(rc))
                      && [](poll(rc) => X(value(rc)))
                      && [](value(rc) => X((!value(rc)) U poll(rc)));
```

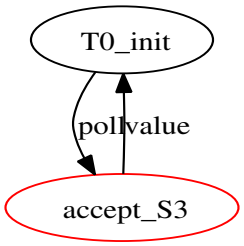


Figure 5.9: the Büchi automaton generated for the poll_sensor LTL formula

For both formulas above, the generated automaton is the same and shown in Figure 5.9. It illustrates how poll requests and responses alternate.

5.4 Implementation

In this section we describe how the different components are implemented as AAL modules. Including the interfaces, the implementation is about 1400 lines of AAL code.

Sensors

Sensors are the easiest components to implement. Each sensor also models a part of the environment and keeps a local variable that represents the reading of that sensor.

Poll requests are received in a main loop and handled by sending back the current value of the sensor. After each reply the value of the sensor is changed non deterministically to model an action of the environment. For example, the motion sensor can non deterministically pick between

`no_motion_detected` and `motion_detected`. Figure 5.10 shows the implementation of the motion sensor. The implementation for the other sensors is similar.

```
module MotionSensor {
  export MotionSensorInterf;

  const no_motion_detected : int = 0;
  const motion_detected : int = 1;
  type sensor_msg = { req : int, reply_c : chan[int] };
  const get : int = 1;
  const poll : chan[sensor_msg] = mkchan of sensor_msg[1];

  active process motion() {
    var value : int = no_motion_detected;

    function next_value() : (v:int) {
      if
      :: true -> v = no_motion_detected;
      :: true -> v = motion_detected;
      :: else -> assert false;
      fi
    }

    function handle_poll_request(m : sensor_msg) {
      if
      :: m.req == get ->
         sel
         :: send(m.reply_c, value) ->
            value = next_value();
         :: timeout -> assert false;
         les
      :: else -> assert false;
      fi
    }

    function run() {
      var m : sensor_msg;
      do
      :: true ->
         sel
         :: recv(poll,m) -> handle_poll_request(m);
         :: timeout -> break;
         les
      :: else -> assert false;
      od
    }
  }
}
```

Figure 5.10: the AAL implementation of the motion sensor

The User

The user is modeled as a loop which non deterministically picks what command to send to the master controller. Figure 5.11 shows some sample code from the implementation of the user module.

```
module User {
  import mc = MasterCtrlInterf;

  active process user() {
    function run() {
      var seq_no : int = 1;
      var reply_c : chan[mc.usr_cmd_reply] =
                    mkchan of mc.usr_cmd_reply[1];
      var m : mc.usr_cmd_reply;
      do
      :: true ->
         sel
         :: send(mc.usr_cmds,{cmd = mc.at_home_day,
                              seq_no = seq_no,
                              param = 0,
                              reply_c = reply_c}) ->
            sel
            :: recv(reply_c,m) ->
               assert m.status == mc.success;
               assert m.seq_no == seq_no;
            :: timeout -> assert false;
            les
         :: ... // send other commands
         les
      od
    }
  }
}
```

Figure 5.11: the AAL implementation of the user module

Appliance Controllers

Each appliance controller is implemented as a main loop in which it decides between receiving a message from the master controller and sending poll messages to sensors. Receiving a message from the master controller takes priority in order to avoid a live lock situation in which the controller only interacts with the sensors and completely avoids the master controller. Poll messages are sent only if the commands channel is empty and the controller is on. After each poll message sent, the controller waits for a reply from the sensor. When the reply is received the control algorithm is applied and

any necessary changes are made (for example, turning the lights on or off, turning on the heater or the AC, etc.) and the controller returns to the main loop. On a message from the master controller, the appliance controller sets its mode accordingly. If the message is `on` then before returning to the main loop, poll messages are sent and the replies handled. This ensures that at least one update is made each time the controller is turned on.

Figure 5.12 shows some sample code from the climate controller. The code not shown follows a similar pattern. Also, the implementation of the other appliance controllers is structured in the same way.

The Master Controller

The master controller is implemented as a main event loop in which it receives commands from the user. Each command is handled by setting the current mode and by sending `on` and `off` messages to the appliance controllers. Once a reply is received from all of them, a success message is sent back to the user. Figure 5.13 shows some sample code from the implementation of the master controller.

5.5 Analysis

During the development of the smart home application type checking and static analysis were always on and they helped identify minor errors that were promptly fixed. For example, forgetting to set a field in a message sent to a component or forgetting to assign the return value of a helper function. We have also set up our analysis to use concolic testing. This part forced us to first specify the interfaces.

Testing a sensor in isolation involves closing the system around it, which in this case, amounts to replacing the receiving of a poll message with a symbolic variable. A sensor does not import any interfaces so no stub modules had to be generated.

Testing the master controller involves generating stubs for the appliance controllers it is connected to. Messages from the user are replaced with symbolic variables.

Closing the system around an appliance controller requires the generation of stub modules for

