SOFTWARE MODEL CHECKING FOR CONFIDENTIALITY

Pavol Černý

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2009

Rajeev Alur
Supervisor of Dissertation

Jianbo Shi
Graduate Group Chairperson

To my parents, Vladimír and Mária Černí

Acknowledgments

This thesis would not have been possible without the help of many people. I would like to express my gratitude to them all.

Rajeev Alur has been a good and patient advisor. I can only hope that some of his ability to think clearly and to find compelling solutions to intricate problems has been imprinted on me. I learned from him how to be a researcher. It would be impossible to pinpoint his contributions large and small to this work; his encouragement has been very important as well.

Andy Gordon, Andre Scedrov, Scott Weinstein and Steve Zdancewic have been members of my thesis committee. I have greatly benefited from their guidance. This thesis was markedly improved because of their critical reading and valuable suggestions. I am especially grateful to Scott for his kind words and scientific collaboration over the past two years.

The faculty, staff and students at the Computer and Information Science department have provided an excellent academic environment. Swarat Chaudhuri and Mikhail Bernadsky have become good friends. Our technical discussions have helped in my work, while our more philosophical discussions have been thoroughly enjoyable.

My friends David, Micah, Mike, Celina and Matt, Cecilia, Colleen, and Marie have made my stay in Philadelphia fun, exciting and memorable. I thank the people in the Penn Newman center and Saint Agatha - Saint James parish who helped me deepen my understanding of the Catholic faith.

My parents Vladimír and Mária Černí have taught me the most important things

I know. I am very grateful to them for their love and support. It is to them that I dedicate this thesis.

ABSTRACT

SOFTWARE MODEL CHECKING FOR CONFIDENTIALITY

Pavol Černý

Rajeev Alur

Protecting confidentiality of data manipulated by programs is a growing concern in various application domains. In particular, for extensible software platforms that allow users to install third party plugins, there is a need for an automated method that can verify that programs preserve confidentiality of data. Our central thesis is that software model checking, an algorithmic, specification-driven program analysis, is an effective way of checking whether programs leak confidential information. Software model checking has emerged as a successful technique for analyzing programs with respect to correctness requirements. However, existing methods and tools are not applicable for specifying and verifying confidentiality properties. In this thesis, we develop a specification framework for confidentiality, novel decision procedures for finite state systems as well as for classes of programs, and an abstraction-based program analysis technique.

A property f over the program variables is said to be *confidential* if the adversary cannot infer the truth of f based on the observed behavior of the program at runtime and the knowledge of the source code of the program. Confidentiality therefore depends not only on individual program executions, as is the case for classical temporal logics, but on sets of observationally equivalent executions. Our specification framework thus consists of a new, richer computation tree model and correspondingly enriched temporal logics.

For finite state systems, we develop an algorithm for the model checking problem for these temporal logics, and we show that the problem is PSPACE-complete for a fragment that is expressive enough to allow specifications of information flow properties such as "agent A does not reveal x (a secret) until agent B reveals y (a password)". For infinite-state software systems, we develop two approaches. First,

we study decidability of confidentiality for programs that access an array. The confidentiality requirement specifies that secret information contained in the array should not be leaked. We develop novel decision procedures for classes of programs that access an array whose length is potentially unbounded, and whose elements range over a potentially infinite, ordered data domain. We show that the reachability problem for these programs is decidable, and that the result extends to confidentiality. Second, we develop an automated abstraction-based analysis technique for confidentiality. We show that both over- and under- approximation is needed for sound analysis. Given a program and a confidentiality requirement, our technique produces a formula that is satisfiable if the requirement holds.

We evaluate the abstraction-based technique by analyzing Java bytecode of a set of methods of J2ME midlets for mobile devices. Midlets are third-party programs designed to enhance the capabilities of the device and often have a legitimate reason to access data on the mobile device (such as the list of contacts or a phone book), as well as a legitimate reason to send outgoing messages or requests. We demonstrate that our approach can be effectively used for certification of these programs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Our central thesis is that algorithmic, specification-driven analysis is an effective way of checking whether programs leak confidential information. To this end, we develop a specification framework for confidentiality, novel decision procedures for finite state systems as well as for classes of programs, and an abstraction-based program analysis technique, and evaluate the latter on a set of Java methods.

1.1 Motivation

Security and confidentiality are growing concerns in software and system development [75, 43]. The problem of ensuring confidentiality of data being processed by computing systems is long-standing (see [73] for an early approach), yet increasingly important, as systems are parts of larger networks. The nodes on the network need to interact, often by communicating sensitive data over the network and thus giving opportunity to active or passive (eavesdropping) attackers to gain knowledge about secret data.

Protecting confidentiality is essential in many areas of computing. For example, in health care systems, it is required by law [3], in order to maintain privacy. Second, collaboration between companies (or divisions in the same company, or teams of

researchers) requires data sharing. However, in many cases, parties might not be willing to share all the information. Third, many platforms (such as mobile phones or web browsers) are highly extensible. They allow users to download third-party programs that can access data on a host system, as well as to communicate with the outside world. The question of how to ascertain that an attacker cannot easily get information about classified data is central in all these application areas.

A specific application context consists of Java midlets. Midlets are third-party programs designed to enhance features of mobile devices. J2ME midlets have legitimate reasons to access data on the mobile device (such as the list of contacts or a phone book), and a legitimate reason to send outgoing messages. Therefore an access control mechanism that would prevent the programs from performing either of these tasks would be too restrictive. The recently released report [65] describes several attack vectors through which a malicious midlet can stealthily release private data through connections such as text messages, emails, or http requests. Thus there is a question of how to ensure that a given (possibly malicious) midlet does not leak confidential information.

A verification tool would be very useful in this context. The J2ME security model uses the concept of *protection domains* (see MIDP specification [4]). A protection domain is associated with permissions for security-sensitive APIs. Midlets that need more privileges have to be either signed by a certificate from a trusted certification authority (CA) or registered with the manufacturer of the device, depending on the policy of the vendor. The source code of the midlets is not analyzed, the registration serves only to enable the possibility of tracking the harmful midlets and their authors. A verification tool could thus be used for guaranteeing that registered midlets do not leak confidential information.

We will use a simplified version of the EventSharingMidlet from a J2ME manual [1] as an example. It uses the security-sensitive PIM[1] API. It allows accessing

[1]Personal information management. See https://java.sun.com/javame/index.jsp.

```
  //get the phone number
  number = phoneBook.
             elementAt(selected);
  //test if the number is valid
  if ((number == null)
            ||  (number ==  "")) {
    //output error
  } else {
    String message = inputMessage();
    //send a message
    sendMessage(number,message);
  }
}
```

Figure 1.1: EventSharingMidlet

```
...
if ((number == null)
          ||  (number ==  "")) {
  //output error
} else {
if (contains(phoneBook,"555-55")) {
  String message = inputMessage();
  //send a message
  sendMessage(number,message);
}
}
```

Figure 1.2: EventSharingMidlet (malicious version)

the native data (such as the phone book) of the phone. For this API, a confidentiality requirement might be that phone numbers in the phone book should not be leaked. EventSharingMidlet allows the user to plan an event and send information about it to contacts in her phone book. The core of the midlet is in Figure 1.1. The phone number of the intended receiver is retrieved from the phone book. A test is performed whether this number is valid. If so, a message is prepared and sent. We suppose that the attacker can observe all messages. The property to be kept secret for the example is whether a particular number, say "555-55" is in the phone book. Let us denote it by *secret*. We want to verify that the attacker cannot infer whether *secret* holds or not based on his or her knowledge of the program and observation of the outputs (in this case, the variable `message`). Note that the outgoing message does depend on the variable `phoneBook` (via the control-flow dependency). However, in this case, the answer is that the attacker cannot infer whether the *secret* holds or not. Now let us consider the case when midlet is malicious as in the Figure 1.2. The attacker inserted a test on whether the number "555-55" is in the phone book. Now if a message (any message) is sent, the attacker can infer that *secret* holds.

The notion of confidentiality The example in Figure 1.2 shows that the attacker can learn it even though the secret is never directly revealed in the messages the attacker sees. Therefore our formalization of confidentiality has to capture also the more indirect information flows.

Intuitively, a property f over the program variables is *confidential* if the adversary cannot infer the truth of f based on the observed behavior of the program at runtime and the knowledge of the source code of the program. More precisely, a property f is confidential if for every execution r, there exists another execution r', such that r and r' disagree on the truth of f, but are equivalent to the observer. Two executions are equivalent to the observer, if they produce the same sequence of observations (observations can be, for example, inputs and outputs of the program).

1.2 Current state of the art

Software model checking We investigate the possibilities for using automated verification techniques for verification of confidentiality. Automated software model checkers have made great progress in recent years, having become efficient and widely deployed in industry for both bug finding and certification of correctness. For example, the tool SDV (based on SLAM [13]) is distributed with the development kit for Windows device drivers. Existing software model checkers (such as SLAM, BLAST [52] or Synergy [47]) are designed for checking linear-time properties of programs, and are based on abstraction, symbolic state-space traversal, and counter-example guided abstraction refinement. More precisely, a program is modeled as a logical structure M and a requirement φ is given in a formal language, such as a temporal logic. The model checking problem is then to determine whether φ holds for M. If the system in question is infinite or prohibitively large, we can employ abstraction techniques to produce a smaller system M^A such that if φ holds for M^A, then φ holds for M. On the other hand, the abstraction is often not complete. That

is, if φ does not hold in M^A, then we cannot conclude whether φ holds in M. However, if the decision method returns a counterexample, it is possible to analyze it. It can prove to be a counterexample to φ in the original model M. If it is not, we are not able to conclude whether φ holds in M, but we can use the counterexample to refine the abstraction M^A to make it more precise.

Unfortunately, software model checking techniques and tools are not directly applicable to verification of confidentiality properties. The reasons are two-fold. First, confidentiality is not a property of a single execution, and in fact, is not specifiable in μ-calculus, which is more expressive than the specification languages of these tools. Second, definition of confidentiality involves both universal and existential quantifiers. Therefore, abstraction based on solely over-approximation (or solely on under-approximation) is not sufficient for checking confidentiality. More precisely, let us consider two programs `P1` and `P2`, such that `P1` is an over-approximation of `P2`, that is, the set of executions of `P2` is included in the set of executions of `P1`. The fact that confidentiality holds for `P1` does not imply that confidentiality holds for `P2` (and vice-versa).

Language-based security Noninterference is a security property often used to ensure confidentiality. Informally, it can be described as follows: "if two input states share the same values of low variables then the behaviors of the program executed from these states are indistinguishable by the observer". See [70] for a survey of the research on noninterference and [63] for a Java-based programming language with a type systems that supports information flow control based on noninterference.

Noninterference is too strong for the specification of confidentiality for the example in Figure 1.1. The reason is, briefly, that the variable `number` depends on the variable `message` via control flow. The definition of confidentiality we present can be seen as a relaxation of noninterference. It is relaxed by allowing the user to specify which predicate(s) should stay secret; noninterference requires that *all* properties

of high variables stay secret. It is well-known that the noninterference requirement needs to be relaxed in various contexts. See [72] for a survey of methods for defining such relaxations via declassification. Compared to these methods, the main benefit of our approach is automation, as our method allows verification of existing programs without requiring annotations by the programmer.

1.3 Contributions

1.3.1 Formalizing the notion of confidentiality

The definition of confidentiality should capture the intuition that a fact is secret if an observer who knows the source code of the program, and can observe (partially) the behavior of the program at runtime, cannot infer whether the fact holds or not. This suggests that the definition of confidentiality should be parameterized by (1) the distinguishing power of the observer, (2) the property to be kept secret, and (3) the executions of interest. We formalize the definition in the standard framework of labeled transition systems, which can represent, for example, semantics of programs. The distinguishing power of the observer is modeled by an equivalence relation on the runs of the system. A property is *conditionally confidential* if, for every run (a sequence of states) of interest, there is an equivalent run such that only one of these two runs satisfies the property. We demonstrate that the parametric definition is flexible enough to capture the confidentiality requirement in many programs and systems. Furthermore, we show that by varying these three parameters, it is possible to capture a variety of *possibilistic* definitions of secrecy found in the literature.

1.3.2 Specification framework

A natural question is whether such a notion of confidentiality can be specified using temporal logics. There are several reasons why such a specification language would

be useful. First, it would give the user a lot of flexibility in specifying what should be kept secret. Second, the model-checking algorithms developed for temporal logics could be used to verify confidentiality. However, we show that confidentiality is not specifiable in logics such as the μ-calculus. On an intuitive level, the reason why this holds can be easily explained by the fact that classical tree logics cannot relate distinct paths in the computation tree.

This motivates our interest in extending temporal logics to reasoning about secrecy requirements of systems. To be able to specify properties such as confidentiality, we propose to enrich the traditional tree model with edges that capture observational indistinguishability: for an agent a, an a-labeled edge is added between two nodes if the observable behaviors of the agent a along the paths to these nodes are identical. Note that above we used an observer observing the program execution. For concurrent programs and protocols, we generalize this setting to multiple observers, one for each agent. We can thus express what an agent has revealed. One can now extend classical tree logics CTL and μ-calculus by a next operator over the new edges in the enriched structure. We denote the operator by EI and we call the logics CTL$\approx$ and $\mu_{\approx}$, respectively. These logics, interpreted over tree models augmented with path equivalences, can express confidentiality properties. To specify that the agent a keeps the value of a boolean variable x secret, we simply have to assert that for all tree nodes, the value of x is different from the value of x in one of its successors along a-labeled edges. We can integrate temporal reasoning with reasoning about confidentiality to specify requirements such as "agent a does not reveal x until event e happens."

1.3.3 Decision problems

Model-checking for finite-state systems We study complexity and decidability of the model checking problem for CTL$\approx$ and $\mu_{\approx}$. We show that if we consider multiple equivalence relations (modeling what multiple agents have revealed), the

model checking problem is undecidable for $\mu_{\approx}$. For CTL$\approx$, we present an algorithm that keeps track of paths equivalent according to an observer via a type of subset construction. We show that the key parameter is the *nesting depth* of the specification. Informally, when we need to evaluate a formula φ after jumping across an a-labeled edge, then an additional layer of subset construction is required to process b-equivalence, for agents $b \neq a$. The complexity of the algorithm is thus nonelementary. We also show the corresponding lower bound.

We show that, if we restrict the nesting depth to 1 the model checking problem for CTL$\approx$ is PSPACE-complete, and is EXPTIME-complete for $\mu_{\approx}$. These fragments of CTL$\approx$ and $\mu_{\approx}$ are expressive enough to capture information flow properties such as "agent A does not reveal x (a secret) until agent B reveals y (a password)" and to capture partial information games.

Array-accessing programs A natural confidentiality requirement for a program is that it should not reveal to the outside world (too much) information about the data structure it is accessing. Verification questions concerning programs are undecidable in general. However, we identify a class of programs for which confidentiality is decidable. The class contain programs that access a single array whose length is potentially unbounded, and whose elements range over pairs from $\Sigma \times D$, where Σ is a finite alphabet and D is a potentially unbounded data domain. The program can have Boolean variables, index variables ranging over array positions, and data variables ranging over D. Programs can access Σ directly, but can only perform equality and order tests on elements of D. The programs are built using assignments, conditionals, and `for`-loops over the array. Even with these restrictions, one can perform interesting computational tasks including searching for a specific value, finding the minimum data value, checking that all values in the array are within specific bounds, or checking for duplicate data values.

We show that the reachability problem, while undecidable in general, is (1)

PSPACE-complete for programs in which the array-accessing `for`-loops are not nested, (2) solvable in EXPSPACE for programs with arbitrarily nested loops if array elements range over a finite data domain, and (3) decidable for a restricted class of programs with doubly-nested loops. The third result establishes connections to automata and logics defining languages over data words. Furthermore, we show that the decidability results for reachability extend to verification of confidentiality properties of programs.

1.3.4 Abstraction-based analysis of Java methods

Successful approaches to software verification of programs with infinite or very large state spaces are based on abstraction. Our method proceeds in two steps. First, we compute a formula φ that is valid if the conditional confidentiality requirement holds. In order to do so, we show that one needs to consider both an over- and an under-approximation of reachable states for every program location. We use user-specified invariants for over-approximation. In all the examples we considered, the invariants that were used are simple enough, and could have been discovered by existing techniques for automatic invariant generation. The under-approximation is specified by a bound on the number of loop iterations and a bound on the size of the array.

The second step consists of deciding the validity of the obtained formulas, which involves both universal and existential quantifiers. For a restricted fragment, we devise an efficient decision method using satisfiability solvers for quantifier free formulas. The restriction contains a subset of Java that contains booleans, integers, on which we allow linear arithmetic, as well as data from an unbounded domain D equipped with only equality tests. Furthermore, the programs can have arrays, which are *a priori* unbounded in length and whose elements are from D. We leverage the restrictions on the program expressions, as well as the specific form of the obtained formulas, to devise a decision method based on using an existing SMT solver. The

restriction on the program expressions is that the domain D (over which the universal quantification takes place) has only equality tests. Therefore given a formula φ, it is possible to produce an equivalent formula φ' where the universal quantification takes place over a bounded domain. As φ' can then be seen as a boolean combination of existential formulas with no free variables, its validity can be decided using an SMT solver.

1.3.5 Implementation and experimental evaluation

We have implemented a prototype tool called ConAn (for CONfidentiality ANalysis). It takes as input a program in Java bytecode, a secret and a condition under which the secret should hold. The tool uses WALA [2] library to process Java bytecode. It then performs the analysis on an intermediate representation called WALA IR and produces formulas whose satisfiability is subsequently checked by the SMT (satisfiability modulo theories) solver Yices [37].

We confirmed feasibility of our solution by checking confidentiality for methods from J2ME midlets, as well as for data- structure accessing methods from the core Java library. The main practical question is whether the formulas produced can be decided by existing tools in reasonable time. The running times for the methods we analyzed were all under two seconds. The size of the methods we analyzed was small, with the largest one having little over 100 lines. However, the methods we chose contain parts of the midlets that are key from the security point of view, that is, the methods that access the confidential data (such as the phone book) and send out messages. Furthermore, we show that the method size is typical for midlets by presenting statistics for lines of code per method for 20 most downloaded open source midlets.

	Reachability	**Conditional Confidentiality**
Specification language	CTL, μ-calculus	**CTL$\approx$, $\mu_{\approx}$-calculus**
Finite-state systems	NL-complete	**PSPACE-complete**
Array-accessing programs	**PSPACE-complete (non-nested loops)**	**PSPACE-complete (non-nested loops)**
Programs (Java methods)	Over-approximation for sound analysis (of unreachability)	**Both over- and under-approximations needed for sound analysis**

Table 1.1: Summary of results

1.4 Summary

Table 1.1 provides a summary of the results and contributions of the dissertation in the context of previous results on reachability analysis. Our results are printed in bold. First, in Chapter 2, we provide a definition of *conditional confidentiality* that is flexible (i.e., it is possible to specify what property should be kept secret and the condition under which its secrecy should hold) and that captures direct and indirect information flows. We show (also in Chapter 2) that even though conditional confidentiality is not expressible in standard temporal logics such as the μ-calculus, it is possible to extend these logics in a way that allows specification of both temporal and confidentiality properties. The frameworks (CTL$\approx$ and $\mu_{\approx}$) allow specification of properties such as "a secret is revealed only after a certain condition is met". Second, in Chapter 3, we show that the problem of checking confidentiality for finite-state system is PSPACE-complete. Third, confidentiality requirements often involve (complex) data structures. We examine programs that access arrays and show that for certain classes the reachability and confidentiality properties are decidable (Chapter 4). Fourth, we show in Chapter 5 that using solely over- (under-) approximation is not sufficient for sound analysis of programs for conditional confidentiality. We present an analysis approach based on both over-

and under- approximation. Furthermore, for a certain restricted class of programs, we develop an efficient implementation based on an SMT-solver and we provide experimental evaluation of our techniques.

Joint work

Chapters 2 and 3 evolved from two articles: *Preserving Secrecy under Refinement* [8] that appeared in the 33rd International Colloquium on Automata, Languages and Programming (ICALP 2006) and was coauthored with Rajeev Alur and Steve Zdancewic, and *Model Checking on Trees with Path Equivalences* [6] that appeared in the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007) and was coauthored with Rajeev Alur and Swarat Chaudhuri. Chapter 4 is based on joint work with Rajeev Alur and Scott Weinstein *Algorithmic Analysis of Array-Accessing Programs* [7] that will be presented in the 18th Conference on Computer Science Logic (CSL 2009). Chapter 5 and 6 contain results from a joint paper with Rajeev Alur *Automated Analysis of Java Methods for Confidentiality* [24] that was published at the 21st International Conference on Computer Aided Verification (CAV 2009).

Chapter 2

Specifying Confidentiality

We introduce a general framework for specifying confidentiality. We first note that confidentiality is dependent on the distinguishing power of the observer, and is parameterized by the fact that should be kept secret, as well as the circumstances under which it should be kept secret.

We will use the standard verification framework — labeled transition systems. The distinguishing power of the observer is modeled by an equivalence relation on runs of the system. The property to be kept secret is, in general, a property of the run. We give the user also the power to specify the set of runs that are of interest, that is, the set of runs for which the confidentiality should be preserved. A property is confidential if, for every run of interest, there is an equivalent run such that exactly one of these two runs satisfies the property. We introduce a formal definition of confidentiality along these lines for transition systems. Turning to programs, we then provide an equivalent characterization of confidentiality in the style of classical data flow definitions.

A natural question is whether such a notion of confidentiality can be specified using temporal logics. There are several reasons why such a specification language would be useful. First, it would give the user a lot of flexibility in specifying what, under what conditions, and when should be kept secret. Second, the model-checking

algorithms developed for temporal logics could be used to verify confidentiality. Temporal logics have been successfully used for specifying and verifying requirements of reactive systems such as distributed protocols [27, 67]. In particular, in the branching-time approach, a system is modeled as a labeled tree whose paths correspond to executions of the system; a specification describes a set of correct trees; and verification reduces to a membership question [55]. Typical branching-time specification languages include CTL, the μ-calculus, and tree automata [54, 28]. The theoretical foundations of this approach are now well understood, and model checkers such as SMV implement highly optimized algorithms for verifying branching-time requirements of finite-state systems [23, 26]. However, we show that confidentiality is not specifiable in logics such as the μ-calculus. On an intuitive level, the reason why this holds can be easily explained by the fact that classical tree logics cannot relate distinct paths in the computation tree and that confidentiality depends on a set of equivalent runs.