```
module ClimateCtrl {
  export ClimateCtrlInterf;
  import ts = TemperatureSensorInterf;
  ... // the type and constant definitions corresponding to the interface
  const hysteresis : int = 2;

  active process climate() {
    var state : int = off;
    var curr_temp : int = 25; var target_temp : int = 25;
    var from_ts : chan[int] = mkchan of int[1];
    var heater : int = off; var cooler : int = off;

    function poll_sensors_and_update() {
      var temp : int;
      send(ts.poll, {req = ts.get, reply_c = from_ts});
      recv(from_ts, temp);
      curr_temp = temp;
      if
      :: curr_temp <= target_temp - hysteresis -> heater = on; cooler = off;
      :: curr_temp >= target_temp + hysteresis -> heater = off; cooler = on;
      :: else -> heater = off; cooler = off;
      fi
    }
    function handle_on(seq_no : int, reply_c:chan[cmd_reply]) {
      state = on;
      poll_sensors_and_update();
      send(reply_c,{seq_no = seq_no, status = success, value = 0});
    }
    function handle_command(m : cmd_msg) {
      if
      :: m.cmd == on  -> handle_on(m.seq_no, m.reply_c);
      :: ...
      fi
    }

    function run() {
      var m : cmd_msg;
      var len : int = 0;
      do
      :: true ->
         len = length(cmds);
         if
         :: len == 0 && state == 0 -> poll_sensors_and_update();
         :: else -> sel
                    :: recv(cmds,m) -> handle_command(m);
                    :: timeout -> assert false;
                    les
         fi
      :: else -> assert false;
      od
    }
  }
}
```

Figure 5.12: the AAL implementation of the climate controller

```
module MasterCtrl {
  export MasterCtrlInterf;
  import ic = IntLightCtrlInterf;
  ... // type and constant definitions from the interface

  const usr_cmds : chan[usr_cmd_msg] = mkchan of usr_cmd_msg[1];

  active process master() {
    var mode : int = at_home_day;
    var from_ic : chan[ic.cmd_reply] = mkchan of ic.cmd_reply[1];
    ...
    function operate_int_light(seq_no : int, op:int) {
      var m : ic.cmd_reply;
      send(ic.cmds, {seq_no = seq_no, cmd = op, reply_c = from_ic});
      recv(from_ic, m);
    }

    function handle_at_home_day(seq_no : int, reply_c : chan[usr_cmd_reply]) {
      var t : int;
      printf("mc: handling at_home_day, seq_no = %d\n", seq_no);
      mode = at_home_day;
      operate_int_light(seq_no, ic.on);
      ... // interact with the other appliance controllers
      send(reply_c, {seq_no = seq_no, status = success, value = 0});
    }

    function handle_usr_msg(m : usr_cmd_msg) {
      if
      :: m.cmd == at_home_day   -> handle_at_home_day(m.seq_no, m.reply_c);
      :: ...
      fi
    }

    function run() {
      var m : usr_cmd_msg;
      do
      :: true ->
         sel
         :: recv(usr_cmds,m) -> handle_usr_msg(m);
         :: timeout -> assert false;
         les
      :: else -> assert false;
      od
    }
  }
}
```

Figure 5.13: the AAL implementation of the master controller

the sensors it uses. Our first attempts at testing the climate controller revealed a problem with the specification of the temperature sensor. The generated stub module was sending more messages

than it was supposed to as replies to a poll request.

We were using the common LTL formula:

```
ltl poll_sensor(rc): [](poll(rc) => X(<>value(rc,v)));
```

To inspect the problem we looked at the automaton generated for this formula (Figure 5.14). We saw that the automaton was not quite what we expected.

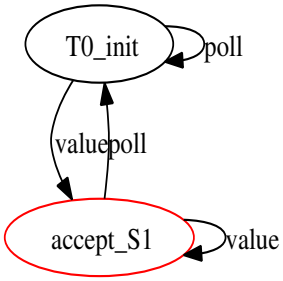


Figure 5.14: the Büchi automaton generated for the initial poll_sensor LTL formula

The automaton shows that the sequence of messages can start with a reply which should not be allowed as no poll request was made. To fix this problem we added the constraint that there should not be a `value` before a `poll` at the beginning of the execution. This is formalized in LTL as: `((!value(rc)) U poll(rc))`. The change did not completely fixed our problem.

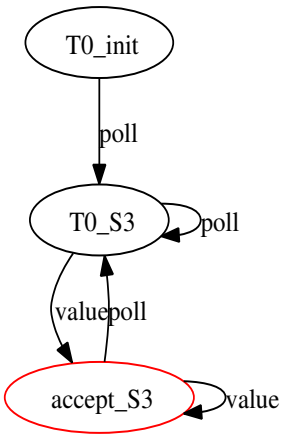


Figure 5.15: the Büchi automaton generated for the poll_sensor LTL formula after removing the possibility of initial value messages

The new automaton (Figure 5.15) looks closer to what we expected, but it allows any number of contiguous poll requests to be answered by any number of contiguous value replies. This does not correspond to what we want: a single poll request followed by a single value reply.

To remove the possibility of multiple contiguous poll requests we have imposed the restriction that a poll request is immediately followed by a value reply. To remove the possibility of multiple value replies to the same poll request we have imposed the restriction that after we send a value reply, no more values are sent until a poll message is received.

The new LTL formula is the one presented in the Specifications section:

```
ltl poll_sensor(rc) :    ((!value(rc)) U poll(rc))
                         && [](poll(rc) => X(value(rc)))
                         && [](value(rc) => X((!value(rc)) U poll(rc)));
```

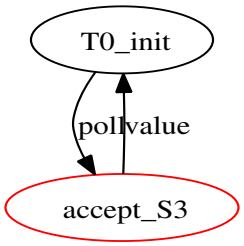


Figure 5.16: the Büchi automaton generated for the poll_sensor LTL formula

The automata generated (Figure 5.16) corresponds exactly to the desired behavior of a sensor.

Chapter 6

Conclusions and Future Directions

In this chapter we present a summary of the thesis and hint at how our work could be further extended in the future.

6.1 Summary

Current practices for embedded systems software development do not structure the source code in a way that easily lends itself to analysis. The software is developed in an "analysis-agnostic" style that often obfuscates design abstractions and hampers the analysis. The analysis, if carried at all, is a post-mortem activity and cannot easily exploit available knowledge about the system. The source code has to be manually inspected in order to reconstruct a model of how it works and use this model as input for the analysis tools. For example, when trying to apply model checking all relevant state must first be identified so it can be tracked. Usually this state is not all in the same place as it might be stored in multiple variables spread across multiple files. As the source code grows larger it becomes difficult to identify all the parts of the state.

Analysis tools tend to be too general and do not exploit available knowledge about the system being analyzed. Furthermore, they do not provide facilities for users to customize the analysis to the needs of the application. For example, knowing that the system uses a priority-based scheduler could reduce the number of executions a model checker has to explore. Without having a way to customize the model checking algorithm for this particular case the benefit might be lost.

In this thesis we have introduced a new methodology for the development and analysis of em-

bedded systems software. We call this methodology "analysis-aware" because it connects the way software is developed with the way software is analyzed. We have built a framework for software development and analysis in which the analysis happens interactively as the software is being written. The framework is composed of two parts: a new executable specification language, called AAL, that can also be seen as a programming language, and a set of analysis tools based on static analysis, testing, and model checking.

Designing a new language allowed us to include features that lead to an analysis-friendly code structure. The source code is organized in modules that implement the functionality specified in interfaces. Interfaces include specifications that allow our analysis tools to automatically generate stub modules which are used to close the system when analyzing other parts of the code. There are no global variables which reduces the risk of race-condition errors. The language provides primitives for implementing testers and model checkers directly in the language. Users can write their own model checking algorithms that exploit the particularities of the system they are designing.

We have defined a type system and a formal semantics for AAL. Having a new language made this part easier than trying to define the semantics of an existing language. Features that are not essential and that are hard to specify were left out or replaced with better-behaved features (e.g., using references instead of pointers).

On the analysis side, we have introduced a new approach to static analysis. A rule-based engine is used to maintain a knowledge base of facts about the program. Static analysis checks are encoded as queries of the knowledge base. For more involved checks, that cannot be answered directly, the static analysis rules can be encoded as rules in the language of the rule-based engine that compute the result. Our static analysis framework is extensible with user-defined checks. The encoding of a static analysis rule tends to be short and closely correspond to the definition of the rule. We provide some examples of common static analysis checks that can be used by the user as a starting point for developing new checks.

We have developed a method for closing an open system based on interface specifications. Because the analysis happens interactively as the code is being developed, it is common for the system to be

open and have unimplemented components. Closing the system is necessary to perform any form of dynamic analysis. We have built a concolic testing framework that dynamically generates and executes test cases in order to explore as many paths of the code as possible.

We have described how model checking algorithms can be implemented directly in AAL as scheduler processes. We give an example of such an algorithm that can be used by the users as a starting point for defining custom model checking algorithms.

We have applied the analysis-aware methodology to the development of a smart home application. We have demonstrated how the specifications of the application are encoded as interface and how to build and analyze models implementing these interfaces.

6.2 Future Directions

Language Features

There are multiple directions in which the AAL language can evolve. The type system could be extended to support variant types (also called algebraic data types) and generics. The values of a variant type are grouped into classes according to the constructor that was used to create the value. Each constructor can take as arguments values of other types. Pattern matching is used to deconstruct a value of a variant type. Variant types can be used to build more interesting data structures. For example, a (non-empty) binary tree in which the nodes hold values of type T could be defined like this: `type Tree[T] = Leaf(T) | Node(T,Tree[T],Tree[T]);`

Another extension would be the addition of function declarations to interfaces. This would allow the implementation of abstract data types as modules. The specification of LTL formulas defined in an interface could be extended to include function calls of the functions declared in the interface.

More advanced iteration constructs would be interesting to have. For example, to iterate over the elements of a set `s` one could use a statement like: `for e in s do stmt_list od`.