This motivates our interest in extending temporal logics to reasoning about secrecy requirements of software systems. To be able to specify properties such as confidentiality, we propose to enrich the traditional tree model with "jump-edges" that capture observational indistinguishability. More precisely, consider a tree T whose nodes are labeled with truth values to atomic propositions P. For an agent a, if the assignment to the propositions $O(a) \subseteq P$ captures the observable behavior of a, then two tree nodes are considered a-equivalent if the paths from the root to these nodes agree on the values of propositions in $O(a)$ at every step. We convert the tree T into a graph $IG(T)$ by adding, for every agent a of interest, an a-labeled edge between every pair of a-equivalent nodes and $\bar{a}$-labeled edge between every pair of a-inequivalent nodes at the same level. One can view $IG(T)$ as a Kripke model, where both nodes and edges have labels, and interpret standard tree logics over it.

Tree logics interpreted over tree models augmented with (in)equivalence edges have rich expressiveness. To specify that the agent a keeps the value of a variable

x secret, we simply have to assert that for all tree nodes, the value of x is different from the value of x in one of its successors along a-labeled edges. One can integrate temporal reasoning with secrecy to specify requirements such as "agent a does not reveal x unless agent b reveals y." We also show that using the tree models with jump edges, partial information games are expressible in our framework. Games are useful for specifying requirements as well as for formulating synthesis questions. In *partial information* games, the strategy can depend only the sequence of observations, rather than the complete execution of the system.

2.1 Confidentiality requirements

A *transition system* (TS) T is a tuple $(Q, \delta, \lambda, q_0)$, where Q is a set of states, $\delta \subseteq Q \times Q$, is a transition relation, $\lambda : Q \to 2^P$ is a map labeling each node with a set of propositions, and $q_I \in Q$ is an initial state.

A sequence $r = q_0 q_1 \ldots$ of states is a *run* of the transition system T iff q_0 is the initial state q_I and $\forall i : 0 \leq i < |r| \Rightarrow (q_i, q_{i+1}) \in \delta$. Let $R(T)$ be the set of all runs of the TS T.

A *property* α is a subset of the set of runs, i.e. $\alpha \subseteq R(T)$. A *state-property* is a property that depends only on the last state of a run. Formally, α is a state-property iff there is a set of states $Q_\alpha \subseteq Q$ such that $r \in \alpha$ iff $r = q_0 q_1 \ldots q_n$ and q_n is in Q_α.

Given this model of systems, we want to define what an observer can see and what he or she can infer based on those observations. The observer cannot see everything about the current run of the system, that is to say, in general, several runs can correspond to the same observation. Let O be a subset of the set of propositions P, defining the set of observables. We will use the set O to define equivalences on paths in a transition system T as follows. Let the map $Tr : Q \to 2^P$, defined as $Tr(q) = \lambda(q) \cap O$ for all q, return the observables at a state q of T. We lift this map to runs of T by defining $Tr(q_0 q_1 \ldots) = Tr(q_0)\, Tr(q_1) \ldots$. Runs r and r' are *equivalent*

(written as $r \approx_O r'$) iff $Tr(r) = Tr(r')$.

For a property α, the observer is able to conclude that α holds, if α holds for all the runs that correspond to his or her observations. He or she is able to conclude that α does not hold, if it does not hold for all the runs that correspond to the observations. The third possibility is that the observer is not able to conclude whether α holds or not. We will thus need to use a three-valued domain, $\{\top, \bot, m\}$ (true, false, maybe), and a partial order that models the knowledge the observer has. $\sqsubseteq$ is the following partial order on $\{\top, \bot, m\}$: $m \sqsubseteq m, m \sqsubseteq \top, m \sqsubseteq \bot, \bot \sqsubseteq \bot, \top \sqsubseteq \top$.

Function IP – *inferable properties*, is a function that, given a run r, a property α, and the set of observables O, represents the knowledge of the observer about the property α after the run r. $IP(r, \alpha, O) = \top$ if $\forall r' : r' \approx_O r \Rightarrow r' \in \alpha$, $IP(r, \alpha, O) = \bot$ if $\forall r' : r' \approx_O r \Rightarrow r' \notin \alpha$ and $IP(r, \alpha, O) = m$ otherwise.

Our notion of confidentiality depends on three parameters: (1) the distinguishing power of the observer, modeled by the set O, (2) the property to be kept secret α, and (3) the set of runs of interest, β: instead of requiring a property α to be secret in every run of the system, we may want to focus only on a subset β of runs that are of interest, e.g. the set of all terminating runs. This leads to the following formalization of conditional confidentiality:

Definition 1. (*Conditional confidentiality*) Let T be a labeled transition system and α and β be two properties. The property α is a *conditionally confidential* in β w.r.t. O if for all $r \in \beta$, $IP(r, \alpha, O) = m$.

Example 1. Consider the following two programs. All the variables in programs A and B are boolean.

A: x=?; y=0; z=x; send z;

B: x=?; y=0; z=y; send z;

It is easy to see how they can be modeled as transition systems in our framework.

The proposition in states encode valuations of variables. There are two more propositions: M_1, encoding whether a new message was sent, M_2, encoding the value of the message. The set O of observable propositions is composed of the two propositions M_1 and M_2. Note that the input ($x = ?$) is not seen by the observer.

We want to analyze what an observer might infer about whether or not $x = 0$ during the execution of the program if he or she can observe what the program sends. Let us suppose that the observer sees an execution where 0 was sent. For the program A the observer, after having seen the execution, can conclude that $x = 0$ holds. For the program B the observer does not know whether $x = 0$. We can conclude that for program A, the state property $x = 0$ is not a secret in the set of all runs w.r.t. $\approx$ and it is a secret for program B.

We present the following examples in order to show the relation of our definition to several standard information-flow properties such as noninterference or Perfect Security Property. Note that we need to vary the three parameters O, α, and β.

Noninterference

Consider the standard formulation of termination insensitive noninterference [70]. It is defined using low and high variables, where low variables are visible to the observer and high variables are not. Noninterference can be then formulated informally as follows: "if two input states share the same values of low variables then the behaviors of the program executed from these states are indistinguishable by the observer". The observer thus does not see any variation in the valuation of low variables, caused by variation in the valuation of high variables.

We show that in our setting, we can specify that the observer cannot infer a (user-specified) property based on his observation of low variables. Consider a classic requirement such as "a secret key should stay secret." In our framework, this can be expressed as "the property P of the secret key stays secret", that is, the property P will not be revealed.

In this sense, our framework of conditional confidentiality can be seen as a relaxation of noninterference. It enables specification of the properties that should stay secret, as opposed to noninterference, which requires that there is no information flow from high to low variables, that is, it requires that all properties stay secret.

We now formalize the idea that conditional confidentiality is a relaxation of noninterference. We consider a deterministic transition system T, and the standard definition of noninterference as described above. We then construct a set NI of properties, such that noninterference holds for T if and only if all the properties in NI are conditionally confidential.

We model noninterference for deterministic programs. A transition system is deterministic if each node has at most one outgoing edge. Such a system is a model of the state space of deterministic programs. The only nondeterminism is allowed in the choice of initial state. This models the input of the program. The state labeling function λ can be used to encode the valuation of the variables. The set L of observable propositions is composed of the propositions encoding the low variables. We denote the equivalence relation on states induced by L by $\approx_L$.

Noninterference can be then formalized as follows. Noninterference holds for the transition system T, if for all nodes s_0, s_1, s'_0, s'_1 we have: ($s_0 \rightarrow^* s_1$ and $s'_0 \rightarrow^* s'_1$ and $s_0 \sim_L s'_0$) implies $s_1 \sim_L s'_1$, where $\rightarrow^*$ is the reflexive and transitive closure of the transition relation $\rightarrow$.

In what follows, we will use the notation $s|_H$ ($s|_L$) to denote the restriction of a state s to high (low) variables.

We extend the definition of conditional confidentiality to sets of properties. Let $\mathcal{S}$ be a set of properties that does not include the empty set and the set of all runs (as the truth of these properties is leaked tautologically). Let β be the set of runs of interest, as before. The set $\mathcal{S}$ is conditionally confidential in β w.r.t. O if for all properties $\alpha \in \mathcal{S}$, α is conditionally confidential in β w.r.t. O.

We define a particular set of properties $\mathcal{P}$ that characterizes noninterference.

Informally, it can be characterized as the set of all subsets of valuations of high variables. To formalize this definition, we define the following notions. An *H-valuation* A is a valuation (i.e. a truth assignment) to all propositions encoding high variables. An *H-property* is a property (i.e. a subset of runs) defined by a set of H-valuations as follows: an H-property α defined by a set of valuation ρ holds for r, i.e. $r \in \alpha$ if and only if there exists a valuation $A \in \rho$ such that for the first state of r, that is, r_0, we have $r_0|_H = A$. Let $\mathcal{P}$ be the set of all possible H-properties α. Finally, let β be the set of all terminated runs and recall that L is the set of propositions encoding low variables. We can now state the desired lemma.

Proposition 1. *$\mathcal{P}$ is conditionally confidential in β w.r.t. O iff noninterference holds for T.*

Proof. Let us suppose that noninterference does not hold for T. We show that this implies that $\mathcal{P}$ is not conditionally confidential.

If noninterference does not hold, this implies that there exist states s_0, s_1, s_0', s_1' such that $s_0 \rightarrow^* s_1$, $s_0' \rightarrow^* s_1'$, $s_0 \sim_L s_0'$ and $s_1 \not\sim_L s_1'$. Let us consider the H-property α defined by the following set of H-valuations:

$$\{s|_H \mid s|_L = s_0|_L \wedge \exists s' : s \rightarrow^* s' \wedge s'|_L = s_1|_L\}$$

We first note that α does not define an empty set of runs, as evidenced by the run represented by $s_0 \rightarrow^* s_1$, and it does not contain all runs, as evidenced by the run $s_0' \rightarrow^* s_1'$.

Secondly, the property α is not conditionally confidential. The reason is that when an observer (who see the values of the low variables) sees that the values of low variables at the start are equal to $s_0|_L$ and that at the end of the run are equal to $s_1|_L$, then the observer can conclude that α holds. As α is in $\mathcal{P}$, we can conclude that $\mathcal{P}$ is not conditionally confidential.

It is straightforward to prove that if noninterference holds, then $\mathcal{P}$ is conditionally confidential.

□

Perfect Security Property

Let us consider the Perfect Security Property (PSP) [83]. It is an information-flow property defined in a trace-based setting. In order to define it, we divide the trace labels into low-security and high-security categories. The observer knows the specification of the system - i.e. the set of all possible traces (sequences of labels) and he or she can observe low-security labels. PSP ensures that the observer cannot deduce any information about occurrences of high-security events.

We can model the PSP in our framework by choosing an appropriate labeling of vertices in a standard way. The set of observables O_{psp} will contain propositions corresponding to low-security labels. For each high-security label h, we define the property α_h: a run r is in α_h if h occurs in r. Now we can conclude that PSP holds iff α_h is secret in β_{all} w.r.t. O_{psp} for all high-security labels h, where β_{all} is the set of all runs of the system.

2.2 Confidentiality for programs

We consider Control Flow Graphs (CFGs) of programs. A CFG G is a tuple (V, E, L, i, e), where V is a finite set of nodes, $E \subseteq V \times V$ is a set of edges, i is the entry node, and e is the exit node. We assume, without loss of generality, that a CFG of a program has a unique entry and a unique exit node.

The function L labels each node in V with a command defined by the following syntax:

```
C ::=   v = Expr
      | assume BExpr
```

The commands include assignment and sequential composition. The command `assume BExpr` ensures that `BExpr` holds in the current execution. If it is not the case,

```
result = -1; i = 0;
while (i < n) {
  if (A[i]==key) { result=A[i]; }
  i++;
}
hist = append(hist,result);
```

Figure 2.1: Program `ArraySearch`

the execution fails. Note that the language of expressions (and boolean expressions) is left unspecified here. The exact details are not relevant for the definition of confidentiality; the framework will be instantiated in later chapters to allow automated analysis.

The only restriction we place on the CFG is that of reducibility [5]. A CFG is *reducible* if for each loop it is possible to uniquely identify a *loop header*, i.e. the entry point to the loop. Reducible CFGs capture all the standard intraprocedural control flow constructs and are more easily amenable to automated analysis. For example, a statement `if B then P1 else P2` can be captured using `assume` statements as follows: The node preceding the conditional would have two successors, one labeled with `assume B; P1`, the other labeled with `assume (not B);P2`, assuming that `P1` and `P2` contain only straight-line code. We do not allow procedure calls.

Example 2. Let us consider the program `ArraySearch` in Figure 2.1. The program takes an array and an integer `key` as an input. It scans through the array to find if there is an element whose value is equal to `key`, and if so, returns this element. The CFG of this program is in Figure 2.2.

In what follows, the CFGs are assumed to be annotated with assignments to a history variable `hist`. The variable is of type list and it stores the sequence that the observer can see. The first command of a program initializes `hist` to the empty list. Where the other annotations with an assignment to `hist` are placed depends on a particular security model. If an observer can see every change of the value of a

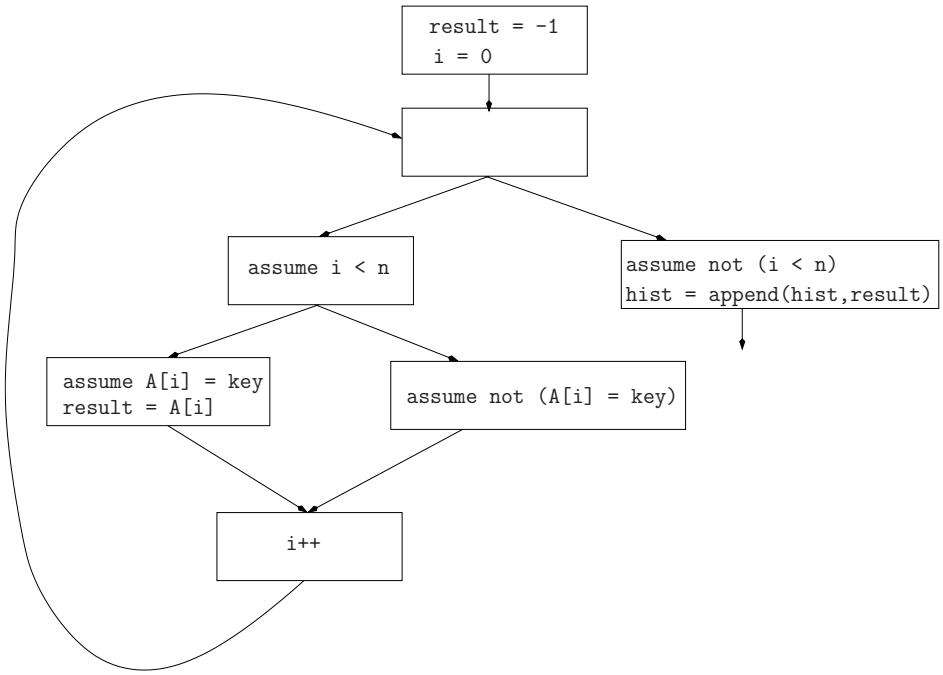

Figure 2.2: CFG

variable, then every command that can change the value of the variable is annotated with an assignment to `hist`. If an observer sees only values sent via a particular API, the calls to this API are annotated. For example, a call of a method `send(d)` which sends a message (visible to the observer) containing the value of variable `d`, is annotated by a command that appends the value of `d` to the variable `hist` (`hist := append(hist,d)`). Let us emphasize that this annotation is not program-specific, and can be done automatically (and is done automatically by our tool). In what follows, we will assume that `hist` contains of a list of values from the data domain D. (The definition and the analysis can be extended to capture also boolean values being visible.)

Let us fix a CFG G. Let V be the set of its variables. A *state* is a valuation of the variables V. Given a node l, the set R_l denotes a set of states that are reachable at l. An *observation* is a sequence of data values. It represents what the observer

sees during an execution of the program.

We can now instantiate Definition 1 for programs. The following version will allow simpler definition and proofs in the rest of this thesis. To define the distinguishing power of the observer, we will use the `hist` variable. Let e be the exit node of the program. Let *secret* and *cond* be predicates over states of the program.

Definition 2. Let h be an observation, let s_0, s_1, s_2 be states. The predicate *secret* is confidential w.r.t. the condition *cond* if and only if

$$\begin{aligned} &\forall h(\exists s_0 : s_0 \in R_e \wedge s_0 \models cond \wedge s_0[hist] = h) \Rightarrow \\ &\quad (\exists s_1 : s_1 \in R_e \wedge s_1 \models secret \wedge s_1[hist] = h \wedge \\ &\quad \exists s_2 : s_2 \in R_e \wedge s_2 \not\models secret \wedge s_2[hist] = h) \end{aligned} \tag{2.1}$$

We rephrase the definition in order to convey the intuition behind it. We say that an execution (a sequence of states) produces a observation h if $s[hist] = h$, where s is the last state of the execution. Two executions are equivalent iff they produce the same observation. This notion of equivalence captures when the observer cannot distinguish between two executions. Let us call a observation h feasible, if there exists a a state s in R_e, such that $s[hist] = h$. Intuitively, the definition says that for all feasible observations h, if there exists an execution for which the condition *cond* holds, then there exists an equivalent execution for which *secret* holds, and an equivalent execution for which $\neg$*secret* holds. Therefore the definition ensures that the observer cannot infer whether *secret* holds or not.

Example 3. Let us consider again the program `ArraySearch` in Figure 2.1. The secret we would like to protect is whether the array contains 7. We therefore define *secret* to be $\exists i : A[i] = 7$. Now let us consider the observations the observer sees. Such an observation contains a single number, the final value of `result`. If the observer sees the value 7, he or she can conclude that 7 is in the array. Therefore confidentiality does not hold. However, the program should preserve confidentiality as long as `key` is not equal to 7. Thus we set *cond* to be `key` $\neq 7$. In this case, it is

easy to see that confidentiality is preserved. Intuitively, by observing the final value of the `result`, the observer only knows that this value is in the array. If the size of the array is at least 2, the observer does not know whether 7 is or is not in the array. As the size of the array is unknown to the observer, we can conclude that the confidentiality of the secret is preserved. (Note however, that if the observer knows that the size of the array is 1, the confidentiality of *secret* does not hold. If the final value of `result` is not equal to -1, and is not equal to 7, then the observer can infer that the array does not contain 7.)

2.3 Specifying confidentiality in temporal logics

We have presented a definition of confidentiality. We now examine the possibilities of specifying confidentiality in temporal logics, and combining notions of confidentiality and time in such specifications.

Specifying confidentiality in classical temporal logics

It is well-known that confidentiality cannot be expressed as a predicate on a single trace and hence cannot be specified in linear-time specification languages such as linear temporal logic (see, for example, [60], for a proof). We prove that confidentiality is not a branching-time property either.

Let P be a set of propositions. We consider labeled, unranked, unordered, infinite trees of the form $T = (V, E, \lambda, r)$, where V is an infinite set of nodes, $E \subseteq V \times V$ is a set of tree edges, $\lambda : V \to 2^P$ is a map labeling each node with the set of propositions holding there, and $r \in V$ is the root of the tree. A *path* in T is a sequence of nodes $\pi = v_0 v_1 v_2 \ldots$ such that $v_0 = r$ and for all i, v_i is the parent of v_{i+1}. Note that each node can be associated with a unique path (the path that leads from the root to this node) and vice-versa. Note that a tree can be seen as a transition system, where states correspond to vertices of the tree and edges are the parent-child edges. We

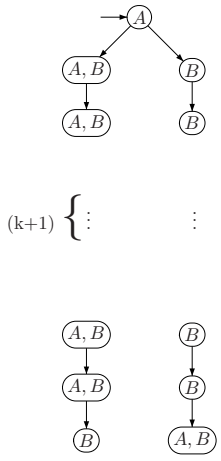

Figure 2.3: Confidentiality is not a regular property.

now show that conditional confidentiality is not specifiable in branching-time logics on trees, and therefore it is not specifiable for transition system.

Let $P = \{A, B\}$, and let α be a state-property that holds for a run iff its last state is labeled by A. The set of observables O is the set $\{B\}$. Let β be the set of runs that contain every run, except of the run that contains only the root node.

Theorem 2. *The set S of trees T such that α is secret in β is not a regular tree-language.*

Proof. For a proof by contradiction, suppose that S is regular. The fact that α is secret in β corresponds to the fact that at each depth d ($d > 0$) of the tree, there is a node in α and a node not in α.

It is easy to prove that this is not a regular property. Let us suppose that conditional confidentiality is a regular language, i.e. that there exists a tree automaton $\mathcal{A}$ (with k states) that accepts a tree only if the property α is secret in β. We will prove by contradiction, using a technique similar to pumping lemma, that this is not the case.

Consider the tree T in Figure 2.3. In T, the confidentiality of α is preserved, therefore there exists an accepting run of $\mathcal{A}$ on T. As $k + 1$ is the length of the part

of the left branch where A holds, there are at least two distinct positions that will be labeled by the same state. Therefore, one could cut the left branch between those two positions, leading to a tree where at a certain depth l, there will be only nodes where $\neg A$ holds. Thus the property α will not be secret for the trace of length l, as the observer will be able to conclude that α is false. □

Corollary 3. *The set of trees T such that α is secret in β is not definable in μ-calculus.*

Proof. The results follows from Theorem 2 and the fact that regular tree languages and μ-calculus have the same expressive power [38]. □

Note that it is possible to devise algorithms based on standard model-checking for special cases of our definition of confidentiality. For example, Barthe et al. [15] claim that it is possible to use CTL model-checking to check for noninterference in finite-state systems. However, upon examination, this holds only for a specific definition of noninterference, the one based on functional equivalence relation (as opposed to, e.g., strong equivalence relation). Barthe et al. reduce checking for noninterference to model-checking a CTL formula on self-composition. Self-composition can be viewed as a (sequential or parallel) composition of a program with itself (variables are renamed in the other copy of the program). It can be shown, by a proof similar to the one above, that there is no μ-calculus formula that characterizes the general definition of noninterference on self-composition.

2.3.1 Trees with path equivalences

We have shown that confidentiality is not specifiable in standard temporal logics interpreted over trees. We show that by enriching the tree model (and correspondingly enriching the logics), we obtain a specification framework that generalizes classical temporal logics, and is expressive enough to capture confidentiality.