The set of built-in functions could be extended to provide access to some of the primitives of the underlying operating system.

Analysis Tools

The static analysis framework could be extended to include more checks. Some of these checks might be more efficient if parts of them are built-in directly as primitives of the rule-based engine. CLIPS, the rule-based engine we have picked, can easily be extended to provide new functions. An example of such extensions would be functions that let us directly interact with an SMT solver from CLIPS.

Incremental analysis could be another extension. We currently analyze a whole file when it is being saved. We could identify the parts that have changed since the last save and focus the analysis on them. Any results previously obtained for the parts that did not change could be reused.

Another extension could be distributed concolic testing. As test cases are dynamically generated they could be distributed to a set of workers that explore them.

We would like to further explore the link between scheduler processes and runtime verification. Just as we use scheduler processes to implement model checking algorithms, they can be used to implement monitors for AAL programs.

Appendix A

Notation

A.1 Preliminaries

In this section we describe the notation used throughout the thesis. We assume familiarity with set theory and the associated notation. Most of our notation follows that in [66].

Relations

A relation R on sets A_i $(i = 1, n)$ is a subset of $\times_{i=1}^{n} A_i = A_1 \times A_2 \times ...A_n$: $R \subseteq \times_{i=1}^{n} A_i$.

For elements $x_i \in A_i$ and a relation R on A_i $(i = 1, n)$ we sometimes write $R(x_1, ..., x_n)$ for $(x_1, ..., x_n) \in R$.

We usually define relations by giving a list of *inference rules* for what elements are in the relation. The rules can be recursive, but at least one of them, called a base case, must be non-recursive.

We use the following format for specifying inference rules:

$$\frac{}{R(a,b)}\ \text{base-case} \qquad\qquad \frac{R(x,y) \qquad R(y,z)}{R(x,z)}\ \text{recursive-rule}$$

The meaning of an inference rule is that if the propositions above the line are true (also called the premises of the rule) then the proposition below the line is true (also called the conclusion of the rule).

To prove that some elements are in a relation we use a *derivation*. A derivation is a chaining of

rules. An example of a derivation is given below:

$$\dfrac{\dfrac{}{R(1,2)}\ \text{base-case} \qquad \dfrac{}{R(2,3)}\ \text{base-case}}{R(2,3)}\ \text{recursive-rule}$$

Functions

A *total function* $f : A \to B$ is defined in terms of the relation graph $graph(f) \subseteq A \times B$, $graph(f) = \{(x, y) \mid f(x) = y\}$ having the following properties:

- *singled-valued*: $(x, y) \in graph(f) \wedge (x, y') \in graph(f) \Rightarrow y = y'$;
- *total*: $\forall x \in A.\exists y \in B.(x, y) \in graph(f)$.

A partial function $f : A \hookrightarrow B$ is just like a function except that the *total* property is not required.

When we say that f is a function we mean that f is a total function.

A partial function $f : A \hookrightarrow B$ is *defined* in $x \in A$ if there exists an element $y \in B$ such that $(x, y) \in graph(f)$.

A partial function $f : A \hookrightarrow B$ is *not defined* in $x \in A$ if there does not exist an element $y \in B$ such that $(x, y) \in graph(f)$.

For a set S we use the notation $S^?$ to represent the set $S \cup \{\perp\}$ where $\perp \notin S$.

Sometimes we want to extend a partial function $f : A \hookrightarrow B$ to a total function $f^? : A \to B^?$:

$$\begin{array}{rcll} f^?(x) & = & f(x) & \text{if } f \text{ is defined in } x \\ & = & \perp & \text{otherwise.} \end{array}$$

Environments

A *T-environment* $\Gamma : I \to T^?$ is the extension of a partial function that associates elements from some index set I (usually names) to elements in some set T.

We use ϵ to denote the empty environment:

$\epsilon(x) = \perp$ for all $x \in I$.

We define the following operations on environments:

Removing an element from an environment:

$$(\Gamma - x)(y) \quad = \quad \perp \qquad \text{if } x = y$$

$$= \quad \Gamma(y) \quad \text{otherwise.}$$

Updating an environment:

$$\Gamma[x = t](y) \quad = \quad t \qquad \text{if } x = y$$

$$= \quad \Gamma(y) \quad \text{otherwise.}$$

The notation $x : t$ is a shorthand for $\Gamma(x) = t$.

The notation $x \in \Gamma$ is a shorthand for $\Gamma(x) \neq \perp$.

For an environment $\Gamma : I \to T^?$, and $s \in S$, the environment $(s, \Gamma) : I \to (S \times T)^?$ is defined as follows:

$$(s, \Gamma)(x) \quad = \quad (s, \Gamma(x)) \quad \text{if } x \in \Gamma$$

$$= \quad \perp \qquad \text{otherwise.}$$

For two environments $\Gamma : I \to T_1^?$ and $\Delta : I \to T_2^?$ we define $\Gamma \cup \Delta : I \to (T_1 \cup T_2)^?$ as follows:

$$(\Gamma \cup \Delta)(x) \quad = \quad \Gamma(x) \quad \text{if } x \in \Gamma$$

$$= \quad \Delta(x) \quad \text{otherwise.}$$

Sometimes we implicitly lift a set of pairs P to the corresponding environment. This can be done only if P is the graph of some partial function (i.e., it does not contain two elements with the same item on the first position).

We use environments to represent the contexts of judgments.

Judgements

A *judgment* has the form $A_1, A_2, ..., A_n \vdash B$ with the semantics that whenever all A_i, $i = 1, n$, are *true* then B is also *true*.

A judgement $A_1, A_2, ..., A_n \vdash B$ is equivalent to the logic formula $A_1, A_2, ..., A_n \to B$.

Judgements are used to simplify the format of inference rules and proofs.

Sometimes in a judgment we use one or more environments instead of A_i, $i = 1, n$.

$\Gamma \vdash B$ stands for $x_1 : t_1, x_2 : t_2, ..., x_n : t_n \vdash B$ where x_i, $i = 1, n$, are all the points where Γ is not $\perp$.

A.2 Language Syntax

We use Extended Backus-Naur Form [2] to define the syntax of a programming language. We define the syntax as a grammar with *syntactic forms* being the nonterminals and *tokens* being the terminal symbols.

Each syntactic form is defined by a set of rules called production rules which, much like inference rules, specify what strings are in the language. An example of a production rule is:

```
expression ::= number
```

where `expression` is the syntactic form of expressions and `number` is a token representing integer numbers.

Multiple rules can be combined in a single one using `|` as a separator like in the following example:

```
expression ::= number
          | expression '+' expression
          | expression '-' expression
          | '(' expression ')'
```

A string is in the language if and only if a *parse tree* can be constructed for it. Much like a derivation, a parse tree for a string s demonstrates how the production rules were (recursively) applied to establish that s is in the language. For example, the parse tree showing that `(5 + 4) - 29` is in the language of expressions is depicted in Figure A.1.

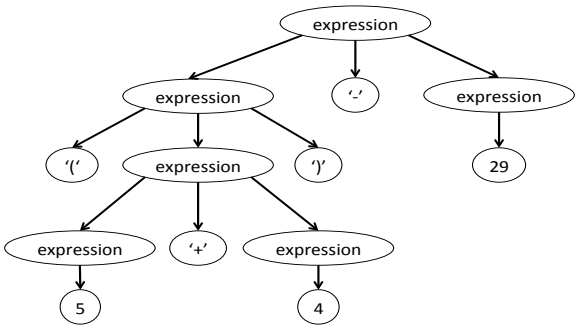


Figure A.1: parse tree showing that "(5+4)-29" is in the language of expressions

An *abstract syntax tree*(AST) is a more compact representation of a parse tree where irrelevant details have been left out (like the use of parentheses). For example, the AST for the previous parse

tree is shown in Figure A.2.

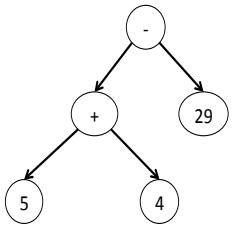


Figure A.2: example of an abstract syntax tree

A.3 Proof Format

Most of our proofs involve some form of induction, the main ones being structural induction and induction on derivations.

Structural Induction

We use structural induction to prove that a proposition P holds for all elements of a syntactic form, for example, to show that something is true for any expression.

Structural induction is an instance of well-founded induction. The set of syntactic forms is a well-founded partial order with the "smaller" relation being defined as follows. For two syntactic forms sf_1 and sf_2, $sf_1 < sf_2$ if and only if sf_1 is a component of sf_2. In other words, the parse tree of sf_1 is a subtree of the parse tree of sf_2. The smallest syntactic forms are the ones composed of only terminals.

The proofs proceeds by a case split on all possible production rules that could have been used last to show that the syntactic form is in the language. The production rules that consist of only terminals are called the bases cases of the induction, the other (recursive) cases are the induction steps.

We start by first proving each bases case. Next, for each recursive case the induction hypothesis is that the property holds for each component of the syntactic form because these components are smaller.

Induction on Derivations

For a relation R defined using a set of inference rules, we can prove properties of the form: "if $R(x)$ then $P(x)$" (if an element x is in the relation R then some other property P also holds for x) by induction on the derivation of $R(x)$ (sometimes also called induction on the proof of $R(x)$).

Induction on derivations is similar to structural induction. It is also an instance of well-founded induction. There is a well-founded partial order on derivations. A derivation d_1 is smaller than a derivation d_2 if d_1 appears in d_2.

The proof proceeds by a case split on all the possible rules that could have been used to infer $R(x)$. The rules that are not recursive are called the base cases, the other ones are called the induction steps.

For each base case we prove that $P(x)$ holds for it. For each induction step we can assume the induction hypothesis that $R(y) \rightarrow P(y)$ for each premise y used in the recursive inference rule of $R(x)$ (because $y < x$).

Appendix B

Details of the Semantics

B.1 Proof of Lemma 3

Proof. We proceed by induction on the proof of $\Gamma, H \vdash Rho$.

Case 1: the proof has the form:

$$\frac{}{\epsilon, H \vdash \epsilon}\ \text{var-env-empty}$$

This case is trivially true because $\epsilon(x) = T$ is false for all x.

Case 2: The proof has the form:

$$\frac{H \vdash v : T_1 \qquad \Gamma, H \vdash Rho}{\Gamma[y = T_1], H \vdash Rho[y = v]}\ \text{var-env-var}$$

Case 2.1: $x = y$

From the definition of variable environments we have: $Rho[y = v](x) = Rho[y = v](y) = v$.

From the definition of type environments we have: $\Gamma[y = T_1](x) = \Gamma[y = T_1](y) = T_1$.

From the premise we have $\Gamma[y = T)1](x) = T$ which implies $T = T_1$.

From the premise of the rule we have $H \vdash v : T_1$ which implies $H \vdash v : T$.

Case 2.2: $x \neq y$

From the definition of variable environments we have $Rho[y = v](x) = Rho(x)$. From the definition of type environments we have $\Gamma[y = T_1](x) = \Gamma(x)$.

By the induction hypothesis we have that there exists a value v' such that $Rho(x) = v'$ and $H \vdash v' : T$. □

B.2 Expression Evaluation

$Eval(c, Rho, H, Locs) = (c, Locs)$

$Eval(l, Rho, H, Locs) = (l, Locs)$

$Eval(x, Rho, H, Locs) = (Rho[x], Locs)$

$Eval(unop(e), Rho, H, Locs_1) = (delta\_unop(unop, v), Locs_2)$ where