We will use the notion of trees defined above. We start by generalizing the definition of observables. We replace the set O of observable propositions by a map $O_M : A \to 2^P$ defining the set of observables for an agent. The set A is a fixed set of agents. We use the map O_M to define equivalences among paths in a tree T as follows. Let the map $Tr_a : V \to 2^P$, defined as $Tr_a(v) = \lambda(v) \cap O_M(a)$ for all v, return the observables of a at a node v of T. We lift this map to paths in T by defining $Tr_a(v_0 v_1 \ldots) = Tr_a(v_0) Tr_a(v_1) \ldots$. Let u and v be two nodes of T and let π be a path leading from the root to u and π' a path leading from the root to v. Nodes u and v are *a-equivalent* (written as $u \approx_a v$) iff $Tr_a(\pi) = Tr_a(\pi')$.

We define the *equivalence graph* $IG(T)$ of a tree T as the node and edge-labeled graph where: (1) the set of nodes is the set V of nodes of T; (2) the *root node* of $IG(T)$ is the root r of T; (3) the node-labeling map λ is the same as in T; (4) there is an *unlabeled edge* from node u to node v (in this case, we write $u \to v$) iff (u, v) is an edge in T; (5) for each agent a, there is an edge labeled a from u to v (we write $u \xrightarrow{a} v$) iff $u \approx_a v$. Intuitively, the structure $IG(T)$ uses a-labeled edges to capture equivalence and defined by the relation $\approx_a$. We can now view $IG(T)$ as a Kripke structure rooted at r. It is on this structure that we interpret our logics. Fig. 2.4-(a) depicts a tree T with path equivalences. We have two agents a and a' satisfying $O_M(a) = \{p_1, p_2\}$ and $O_M(a') = \{p_2, p_3\}$, and the nodes $u_1, u_2, \ldots$ are labeled as in the figure. Now it is easy to check that, for instance, $u_2 \approx_{a'} u_3$. Consequently, the edges of the equivalence graph $IG(T)$, part of which is shown in Fig. 2.4-(b), include $u_2 \xrightarrow{a'} u_3$ (and $u_3 \xrightarrow{a'} u_2$.)

The above definition of a-equivalence can be considered time sensitive in the sense that it can model an observer who knows that a transition has occurred even if the observation has not changed. We consider also the following time insensitive equivalence. Let $\equiv_w$ be the smallest congruence on sequences of sets of propositions such that $U \equiv_w UU$, where U is a set of propositions. This relation is sometimes called stuttering congruence. Once more, let u and v be two nodes of T and let

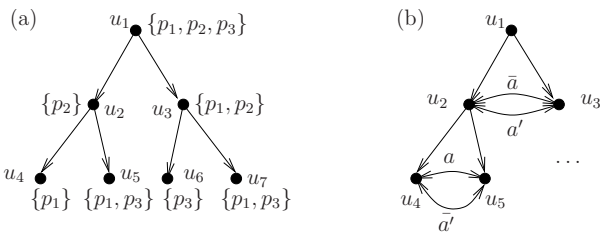

Figure 2.4: (a) A tree with equivalences (b) Part of its equivalence graph

π be a path leading from the root to u and π' a path leading from the root to v. Nodes u and v are *weakly a-equivalent* (written as $u \approx_a^w v$) iff $Tr_a(\pi) \equiv_w Tr_a(\pi')$. The *weak-equivalence graph* $IG^w(T)$ graph is defined similarly as $IG(T)$, with $\approx_a^w$ replacing $\approx_a$.

2.3.2 Branching-time logics on equivalence graphs

In this section, we interpret branching-time temporal logics on equivalence graphs and apply this interpretation to express some natural information-flow and partial-information requirements.

Note that we will restrict our attention to state-properties, that is, to properties of runs that only depend on the last state of the run.

$\mu_\approx$-calculus The $\mu_\approx$-calculus has modalities to reason about edges labeled a, for any agent a, as well as unlabeled edges. For example, we have formulas such as $\langle a \rangle \varphi$, which holds at a node u iff there is a node v satisfying φ such that $u \xrightarrow{a} v$. In order to increase the expressiveness of the logic (without increasing the complexity of model checking), we add an operator $\langle \bar{a} \rangle$ to the syntax. The formula $\langle \bar{a} \rangle \varphi$ holds at a node u if there is another node v satisfying φ on the same level of $IG(T)$ (i.e., with the same distance from the root) that is not a-equivalent to u. See Example 4 below for an example of a property specified using the $\langle \bar{a} \rangle$ operator. To define the semantics of this operator, we will need to refer to nodes that are on the same level. This can be done using an agent sl such that $O_M(sl) = \emptyset$. Intuitively, this agent

does not observe anything, and thus considers all the nodes at the same level to be equivalent. That is, there is an *sl*-labeled edge between every two nodes at the same level.

Formally, let P be the set of propositions labeling our trees, and *Var* be a set of *variables*. Formulas in the $\mu_{\approx}$-calculus are given by the grammar: $\varphi = p \mid \neg\varphi' \mid X \mid \varphi_1 \vee \varphi_2 \mid \langle\,\rangle\varphi' \mid \langle a\rangle\varphi' \mid \langle \bar{a}\rangle\varphi' \mid \mu X.\varphi'(X)$, if X occurs in φ' only under an even number of negations, where $p \in P, a \in A$ and $X \in Var$.

As for semantics, consider the equivalence graph $IG(T)$ of a tree T with path equivalences. A formula φ is interpreted in an *environment* $\mathcal{E}$ that interprets free variables of the formula as sets of nodes in $IG(T)$. The set $[\![\varphi]\!]_{\mathcal{E}}$ of nodes satisfying φ in environment $\mathcal{E}$ is defined inductively in a standard way. We state only a few cases:

- $[\![\langle\,\rangle\varphi]\!]_{\mathcal{E}} = \{u : \text{for some } v, u \to v \text{ and } v \in [\![\varphi]\!]_{\mathcal{E}}\}$;
- $[\![\langle a\rangle\varphi]\!]_{\mathcal{E}} = \{u : \text{for some } v,\ u \xrightarrow{a} v \text{ and } v \in [\![\varphi]\!]_{\mathcal{E}}\}$,
- $[\![\langle \bar{a}\rangle\varphi]\!]_{\mathcal{E}} = \{u : \text{for some } v,\ u \xrightarrow{sl} v \text{ and } \neg(u \xrightarrow{a} v) \text{ and } v \in [\![\varphi]\!]_{\mathcal{E}}\}$.

If φ is a closed formula, its satisfaction by u is independent of the environment. If u satisfies φ in this case, then we write $u \models \varphi$. If $IG(T)$ has root r, then T satisfies φ ($T \models \varphi$) iff $r \models \varphi$.

$\mu^w_{\approx}$-calculus For reasoning on the model $IG^w(T)$, we use a fragment of $\mu_{\approx}$-calculus called $\mu^w_{\approx}$-calculus that does not contain the operator $\langle \bar{a}\rangle$, since in this case the same level predicate is not meaningful. If the root r of $IG^w(T)$ satisfies a closed $\mu^w_{\approx}$-calculus formula φ, then T satisfies φ (written $T \models \varphi$).

CTL$\approx$ As we shall see in Chapter 3, the full $\mu_{\approx}$-calculus over equivalence trees turns out to have an undecidable model checking problem[1]. Consequently, we are

[1]One may wonder if monadic second order logic (MSO) is of any interest in this context. It turns out that a single path equivalence relation suffices to encode the "same-level" predicate on trees studied in the literature [57]. This implies that model checking MSO on trees with path equivalences is undecidable even for single-agent systems.

interested in a simple fragment called CTL$\approx$ that is very similar to CTL interpreted on equivalence trees. Not only is this logic decidable, but it is also expressive enough for most of our illustrative examples.

Formulas of CTL$\approx$ are given by: $\varphi = p \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi' \mid EX\,\varphi' \mid EI_a\,\varphi' \mid EI_{\bar{a}}\,\varphi' \mid \varphi_1\,EU\,\varphi_2 \mid EG\,\varphi'$, where $p \in P$ and $a \in A$ as before. Following CTL conventions, let us use the following abbreviations $EF\,\varphi$ and $AG\,\varphi$. We also write $AX\,\varphi$, $AI_a\,\varphi$ and $AI_{\bar{a}}\varphi$ as shorthand for $\neg EX\,\neg\varphi$, $\neg EI_a\,\neg\varphi$, and $\neg EI_{\bar{a}}\,\neg\varphi$. We define the semantics of CTL$\approx$ using a map $\Psi : \varphi \mapsto \psi$ that rewrites a CTL$\approx$ formula φ as a $\mu_{\approx}$-calculus formula ψ. The function Ψ is defined inductively in the standard way. We state the definition only for a few cases: $\Psi(EI_a\,\varphi') = \langle a\rangle\Psi(\varphi')$ and $\Psi(EI_{\bar{a}}\,\varphi') = \langle \bar{a}\rangle\Psi(\varphi')$. A tree T with path equivalences satisfies a CTL$\approx$ formula φ iff it satisfies $\Psi(\varphi)$.

CTL$\approx^w$ We also consider the logic CTL$\approx^w$ for reasoning on the model with weak path equivalences $IG^w(T)$. This logic does not contain the operator $EI_{\bar{a}}$, but otherwise is same as CTL$\approx$. Its semantics is defined on $IG^w(T)$.

Semantics on finite Kripke structures. We use finite Kripke structures to model finite-state systems. Formally, a Kripke structure K is a tuple $(Q, \rightarrow \subseteq Q \times Q, \lambda : Q \rightarrow 2^P, r)$, where Q is a finite set of states, $\rightarrow$ is a transition function, $\lambda : Q \rightarrow 2^P$ is a map labeling each state with the set of propositions, and $r \in Q$ is the initial state.

We want to define when a Kripke structure K satisfies a CTL$\approx$ (CTL$\approx^w$,$\mu_{\approx}$,$\mu^w_{\approx}$) formula φ. Note that it is not possible to define whether or not the formula holds in a particular state of K. The reason is that the equivalence relations are relations on paths in the structure, rather than on states of the structure. Thus, given a state s, it is not possible to determine which states are equivalent to s. This also implies that whether or not a given Kripke structure K satisfies φ can be defined inductively on the structure of φ on the tree unrolling of K. For a node in this tree, there is a unique path leading to it, so the set of equivalent nodes is well-defined. Given a Kripke structure K, let T_K be its tree unrolling. T_K can be seen as a tree with

(weak) path equivalences (which are determined by the set of agents A). Then we define $K \models \varphi$ iff $T_K \models \varphi$.

We are now ready to show how our definition of confidentiality can be specified in CTL$\approx$. Let α and β be state properties defined by propositions $a, b \in P$, that is we have that $r \in \alpha(\beta)$ iff the last state of r is labeled by $a(b)$. Let O be a set of observable propositions, and let c be an agent such that $O_M(c) = O$.

Lemma 4. *The property α is confidential in β w.r.t. O if and only if φ holds, where*

$$\varphi = AG(b \rightarrow (a \leftrightarrow EI_c \neg a)).$$

In the rest of this section, we will demonstrate how logics on trees with path equivalences aid specification.

Example 4. Consider the game of Battleship. In our formulation, each player owns a grid whose cells are filled with 0's and 1's, and at each round, a player asks another player about the contents of a cell. A central requirement is that player a does not reveal information about the contents of a cell (i, j) at any time unless the opponent asks specifically for them. To see how this property may be unintentionally violated in an automated Battleship game, consider an implementation where rows in a's grid are represented as linked lists that a iterates through to answer a query about a cell. Now, if a is asked about an element in an empty row, it gives an answer immediately (as it has nothing to iterate over). If the row is non-empty, it must iterate through a non-empty list and spend more time "thinking". Thus, a's opponent may glean information about whether a row in a's board is nonempty by tracking the time a takes to answer a query.

We can write a requirement forbidding the above scenario in CTL$\approx$. Let propositions c_{ij} and ask_{ij} be true at points in a play respectively iff cell c_{ij} contains 1 and a receives a request to reveal the contents of cell (i, j). We omit the full definition of a-equivalence in this version; roughly, observables of a include the requests a receives, the answers it gives, and a "silent proposition" τ that holds when a is "thinking".

Now consider the CTL$\approx$ property $\varphi = \neg(\neg ask_{ij}\, EU\, (AI_a\, c_{ij} \vee AI_a\, \neg c_{ij}))$, which asserts that there is no play with a node such that: (1) all behaviors a-equivalent to the play till this point lead to nodes where the content of (i, j) is the same, and (2) no explicit request for the contents of cell (i, j) is made by the opponent prior to this point. This ensures that the adversary cannot infer the contents of (i, j) by watching a's observables. On the contrary, in the case when $AI_a\varphi$ holds at any reachable node of the tree for some secret property φ, then an observer of a can infer the property φ by watching a's actions till that point. In other words, a *leaks* the secret φ.

Example 5. Logics on trees with path equivalences may be used to specify properties of systems where participants have partial information. Consider a Kripke structure representing a *blindfold reachability game* played by an agent a. At each round, an *active node* represents the current state of the game, and when a takes an action, a child of the current active node becomes the new active node. Because of partial information, however, a given action may cause different children of the current node to become active. We say that a has a winning strategy in this game if it can decide on a sequence of actions a priori, execute actions in it in succession, and no matter what actual path in the tree is taken, end in a node satisfying a target proposition p. Letting two paths be a-equivalent iff they agree on the sequence of actions of a, we find that a has a winning strategy in this game iff the tree satisfies the CTL$\approx$ requirement $EF\, (AI_a\, p)$.

Now consider an *adaptive reachability game*, where a can choose actions to guide the game while it is in progress. Let some of the tree nodes be now labeled with a *control proposition* b. At each round, a is now able to pick, along with an action, one of the *control formulas* b and $\neg b$. At any given point, partial information may cause different children of the current node to become active; however, the new active node is guaranteed to satisfy the control formula chosen at the current round. Let us define a-equivalence as before. It can be shown that a has a strategy to reach a node satisfying a target proposition p iff the game tree satisfies the $\mu_{\approx}$-calculus

formula $\varphi = \mu X.(p \vee [a][\,](b \wedge X) \vee [a][\,](\neg b \wedge X))$.

Example 6. In various protocols involving multiple agents, a need for properties involving secrecy and time arises often. For example in the case of auction protocols (studied in security literature, see e.g. [25]), the following property is important. Agent a's bid is not revealed before the auctioneer reveals all the bids. In order to illustrate how such requirements can be expressed in our logic, we present the following formula, which states that agent a does not reveal p (a secret) before agent b reveals q: $\varphi = \neg((EI_b\, q \wedge EI_b\, \neg q)\, EU\, (AI_a\, p \vee AI_a\, \neg p))$. The formula expresses that it is not the case that b does not reveal q $(EI_b\, q \wedge EI_b\, \neg q)$ until a reveals p $(AI_a\, p \vee AI_a\, \neg p)$. Now let us consider agents who make only time-insensitive observations, i.e. ones who cannot tell that an agent has performed an operation if the observables have not changed. This can be modeled using the weak-equivalence graph. The correctness of the protocol can thus be established by model checking the formula φ on $IG^w(T)$.

Example 7. Consider a system that is being observed by a low-security observer. We define low-security (low) and high-security (high) variables, where low variables are visible to the observer and high variables are not. We now show how to specify in CTL$\approx$ the following requirement R: " The sequence of valuations of the low variables is the same along all execution paths." Consider for example the case when there is a secret input, i.e. an input to a high variable. If the above requirement R is satisfied, the observer cannot infer any property of the secret input, since there cannot be any flow of information from the high input to low variables. (Note however that the requirement R is even stronger, it prevents e.g. inputs to low variables.)

The values of variables are encoded by propositions from a set P. We have one proposition for every bit of every variable. We will use only one agent a. The subset of propositions observable by the agent is the set of all those propositions that encode low variables. The requirement R is satisfied iff the following CTL$\approx$ formula

holds: $AG\, AI_{\bar{a}}\, false$. This property says that for each node, there does *not* exist an a-nonequivalent node at the same level of the execution tree. This implies that all nodes at the same level are a-equivalent, and therefore have the same valuations of low variables. Notice that this property cannot be captured without the $AI_{\bar{a}}$ ($EI_{\bar{a}}$) operators, since we need to refer to all nodes at the same level.

Chapter 3

Model Checking Confidentiality Properties for Finite State Systems

In this chapter, we present a model checking algorithm for CTL$\approx$ and the $\mu_{\approx}$-calculus.

As described in the previous chapter, we enriched the traditional tree model with "jump-edges" that capture observational indistinguishability. Consider a tree T whose nodes are labeled with truth values to atomic propositions P. For an agent a, if the assignment to the propositions $O(a) \subseteq P$ captures the observable behavior of a, then two tree nodes are considered a-equivalent if the paths from the root to these nodes agree on the values of propositions in $O(a)$ at every step. We convert the tree T into a graph $IG(T)$ by adding, for every agent a of interest, an a-labeled edge between every pair of a-equivalent nodes and $\bar{a}$-labeled edge between every pair of a-inequivalent nodes at the same level. One can view $IG(T)$ as a Kripke model, where both nodes and edges have labels, and interpret standard tree logics over it.

In our formulation, the model checking question is to decide whether $IG(T_K)$ satisfies a tree logic formula φ, where T_K is the tree unfolding of a finite-state model

K. Keeping track of paths equivalent with respect to one agent requires a subset construction leading to PSPACE complexity. We show that this construction can be generalized, and the key parameter is the *nesting depth* of the specification. Informally, when we need to evaluate a formula φ after jumping across an a-labeled edge, then an additional layer of subset construction is required to process b-equivalence, for agents $b \neq a$. We show that, if we restrict the nesting depth to 1, as is the case for all our example specifications, the model checking problem for a CTL-like logic is PSPACE-complete, and EXPTIME-complete for μ-calculus. When nesting depth is unbounded, model checking for CTL$\approx$ (the CTL-like logic) becomes nonelementary, and is undecidable for μ-calculus.

3.1 Finite model $FM^{\varphi}(K)$

Recall that $K \models \varphi$ is defined in terms of an infinite state structure. However, we can still apply model checking on a finite state system. This is because for a given CTL$\approx$ formula φ and a given Kripke structure K, we can find a finite model $FM^{\varphi}(K)$ such that $FM^{\varphi}(K) \models \varphi$ iff $T_K \models \varphi$. Let A_{φ} be the set of agents that appear in φ.

The *nesting depth* of a CTL$\approx$ formula φ is intuitively the maximum number of nestings between equivalence operators EI_a, $EI_{\bar{a}}$, for different agents a. The only exception is the nesting of EI_a operators for the same agent, which does not contribute to nesting depth. For example, the nesting depth of $EI_a\, p$ is 1, $(EI_a\, p)\, EU\, (EI_b\, r)$ is also 1, while for $EI_a\, EI_{\bar{a}}\, p$ it is 2. On the other hand, $EI_a\, EI_a\, p$ and $EI_a\, (\varphi_1\, EU\, EI_a\, \varphi_2)$ have a nesting depth of 1. The nesting depth of φ will be denoted by $nd(\varphi)$. Formally, the nesting depth is defined as follows. We will use an auxiliary function that takes two parameters: $nd(\varphi, a)$, where a is an agent. Let c be an agent that does not appear in φ. The function $nd(\varphi, a)$ is then defined as follows:

$$nd(\varphi, a) = 0 \text{ if } \varphi = p$$

$$nd(\varphi, a) = nd(\varphi_1) \text{ if } \varphi = \neg(\varphi_1), EX\,\varphi_1, EI_a\,\varphi_1, EG\,\varphi_1$$

$$nd(\varphi, a) = \max(nd(\varphi_1, a), nd(\varphi_2, a)) \text{ if } \varphi = \varphi_1 \vee \varphi_2, \varphi_1\,EU\,\varphi_2$$

$$nd(\varphi, a) = nd(\varphi_1, b) + 1 \text{ if } \varphi = EI_b\,\varphi_1 \text{ where } b \neq a$$

$$nd(\varphi, a) = nd(\varphi_1, c) + 1 \text{ if } \varphi = EI_{\bar{b}}\,\varphi_1 \text{ (for any } b)$$

The function $nd(\varphi)$ can then be defined as $nd(\varphi, a)$. The complexity of model checking of a CTL$\approx$ formula φ grows rapidly with the nesting depth of φ. However, as we show, the nesting of EI_a operators for the same agent does not contribute to the growth in complexity of the problem. This distinction is especially important in the case of the $\mu_{\approx}$-calculus, where the formulas with unbounded nesting depth are undecidable in general. However, formulas where only $\langle a \rangle$ operators for the same agent are nested unboundedly are in a decidable (EXPTIME-complete) fragment. This fragment allows for example specification of adaptive partial-information games (see Section 2.3.2).

We first give the intuition behind the construction of the finite state model $FM^{\varphi}(K)$. The states of this model carry enough information so that the semantics of CTL$\approx$ formulas can be defined on these states in such a way that $FM^{\varphi}(K) \models \varphi$ iff $T_K \models \varphi$. Consider the case when φ is a CTL formula. To determine whether φ holds at a node s of T_K, we only need to know to which state of K the node s corresponds, because if two nodes in T_K correspond to the same state of K, they satisfy the same CTL formulas. Now consider $\varphi \equiv EI_a\,\varphi_1$, where φ_1 is a CTL formula. Let S be the set of a-equivalent nodes of T_K. In order to determine whether $EI_a\varphi_1$ holds at s, one needs to know to which state of K the node s corresponds and to which states of K the nodes in S correspond. The amount of information needed is thus finite, and can be stored as a pair (s, U) such that $s \in Q, U \subseteq Q$, where Q is the set of states of K. We also need to know how to update this information across transitions. There

are two key ideas: First, the transition relation $(s, U) \rightarrow (t, V)$ on these pairs can be computed locally - the set of nodes V equivalent to t will be all those nodes v that have the same observation as t and that have predecessors equivalent to s, i.e. stored in U. Second, we can also define an a-transition ($\xrightarrow{a}$) on these tuples locally, since the tuple stores the set of nodes that are mutually a-equivalent. The transition is thus defined as follows: $(s, U) \xrightarrow{a} (t, U)$ for $t \in U$.

This construction lends itself to generalization in three ways: we can have multiple agents, we can store information needed for $\bar{a}$ transitions, and we can keep enough information to allow nesting of equivalence and nonequivalence operators. This leads to a definition of the finite-state model of $FM^{\varphi}(K)$. Note that in order to allow for nesting of equivalence operators, it is not enough to store only a set of a-equivalent nodes U for all agents a. In fact, for each node in U, we need to store the set of its b-equivalent nodes (where $b \neq a$), etc. We store this information as a tree whose nodes are labeled by states of K. Formally, we define $FM^{\varphi}(K)$ as follows:

States of $FM^{\varphi}(K)$: A state W of $FM^{\varphi}(K)$ is a tree of depth at most $nd(\varphi)$. The vertices of these trees are labeled by states of K and edges are labeled by a or $\bar{a}$, where a is in A_{φ}. We require that if a subtree is an a-child of its parent, then it itself does not have a-children. For all nodes in W, we require that no two of its a-children are isomorphic (similarly for $\bar{a}$-children). The state W is labeled by the same propositions as its root in the original Kripke structure K.

The intuition behind the definition is simple: a node s in T_K corresponds to a state W, if s is a root of W, the a-equivalent nodes of s correspond to a-children of W and this correspondence continues to depth $nd(\varphi)$. Such a state thus carries enough information to allow checking whether or not φ of nesting depth $nd(\varphi)$ holds. An example of a state W is in Figure 3.1. It stores the information about a node s, which has two a-equivalent nodes u and v, one a-nonequivalent node t and one b-equivalent node x.

If a subtree rooted at u is an a-child of its parent s, it does not need to have

a-children, since the nodes that are a-equivalent to u are a-equivalent to s. Therefore we do not need to replicate these nodes as children of u. The main reason is that for a subtree of depth d, the a-siblings store more information (they are trees of depths (at most) d) than would a-children - subtrees of depth (at most) $d-1$. This is what allows arbitrary nesting of EI_a (or $\langle a \rangle$) operators for the same agent a.

We can bound the number of states in $FM^{\varphi}(K)$. To state an upper bound, we will use the following function exp: $\exp(a, b, 0) = a$, $\exp(a, b, n+1) = a * b * 2^{\exp(a,b,n)}$. Considering how a state is constructed (it does not have isomorphic a-children), we can conclude that $FM^{\varphi}(K)$ has less than $\exp(|K|, 2 * |A_{\varphi}|, nd(\varphi))$ states.

Transition relation of $FM^{\varphi}(K)$: We explained above how a transition function is determined locally for tuples of the form (s, U) representing the node and a set of its a-equivalent nodes. The construction can be generalized to states of $FM^{\varphi}(K)$. We abuse the notation slightly and use the same notation for transition relations $\rightarrow, \xrightarrow{a}, \xrightarrow{\bar{a}}$ as is used in T_K. Given a state W, $root(W)$ refers to its root (a node in K). a-child of W refers to the tree rooted at a node that is an a-child of the root of W. For a state W of depth n, transition relation $\rightarrow$ is defined recursively on n.

- $n = 0$: Trees W and W' are of depth 0, i.e. they contain only a root without any children. $W \rightarrow W'$ if $root(W) \rightarrow root(W')$ in the Kripke structure K.
- $n = k + 1$: $W \rightarrow W'$ iff $root(W) \rightarrow root(W')$ in K and
  - V is an a-child of W' iff $Tr_a(root(W')) = Tr_a(root(V))$ and there exists an a-child U of W, such that $U \rightarrow V$
  - V is an $\bar{a}$-child of W' iff either there exists an $\bar{a}$-child U of W, such that $U \rightarrow V$ or there exists an a-child U', such that $U' \rightarrow V$ and $Tr_a(root(W')) \neq Tr_a(root(V))$.

An example of a transition $W \rightarrow W'$ transition in $FM^{\varphi}(K)$ is in Figure 3.1. The figure captures the following situation: There is a transition in K from s (the root

of W) to s', and from the a-equivalent node u to u' and the node u' is a-equivalent to s' (similarly for the b-equivalent node x). The node v is a-equivalent to s and it has a transition to v' in K. However, v' is not a-equivalent to s'. The node t is non-equivalent to s, thus its successor t' will be nonequivalent to s'. The subtrees T_u, T_v, T_t, T_x need to be transformed in a similar way.

We defined the structure $FM^{\varphi}(K)$ in order to keep information about a-equivalent nodes locally. Now we use this information to define a-transitions (transitions of the form $W \xrightarrow{a} W'$). The idea is that on an a-transition, we go from a state W to a state W' represented by an a-child of W. In general, this transition leads from a tree of depth n to a tree with depth $n-1$ (this is true for b-children where $b \neq a$ and all $(\bar{a})$-children). However, for a-children we leverage the fact that a-children of a parent are mutually equivalent, which enables us to construct a tree such that the depth of a-children does not decrease. Transition relations $\xrightarrow{a}, \xrightarrow{\bar{a}}$ are defined as follows:

- $W \xrightarrow{a} W'$ iff W' can be constructed as follows: Let V be an a-child of W. Let V' be V with other a-children of W as a-children (note that V did not have a-children). Let V'' be W without all the a-children, and we remove the leaves for all the other children (to ensure that the depth of W' is smaller or equal to $nd(\varphi)$). Finally, let W' be V' with V'' as an a-child.

- $W \xrightarrow{\bar{a}} W'$ if W' is a $\bar{a}$-child of W

An example of an $\xrightarrow{a}$ transition in $FM^{\varphi}(K)$ $W \xrightarrow{a} W''$ is in Figure 3.1. The idea is that W'' will be a subtree rooted at an a-child of s, which in this case is the subtree rooted at u. However, as explained above, we add as a-children subtrees rooted at a-siblings of u (in this case, the subtree rooted at v) and the subtree rooted at the parent s and its subtrees (except the a-children). We modify these subtrees (T_t^c and T_x^c in the figure) by removing the leaves.

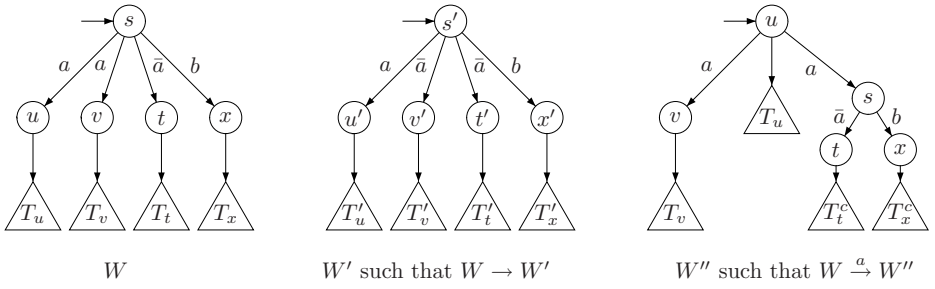

Figure 3.1: States and transitions of $FM^{\varphi}(K)$

3.2 CTL≈

We want to prove that the finite state model $FM^{\varphi}(K)$ is adequate for evaluating the formula φ, i.e. that for each node s of T_K there is a corresponding state W in $FM^{\varphi}(K)$, such that φ holds in s iff it holds in W. In order to state this claim, we need to define the correspondence between the states of T_K and $FM^{\varphi}(K)$. We will do so using a family of functions Ω^n_a, where a is an agent and n will correspond to $nd(\varphi)$. The definition is structured in a similar way as the definition of the function nd. Each Ω^n_a is a function that relates a node in T_K to a node of $FM^{\varphi}(K)$. It is defined recursively as follows:

- $n = 0$: $\Omega^0_a(u)$, for all a, is a tree of depth 0, whose root is u.
- $n = k + 1$: $\Omega^{k+1}_a(u) = W$ iff W can be constructed as follows: $root(W) = u$.
  - Consider the set S of all a-equivalent nodes of u. For every node v in this set, compute $V = \Omega^k_a(v)$. Let V' be V without a-children. Add V' as an a-child to W. For every node r at the same depth as u, that is not a-equivalent to u, add $R = \Omega^k_c(r)$ as an $\bar{a}$-child to W, where c is an agent that does not appear in φ.
  - For all agents b such that $b \neq a$, consider the set S of all b-equivalent nodes of u. For every node v in this set, compute $V = \Omega^{k-1}_b(v)$. Let V'

be V without b-children. Add V' as a b-child to W. For every node r at the same depth as u, that is not b-equivalent to u, add $R = \Omega_c^k(r)$ as an $\bar{a}$-child to W, where c is an agent that does not appear in φ.

We define the function Ω^n as Ω_c^n, where c is an agent that does not appear in φ.

The following lemma asserts that the construction of $FM^\varphi(K)$ is correct. It implies that model checking of CTL$\approx$ formula φ can be performed on $FM^\varphi(K)$. It is proven by induction on the nesting depth of the formula.

Lemma 5. *$T_K, s \models \varphi$ iff $FM^\varphi(K), \Omega^n(s) \models \varphi$, where $n = nd(\varphi)$*

Proof. The proof is by structural induction on φ. We show the proof for three operators – the proof is similar in the other cases. Recall that $\Omega^n(s)$ is $\Omega_c^n(s)$, where c is an agent that does not appear in φ.

- $\varphi \equiv EX\ \varphi_1$: Let us assume that $T_K, s \models \varphi$. As $T_K, s \models EX\ \varphi_1$, we have that there exists t such that $T_K, t \models \varphi_1$ and $s \to t$. By induction hypothesis, we obtain that $FM^\varphi(K), \Omega^n(t) \models \varphi_1$. From the definitions of Ω^n and $FM^\varphi(K)$, we have that $\Omega_c^n(s) \to \Omega_c^n(t)$. Combining the two facts, we can conclude that $FM^\varphi(K), \Omega_c^n(s) \models \varphi$. The reasoning can be reversed to prove the other implication.

- $\varphi \equiv EI_a\ \varphi_1$, where $a \neq c$: Let us assume that $T_K, s \models \varphi$. As $T_K, s \models EI_a\ \varphi_1$, we have that there exists t such that $T_K, t \models \varphi_1$ and $s \xrightarrow{a} t$. By induction hypothesis, we obtain that $FM^\varphi(K), \Omega_a^{n-1}(t) \models \varphi_1$. From the definitions of Ω^n and $FM^\varphi(K)$, we have that $\Omega_c^n(s) \xrightarrow{a} \Omega_a^{n-1}(t)$. Combining the two facts, we can conclude that $FM^\varphi(K), \Omega_c^n(s) \models \varphi$. The reasoning can be reversed to prove the other implication.

- $\varphi \equiv EI_c\ \varphi_1$: Let us assume that $T_K, s \models \varphi$. As $T_K, s \models EI_c\ \varphi_1$, we have that there exists t such that $T_K, t \models \varphi_1$ and $s \xrightarrow{c} t$. By induction hypothesis, we obtain that $FM^\varphi(K), \Omega_c^n(t) \models \varphi_1$. From the definitions of Ω^n and $FM^\varphi(K)$,

we have that $\Omega_c^n(s) \xrightarrow{a} \Omega_c^n(t)$. Combining the two facts, we can conclude that $FM^{\varphi}(K), \Omega_c^n(s) \models \varphi$. The reasoning can be reversed to prove the other implication.

□

A nesting-free formula is a formula with nesting depth at most 1. Thus it is a formula that can refer to operators EI_a, $EI_{\bar{a}}$ for different agents, but it can nest only the EI_a operators for the same agent. All of the example properties mentioned in Section 2.3.2 are expressed in this fragment.

Theorem 6. *The model checking problem for nesting-free formulas of CTL≈ is PSPACE-complete.*

Proof. We show that the problem is in PSPACE using Lemma 5. The lemma shows that it is possible to reduce CTL≈ model checking to CTL model checking on an exponentially larger structure $FM^{\varphi}(K)$, whose number of states is less than $|K| * 2 * |A| * 2^{|K|}$. Note however, that it is not necessary to construct the structure ahead of time, since the transition function can be computed locally. Instead, the non-deterministic model checking algorithm for CTL [55] that uses only logarithmic space in terms of the size of the structure can be used. Therefore the model checking problem for nesting-free formulas is in PSPACE. The lower bound is obtained by reduction from equivalence checking of nondeterministic finite automata. Note that the reduction in fact encodes the equivalence check as a conditional confidentiality formula. Therefore we also have that checking conditional confidentiality is PSPACE-complete. □

For general CTL≈ formulas we obtain the following result.

Theorem 7. *For a fixed CTL≈ formula φ such that $nd(\varphi) \geq 2$, the model checking problem is decidable in space polynomial in* $\exp(|K|, 2 * |A|, nd(\varphi) - 1)$.

Proof. Similarly as in the proof of Theorem 6, we consider the structure $FM^{\varphi}(K)$. As noted above, the number of states of this structure is bounded from above by the expression $\exp(|K|, 2*|A_{\varphi}|, nd(\varphi))$. However, again as in the proof of Theorem 6, the non-deterministic model checking algorithm for CTL is used. This algorithm uses only logarithmic space in terms of the size of the structure, thus we can conclude that the model checking problem is decidable in space polynomial in $\exp(|K|, 2 * |A|, nd(\varphi) - 1)$. □

In order to state the lower bound, we define the function $Tower(n, k)$ by $Tower(n, 1) = n$ and $Tower(n, k + 1) = 2^{Tower(n,k)}$.

Theorem 8. *For every algorithm $\mathcal{A}$ for the model checking problem of CTL$\approx$, and each $i \geq 1$, there is a Kripke structure K with n states and a CTL$\approx$ formula φ such that $nd(\varphi) = i$ such that $\mathcal{A}$ runs on K and φ in time $\Omega(Tower(n, i))$.*

Proof. In order to obtain a lower bound for the model checking problem for CTL$\approx$ formulas, we can encode Shilov and Garanina's Act-CTL-K_n [76] in CTL$\approx$ and use the fact that model checking for Act-CTL-K_n has a nonelementary lower bound.

Act-CTL-K_n is a logic similar to CTL with actions augmented with knowledge operators. In order to encode it in CTL$\approx$, we need to show how to encode the knowledge operators and the actions. We consider the version of Act-CTL-K_n where the knowledge operators K_a in are given perfect-recall semantics. Each agent a can observe only a part of the global state (as specified by the agent's observation function). The perfect-recall semantics means that each agent remembers the whole history of his observations. This semantics can be easily captured in CTL$\approx$ by defining the equivalence relation $\approx_a$ to be such that two paths (sequences of global states) are equivalent iff the corresponding sequences of states of agent a are the same, and by encoding the knowledge operator K_a corresponding to agent a can as follows: $K_a\varphi = \neg\, EI_a \neg\, \varphi$. The actions are encoded of Act-CTL-K_n are encoded in a standard way: edge labels are transferred from edges to nodes. Given an a-labeled

edge between nodes s and r, we create a label (a, s) and label the node r with this label. Formulas such as $EX_a\varphi$ are then encoded as $EX((a, s) \wedge \varphi)$.

As the model checking problem for Act-CTL-K_n has a nonelementary lower bound, we can infer that so does the model checking problem for CTL$\approx$. □

3.3 $\mu_\approx$-calculus

We now consider the model checking problem for $\mu_\approx$-calculus formulas on trees with path equivalences.

Theorem 9. *The model checking problem for the $\mu_\approx$-calculus is undecidable.*

Proof. We can prove that the problem is undecidable by encoding by encoding Shilov and Garanina's μ-calculus of knowledge with perfect recall semantics - (μPLK_n) - [76].

In general, this problem is undecidable. We can prove it by encoding Shilov and Garanina's μ-calculus of knowledge - (μPLK_n)[76]. This logic can be encoded in $\mu_\approx$-calculus over trees with path equivalences in a similar way as Act-CTL-K_n was encoded to CTL$\approx$. The encoding is very similar to the one used in the proof of Theorem 8. The knowledge operator corresponding to an agent a can be encoded using the EI_a operator. Similarly to Act-CTL-K_n, μPLK_n is a temporal logic with actions. The operators that are parameterized by actions are encoded as indicated in the the proof of Theorem 8.

As the model checking problem for μPLK_n is undecidable, we can conclude that so is the model checking problem for $\mu_\approx$-calculus. □

We have shown that the model checking problem is undecidable in general for the $\mu_\approx$-calculus. However, we identify a decidable fragment of the $\mu_\approx$-calculus as follows. Define the set $Subf(\varphi)$ of subformulas of a formula φ inductively as: (1) for $\varphi = p$ or $\neg p$, $Subf(\varphi) = \{\varphi\}$; (2) if φ equals $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$, then $Subf(\varphi) =$

$\{\varphi\} \cup \mathit{Subf}(\varphi_1) \cup \mathit{Subf}(\varphi_2)$; (3) if φ equals $\langle\rangle\varphi'$, $[\,]\varphi'$, $\langle a\rangle\varphi'$ or $[a]\varphi'$, for arbitrary a, we have $\mathit{Subf}(\varphi) = \mathit{Subf}(\varphi')$; and (4) for $\varphi = \mu X.\varphi'$ or $\nu X.\varphi'$, we have $\mathit{Subf}(\varphi) = \{X\} \cup \mathit{Subf}(\varphi')$. Now, let us only consider "well-named" formulas, i.e., closed formulas φ where for each variable X appearing in φ, there is a unique binding formula $\mu X.\varphi'$ or $\nu X.\varphi'$ in $\mathit{Subf}(\varphi)$ such that $X \in \mathit{Subf}(\varphi')$. As for the μ-calculus, every closed $\mu_\approx$-calculus formula can be rewritten in a well-named form. Now construct the graph G_φ with node set $\mathit{Subf}(\varphi)$ and edges as below:

1. for each node φ of the form $\langle\rangle\varphi'$, $[\,]\varphi'$, $\langle a\rangle\varphi'$, $[a]\varphi'$, $\mu X.\varphi'$, or $\nu X.\varphi'$, add an edge from φ to φ'.

2. for each node X, where $X \in \mathit{Var}$, add an edge from X to the unique subformula of the form $\mu X.\varphi'$ or $\nu X.\varphi'$ that binds it.

Intuitively, G_φ captures the operational semantics of φ. If there is a path from φ' to φ'' in $G_{\varphi'}$, then evaluation of φ' requires the evaluation of φ'' (note that to evaluate $\varphi' = X$, we must recursively evaluate the formula φ'' binding X).

A formula is said to be a-modal (resp. $\bar{a}$-modal) if it is of the form $\langle a\rangle\varphi$ or $[a]\varphi$ (resp. $\langle\bar{a}\rangle\varphi$ or $[\bar{a}]\varphi$). Let $\pi = \psi_1\psi_2\ldots\psi_m$ be a path in G_φ. The *nesting distance* of π is k, where k is the length of the maximum subsequence $\pi' = \psi'_1\psi'_2\ldots\psi'_k$ in π such that: (1) each ψ'_i is an a-modal formula for some agent a; and (2) for each i, if ψ'_i is a-modal and ψ'_{i+1} is a'-modal, then $a \neq a'$. A formula φ has *nesting depth* k if k is the least upper bound on the nesting distance of any path in G_φ. Note that such a k may not exist—if it does, then φ is said to have a *bounded* nesting depth. For instance, the formula $\varphi_1 = \nu X.([a_1]\langle a_2\rangle p \wedge \langle a_1\rangle[\,][a_1]X)$ is bounded, while $\varphi_2 = \mu X.(p \vee \langle\rangle[a_1]\langle a_2\rangle X)$ is not.

The proof of the following theorem gives rise to a a model checking algorithm for the fragment of the $\mu_\approx$-calculus with bounded nesting depth. We show also a lower bound.

Theorem 10. *The model checking problem for a Kripke structure K and a $\mu_{\approx}$-calculus formula φ with nesting depth k is solvable in time* $\exp(|K|, 2 * |A|, k)$. *Also, for every algorithm $\mathcal{A}$ for this problem and every $i \geq 1$, there is a Kripke structure K with n states and a formula φ such that $\mathcal{A}$ runs on K and φ in time* $\Omega(Tower(n, i))$.

Proof. Recall that the relation $K \models \varphi$ is defined in terms of an infinite-state structure T_K. The model checking algorithm that we use is based on a finite-state model checking algorithm. The reason is that we use a finite-state structure $FM^{\varphi}(K)$. We can prove that the finite state model $FM^{\varphi}(K)$ is adequate for evaluating $\mu_{\approx}$formulas. That is, given a formula φ, for each node s of T_K there is a corresponding state W in $FM^{\varphi}(K)$, such that φ holds at s iff it holds at W. The proof is analogous to the proof of Lemma 5.

The size of $FM^{\varphi}(K)$ is less than $\exp(|K|, 2 * |A_{\varphi}|, nd(\varphi))$. The complexity of the standard CTL model checking algorithm is linear in the number of states of the structure. Therefore we conclude that the model checking problem for a $\mu_{\approx}$-calculus formula φ with nesting depth k is solvable in time $\exp(|K|, 2 * |A|, k)$.

We obtain the lower bound by encoding CTL$\approx$ into the fragment of the $\mu_{\approx}$-calculus with bounded nesting depth. This is possible, as all CTL$\approx$ formulas have bounded nested depth. □

Now consider the model checking problem for *nesting-free formulas*, i.e., formulas with nesting depth 1. Recall that such formulas can express all properties involving a single agent.

Theorem 11. *Model checking nesting-free $\mu_{\approx}$-calculus formulas is EXPTIME-complete. Model checking single-agent, alternation-free $\mu_{\approx}$-calculus formulas is EXPTIME-hard.*

Proof. We first show that the model checking nesting-free $\mu_{\approx}$-calculus formulas is in EXPTIME. Given a Kripke structure K and a set of agents A, we construct the structure $FM^{\varphi}(K)$ in exponential time. We can interpret nesting-free $\mu_{\approx}$-calculus

formulas on $FM^{\varphi}(K)$. As in Lemma 5, we can show that K satisfies a nesting-free formula φ iff $FM^{\varphi}(K)$ satisfies φ.

For a lower bound, we turn to the model of *space-bounded private alternating Turing machines* (PA-TM) introduced by Reif [69]. Let PALOGSPACE be the class of languages recognized by such machines using logarithmic space — Reif shows that PALOGSPACE = EXPTIME.

Given a PA-TM M and its input w, we construct a Kripke structure K, with initial state s_0, such that $K, s_0 \models \varphi$ iff M accepts w, where φ is a $\mu_{\approx}$formula defined below. The structure of the proof is similar to the proof that the alternation-free modal μ-calculus is complete for PTIME and consequently alternating LOGSPACE [55].

The states of K will be the states of M augmented with contents of the tape, and marked by e (the existential nodes) and u (the universal nodes). The existential player does not have full information (he or she does not see the contents of the private tapes). This will be captured using the path equivalence relation of an observer a. The PA-TM accepts iff there is a winning strategy for the existential player. This can be captured by the $\mu_{\approx}$-formula $\varphi = \mu X.(p \vee (e \wedge \langle\,\rangle\langle a\rangle X) \vee (u \wedge [\,a]X))$.