$(v, Locs_2) = Eval(e, Rho, H, Locs_1)$

$Eval(binop(e_1, e_2), Rho, H, Locs_1) = (delta\_binop(binop, v_1, v_2), Locs_3)$ where

$(v_1, Locs_2) = Eval(e_1, Rho, H, Locs_1)$

$(v_2, Locs_3) = Eval(e_2, Rho, H, Locs_2)$

for $binop \neq /$ and $binop \neq \%$

$Eval(binop(e_1, e_2), Rho, H, Locs_1) = (delta\_binop(binop, v_1, v_2), Locs_3)$ where

$(v_1, Locs_2) = Eval(e_1, Rho, H, Locs_1)$

$(v_2, Locs_3) = Eval(e_2, Rho, H, Locs_2)$ with $to\_int(v_2) \neq 0$

for $binop = /$ or $binop = \%$

$Eval(binop(e_1, e_2), Rho, H, Locs_1) = \perp$ where

$(v_1, Locs_2) = Eval(e_1, Rho, H, Locs_1)$

$(v_2, Locs_3) = Eval(e_2, Rho, H, Locs_2)$ with $to\_int(v_2) = 0$

for $binop = /$ or $binop = \%$

$Eval(\{f_1 = e_1, ..., f_n = e_n\}, Rho, H, Locs) = (\{f_1 = v_1, ..., f_n = v_n\}, Locs_n)$ where

$(v_1, Locs_1) = Eval(e_1, Rho, H, Locs)$

...

$(v_n, Locs_n) = Eval(en, Rho, H, Locs_{n-1})$

$Eval(e.fi, Rho, H, Locs_1) = (vi, Locs_2)$ where

$(\{f_1 = v1, ..., f_i = v_i, ..., f_n = v_n\}, Locs_2) = Eval(e, Rho, H, Locs_1)$

$Eval(ref\ e, Rho, H, Locs_1) = (l, Locs_2 \cup \{(l, ref\ v)\})$ where

$(v, Locs_2) = Eval(e, Rho, H, Locs_1)$

$l \notin H \cup Locs_2$

$Eval(e\$, Rho, H, Locs_1) = (v, Locs_2)$ where

$(l, Locs_2) = Eval(e, Rho, H, Locs_1)$

$(l, ref\ v) in H \cup Locs_2$

$Eval(mkchan\ \text{of}\ T[esz], Rho, H, Locs_1) = (l, Locs_2 \cup \{(l, chan(n, []))\})$ where

$(v, Locs_2) = Eval(esz, Rho, H, Locs_1)$

$n = to\_int(v)$

$n > 0$

$l \notin H \cup Locs_2$

$Eval(mkarray\ \text{of}\ T[esz], Rho, H, Locs_1) = (l, Locs_2 \cup \{(l, array[v1, .., vn])\})$ where

$(v, Locs_2) = Eval(esz, Rho, H, Locs_1)$

$n = to_int(v)$

$n > 0$

$v_i = default(T),\ i = 1, n$

$l \notin H \cup Locs_2$

$Eval(e[idx], Rho, H, Locs_1) = (vi, Locs_3)$ where

$(l, Locs_2) = Eval(e, Rho, H, Locs_1)$

$(l, array[v1, ..., vn]) \in H \cup Locs2$

$(vidx, Locs_3) = Eval(idx, Rho, H, Locs_3)$

$i = to\_int(vidx) + 1$

$0 < i \leq n$

$Eval([], Rho, H, Locs) = (l, Locs \cup (l, \{\}))$

$Eval([e_1, ..., e_n], Rho, H, Locs) = (l, Locs_n \cup \{(l, [v_1, ..., v_n])\})$ where

$(v_1, Locs_1) = Eval(e_1, Rho, H, Locs)$

...

$(v_n, Locs_n) = Eval(e_n, Rho, H, Locs_{n-1})$

$l \notin H \cup Locs_n$

$Eval(\{\}, Rho, H, Locs) = (l, Locs \cup \{(l, \{\})\})$

$Eval(\{e_1, ..., e_n\}, Rho, H, Locs) = (l, Locs_n \cup \{(l, \{v_1, ..., v_n\})\})$ where

$(v_1, Locs_1) = Eval(e_1, Rho, H, Locs)$

...

$(v_n, Locs_n) = Eval(e_n, Rho, H, Locs_{n-1})$

$l \notin H \cup Locs_n$

$Eval(\{e_k1 : ev_1, ..., ek_n : ev_n\}, Rho, H, Locs) = (l, Locs_{2n} \cup (l, \{k_1 : v_1, ..., k_n : v_n\})\}$ where

$(k_1, Locs_1) = Eval(ek_1, Rho, H, Locs)$

$(v_1, Locs_2) = Eval(ev_1, Rho, H, Locs_1)$

...

$(k_n, Locs_{2n-1}) = Eval(ek_n, Rho, H, Locs_{2n-2})$

$(v_n, Locs_{2n}) = Eval(ev_n, Rho, H, Locs_{2n-1})$

$l \notin H \cup Locs_{2n}$

B.3 Proof of Theorem 1

Proof. Structural induction on e. We only give a scketch of the proof here and show how it can be carried for some cases. The other cases can be carried following the same pattern.

Case 1: e is a constant c.

From the definition of $Eval$ we have $Eval(c, Rho, H, Locs_1) = (c, Locs_1)$. We have to prove that $H \cup Locs_2 \vdash c : T$.

The proof of $\Gamma \vdash e : T$ must have had the form: $\dfrac{H \vdash c : T}{\Gamma, H \vdash c : T}$ expr-val

Because it is the only applicable rule, its premise must hold and so we have $H \vdash c : T$ which implies $H \cup Locs_2 \vdash c : T$.

Case 2: e is a location l. This case is similar to the case for a constant.

Case 3: e is a variable x.

From the definition of $Eval$ we have $Eval(x, Rho, H, Locs_1) = (Rho[x], Locs_1)$.

The proof of $\Gamma \vdash e : T$ must have had the form:

$$\frac{\Gamma(x) = T}{\Gamma, H \vdash x : T} \text{ expr-var}$$

Since it is the only applicable rule, its premise must hold so we have $\Gamma(x) = T$.

The premise of the theorem includes $\Gamma \vdash Rho$.

We can apply Lemma 3 to get that there exists a value v such that $Rho[x] = v$ and $H \vdash v : T$ which implies $H \cup Locs_2 \vdash v : T$.

Case 4: e is $ref\ e_1$.

The proof of $\Gamma \vdash e : T$ must have had the form: $\dfrac{\Gamma, H \vdash e_1 : T_1}{\Gamma, H \vdash ref\ e_1 : ref[T]}$ expr-ref

The expression e_1 is smaller than e and from the induction hypothesis we have that there exists a value v such that either $Eval(e_1, Rho, Locs_1) = (v, Locs'_1)$ and $H \vdash v : T$, or e_1 contains a division by zero.

If e_1 contains a division by zero the e also contains a division by zero because e_1 is a sub-expression of e.

If e_1 does not contain a division by zero then we can apply the definition of $Eval$ and we have:

$Eval(ref\ e_1, Rho, H, Locs) = (l, Locs'_1 \cup \{(l, ref\ v)\})$ where