We have thus reduced recognition by a PALOGSPACE-machine to the model checking problem for an alternation-free, single-agent $\mu_{\approx}$-calculus formula. The latter problem is thus EXPTIME-hard. As the nesting-free fragment of $\mu_{\approx}$captures all single agent formulas, we conclude, using Theorem 10, that model checking nesting-free $\mu_{\approx}$-calculus formulas is EXPTIME-complete. □

Chapter 4

Decision Procedures for Confidentiality for Array-Accessing Programs

We develop program analysis methods for confidentiality for programs. We have introduced the definition of conditional confidentiality and we have shown how it can be checked on finite-state systems. In this chapter, we develop decision procedures for classes of programs. Verification questions concerning programs are undecidable in general. However, we identify a class of programs for which confidentiality is decidable. The class contains programs that access a data structure. The confidentiality requirement can for example specify what information about the data stored in the data structure can be revealed. The decision procedures for confidentiality is based on algorithms for deciding reachability. The latter are of independent interest, as detailed below, both from a theoretical point of view (due to connections to logics on data words) as well as from practical point of view, due to potential use as back-end decision procedures for software model checkers.

For finite-state programs — programs whose data variables range over finite types such as Boolean, the number of bits needed to encode a program state is a priori

bounded, and verification questions such as reachability are decidable. This result forms the basis of recent tools for software model checking [13, 52, 47].

A natural question is then whether it is possible to extend the Boolean program model without losing decidability or worsening computational complexity of the reachability problem. The first idea might be to add integer variables. However, adding expressions permitting (Presburger) integer arithmetic would cause undecidability. Therefore, one can investigate the possibility of adding only equality and order tests on integers to the language. Reachability in such programs is decidable, but perhaps the programs themselves are not too interesting. We show that it is possible to extend the model further.

We focus on algorithmic verification of programs that have a read-only access to a single array. The length of the input array is potentially unbounded. The elements of the array range over $\Sigma \times D$, where Σ is a finite set, and D is a data domain that is potentially unbounded and totally ordered. The array is thus modeled as a *data word*, that is, a sequence of pairs in $\Sigma \times D$. For example, integer arrays are easily captured by setting D to be $\mathbb{N}$ and Σ to be a singleton set. The program can have Boolean variables, index variables ranging over array positions, and data variables ranging over D. Programs can access Σ directly, but can only perform equality and order tests on elements of D. The expressions in the program can use constants in D, and equality tests and ordering over index and data variables. The programs are built using assignments, conditionals, and `for`-loops over the array. Even with these restrictions, one can perform interesting computational tasks including searching for a specific value, finding the minimum data value, checking that all values in the array are within specific bounds, or checking for duplicate data values. Array is a heavily used data structure. For example, Java midlets designed to enhance features of mobile devices include simple programs accessing the address books, and our methods can lead to an automatic verification tool that certifies their correctness before being downloaded. In order to analyze programs statically, it is often necessary to check

relationships among values in the array, as well as their relationships to values of other variables and constants. For example, in the case of indirect addressing, it is needed to check that all the values in the array fall within certain bounds. For programs that fall outside the restrictions mentioned above, it is possible to use abstract interpretation [30] techniques such as predicate abstraction [45] to abstract some of the features of the program, and analyze the property of interest on the abstract program. As the abstract programs are nondeterministic, we consider nondeterministic programs.

Our first result is that the reachability problem for programs in which there are no *nested* loops is decidable. The construction is by mapping such a program to a finite-state abstract transition system such that every finite path in the abstract system is feasible in the original program for an appropriately chosen array. We show that the reachability problem for programs with non-nested loops is PSPACE-complete, which is the same complexity as that for finite-state programs with only Boolean variables. The latter is the basis of successful software verification tools, and therefore we believe that, coupled with abstraction techniques, our decision procedure can potentially be the basis of a software model checking tool that better handles data structures with unbounded size.

Our second result establishes decidability of the reachability problem for programs with arbitrary nesting of loops that do not use index variables, under the assumption that the data domain is finite. The algorithm can be used for bounded model checking of such programs. In this case, the array can be viewed as a finite word over the finite alphabet of data values. The traversal order of a program with nested loops and index/data variables does not directly correspond to classical extensions of automata with multiple passes and/or pebbles (see for example [41]). We show that the set of arrays for which a particular Boolean state is reachable is regular, and reachability is solvable in space polynomial in the number of states of the program, which itself is exponential in the number of variables.

Our third result shows decidability of reachability for programs with doubly-nested loops with some restrictions on the allowed expressions. The resulting complexity is non-elementary, and the interest is mainly due to the theoretical connections with the recently well-studied notions of automata and logics over *data words* [19, 17, 53]. Among different kinds of automata over data words that have been studied, *data automata* [19] emerged as a good candidate definition for the notion of regularity for languages on data words. A data automaton first rewrites the Σ-component to another finite alphabet Γ using a nondeterministic finite-state transducer, and then checks, for every data value d, whether the word over Γ obtained by deleting all the positions in which the data value is not equal to d, belongs to a regular language over Γ. In order to show decidability of the reachability problem for programs with doubly nested loops, we extend this definition as follows: An *extended data automaton* first rewrites the data word as in case of data automata. For every data value d, the corresponding projection, obtained by replacing each position with data value different from d by the special symbol 0, is required to be in a regular language over $\Gamma \cup \{0\}$. We prove that the reachability problem for extended data automata can be reduced to emptiness of multi-counter automata (or equivalently, to Petri nets reachability), and is thus decidable. We then show that a program containing doubly-nested loops can be simulated, under some restrictions, by an extended data automaton. Relaxing these restrictions leads to undecidability of the reachability problem for programs with doubly-nested loops.

Analyzing the reachability problem for programs brings a new dimension to investigations on logics and automata on data words. We establish some new connections, in terms of expressiveness and decidability boundaries, between programs, logics, and automata over data words. Bojanczyk et al. [19] consider logics on data words that use two binary predicates on positions of the word: (1) an equivalence relation $\approx$, such that $i \approx j$ if the data values at positions i and j are equal, and (2) an order $\prec$ which gives access to order on data values, in addition to standard successor (+1)

and order $<$ predicates. They show that while the first order logic with two variables, $\mathrm{FO}^2(\approx, <, +1)$, is decidable, introducing order on data values causes undecidability, that is, $\mathrm{FO}^2(\approx, \prec, <, +1)$ is undecidable. In this context, our result on programs with non-nested loops is perhaps surprising, as we show that the undecidability does not carry over to these programs, even though they access order on the data domain and have an arbitrary number of index and data variables.

We show that our decidability results concerning reachability problem for various classes of array-accessing programs lead to decidability of confidentiality properties. Note that this is not immediate, as the definition of confidentiality involves quantifier alternation on runs. For confidentiality, we require that for all runs, there exists an equivalent run with certain properties. Nevertheless, we will show how the confidentiality question reduces to two reachability queries under certain restrictions.

4.1 Programs

In this section, we define the syntax and semantics of programs that we will consider. We start by defining arrays. Let D be an infinite set of data values. We will consider domains D equipped with equality $(D, =)$, or with both equality and linear order $(D, =, <)$. Let Σ be a finite set of symbols. An array is a data word $w \in (\Sigma \times D)^*$. The program can access the elements of the array via indices into the array.

Syntax. The programs have one array variable `A`. Variables `b`, `b1`, `b2`, ... are boolean. Variables `p`, `p1`, `p2`, ... range over $\mathbb{N}$, and are called index variables. Variables `i`, `j`, `i1`, `i2`, ... range over $\mathbb{N}$ and are called loop variables. Variables `v`, `v1`, `v2`, ... range over D and are called data variables. Constants `c`, `c1`, `c2`, ... are in D, and constants `s`, `s1`, `s2`, ... are in Σ. We make a distinction between loop and index variables because loop variables cannot be modified outside of the loop header.

Index expressions `IE` are defined by the following grammar `IE ::= p | i`. Data expressions `DE` are of the form `DE ::  = v | c | A[IE].d`, where `A[IE].d` accesses

the data part of the array. Σ-expressions `SE` are of the form `SE :: = s | A[IE].s`, where `A[IE].s` accesses the Σ part of the array. Boolean expressions are defined by the following grammar:

```
B :: =    true | b | B and B | not B
        | IE = IE | IE < IE
        | DE = DE | DE < DE
        | SE = SE
```

The programs are defined by the grammar:

```
P :: =  skip | { P }
      | b:=B | p:=IE | v:=DE
      | if B then P else P
      | if * then P else P
      | for i:=1 to length(A) do P
      | P;P
```

The commands include a nondeterministic conditional. We consider nondeterministic programs, in order to enable modeling of abstracted programs. Software model checking approaches [45, 13, 52] often rely on predicate abstraction. For example, if the original program contains an assignment of the form `b := E`, where `E` is a complicated expression that falls out of scope of the intended analysis, the assignment is abstracted into a nondeterministic assignment to `b`. This is modeled as `if * then b:=true else b:=false` in the language presented here.

Semantics. A *global state* of the program is a valuation of its boolean, loop, index and data variables, as well as of the array variable. We denote global states by g, g_1, and the set of global states by G. For a boolean, index, loop or data variable `v`, we denote the value of `v` by $g[\mathtt{v}]$. The value of the array variable `A` is a word $w \in (\Sigma \times D)^*$. It is denoted by $g[\mathtt{A}]$. The length of the array at global state g is

denoted by $l(g[\texttt{A}])$ and evaluates to the length of w. Note that the length and the contents of the array do not change over the course of the computation.

Semantics of boolean, index, data and Σ expressions is a partial function: $[\![\texttt{B}]\!] : G \times \mathbf{B}$, $[\![\texttt{IE}]\!] : G \times \mathbb{N}$, $[\![\texttt{DE}]\!] : G \times D$ and $[\![\texttt{SE}]\!] : G \times \Sigma$. It is not defined only in cases when there is an array access out of bounds. For example, in a state g where $g[\texttt{A}]$ is a word of length 10 and $g[\texttt{p}]$ is 20, the semantics of the expression `A[p].d` is undefined. The semantics of commands is defined as a relation on G, $[\![\texttt{P}]\!] \subseteq G \times G$.

- $(g, g) \in [\![\texttt{skip}]\!]$, for all g in G
- $(g, g') \in [\![\texttt{v:=E}]\!]$, iff $g' = g[\mathbf{v} \leftarrow [\![\texttt{E}]\!](g)]$, for any assignment.
- $(g, g') \in [\![\texttt{if B then P1 else P2}]\!]$ iff $[\![\texttt{B}]\!](g) = \mathit{true}$ and $(g, g') \in [\![\texttt{P1}]\!]$ or $[\![\texttt{B}]\!](g) = \mathit{false}$ and $(g, g') \in [\![\texttt{P2}]\!]$.
- $(g, g') \in [\![\texttt{if * then P1 else P2}]\!]$ iff $(g, g') \in [\![\texttt{P1}]\!]$ or $(g, g') \in [\![\texttt{P2}]\!]$.
- $(g, g) \in [\![\texttt{for i1:=1 to length(A) do P}]\!]$ iff $l(g[\texttt{A}]) = 0$.
- $(g, g') \in [\![\texttt{for i1:=1 to length(A) do P}]\!]$ iff $l(g[\texttt{A}]) > 0$ and there exist $g_1, g_2, \ldots, g_{l+1}$, where $l = l(g[\texttt{A}])$, such that $g_1 = g[\texttt{i1} \leftarrow 1]$, $g_{l+1} = g'$, and for all i such that $1 \leq i \leq l$, we have that there exists a g'_{i+1}, such that $(g_i, g'_{i+1}) \in [\![\texttt{P}]\!]$ and $g_{i+1} = g'_{i+1}[\texttt{i1} \leftarrow i + 1]$.
- $(g, g') \in [\![\texttt{P1;P2}]\!]$ iff there exists g'' such that $(g, g'') \in [\![\texttt{P1}]\!]$ and $(g'', g') \in [\![\texttt{P2}]\!]$.

Given a program, a global state is *initial* if either i) the array variable contains a nonempty word, all boolean variables are set to *false*, all index and loop variables are set to 1, and all data variables are set to the same value as the first element of the array; or ii) the array variable contains an empty word, all boolean variables are set to *false*, all index and loop variables are set to 1, and all data variables are set to constant $c_D \in D$. The intention is that the only unspecified part of the initial state, the part that models input of the program, is the array.

Note that for the programs we have defined, where the only iteration allowed is over the array, the termination is guaranteed. Therefore for all initial global states g_I there exists a global state g such that $(g_I, g) \in [\![\texttt{P}]\!]$.

A boolean state is a valuation of all the boolean variables of a program. For a given global state g, we denote the corresponding boolean state by $bool(g)$. For any boolean variable b of the program, we have that $bool(g)[\texttt{b}] = g[\texttt{b}]$. We denote boolean states by m, m_1 and the set of boolean states by M.

Restricted fragments. We classify programs using the nesting depth of loops. We denote programs with only non-nested loops by `ND1`, programs with nesting depth at most 2 by `ND2`, etc. Restricted-`ND2` programs are programs with nesting depth at most 2, that do not use index or data variables, and do not refer to order on data or indices. Furthermore, a key restriction, such that if it is lifted, the reachability problem becomes undecidable, is a restriction on the syntax of the code inside the inner loop. Let `P1` be the code inside an inner loop, and let `i` be the loop variable of the outer loop and let `j` be the loop variable for the inner loop. `P1` must be of the following form: `if A[i].d=A[j].d then P2 else P3`. Furthermore, `P3` cannot refer to `A[j]`, i.e. it does not contain occurrences of `A[j].d` or `A[j].s`.

Examples. We present three illustrative examples for the classes of programs we defined.

Example 8. We consider a simple array accessing program that scans through an array to find a minimal data value. It has one index variable, `min`, and it is an `ND1` program, as it does not contain nested loops. Note that by definition of program semantics, `min` is initialized to 1.

```
for i:= 1 to length(A) do {
  if A[i].d < A[min].d then {min := i}
}
```

The correctness requirement for this program is that the index `min` points to a

```
b:=true;
for i:= 1 to length(A) do {
  if A[i].d < v then b:=false
                else skip;
  v := A[i].d
}
```

Figure 4.1: Example 2

```
b:=false;
for i:= 1 to length(A) do {
  for j:= 1 to length(A) do
    if (A[i].d = A[j].d) then {
    if (not (i = j)) then b:=true
                     else skip
    } else skip
}
```

Figure 4.2: Example 3

minimal element, that is $\forall$ `i: A[i]` $\geq$ `A[min]`. Verifying the correctness of the program can be reduced to checking reachability, as the requirement itself can be expressed as a program.

```
b:= true;
for i:= 1 to length(A) do {
  if A[i].d < A[min].d then {min := i}
}
for i:= 1 to length(A) do {
  if A[i].d < A[min].d then {b:=false}
}
```

We can now ask a reachability question: Does the control reach the end of the program in a state where `b == false` holds?

Example 9. Figure 4.1 shows an `ND1` program that tests whether the array is sorted. It uses one data variable called `v` (note that by definition of the semantics, `v` is initialized to the same value as the first element of the array).

Example 10. The Restricted-`ND2` program in Figure 4.2 tests whether there is a data value that appears twice in the array.

4.2 Reachability

Given a program P, a boolean state m is *reachable* if and only if there exists an initial global state g_I and a global state g such that $(g_I, g) \in [\![\texttt{P}]\!]$ and $bool(g) = m$. The reachability problem is to determine, for a given program P and a given boolean state m, whether m is reachable.

Local states. We will use a notion of a *local state*. Given a program, a local state is a valuation of all its boolean, index, loop, and data variables, as well as the values of array elements corresponding to index and loop variables. For each index and loop variable v, local states have an additional variable A_v that stores the value of the array element at position given by v. The main difference between local and global states is that local states do not contain valuation of the array, they only store a finite number of values from the unbounded domain D.

Notation for local states For a given global state g, we denote the corresponding local state by $loc(g)$. For any variable v of the program, we have that $loc(g)[\texttt{v}] = g[\texttt{v}]$. If v is an index or a loop variable, we also have that $loc(g)[\texttt{A\_v}] = [\![\texttt{A[v]}]\!](g)$. We denote local states by q, q_1, and the set of local states by Q. A local state q is *initial* if there exists an initial global state g_I such that $loc(g_I) = q$.

Normal form In order to simplify the presentation of proofs of the decidability results, we will first translate the programs into a normal form. A program is in normal form if the branches of `if` statements do not contain loops.

We define a translation function $norm(\texttt{P})$, that given a program P returns an equivalent program in normal form. We use an auxiliary function $assume(\texttt{B, P})$, and we set $norm(\texttt{P}) = assume(\texttt{true, P})$. The function $assume(\texttt{B, P})$ is defined inductively as follows:

- $assume(\texttt{B, skip}) = \texttt{skip}$.

- *assume*(B, v:=E) = `if B then v:=E else skip`,
  if B is not `true`, and `v:=E` otherwise.

- *assume*(B, if B1 then P1 else P2) =
  `b := B1;`
  assume(B and b, P1);
  assume(B and (not b), P2),
  where `b` is a new boolean variable

- *assume*(B, if * then P1 else P2) =
  `if * then b:=true else b:=false;`
  assume(B and b, P1);
  assume(B and (not b), P2),
  where `b` is a new boolean variable

- *assume*(B, for i1:=1 to length(A) do P) =
  `for i1:=1 to length(A) do` *assume*(B, P).

- *assume*(B, P1;P2) =
  assume(B, P1); *assume*(B, P2).

The program *norm*(P) has more variables than the program P. However, intuitively the programs *norm*(P) and P compute the same function on the common variables. We now formalize this notion.

Let P be a program, let G be its set of global states and let V be its set of variables. Similarly, let G' and V' be the sets of states and variables of P'. Furthermore, we will assume that $V \subseteq V'$. We define an equivalence relation $\sim_{V,V'}$ as follows. We have that $[\![\texttt{P}]\!] \sim_{V,V'} [\![\texttt{P'}]\!]$ if and only if for all $g'_1, g'_2 \in G'$ it holds that $(g'_1, g'_2) \in [\![\texttt{P'}]\!]$ iff $(\pi_{V,V'}(g'_1), \pi_{V,V'}(g'_2)) \in [\![\texttt{P}]\!]$, where $\pi_{V,V'} : G' \to G$ as follows: $\pi_{V,V'}(g') = g$ iff g and g' agree on variables from V. The following lemma is proven by induction on the structure of the program P.

Lemma 12. *Let* P *be a program, let* V *be its set of variables, and let* V' *be the set of variables of* $norm(P)$*. We have that* $[\![norm(P)]\!] \sim_{V,V'} [\![P]\!]$*. Furthermore, the nesting depth of loops is the same in* $norm(P)$ *as it is in* P*. The number of boolean variables in* $norm(P)$ *increased by at most the number of* `if` *statements in* P*.*

Theorem 13. *Reachability for* ND1 *programs is decidable. The problem is* PSPACE*-complete.*

The structure of the proof is as follows. We first characterize the semantics of a program in terms of a transition system T whose states are (tuples of) local states. Let us first consider the following simple program P: `for i1:=1 to length(A) do P1`. Here, and in the rest of the proof, we assume that the length of the array is non-zero. (In the case the length of the array is zero, the program effectively contains no loops, and reachability can be computed in time polynomial in number of variables.) The program P can be seen as a transition system whose states are local states of P and which processes an input word in $\Sigma \times D$, with each iteration consuming one symbol of the word. For sequential composition of commands, a product construction (augmented with some bookkeeping) is used.

Note that T is still an infinite-state system, as its states store values from D. Therefore, we construct a finite state system T^α that abstracts the infinite part of the local states, that is the values of index, loop and data variables. The abstract state transition system T^α keeps only order and equality information on the index, loop and data variables. Let IV be the set of index and loop variables of P. Let DV be the set of data variables of P. An abstract state is a tuple (m, SI, SD), where m is a boolean state in M; SI defines an equivalence relation on IV and a total order on equivalence classes of this relation; and SD defines an equivalence relation on $DV \cup IV$ and a total order on equivalence classes of this relation. An abstract state represents a set of local states. For example, if a program has an index variable `p1`, a loop variable `i1` and a data variable `v1`, a possible abstract state is $(m, \texttt{p1} < \texttt{i1}, \texttt{p1} = \texttt{i1} < \texttt{v1})$. This abstract state represents a set of concrete states

whose boolean state is m and, the value of `p1` is less than the value of `i1`, the value of the array at position `p1` is the same as the value of the array at position `i1`, which is less than the value of `v1`.

We show that reachability of a boolean state m can be decided on the abstract system, in the sense that m is reachable in T if and only if it is reachable in T^α. (A boolean state m is reachable in T^α iff there exist SI and SD such that (m, SI, SD) is reachable in T^α.) The main part of the proof shows that every finite path in the abstract transition system is feasible in the concrete transition system.

The first idea for a proof might be to show that the abstraction defines a bisimulation between abstract and concrete transition systems. However, this is not the case. We present a simple counterexample. Let us consider a program `P` and let us focus on two data variables `v1` and `v2`. Let q_1 be a local state such that its boolean component is m, the value of `v1` at q is 5 and the value of `v2` at q is 6. The abstract state corresponding to r_1, r_1^α is thus (m, SI, SD), where SD, the order on data and index variables, includes $\texttt{v1} < \texttt{v2}$. Furthermore, let us suppose that the program is such that the abstract state r_1^α can transition (in a way that does not change the values of `v1` and `v2`) to an abstract state r_2^α that requires that another data variable `v3` has a value greater than the value of `v1`, but smaller than the value of `v2`. Note now that the concrete state r_1 cannot transition to any state that would correspond to the order on data variables required by r_2^α, because there is no value between 5 and 6.

In a key part of the proof, we show that if an abstract state r_2^α is reachable from r_1^α, then there exists a state r_1 (abstracted by r_1^α) and a state r_2 (abstracted by r_2^α) such that r_2 is reachable from r_1. The main idea for proof by induction is that we can choose r_1 in such a way that the gaps between values are large enough. More precisely, if (1) r_1^α requires that e.g. `v1 > v2` for two data variables `v1` and `v2` and (2) r_2^α is reachable from r_1^α in k steps, then it is sufficient to choose r_1 such that $\texttt{v1} - \texttt{v2} > 2^k$.

Proof of Theorem 13. To simplify the presentation, we will suppose that there are no constants in D in the programs. At the end of this subsection, we will explain how the proof that follows can be extended to programs with constants from D.

Transition system semantics. We show that for programs that contain only non-nested loops and are in normal form, $[\![\texttt{P}]\!]$ can be represented by a triple (e, T, f), where $T = (R, \delta \subseteq R \times (\Sigma \times D) \times R, F)$ is a transition system whose set of states is R. The set $F \subseteq R$ is the set of final states. The transition relation δ will simulate executions of the loops that appear in the program. Its input will be, in addition to a state from R, also a pair (a, d) from $(\Sigma \times D)$ representing the current element of the input array. The relation e is a subset of $Q \times R$ and the relation f is a subset of $F \times Q$. The relation e will represent the loop-free part of the program before the first non-nested loop, and the relation f will represent the loop-free part of the program after the last non-nested loops. Recall that for program in normal form, loops do not appear in branches of `if` statements.

We define a function $[\![\texttt{P}]\!]^t$ which for loop-free programs returns a binary relation over Q, and returns a triple (e, T, f) for programs that contain non-nested loops. Intuitively, a loop free program `P` will be represented by a binary relation over Q. For a loop command we use the relation representing the (loop free) body of the loop to construct a transition system. For sequential composition of commands, a product construction augmented with some bookkeeping is used. We explain the construction for two sequentially composed loops that iterate through the array. The transition system is a product of the transition systems defined by the two loops, and the bookkeeping part ensures that the second loop starts from a state where the first loop finished.

Construction of $[\![\texttt{P}]\!]^t$ For the following commands `P`: `skip`, `v := E`, `if B then P1 else P2`, `if * then P1 else P2`, $[\![\texttt{P}]\!]^t$ is defined by

$$[\![\texttt{P}]\!]^t = \{(loc(g), loc(g'))|(g, g') \in [\![\texttt{P}]\!]\}.$$

Note that for the conditionals, we have that `P1` and `P2` are loop-free. For loops and sequential composition we have:

- $[\![\texttt{for i1:=1 to length(A) do P}]\!]^t = (e, (Q, \delta, Q), f)$, where e and f are identity relations on Q, and $(q, (a, d), q') \in \delta$ if there exists a local state $q'' \in Q$ such that $(q, q'') \in [\![\texttt{P}]\!]^t$ and $q' = q''[\texttt{i1} = i + 1, \texttt{A\_i1} = (a, d)]$, where $i = q[\texttt{i1}]$. (Note that `P` is loop free.)

- $[\![\texttt{P1;P2}]\!]^t$ is defined as follows:

  1. If $[\![\texttt{P1}]\!]^t = f_1$ and $[\![\texttt{P2}]\!]^t = f_2$, then $[\![\texttt{P1;P2}]\!]^t = f_1 \circ f_2$.
  2. If $[\![\texttt{P1}]\!]^t = f_1$ and $[\![\texttt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\texttt{P1;P2}]\!]^t = ((f_1 \circ e_2), T_2, f_2)$.
  3. If $[\![\texttt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\texttt{P2}]\!]^t = f_2$, then $[\![\texttt{P1;P2}]\!]^t = (e_1, T_1, (f_1 \circ f_2))$.
  4. If $[\![\texttt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\texttt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\texttt{P1;P2}]\!]^t = (e, T, f)$, where the components are defined as follows. Let $T_1 = (R_1, \delta_1, F_1)$ and $T_2 = (R_2, \delta_2, F_2)$. The transition system $T = (R, \delta, F)$ is defined as follows: $R = R_1 \times R_2 \times R_2$, $((r_1, r_2, r_3), (a, d), (r_1', r_2', r_3')) \in \delta$ iff $(r_1, (a, d), r_1') \in \delta_1$, $r_2 = r_2'$, and $(r_3, (a, d), r_3') \in \delta_2$. A state (r_1, r_2, r_3) is in F if and only if $r_1 \in F_1$, $r_3 \in F_2$, and $(r_1, r_2) \in (f_1 \circ e_2)$. The function e is defined in the following way: $(q, (r_1, r_2, r_3)) \in e$ if and only if $r_2 = r_3$ and $(q, r_1) \in e_1$. For the function f, we have $((r_1, r_2, r_3), q) \in f$ if $(r_1, r_2, r_3) \in F$ and $(q, r_3) \in f_2$.

We now show that $[\![\texttt{P}]\!]^t = (e, T, f)$ captures the semantics of `P`. In what follows, we suppose that the program that we analyze contains at least one non-nested loop, and therefore $[\![\texttt{P}]\!]^t$ has the form (e, T, f).

Given a transition system $T = (R, \delta, F)$, where δ is a subset of $R \times (\Sigma \times D) \times R$, we extend the definition of δ to words in $(\Sigma \times D)^*$. We define a relation δ^* on $R \times (\Sigma \times D)^* \times R$ as follows: for $w = w_1 \ldots w_l$ we have that $(r, w, r') \in \delta^*$ iff

$\exists r_1, \ldots r_{l+1}$ such that $r = r_1$, $r' = r_{l+1}$ and for all i such that $1 \leq i \leq l$ we have that $(r_i, w_i, r_{i+1}) \in \delta$.

Given a word w in $(\Sigma \times D)^*$, we say that q_2 *is w-reachable from* q_1 *in* $[\![P]\!]^t$ iff $[\![\mathtt{P}]\!]^t = (e, T, f)$, $T = (R, \delta, F)$ and there exist $r_1, r_2 \in R$ such that $(q_1, r_1) \in e$, $(r_2, q_2) \in f$, and $(r_1, w, r_2) \in \delta^*$.

Lemma 14. *A local state q_2 is w-reachable from q_1 in $[\![P]\!]^t$ if and only if there exist states g_1 and g_2 such that $loc(g_1) = q_1$, $g_1[A] = w$, $loc(g_2) = q_2$, $g_2[A] = w$ and $(g_1, g_2) \in [\![P]\!]$.*

Proof. The proof uses induction on the structure of the program P. □

A boolean state m is *w-reachable in* $[\![P]\!]^t$ if there exist an initial local state q_I, a local state q such that $bool(q) = m$ and q is w-reachable from q_I in $[\![\mathtt{P}]\!]^t$.

The next lemma follows from Lemma 14.

Lemma 15. *Given a program P, a boolean state m is reachable if and only if there exists a word $w \in (\Sigma \times D)^*$ such that m is w-reachable in $[\![P]\!]^t$.*

Furthermore, if $[\![\mathtt{P}]\!]^t = (e, T, f)$ and $T = (R, \delta, F)$, we have that $R = Q^{2k-1}$, where k is the number of loops in P.

Abstract transition system. We fix a program P for the rest of this subsection. Let $[\![\mathtt{P}]\!]^t$ be (e, T, f), where $T = (R, \delta, F)$, and $R = Q^{2k-1}$. We show that we can find a finite state system T^α (and corresponding relations e^α and f^α) such that we can reduce reachability in T to reachability in T^α. The main idea in the construction of the abstract transition system is that it will keep track of only the order of index and data variables, not their values.

We will need an abstract version of the set Q. Let IV be the set of index and loop variables of P. Let DV be the set of data variables of P. An abstract state is a tuple (m, SI, SD), where m is a boolean state in M, SI is a total order on equivalence classes on IV and SD is a total order on equivalence classes on $DV \cup IV$. For example,

if a program has an index variable `p1`, a loop variable `i1` and a data variable `d1`, a possible abstract state is $(m, \texttt{p1} < \texttt{i1}, \texttt{p1} = \texttt{i1} < \texttt{d1})$. This means that the program is in a boolean state m, `p1` is less than `i1`, and `A[p1]` is equal to `A[i1]` and is less than `d1`. Let Q^α be the set of abstract states.

We will also need an abstract version of R, the set of states of T. We consider sets IV^{2k-1} and DV^{2k-1}, where there are $2k-1$ copies of each variable. Let SI_R be a total order on IV^{2k-1} and let SD_R be a total order on $DV^{2k-1} \cup IV^{2k-1}$. We will consider the set $U = M^{2k-1}$. Let R^α be the set of abstract states of the form (u, SI_R, SD_R), where u is in U.

The abstraction function $\alpha_Q : Q \to Q^\alpha$ can be defined straightforwardly: $\alpha_Q(q) = (m, SI, SD)$ iff $bool(q) = m$ and for all index and loop variables `p1`, `p2`, we have that `p1` $<$ `p2` in SI iff $q[\texttt{p1}] < q[\texttt{p2}]$, and `p1 = p2` in SI iff $q[\texttt{p1}] = q[\texttt{p2}]$. The definition is similar for SD. We present the case of one index variable `p1` and one data variable `v1`. We have that `p1` $<$ `v1` in SD if and only if $[\![\texttt{A[p1]}]\!] < q[\texttt{v1}]$, and `p1` $=$ `v1` in SD if and only if $[\![\texttt{A[p1]}]\!] = q[\texttt{v1}]$. We define the abstraction function $\alpha_R : R \to R^\alpha$ similarly.

We now define the abstract transition system. More precisely, we define $[\![\texttt{P}]\!]^\alpha = (e^\alpha, T^\alpha, f^\alpha)$ using $[\![\texttt{P}]\!]^t$ as follows: Let $T^\alpha = (R^\alpha, \delta^\alpha, F^\alpha)$. The transition relation $\delta^\alpha \subseteq R^\alpha \times R^\alpha$ is defined in a standard way: $\delta^\alpha(r_1^\alpha, r_2^\alpha)$ iff there exist r_1, r_2 and a pair $(a, d) \in (\Sigma \times D)^*$, such that $(r_1, (a, d), r_2) \in \delta$ and $\alpha(r_1) = r_1^\alpha$ and $\alpha(r_2) = r_2^\alpha$. The set F^α of final states is defined as follows: $r^\alpha \in F^\alpha$ iff there exists $r \in F$ and $\alpha(r) = r^\alpha$. The relation $\delta^{\alpha *}$ denotes the transitive closure of δ^α. Given a relation e on $Q \times R$, we define its abstract version e^α on $Q^\alpha \times R^\alpha$ similarly to the definition of the abstract transition relation. Also, given a relation f on $R \times Q$, we define its abstract version f^α on $R^\alpha \times Q^\alpha$.

The following lemma is the key part of the proof. It relates reachability of a boolean state in the abstract and concrete systems.

Lemma 16. *For all r_1^α, r_2^α in R^α,we have that $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$ if and only if there exist $r_1, r_2 \in R$ and a word $w \in (\Sigma \times D)^*$ such that $\alpha(r_1) = r_1^\alpha$, $\alpha(r_2) = r_2^\alpha$, and $\delta^*(r_1, w, r_2)$.*

Proof. It is straightforward to prove that if there exist r_1, r_2 and w such that $\alpha(r_1) = r_1^\alpha$, $\alpha(r_2) = r_2^\alpha$, and $\delta^*(r_1, w, r_2)$ then $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$. We only need to apply the definition of δ^α inductively.

The proof of the other implication uses induction on the length of the path from r_1^α to r_2^α that witnesses $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$. We will also need the following notion: The relation $Gap(r, o)$ holds for $r \in R$ and $o \in \mathbb{N}$ iff for all data variables (and values pointed to by index variables) $\mathtt{v1}$, $\mathtt{v2}$, we have that if $r[\mathtt{v1}] > r[\mathtt{v2}]$, then $r[\mathtt{v1}] - r[\mathtt{v2}] \geq o$. The relation $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$ is defined as follows: $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$ if there exists a state $r_3^\alpha \in R^\alpha$ such that $\delta^\alpha(r_1^\alpha, r_3^\alpha)$ and $\delta_{k-1}^\alpha(r_3^\alpha, r_2^\alpha)$ for $k > 1$; and $\delta_1^\alpha = \delta^\alpha$.

We will prove the following inductive claim: If $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$, then for all r_1 such that $\alpha_R(r_1) = r_1^\alpha$ and $Gap(r_1, 2^k)$, there exists r_2 and a word $w \in (\Sigma \times D)^k$, such that $(r_1, w, r_2) \in \delta$, and $\alpha_R(r_2) = r_2^\alpha$.

The base case, where $k = 0$ is straightforward. For the inductive case, suppose that $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$. Then there exists a state $r_3^\alpha \in R^\alpha$ such that $\delta^\alpha(r_1^\alpha, r_3^\alpha)$ and $\delta_{k-1}^\alpha(r_3^\alpha, r_2^\alpha)$. Let r_1 be such that $\alpha(r_1) = r_1^\alpha$ and $Gap(r_1, 2^k)$. (It is easy to show that such r_1 exists for all r_1^α.) We need to find a state $r_3 \in R$ and a pair $(a, d) \in \Sigma \times D$ such that $(r_1, (a, d), r_3) \in \delta$, $\alpha_R(r_3) = r_3^\alpha$ and $Gap(r_3, 2^{k-1})$. This is done by case analysis of the transition $\delta^\alpha(r_1^\alpha, r_3^\alpha)$. Informally, the transition can require that the data value d of the current position (the position pointed to by the loop variable) has to be between two stored values, but as $Gap(r_1, 2^k)$ holds, we can always choose d such that we ensure that $Gap(r_3, 2^{k-1})$. We can conclude by using induction hypothesis for $\delta_{k-1}^\alpha(r_3^\alpha, r_2^\alpha)$. □

A boolean state m is *reachable in* $[\![P]\!]^\alpha$ if there exists an initial state g_I, an abstract state q_I^α such that $\alpha(loc(g_I)) = q_I^\alpha$, and states $q_2^\alpha \in Q^\alpha$, $r_1^\alpha, r_2^\alpha \in R^\alpha$ such that $(q_I^\alpha, r_1^\alpha) \in e^\alpha$, $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$, $(r_2^\alpha, q_2^\alpha) \in f^\alpha$, and $bool(q_2^\alpha) = m$.

Lemma 17. *A boolean state m is reachable if and only if it is reachable in $[\![P]\!]^\alpha$.*

Proof. The proof uses Lemmas 15 and 16. □

Complexity The proofs of the preceding lemmas give rise to an algorithm for deciding reachability of a boolean state m. The algorithm tests reachability of m in the abstract transition system. We show that the algorithm is in PSPACE. The number of states in T^α depends exponentially the number of variables in the program. Furthermore, given two abstract states, r_1^α and r_2^α, one can decide (in polynomial time in the number of variables), whether the tuple (r_1^α, r_2^α) is in δ^α.

In order to show that the problem is PSPACE-hard, we can reduce SUCCINCT-REACHABILITY (see [66]) to our reachability problem. Note that the resulting instance will use only boolean variables, not data or index variables.

This completes the proof of Theorem 13.

As noted above, we presented the proof for programs without constants in D. The proof can be extended to programs with constants in a straightforward way: Let c_1 be the smallest and let c_2 be the greatest constant that a given program `P` uses. The abstract system $[\![\texttt{P}]\!]^\alpha$ will need to track the values between c_1 and c_2 precisely, and track only the order between the stored values for values less than c_1 or greater than c_2. The resulting system will thus still have a finite number of states. The reachability problem can be solved in space polynomial in the number of variables and the size (number of bits) of the largest constant.

4.2.1 Finite data domain

In this subsection, we consider the case when the data domain D is finite. We also syntactically restrict the programs: we consider programs which do not have index variables and which do not contain expressions of the form `IE = IE` and `IE` $<$ `IE`, that is the index expressions (consisting now only of loop variables) are not compared. We call these programs *index-free*. The reason we consider this restriction

is that in this case, the local state needs only to store a fixed number of data values. As the data values are from a finite domain, the set of local states is finite.

Since the set of local states is finite, there is a natural question about how such programs are related to finite state automata on words. Let us consider an execution of a program `P`. The traversal order of this execution is different from standard finite state automata, as the program reads the input array many times. The number of times `P` scans the array in fact depends on the length of the input word, and is therefore unbounded. If n is the length of the input word, and k is the nesting depth of loops in a program `P`, then `P` scans the array n^k times.

We show that for index-free programs on a finite domain D, we can allow the nesting depth of loops to be arbitrary without losing decidability of the reachability problem.

Theorem 18. *Reachability is decidable for index-free programs if the data domain D is finite. The problem is in* EXPSPACE.

We start by describing the structure of the proof. We will show how all the traversals of the array can be simulated by a finite state system. We explain the main idea of the construction using a program with a doubly nested loop. The number of iterations of the inner loop depends on the length of the input word. However, each iteration can be characterized by a pair of states of the program - a state q_1 in which it begins, and a state q_2 in which it ends. We can thus reduce the reachability problem to the reachability problem in a finite state system T whose states will contain sets of pairs of states of the original program (and, in addition, some bookkeeping information). The number of states of T is thus doubly-exponential in the number of variables of the program. The reachability problem can therefore be solved in EXPSPACE.

The full proof of Theorem 18 follows. We will first construct the finite state system T.

Construction of a finite-state transition system. In the rest of this subsection, we fix a program `P`. A program in normal form can be seen as a sequence $f_1L_1f_2L_2 \ldots f_kL_kf_{k+1}$, where f_i is a binary relation over Q representing a loop-free part of the program, and L_i is a loop, i.e. a command of the form `for i1:=1 to length(A) do P1`. We present the construction of such a sequence $Seq(\texttt{P})$.

For the following commands `P`: `skip`, `v := E`, `if B then P1 else P2`, `if * then P1 else P2`, $Seq(\texttt{P})$ is defined by

$$Seq(\texttt{P}) = \{(loc(g), loc(g'))|(g, g') \in [\![\texttt{P}]\!]\}.$$

Note that for the conditionals, we have that `P1` and `P2` are loop free. For loops and sequential composition we have:

- $Seq(\texttt{for i1:=1 to length(A) do P}) = f_1Lf_2$, where $(q, q) \in f_1$ and $(q, q) \in f_2$, for all $q \in Q$, and `L = for i1:=1 to length(A) do P`.
- $Seq(\texttt{P1;P2})$ is defined as follows: If $Seq(\texttt{P1}) = f_1L_1f_2L_2 \ldots f_kL_kf_{k+1}$ and $Seq(\texttt{P2}) = f'_1L'_1f'_2L'_2 \ldots f'_{k'}L'_{k'}f'_{k'+1}$, then $Seq(\texttt{P1;P2}) = f_1L_1f_2L_2 \ldots f_kL_k(f_{k+1} \circ f'_1)L'_1f'_2L'_2 \ldots f'_{k'}L'_{k'}f'_{k'+1}$.

Given a program `P`, we construct a finite-state transition system $FS(\texttt{P}) = (S, D, \delta, s_0)$. Let L be the set of all loop commands that appear in the program. Let S_P be a set of all tuples of the form $Q \times Q \times L \times Q$. The set S is then $2^{S_P} \cup \{s_0\}$, where s_0 will be the initial state.

Given two states q_1 and q_2, we say that a state s of $FS(\texttt{P})$ models a triple $(q_1, \texttt{P}, q_2)$ (denoted by $s \models (q_1, \texttt{P}, q_2)$) iff $Seq(\texttt{P}) = f_1L_1f_2L_2 \ldots f_{k+1}$ and there exist $q^1_1q^2_1q^1_2q^2_2 \ldots q^1_kq^2_kq^1_{k+1}$ such that for all i, if $1 \leq i \leq k$, then $(q^1_i, q^2_i) \in f_i$, there exists a state q such that $(q^2_i, q, L_i, q^1_{i+1})$ is in s, $q^1_1 = q_1$ and $q^1_{k+1} = q_2$.

A state s is called *starting* iff there exist an initial local state q_1 and a local state q_2 such that $s \models (q_1, \texttt{P}, q_2)$ and for all tuples $(q_1, q_2, \texttt{L}, q_3)$ in s, we have that $q_1 = q_2$.

A state s is called *ending* iff for all tuples $(q_1, q_2, \texttt{L}, q_3)$ in s, we have that $q_2 = q_3$.

The transition relation $\delta \subseteq S \times ((\Sigma \times D) \cup \{\epsilon\}) \times S$ is defined as follows. The initial state s_0 transitions on ϵ to a state s iff s is a starting state. In addition, we have that $(s_1, (a, d), s_2) \in \delta$ iff for all tuples $t \in s$, there exists a tuple $t' \in s'$ such that $t \xrightarrow{(a,d),s,s'} t'$, and for all tuples $t' \in s'$, there exists a tuple $t \in s$ such that $t \xrightarrow{(a,d),s,s'} t'$. The auxiliary relation $t \xrightarrow{(a,d),s,s'} t'$ is defined as follows: $(q_1, q_2, \texttt{L}, q_3) \xrightarrow{(a,d),s,s'} (q_1', q_2', L, q_3')$ iff `L = for i1 := 1 to length(A) do P`, $q_1 = q_1'$, $q_3 = q_3'$, and there exist a state q'' such that $s \models (q_2, P, q_2'')$ and $q_2' = q_2''[\texttt{A\_i1} = (a, d)]$.

Given a data word w in $(\Sigma \times D)$, we say that q_2 *is w-reachable from q_1 in $FS(\texttt{P})$* iff there exist a starting state s, an ending state s', a word $w = w_1 w_2 \ldots w_l$, and states $s_1, s_2, \ldots, s_{l+1}$ such that $s \models (q_1, P, q_2)$, $s = s_1$, $s' = s_{l+1}$, and $(s_i, w_i, s_{i+1}) \in \delta$, for all i such that $1 \leq i \leq l$.

Lemma 19. *A local state q_2 is w-reachable from q_1 in $FS(\texttt{P})$ if and only if there exist states g_1 and g_2 such that $loc(g_1) = q_1$, $g_1[\texttt{A}] = w$, $loc(g_2) = q_2$, $g_2[\texttt{A}] = w$ and $(g_1, g_2) \in [\![\texttt{P}]\!]$.*

Proof. The proof uses induction on the nesting depth of `P`. The inductive step is proven by induction on the number of sequentially composed loops in the program. □

A boolean state m is *w-reachable in $Seq(\texttt{P})$* if there exist an initial local state q_I and a local state q, such that $bool(q) = m$, and q is w-reachable from q_I in $FS(\texttt{P})$.

The proof of the following lemma uses Lemma 19.

Lemma 20. *A boolean state m is reachable if and only if it is reachable in $FS(\texttt{P})$.*

Lemma 20 reduces the reachability problem to the reachability problem in a finite state transition system whose size is doubly exponential in the number of variables of the program. We also have that given two states of $FS(\texttt{P})$, s_1 and s_2, and a pair $(a, d) \in (\Sigma \times D)$, it is possible to decide (in polynomial time in the number of variables), whether $(s_1, (a, d), s_2) \in \delta$. Therefore we have that the problem of deciding reachability is in EXPSPACE. This concludes the proof of Theorem 18.