$(v, Locs'_1) = Eval(e_1, Rho, H, Locs_1)$

$l \notin H \cup Locs'_1$.

We have to prove that $H \cup Locs_2 \vdash l : T$.

$Locs_2 = Locs'_1 \cup \{(l, ref\ v)\}$.

By applying the typing rule for locations that are references we get $Locs_2 \vdash l : ref[T]$ which implies $H \cup Locs_2 \vdash l : ref[T]$.

Case 5: e is $e1\$$.

The proof of $\Gamma \vdash e : T$ must have had the form:

$$\frac{\Gamma, H \vdash e_1 : ref[T]}{\Gamma, H \vdash e_1\$: T} \text{ expr-deref}$$

The expression e_1 is smaller than e and from the induction hypothesis we have that there exists a

value v such that either $Eval(e_1, Rho, Locs_1) = (l, Locs_2)$ and $H \cup Locs_2 \vdash l : ref[T]$, or e_1 contains a division by zero.

If e_1 contains a division by zero the e also contains a division by zero because e_1 is a sub-expression of e.

If e_1 does not contain a division by zero then we can apply the definition of $Eval$ and we have: $Eval(e_1\$, Rho, H, Locs_1) = (v, Locs_2)$ where

$(l, Locs_2) = Eval(e_1, Rho, H, Locs_1)$

$(l, ref\ v) in H \cup Locs_2$.

We have to prove that $H \cup Locs_2 \vdash v : T$.

From the definition of $Eval$ for this case we have $(l, ref\ v) in H \cup Locs_2$ which implies

$H \cup Locs_2 \vdash v : T$.

Case 6: e is $unop(e_1)$. This case follows directly from Lemma 1 with the special case that if e_1 contains a division by zero then e also contains a division by zero.

Case 7: e is $binop(e_1, e_2)$. This case follows directly from Lemma 2 with the special case that if either e_1 or e_2 contain a division by zero then e also contains a division by zero. Note that Lemma 2 guarantees that the only case where the binary operation does not produce a value when e_1 and e_2 produce values (do not contain division by zero) is the case where e contains a division by zero. □

Bibliography

[1] Datalog User Manual, http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html.

[2] Extended Backus-Naur Form (EBNF): ISO/IEC 14977, 1996.

[3] Flymake: an on-the-fly syntax checker for Emacs, http://flymake.sourceforge.net/.

[4] ISO/IEC 13211: Information technology - Programming languages - Prolog. International Organization for Standardization, Geneva.

[5] WebSphere ILOG JRules BRMS, http://www-01.ibm.com/software/integration/business-rule-management/jrules-family/.

[6] *Standard ECMA-367: Eiffel: Analysis, Design and Programming Language, 2nd Edition.* ECMA International, 2006.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[8] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Formal Methods for Components and Objects*, 4111:364–387, 2006.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 3362:49–69, 2005.

[10] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach.* Springer-Verlag, 2007.

[11] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[12] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[13] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. *Computer Aided Verification*, 6174:354–359, 2010.

[14] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. *ACM SIGPLAN Notices*, 44(1):114–126, 2009.

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *8th USENIX Symposium on Operating Systems Design and Implementation*, 8:209–224, 2008.

[16] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. *Computer Aided Verification*, 2404:359–364, 2002.

[17] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[18] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. *Theorem Proving in Higher Order Logics*, 5674:23–42, 2009.

[19] David R. Cok and Joseph R. Kiniry. Esc/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 3362:108–128, 2005.

[20] ConsumerReports. Keep your phone safe, http://www.consumerreports.org/cro/magazine/2013/06/keep-your-phone-safe/index.htm, June 2013.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[22] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. *Programming Languages and Systems*, 3444:21–30, 2005.

[23] Oege De Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. *Generative and Transformational Techniques in Software Engineering II*, 5235:78–133, 2008.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.

[25] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[26] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[27] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver, http://yices.csl.sri.com.

[28] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. *ACM SIGPLAN Notices*, 46(1):411–422, 2011.

[29] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *4th USENIX Symposium on Operating System Design and Implementation*, 4:1–16, 2000.

[30] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. *ACM Symposium on Applied Computing*, pages 2103–2110, 2010.

[31] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. *Computer Aided Verification*, 4590:173–177, 2007.

[32] Mihai Florian. SCALE: source code analyzer for locating errors. Master's thesis, California Institute of Technology, 2010.

[33] Mihai Florian, Ed Gamble, and Gerard J. Holzmann. Logic Model Checking of Time-Periodic Real-Time Systems. *AIAA Infotech@Aerospace*, 2012.

[34] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. *4th USENIX Windows System Symposium*, pages 59–68, 2000.

[35] Patrice Godefroid. Model checking for programming languages using VeriSoft. *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[36] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, 2005.

[37] Patrice Godefroid, Michael Y. Levin, David A. Molnar, et al. Automated Whitebox Fuzz Testing. *Network and Distributed System Security*, 8:151–166, 2008.

[38] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[39] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[40] Gerard J. Holzmann. Static source code checking for user-defined properties. *Integrated Design and Process Technology*, 2, 2002.

[41] Gerard J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2004.

[42] Gerard J. Holzmann. Reliable software development: extending the programmer's toolbox. *European Joint Conferences on Theory and Practice of Software*, 2011.

[43] Gerard J. Holzmann and Mihai Florian. Model checking with bounded context switching. *Formal Aspects of Computing*, 23(3):365–389, 2011.

[44] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling Large Verification Problems with the Swarm Tool. *SPIN*, 5156:134–143, 2008.

[45] Bart Jacobs and Frank Piessens. The verifast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

[46] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Lecture Notes in Computer Science*, 6617:41–55, 2011.

[47] Capers Jones. Measuring Defect Potentials and Defect Removal Efficiency. *Crosstalk, The Journal of Defense Software Engineering*, 2008.

[48] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[49] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[50] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[51] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, 523:175–188, 1999.

[52] K. Rustan M. Leino. This is Boogie 2. *Manuscript KRML*, 178, 2008.

[53] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. *Logic for Programming, Artificial Intelligence, and Reasoning*, 6355:348–370, 2010.

[54] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. *40th Annual Hawaii International Conference on System Sciences*, pages 261a–261a, 2007.

[55] Nancy G. Leveson and Clark Savage Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[56] Kenneth L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[57] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice Hall, 1988.

[58] Bertrand Meyer. Eiffel as a framework for verification. *Verified Software: Theories, Tools, Experiments*, 4171:301–307, 2008.

[59] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[60] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM SIGPLAN Notices*, 42(6):446–455, 2007.

[61] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[62] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[63] Hanne R. Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., 1992.

[64] Denis Oddoux and Paul Gastin. LTL 2 BA, http://www.lsv.ens-cachan.fr/ gastin/ltl2ba/.

[65] Scott Owens. A sound semantics for OCaml light. *Programming Languages and Systems*, 4960:1–15, 2008.

[66] Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002.

[67] Amir Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[68] Mark Proctor, Michael Neale, Peter Lin, and Michael Frandsen. Drools documentation. *JBoss*, 5(05):2008, 2008.

[69] Gary Riley. CLIPS: A Tool for Building Expert Systems, http://clipsrules.sourceforge.net, August 2013.

[70] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30(5):263–272, September 2005.

[71] Jeremy Siek. Type Safety in Five Easy Lemmas, http://siek.blogspot.com/2012/08/type-safety-in-five-easy-lemmas.html.

[72] Jeremy Siek. Type Safety in Three Easy Lemmas, http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

[73] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. *First Symposium on Logic in Computer Science*, 1986.

[74] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode: an eclipse plugin for semantic code search. *22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 880–881, 2007.

[75] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[76] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An Operational Semantics and Type Safety Proof for C++-like Multiple Inheritance. Technical Report RC23709, IBM, 2005.

[77] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3):16, 2007.