4.3 Programs, automata and logics on data words

In this section, we will examine the decidability boundary for array-accessing programs, and compare the expressive power of these programs to that of logics and automata on data words. We will show that the reachability problem for Restricted-ND2 programs is decidable, and that it is undecidable for full ND2 programs. We start by reviewing the results on automata and logics on data words, as these will be needed for the decidability proof. We will reduce the reachability problem for Restricted-ND2 programs to the nonemptiness problem of extended data automata, a new variation of data automata. The latter is a definition intended to correspond to the notion of regular automata on finite words.

4.3.1 Background

We briefly review the results on automata and logics on data words from [19]. Recall that a data word is a sequence of pairs $\Sigma \times D$. A *data language* is a set of data words. Let w be a data word $(a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$. The string $str(w) = a_1 a_2 \dots a_n$ is called the string projection of w. Given a data language L, we write $str(L)$ to denote the set $\{str(w) \mid w \in L\}$. A class is a maximal set of positions in a data word with the same data value. Let $\mathcal{S}(w)$ be the set of all classes of the data word w. For a class X in $\mathcal{S}(w)$ with positions $i_1 < \ldots < i_k$, the class string $str(w, X)$ is $a_{i_1} \dots a_{i_k}$.

Data automata A *data automaton* (DA) $\mathcal{A} = (G, C)$ consists of a transducer G and a class automaton C. The transducer G is a nondeterministic finite-state letter-to-letter transducer from Σ to Γ and C is a finite-state automaton on Γ. A data word $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ is accepted by a data automaton $\mathcal{A}$ if there is an accepting run of G on the string projection of w, yielding an output string $b = b_1 \dots b_n$, and for each class X in $\mathcal{S}(w')$, the class automaton C accepts $str(w', X)$, where $w' = w'_1 \dots w'_n$ is defined by $w'_i = (b_i, d_i)$, for all i such that $1 \leq i \leq n$. Given a DA $\mathcal{A}$, $L(\mathcal{A})$ is the language of data words accepted by $\mathcal{A}$. The

nonemptiness problem for data automata is decidable. The proof is by reduction to a computationally complex problem, the reachability problem in Petri nets.

Logics on data words We define logics whose models are data words. Following [19], we consider two predicates on positions in a data word whose definition also involves the data values at these positions. The predicate $i \approx j$ is satisfied if both positions i and j have the same data value. The predicate $i \prec j$ is satisfied if the data value at position i is smaller than the data value at position j. Furthermore, standard successor and order predicates on positions in a data word are used.

Let us first consider logics that use the $\approx$ predicate and not the $\prec$ predicate. We first note that for a first order logic $\mathrm{FO}(\approx, <, +1)$ satisfiability is undecidable, even if we restrict the number of variables to three. If we restrict the number of variables to two, the logic becomes decidable, and the proof is by reduction to the nonemptiness problem of data automata. The decidability naturally extends to existentially quantified second order monadic logic with two first order variables, denoted by $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. Moreover, $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ is precisely equivalent in expressive power to data automata. The predicate $\oplus 1$ denotes the class successor, and $i \oplus 1 = j$ is satisfied if i and j are two successive positions in the same class of the data word. Furthermore, the logic $\mathrm{EMSO}^2(\approx, <, +\omega, \oplus 1)$ is included in $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. The symbol $+\omega$ represents all predicates of the form $+k$, $k \in \mathbb{N}$, i.e. the logic includes all predicates $i + 2 = j$, $i + 3 = j$, etc.

Example 11. We present a data automaton $\mathcal{A}$ such that $str(L(\mathcal{A}))$, the set of string projections, is exactly the set of all words over $\{a, b, c\}$ that contain the same number of as, bs, and cs. The transducer of $\mathcal{A}$ computes the identity function, i.e. it accepts all words and its output string is the same as its input string. The class automaton ensures, for each class, that the class contains exactly one occurrence of a, one occurrence of b and one occurrence of c.

4.3.2 Extended data automata

Position-preserving class string Note that the class automaton does not know the positions of symbols in the word w. The symbols from other classes have simply been erased. However, let us consider a program with a doubly-nested loop where `i` is the loop variable of the outer loop and `j` is the loop variable of the inner loop, and let us suppose that the program inside the inner loop is of the form: `if (A[i].d=A[j].d) then P1 else P2`.

The inner loop of the program scans the array from left to right and and modifies the state in two different ways (given by `P1` and `P2`), depending on whether `(A[i].d=A[j].d)` holds or not. Simply erasing the positions from other classes seems therefore not good enough. We thus define an extension of the notion of class string and a corresponding extension of the class automaton.

Given a data word $w \in (\Sigma \times D)^*$, a *position-preserving class string* $pstr(w, X)$ is a string over $\Sigma \cup \{0\}$. (We assume that $0 \notin \Sigma$.) Let $w = w_1 w_2 \dots w_n$, let i be a position in w, and let w_i be (a_i, d_i). The string $v = pstr(w, X)$ has the same length as w, and for v_i we have that $v_i = a_i$ iff $i \in X$, and $v_i = 0$ otherwise. That is, for each position i which does not belong to X, the symbol from Σ at the position i is replaced by 0.

An *extended data automaton* (EDA) $\mathcal{E} = (G, C)$ consists of a transducer G and a class automaton C. The transducer G is a finite-state letter-to-letter transducer from Σ to Γ and C is a finite-state automaton over $\Gamma \cup \{0\}$. A data word $w = w_1 \dots w_n$ is accepted by the EDA $\mathcal{E}$ if there is an accepting run of G on the string projection of w, yielding an output string $b = b_1 \dots b_n$, and for each class X in $\mathcal{S}(w')$, the class automaton C accepts $pstr(w', X)$, where $w' = w'_1 \dots w'_n$ is defined as follows: $w'_i = (b_i, d_i)$, for all i such that $1 \leq i \leq n$. Given an EDA $\mathcal{E}$, $L(\mathcal{E})$ is the language of data words accepted by $\mathcal{E}$.

Example 12. We consider L, a language of data words defined by the following

property: A data word w is in L iff for every class X in $\mathcal{S}(w)$, we have that between every two successive positions in the class, there is exactly one position from another class. We show that there exists an EDA $\mathcal{E} = (G, C)$ such that $L(\mathcal{E}) = L$. The transducer G computes the identity function. The class automaton C is given by the following regular expression: $0^*(\Sigma 0)^*0^*$. It is easy to see that $\mathcal{E}$ accepts L. We first note that for each DA $\mathcal{A}$, it is easy to find an EDA $\mathcal{E}$ such that $L(\mathcal{E}) = L(\mathcal{A})$. We just modify the class automaton C, by adding the tuple $(q, 0, q)$, for each q, to the transition relation. This means that on reading 0 the state of the class automaton does not change.

We will also show in this section that for each EDA $\mathcal{E}$ we can find an equivalent DA $\mathcal{A}$. This might not be obvious at a first glance, as class automata of DAs do not get to see the distances between positions in a class. Indeed, we show that the language from Example 12 cannot be captured by a deterministic DA. However, we show that $\text{EMSO}^2(\approx, +1, \oplus 1)$ and EDAs are expressively equivalent, and since $\text{EMSO}^2(\approx, +1, \oplus 1)$ and DAs are also expressively equivalent, we conclude that for every EDA there exists a DA that accepts the same language. Showing that for every EDA there exists an equivalent $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula also establishes that non-emptiness is decidable for EDAs. However, the proof of decidability of satisfiability of $\text{EMSO}^2(\approx, +1, \oplus 1)$ formulas is rather involved. We present a direct proof for decidability of emptiness for EDAs, as it also gives an intuitive reason why emptiness is decidable fro EDAs.

Theorem 21. *Given an EDA $\mathcal{E}$, it is decidable whether $L(\mathcal{E}) = \emptyset$.*

Proof. Let $\mathcal{E} = (G, C)$ be an EDA, let G be defined by a tuple $(Q_G, \Sigma, \Gamma, \delta_G, q_0^G, F_G)$, and let C be defined by a tuple $(Q_C, \Gamma, \delta_C, q_0^C, F_C)$. We start by describing a more operational view of EDAs. A *run* of an EDA on a data word w is a function ϱ from positions in w to tuples of the form (q, o, c), where $q \in Q_G$ is a state of the transducer G, o (a symbol from Γ) is the output of the transducer, and c is a function from $\mathcal{S}(w)$ to Q_C, the set of states of C. Furthermore, we require that ϱ is

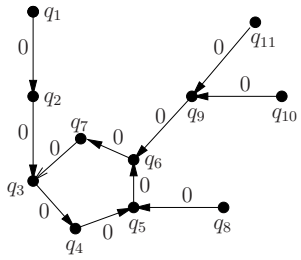

Figure 4.3: A connected component of a graph C_0 corresponding to an EDA $\mathcal{E}$

consistent with δ_G and δ_C, the transition functions of G and C. We define $\varrho(0)$ to be $(q_0^G, \gamma, \lambda X.q_0^C)$, i.e. the transducer and all the copies of the class automaton are in initial states. Furthermore, for each position i, $\varrho(i)$ is equal to (q', o', c') if and only if $w_i = (a, d)$, $\varrho(i-1) = (q, o, c)$ and (i) $(q', o') \in \delta_G(q, a)$, (ii) for the unique X such that $i \in X$ we have $c'(X) \in \delta_C(c(X), o')$, (iii) for X such that $i \notin X$ we have $c'(X) \in \delta_C(c(X), 0)$.

A run is *accepting* iff $\varrho(n) = (q, o, c)$, q is a final state of G and for all X in $\mathcal{S}(w)$, we have that $c(X)$ is a final state of C.

Let us consider the class automaton C. Without loss of generality, we suppose that C is a complete deterministic automaton on $\Gamma \cup \{0\}$. The transition function δ_C defines a directed graph C_0 with states of C as vertices and 0-transitions as edges, i.e. there is an edge (p_1, p_2) in C_0 if and only if $\delta_C(p_1, 0) = p_2$. Every vertex in C_0 has exactly one outgoing edge (and might have multiple incoming edges). Therefore, each connected component of C_0 has exactly one cycle. A vertex is called cyclic if it is part of a cycle, and it is called non-cyclic otherwise. It is easy to see that each connected component is formed by the cyclic vertices and their 0-ancestors. An example of a connected component is in Figure 4.3. The vertex labeled q_6 is cyclic, its ancestors q_9, q_{10}, q_{11} are non-cyclic.

The graph C_0 consists of a number of connected components. We denote these components by C_0^j, for $j \in [1..k]$, where k is the number of the components. Let

W be the set of all non-cyclic vertices. For each non-cyclic vertex v, let $D(v)$ be defined as follows: $D(v) = d$ for non-cyclic vertices connected to a cycle, where d is the length of the unique path connecting v to the closest cyclic vertex. For the graph C_0, we define $D(C_0)$ to be $\max_{v \in W} D(v)$.

Let i be a position in a data word w. The data word $w_1 w_2 \dots w_i$ is denoted by $\mathit{prefix}(w, i)$. Let us consider a position i in a data word w and the set of classes $\mathcal{S}(w)$. Let $\mathcal{S}_{act}(w, i)$ be a set of *active* classes, i.e. classes X such that there is a position in X to the left of the position i. More formally, a class $X \in \mathcal{S}(w)$ is in $\mathcal{S}_{act}(w, i)$ if the string $\mathit{str}(\mathit{prefix}(w, i), X)$ is not equal to 0^i.

Lemma 22. *Let ϱ be a run of $\mathcal{E}$ on w. Let i be a position in w. Let $\varrho(i)$ be (q, o, c). The number N of classes X, such that X is in $\mathcal{S}_{act}(w, i)$ and $c(X)$ is a noncyclic vertex, is bounded by $D(C_0)$, i.e. $N \leq D(C_0)$.*

Proof. Let i be a position in a word w. If $i \leq D(C_0)$, then the number of active classes is at most $D(C_0)$, and we conclude immediately.

Let us consider the case $i > D(C_0)$. Let $\varrho(i)$ be (q, o, c) and let s be the string of length $D(C_0)$ defined by $s = w_{i-D(C_0)+1}\ w_{i-D(C_0)+2} \dots w_i$. There are two possible cases for each class X in $\mathcal{S}(w)$:

- $\mathit{pstr}(s, X) = 0^{D(C_0)}$. Let $\varrho(i - D(C_0)) = (q', o', c')$, and let $c'(X) = v$. We can easily prove that $\delta^*_C(p, 0^e)$ is not in W, for all p and for all $e \geq D(p)$. By definition, $D(C_0) \geq D(q')$. Therefore, we can conclude that $c(X) \notin W$.

- $\mathit{pstr}(s, X) \neq 0^{D(C_0)}$. This is true for at most $D(C_0)$ classes, because, for all positions x, there is exactly one class X, such that the symbol at the position x of the class string $\mathit{pstr}(s, X)$ is not 0.

Therefore we have that $c(X) \in W$ for at most $D(C_0)$ classes. □

We reduce emptiness of EDAs to emptiness of multicounter automata. Multicounter automata are equivalent to Petri nets [40], and thus the emptiness of multicounter automata is decidable. We use the definition of multicounter automata

from [19].

A *multicounter automaton* is a finite, non-deterministic automaton extended by a number k of counters. It can be described as a tuple $(Q, \Sigma, k, \delta, q_I, F)$. The set of states Q, the input alphabet, the initial state $q_I \in Q$ and final states $F \subseteq Q$ are as in a usual finite automaton.

The transition relation is a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \{inc(i), dec(i)\} \times Q$. The idea is that in each step, the automaton can change its state and modify the counters, by incrementing or decrementing them, according to the current state and the current letter on the input (which can be ϵ). Whenever it tries to decrement a counter of value zero the computation stops and rejects. The transition of a multicounter automaton does not depend on the value of the counters in any other way. In particular, it cannot test whether a counter is exactly zero.

Nevertheless, by decrementing a counter k times and incrementing it again afterward it can check that the value of that counter is at least k.

A *configuration* of a multicounter automaton is a tuple $(q, (c_i)_{i \leq n})$, where $q \in Q$ is the current state and $c_i \in \mathbb{N}$ is the value of the counter i. A transition $(p, \epsilon, inc(i), q) \in \delta$ can be applied if the current state is p. For $a \in \Sigma$, a transition $(p, a, inc(i), q) \in \delta$ can be applied if furthermore the current letter is a. In the successor configuration, the state is q, while each counter value is the same as before, except for counter i, which now has value $c_i + 1$. Similarly, a transition $(p, a, dec(i), q) \in \delta$ with $a \in \Sigma \cup \{\epsilon\}$ can be applied if the current state is p, the current letter is a, if $a \in \Sigma$, and counter i is non-zero. In the successor configuration, all counter values are unchanged, except for counter i, which now has value $c_i = c_i - 1$. A run over a word w is a sequence of configurations that is consistent with the transition function. A run is accepting if it starts in the state q_I with all counters empty and ends in a configuration where all counters are empty and the state is final.

Lemma 23. *Let $\mathcal{E}$ be an EDA. A multicounter automaton V such that $str(L(\mathcal{E})) =$*

$L(V)$ can be computed from $\mathcal{E}$.

Proof. We present the construction of a multicounter automaton V that simulates $\mathcal{E}$. The multicounter automaton V simulates the transducer G and a number of copies of C. There is one copy per class in $\mathcal{S}(w)$, where w is the word the automaton is reading. We say that a class automaton performs a 0-transition if the input symbol it reads is 0, and it performs a Γ-transition if the input symbol it reads is from Γ. Intuitively, at each step, the automaton V:

1. Simulates the transducer G using the finite state part (i.e. not the counters).

2. It guesses to which class the current position belongs, and it executes the Γ-transition of the automaton for that class with the symbol that is the output of the transducer at this step. For all the other simulated automata, V executes the 0-transition. (This is sufficient because each position belongs to exactly one equivalence class.)

The counters of the multicounter automaton V correspond to the cyclic vertices in C_0. (In what follows, we call a state of C (non-)cyclic if it corresponds to a (non-)cyclic vertex in C_0.) The value of the counter h corresponds to the number of copies of C currently in the state h. The finite part of the automaton state tracks the number of copies in each non-cyclic state. The key idea of the proof is that the total number of copies in non-cyclic states is finite and bounded (by $D(C_0)$). This fact is implied by Lemma 22.

Furthermore, one copy e of the class automaton is used to keep track of all the classes that are not active yet, i.e. not in $\mathcal{S}_{act}(w, i)$ at step i - thus when a position-preserving class string contains a symbol in Γ for the first time, a new copy of the automaton C is started from the state at which the copy e is.

Let $\gamma \in \Gamma$ be the current input symbol. The automaton works as follows: The first step consists of the automaton V nondeterministically guessing the equivalence class X to which the current position belongs. The copy of the class automaton for

X is then set aside while the second step is performed. That is, if the copy is in state s, then s is remembered in a separate part of the finite state. In the second step, the automaton V simulates 0-transitions for all the other copies (other than the copy that performed the Γ-transition). For copies in non-cyclic states, this is done by a transition modifying the finite state of V. The copies that transition from a non-cyclic to a cyclic state are dealt with by modifying the finite state and increasing the corresponding counter. The copies in cyclic states are tracked in the counters. Note that if we restrict the graph to only cyclic states, each state has exactly one incoming and one outgoing 0-edge. For all the copies in cyclic states, the 0-transition is accomplished by 'relabeling' the counters. This is done by remembering in the the finite state of V for each loop for one particular state to which counter it corresponds. This is then shifted in the direction of the 0-transition.

The third step is to perform the Γ transition for the class X. For the copy of the automaton corresponding to this class, a Γ-transition is performed. That is, if it is in state q, and $\delta(q, \gamma) = q'$, then there are four possibilities:

- If q, q' are cyclic states, the counter corresponding to q is decreased and the counter corresponding to q' is increased.
- If q, q' are non-cyclic state, a transition that changes the state of V is made.
- If q is a cyclic state and q' is a non-cyclic state, the counter corresponding to q is decreased, and the finite state of V is changed to reflect that the number of copies in q' has increased.
- If q is a noncyclic state and q' is a cyclic state, the transition is simulated similarly.

□

This concludes the proof of Theorem 21.

4.3.3 Restricted doubly-nested loops

We will reduce the reachability problem of Restricted-ND2 programs to the emptiness problem of EDAs. The main idea of the proof is that the transducer G guesses an accepting run of the outer loop, while the class automaton C checks that the inner loop can be executed in a way that is consistent with the guess of the transducer. We will need the following notion: For a given program P and a given boolean state m, we consider a language of data words such that the execution of P on a word from this language ends in a global state whose boolean component is m. More precisely, the language $L_m(\mathtt{P})$ is the set of data words w, such that there exist an initial state g_I and a state g, such that $g_I[\mathtt{A}] = w$, $bool(g) = m$, and $(g_I, g) \in [\![\mathtt{P}]\!]$.

Theorem 24. *Reachability for Restricted-ND2 programs is decidable.*

Proof. In this proof, we fix a program P of the following form:

```
for i1 := 1 to length(A) do
   P1;
   for j1: 1 to length(A) do P2;
   P3
```

where P1, P2, P3 are loop free programs. We present the proof for programs of this form. It can be extended for general programs using product construction techniques similar to those from the proof of Theorem 13. Similarly to the proof of Theorem 13, we assume that we assume that the length of the array is non-zero. (In the case the length of the array is zero, the program effectively contains no loops, and reachability can be computed in time polynomial in number of variables.) Recall that according to the definition of Restricted-ND2 programs, P2 must be of the following form: `if A[i].d=A[j].d then P21 else P22`, where P22 cannot contain `A[i1].d` or `A[i1].s`.

Given a boolean state m_r, we construct an EDA $\mathcal{E} = (G, C)$ such that $w \in L_{m_r}(\mathtt{P})$ iff $w \in L(\mathcal{E})$.

The task of the finite state transducer $G = (Q_G, \Sigma, \Gamma, \delta_G, q_0^G, F_G)$ is to guess a run of the program P. The output alphabet Γ consists of tuples in $\Sigma \times M \times M \times V$, where M is the set of boolean states of the program P.

The main idea of the construction is that the transducer G guesses an accepting run of the outer loop, while the class automaton C checks that the inner loop can be executed in a way that is consistent with the guess of the transducer. If a position i is marked with (a_i, m, m', v), the class automaton corresponding to class X such that $i \in X$ will verify that if the inner loop, which ran when the loop variable of the outer loop pointed to i, was started at m, then it will finish at m'.

The set V is defined as $V_C \cup V_C' \cup \{e\}$, with $e \notin V$. The set V_C is the set of all constants from D that appear in the program P. The set V_C' contains a symbol c' for each $c \in V_C$. The symbol e will represent the fact that the current input is not equal to any of the constants in the program.

First, let us summarize the effect of the loop-free subprograms P1 and P3 by relations $f_1, f_3 \subseteq M \times (\Sigma \times V) \times M$. The programs P1 and P3 can access the boolean state, read the value $[\![\mathtt{A[i].s}]\!]$, compare the value $[\![\mathtt{A[i].d}]\!]$ to constants, and modify the boolean state.

The transducer reads a word $a_1 a_2 \ldots a_l \in \Sigma^*$, and produces a word $b_1 b_2 \ldots b_l \in \Gamma^*$ such that:

- $b_1 = (a_1, m, m', v)$, for some m such that there exists a global state g_I such that $(\mathit{bool}(g_I), (a, v), m) \in f_1$.
- for all i such that $1 \leq i < l$, if $b_i = (a_i, m_1, m_2, v)$ and $b_{i+1} = (a_{i+1}, m_1', m_2', v')$, then there exist boolean states m_3, m_1', m_2' such that $(m_2, (a_i, v), m_3) \in f_3$ and $(m_3, (a_{i+1}, v'), m_1') \in f_1$.
- $b_l = (a_l, m, m', v)$, for some $m \in M$ and $v \in V$ such that $(m', (a_l, v), m_r) \in f_3$.

- There is an additional requirement on the fourth component of the tuple (a, m, m', v) that will enable the class automaton to verify that the position of constants has been guessed consistently. The transducer guesses a value in $V_C \cup \{e\}$, but at the rightmost position where it guesses a particular value $v \in V_C$, it outputs v' instead of v. This enables the class automata to check that each value $v \in V_C$ has been guessed for at most one class.

It is straightforward to show that this is possible to do with a finite state transducer.

We now define the class automaton C. The position preserving class string defined by a data value d looks as follows:

$$00(a_1, m_1, m'_1, v)000(a_2, m_2, m'_2, v) \ldots 0(a_l, m_l, m'_l, v)00$$

The task of the class automaton is twofold. First, it checks that if we consider only non-0 elements of the sequence and project to the fourth component of the tuple, the sequence observed is either of the form e^* or v^*v', for a constant v. This ensures that constants have been guessed consistently, i.e. that each constant has been assigned to a unique class, and at most one constant has been assigned to a class.

Second, the class automaton for a class X checks that the inner loops that ran when `i1`, the variable of the outer loop, pointed to one of the positions belonging to X, can run as the transducer has guessed. That is, if the position $i \in X$ has a tuple of the form (a_i, m_i, m'_i, v), the inner loop that started at state m_i, with the value of `i1` equal to i, will finish at state m'_i. It is not difficult to construct a regular automaton this condition. □

The proof of Theorem 24 gives a decision procedure, but one whose running time is non-elementary. The reason is that while the problem of reachability in multicounter automata is decidable, no elementary upper bound is known.

However, the following proposition shows that the problem is at least as hard as the reachability in multicounter automata, which makes it unlikely that a more efficient algorithm exists. The best lower bound for the latter problem is EXPSPACE [58].

Proposition 25. *The reachability problem for multicounter automata can be reduced to the reachability problem for Restricted-*`ND2` *programs in polynomial time.*

Proof. Given a multicounter automaton A, we construct a Restricted-`ND2` program `P` operating on data words that encode runs of A. The proof is similar to the proof of Theorem 14 from [19].

More precisely, we will construct a program `P` with a boolean state m such that $L_m(\texttt{P})$ is non-empty if and only if there is an accepting run of A, i.e. iff $L(A)$ is non-empty.

Let A be defined by the tuple $(Q_A, \Sigma_A, k_A, \delta_A, q_I^A, F_A)$. The array of the program P is a word on $\Sigma \times D$. The set Σ is defined as $Q \cup \{I_j \mid 1 \leq j \leq k_A\} \cup \{D_j \mid 1 \leq j \leq k_A\}$. I_j models the increase operation of the counter j, and similarly, D_j models the decrease operation of the counter j. A transition $(q_1, inc(j), q_2) \in \delta_A$ is encoded by having q_1, I_j and q_2 as Σ values in successive positions. A transition $(q_1, dec(j), q_2) \in \delta_A$ is encoded by having q_1, D_j and q_2 as Σ values in successive positions.

The program `P` consists of two sequentially composed parts. The first part is a single non-nested loop that examines only the Σ part of the data word, and check whether: (a) the first symbol is the initial state of A, (b) the word encodes transitions in δ_A, and (c) the last position contains a state in F_A. The second part uses data values to check that each decrement matches exactly one previous decrement, and each increment matches exactly one subsequent decrement, that is, the counters are never less than zero, and are equal exactly to zero at the end of the computation. This is ensured by requiring that each occurrence of I_j has a different data value, while each occurrence of D_j has the same data value as exactly one preceding I_j.

It is easy to write a Restricted-ND2 program (of polynomial size) that checks the second part above.

```
b:=false;
b1:=true;
for i:= 1 to length(A) do {
  if (A[i].s = I_j) {
    for j:= 1 to length(A) do {
       b1 := checkForD_j
    }
  }
  ...
}
if b1:=true then b:=true;
```

The code fragment above shows the core structure of the program P. The reachability question we will ask is whether a state where b is true is reachable. The variable b is set to true at the end of the program if b1 is true, which will be the case only if no error was found during the tests such as $\texttt{checkForD}_\texttt{j}$ above. This particular test is run if the Σ part of the current element in the outer loop is equal to I_j and the test checks whether the equivalence class of the current element contains exactly two elements, with the other element at a position greater than the current value of i (no order tests need to be used, this can be checked by switching a boolean variable when i = j holds) and its Σ part contains D_j. Note that $\texttt{checkForD}_\texttt{j}$ denotes a few lines of code, it is not a procedure call. □

4.3.4 Undecidable extensions

We show that if we lift the restrictions we imposed on Restricted-ND2 programs, the reachability problem becomes undecidable.

Theorem 26. *The reachability problem for* `ND2` *programs is undecidable.*

Proof. The proof is by reduction from the reachability problem of two-counter automata [62]. We note that the proof also implies that the reachability problem is undecidable even for `ND2` programs that do not use order on the data domain and do not use index or data variables.

The proof uses a reduction from reachability in two-counter automata. Two-counter automaton has a finite set of states and two integer counters. The main difference between two-counter automata and multicounter automata is that a two-counter automaton can test whether the value of a counter is equal to 0. More precisely, a two-counter machine is a tuple (Q, δ, q_0, F), where Q is a set of states, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states. The transition relation δ is a subset of $Q \times ((\{+\} \times \{1,2\} \times Q) \cup (\{-\} \times \{1,2\} \times Q \times Q))$. A configuration is a tuple in $Q \times \mathbb{N} \times \mathbb{N}$ representing the current state and current values of the two counters. At each step, the machine can either increment one of the counters and transition to a new state, or (try to) decrement one of the counter and transition to one of two possible new states depending on whether the value of the counter was 0. The automaton accepts by final state. It is well-known that the nonemptiness problem for multicounter automata is undecidable [62, 56].

Given a two-counter automaton A, we construct an `ND2` program `P` that has a boolean state m such that for all data words w, we have that $w \in L_m$ iff w encodes runs of A. A configuration (q, i, j) is encoded as a data word as follows: The Σ part of the data word will be of the form

$$(\#q\$\{a,b\}^{h_1}\$\{a,b\}^{h_2})^*$$

where $\Sigma = \{\#, \$\} \cup Q \cup \{a, b\}$. The first counter is represented between the first and second \$ symbols and its value is given by the number of times the symbol a occurs. Similarly, the second counter is represented between the second and third \$ symbols. A run is a sequence of configurations, and it will be encoded as a sequence

of encodings of configurations. Note that the maximal value the counters can have in a run (denoted by h_1, h_2 above) does not change during the run in this encoding.

The `ND2` program `P` checks that the run has the form above, the first configuration has an initial state and the values of counters are zero, the last configuration in the input word has a final state, and that the transition relation δ is respected at every step. We describe how the program checks the last condition.

Let us consider two successive configurations, represented as follows:

$$\ldots \#q\$C_1\$C_2\#q'\$C_1'\$C_2' \ldots$$

Let us describe how the program check that the length of the string C_1 is the same as the length of the string C_1'. The program first checks that every data value that appears in the C_1 part appears there exactly once, and appears exactly once in the C_1' part. Similarly, every data value that appears in the C_1' part should appear there exactly once, and should appear exactly once in the C_1 part.

The program guesses a transition and checks that it matches the two successive configurations. We proceed by case analysis on the form of tuples in the transition relation. Let us suppose that the transition guessed is given by the tuple $(q_1, +, 1, q_2)$. The automaton checks that $q = q_1$, $q' = q_2$, the value of the first counter (represented by C_1) has increased by one and the value of the second counter has not changed. We show how a program can check that the value of the first counter increased by one. The program also checks that every data value d in the C_1 and C_1' parts of the data word appears in pairs (a, d) or (b, d), and not in pairs with other symbols from Σ. Furthermore, the program checks that:

- If a data value d appeared in the C_1 part in a pair (a, d), then it appears in the C_2 part in a pair (a, d).

- Let d be the leftmost data value d that appears in the C_1 part in a pair (b, d). The program checks that it appears in the C_2 part in a pair (a, d).

- For all other data values that appear in the C_1 part in a pair (b, d), the program checks that they appear in a pair (b, d) in the C_1' part.

Thus the program determines that the number of a's has increased by one. Checking that the value of the second counter has not changed can be done similarly. Conformance to other transition tuples can be verified analogically. Therefore we can conclude that for all words w, we have that $w \in L_m(\mathtt{P})$ if and only if w represents an accepting run of the two-counter automaton.

This concludes the proof of Theorem 26. □

We investigate the case of programs obtained by adding access to order on the data domain and adding data or index variables to Restricted-ND2 programs. We show that if we add order on the data domain as well as at least one data variable, the reachability problem becomes undecidable. The proof is by reduction from the Post's Correspondence Problem, and uses the encoding developed in the proofs of Propositions 26 and 27 of [19].

Proposition 27. *Reachability for Restricted-ND2 programs that use order on D and at least one data variable is undecidable.*

A natural question, which is now open, is whether it is possible to add only one of these features (order on data domain or data (index) variables) to Restricted-ND2 programs without losing decidability of the reachability problem.

4.3.5 Expressiveness

In this section, we compare expressiveness of logics and automata on data words and array-accessing programs. We make our comparisons in terms of languages of data words these formalisms can define. Due to a lack of space, we present only the results in this subsection.

Language of a program. In order to define the language of a program, we extend the notion of a program by adding a final state. That is, in this section we

will assume that every program $\mathtt{P}$ has a final state m_f, where m is a boolean state of $\mathtt{P}$. The language $L_m(\mathtt{P})$ is the set of data words w, such that there exist an initial state g_I and a state g, such that $g_I[\mathtt{A}] = w$, $bool(g) = m$, and $(g_I, g) \in [\![\mathtt{P}]\!]$. We say that a program $\mathtt{P}$ *accepts* the language $L_m(\mathtt{P})$, where m is the final state.

The following proposition shows that EDAs and EMSO$^2(\approx, +1, \oplus 1)$ are equally expressive. This means that somewhat surprisingly, DAs and EDAs are expressively equivalent.

Proposition 28. *EDAs and EMSO*$^2(\approx, +1, \oplus 1)$ *are equally expressive.*

Proof. The fact that EDAs are at least as expressive follows from two facts mentioned in Section 4.3.2. First, the logic EMSO$^2(\approx, +1, \oplus 1)$ and data automata are equally expressive, and second, for each DA there exists an EDA that accepts the same language on data words.

To show that EMSO$^2(\approx, +1, \oplus 1)$ is at least as expressive as EDAs, we present a construction that given an EDA $\mathcal{E}$ constructs an EMSO$^2(\approx, +1, \oplus 1)$ formula φ such that for all words $w \in D^*$, $w \models \varphi$ iff $w \in L(\mathcal{E})$.

First, we recall a result of [19] that states that EMSO$^2(\approx, +1, \oplus 1)$ and EMSO$^2(\approx, <, +\omega, \oplus 1)$ are expressively equivalent. It is thus sufficient to construct an EMSO$^2(\approx, <, +\omega, \oplus 1)$ formula. The construction is similar to classical simulation of finite state automata in EMSO$^2(+1)$.

Due to space constraints, we present only the core part of the proof that is different from the classical construction. A formula φ that simulates an accepting run of $\mathcal{E}$ is constructed. It needs to simulate the run of the transducer, as well as the run of a priori unbounded number of copies of the class automaton. We present the simulation of the runs of copies of the class automaton C. Note that we cannot mark (via existentially quantified monadic second order variables) each position in the string with the state of all the copies of C. Instead, monadic second order variables will correspond to single states of C, and each position in a word is marked by exactly one of these state predicates. If the position p is in class X, it will be

marked with a state in which the copy of C corresponding to X is at p. The task of the first order part of φ is then to verify, for each class, that the labeling encodes an accepting run of the class automaton. As part of this task, it needs to verify that a correct number of 0 positions appeared between successive class positions. If P_q and P'_q are labels on successive class positions p and p', then one needs to verify that the class automaton that ran with the position-preserving class string as input and thus saw the 0 symbols will indeed be in the state q' after processing the string of 0s followed by the symbol at position p'. The formula that verifies this condition of course depends closely on the transition relation of the class automaton. We will not present the proof for a general transition relation, but will use an illustrative example. Let us suppose that the class automaton (its 0-transitions) are as depicted in Figure 4.3, and let us suppose that position p is labeled by q_1 and position p' with a Γ symbol a is labeled with a some state s such that there is a transition $\delta_C(q_7, a) = s$. The formula now needs to check that the distance between p and p' is $6 + 5i$, for some i, as this would guarantee that the class automaton transitions to q_4 on the initial string. The part of the formula that checks this property is:

$$\begin{aligned}
&\forall x\ \forall y\ (x \oplus 1 = y \wedge P_q(x) \wedge P_{q'}(y)) \rightarrow \\
&(\bigwedge_{1 \leq k \leq 5} \forall y\ ((x + k = y) \rightarrow (x \not\approx y)))\wedge \\
&C_0(x) \leftrightarrow C_1(y) \wedge C_1(x) \leftrightarrow C_2(y) \wedge C_2(x) \leftrightarrow C_3(y)\wedge \\
&C_3(x) \leftrightarrow C_4(y) \wedge C_4(x) \leftrightarrow C_0(y)
\end{aligned}$$

where C_0, C_1, C_2, C_3, C_4 are existentially quantified monadic second order predicates that are used for counting modulo the length of the cycle (which is 5 in the example). Note that this is an $\mathrm{FO}^2(\approx, <, +\omega, \oplus 1)$ formula. □

The following proposition sheds light on the difference between DAs and EDAs. We saw that DAs and EDAs are expressively equivalent. However, one difference

between EDAs and DAs is that deterministic EDAs are more expressive than deterministic DAs. It is the nondeterminism that then levels the difference.

Proposition 29. *Deterministic EDAs are more expressive than deterministic DAs.*

Proof. Let L be the language defined in Example 12. We showed that there is a deterministic EDA $\mathcal{E}$ such that $L(\mathcal{E}) = L$.

We now show that there is no deterministic DA $\mathcal{A} = (G_A, C_A)$ such that $L(\mathcal{A}) = L$. The proof will be by contradiction. We suppose that there is such a data automaton. As the alphabet Σ is a singleton, the Σ part of the data word is determined by the length of the word in this case. We therefore define data words only by their data part in the rest of this proof. Let k be the number of states of the transducer G_A. We consider a data word $w_1 = (d_1 d_2)^{k+1}$, where d_1, d_2 are values in D. This word is in L. There is therefore an accepting run of G_A. Let us consider the even positions in w_1. Clearly, there are two positions $2i$ and $2j$ such that G_A is in the same state at $2i$ as it is at $2j$. We now consider the words $w_2 = (d_1 d_2)^i (d_1 d_2)^{k+1-j} (d_1 d_4)^{j-i}$ and $w_3 = (d_1 d_2)^i (d_1 d_3)^{j-i} (d_1 d_4)^{k+1-j}$. Note that both w_2 and w_3 are in L and the run of the transducer G_A on both of these words is the same as on w_1, as G_A is deterministic and the Σ parts of w_1, w_2, and w_3 are the same.

Now we look at the word $w_4 = (d_1 d_2)^i (d_1 d_3)^{j-i} (d_1 d_2)^{k+1-j}$ and show that it is accepted by $\mathcal{E}$. Again, the run of the transducer is the same as for w_1. The class automaton for the class corresponding to d_1 reads the same input as was the case for w_1. The class automaton for the class corresponding to d_2 reads the same input as was the case for w_2 (here the fact that the transducer is in the same state at $2i$ and $2j$ is used), and the class automaton for the class corresponding to d_3 gets the same input as was the case for w_3. Therefore in each case, the class automaton C_A accepts its input. Thus we have reached a contradiction, as w_4 is not in L. □

We show that nondeterminism adds to the expressive power of EDAs. We will use the following example from [19].

Example 13. Let $L_{\#}$ be the language of data words defined by the following properties: (1) $str(w) = a^*\$a^*$, (2) the data value of the \$-position occurs exactly once, and each other data value occurs precisely twice — once before and once after the \$ sign, and (3) the order of data values in the first a-block is different from the order of data values in the second a-block.

There exists a nondeterministic EDA for this language (we can use the DA constructed in [19] in Example 8). We will use the example to prove the following proposition.

Proposition 30. *Deterministic EDAs are strictly less expressive than EDAs.*

Proof. It remains to prove that there is no deterministic EDA that accepts the language $L_{\#}$. We adapt the proof from the Example 8 from [19]. The proof is by contradiction. Let us assume that there exists a deterministic EDA $\mathcal{E} = (G, C)$ that accepts $L_{\#}$.

Let n be the number of states G. As for the automaton A, we assume (without loss of generality) that it is deterministic. We consider the graph C_0 and its components C_0^j for $j \in [1..k]$, as defined in the proof in Theorem 21. For each component C_0^j let $loop(C_0^j)$ be the length of the loop in C_0^j. Let K_0 be the product of the lengths of the loops, i.e. $K_0 = \prod_{j:1\leq j\leq k} loop(C_0^j)$. Recall the definition of $D(C_0)$ from the proof of Theorem 21. Let K be the smallest multiple of K_0 greater than $D(C_0)$.

Let w be the word $(a, d_1)(a, d_2)...(a, d_{K(n+2)})(\$, d)(a, d_1)(a, d_2)\dots(a, d_{K(n+2)})$, where $d_1, \dots, d_{K(n+2)}, d$ are distinct. The word w is not in the language, but we show that there is an accepting run of $\mathcal{E}$ on this word. Let us consider the following $n+1$ position in the first part of the word: $K, 2*K, 3*K, \dots, (n+1)*K$. There exist p_1 and p_2, $p_1 < p_2$ between 1 and $n+1$ such that G has the same state after reading the position $p_1 * K$ and $p_2 * K$. Let v be the data word obtained by switching the data values at $p_1 * K$ and $p_2 * K$. We can observe that the run of G on v and w is the same.

The position-preserving class strings for classes other than the classes defined by d_{p_1} and d_{p_2} are the same. Consider the position-preserving class string s_v of the word v for the class defined by the value d_{p_1}. It is of the form $0^{p_2-1}(a, d_{p_1})0^{k(n+2)-p_2}0\ldots$. The corresponding string s_w in w is of the form $0^{p_1-1}(a, d_{p_1})0^{k(n+2)-p_1}0\ldots$. It is easy to see that C does not distinguish between the two strings in the following sense: (a) C has the same state after reading the position p_2 of s_v as it does after reading the position p_1 of s_w, and (b) C has the same state after reading the position $K(n+2)+1$ in both s_v and s_w. (We use (a) to show (b).) We now use the same argument for the class defined by d_{p_2}. We have thus shown that the accepting run of $\mathcal{E}$ on v can be sued to construct the accepting run of $\mathcal{E}$ on w, which creates a contradiction. □

We will now compare the expressive power of array-accessing programs to logics and automata on data words. Specifically, we will use the logic $\text{EMSO}^2(\approx, +1, \oplus 1)$ for comparison. Recall that this logic is expressively equivalent to data automata. First, we will show that Restricted-ND2 programs are not as expressive as $\text{EMSO}^2(\approx, +1, \oplus 1)$.

Proposition 31. *Restricted-*ND2 *programs are strictly less expressive than* $\text{EMSO}^2(\approx, +1, \oplus 1)$.

Proof. For every Restricted-ND2 program P and its boolean state m, we can find an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula φ such that $w \in L_m(\mathtt{P})$ iff $w \models \varphi$. The proof of Theorem 24 gives, for each Restricted-ND2 program P and a boolean state m, an equivalent EDA $\mathcal{E}$. In the proof of Proposition 28, we have constructed an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula equivalent to a given EDA.

We will now show that there is a language of data words that can be specified by an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula φ, but not by a Restricted-ND2 program. We will use Example 13. We have stated that the language $L_\#$ can be captured by a nondeterministic EDA, and thus by an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula. There is no Restricted-ND2 program P that captures $L_\#$. The reason is that the programs, as opposed to transducers in DAs, cannot mark the input array in any way. □

Second, we compare the expressive power of ND1 programs and EMSO$^2(\approx, +1, \oplus 1)$.

Proposition 32. *There exists an EMSO$^2(\approx, +1, \oplus 1)$ property that is not expressible by an* ND1 *program.*

Proof. Let us consider the language L of data words w such that every data value that appears in w appears at least twice. It is easy to construct a (deterministic) Restricted-ND2 program that checks this property. The property can thus be specified in EMSO$^2(\approx, +1, \oplus 1)$.

We now show that this property cannot be specified by an ND1 program. For the sake of contradiction, suppose that there exists an ND1 program P with k index and data variables. Let us consider a word $w = w_1 w_2 \dots w_{2(k+1)}$ of length $2(k+1)$, such that corresponding data values are such that for all $i \leq k+1$, $d_{i+1} > d_i$, and there exists a d'_i such that $d_i < d'_i < d_{i+1}$. The positions greater than $k+1$ are defined by $d_{k+1+i} = d_i$. As w is in L, there is an accepting run of P. Let us consider this run after $k+1$ steps. At this point, there is one value d_j among the first $k+1$ values in w that is not stored in a data variable or pointed to by an index variable. Let us now construct a word w' by replacing the value at $k+1+j$ by d'_j. We can show that P accepts w' with the same run, even though w' is not in L. We have thus reached a contradiction.

□

Note that ND1 programs allow order on the data domain, and thus can check a property specifying that the elements in the input data word are in increasing order. It is easy to see that this property is not specifiable in EMSO$^2(\approx, +1, \oplus 1)$. However, if we syntactically restrict ND1 programs not to use order on D, they can be captured by EMSO$^2(\approx, +1, \oplus 1)$ formulas. The reason is that ND1 programs that do not refer to the order on D can be simulated by register automata introduced in [53]. For every register automaton, there is an equivalent data automaton [17]. Another natural question is whether there is an order-invariant property that can

be captured by ND1 programs (that have access to order), but is not expressible in $\text{EMSO}^2(\approx, +1, \oplus 1)$. We leave this question for future work.

4.4 Confidentiality

Our goal in this section is to verify that a program protects confidentiality of a property f of its inputs. That is, we would like to certify that an observer, who knows the source code of the program, and observes the outputs, cannot infer whether the property f holds or not.

We now instantiate Definition 2, that is, we specify more precisely the property to be kept secret, the condition *cond*, and the equivalence relation that determines the distinguishing power of the observer.

First note that Definition 2 is for general control flow graphs, but in this chapter, we consider a more restricted type of iteration — `for`-loops. Another difference is that here we consider nondeterministic programs. However, note that Definition 2 applies equally well to nondeterministic programs.

- The secret will be defined as a property f of the array. That is, f is a set of arrays, or data words. For example, the property f can ask whether the array contains the data value 7, that is $f \equiv (\exists i.\ [\![\texttt{A[i].d}]\!] = 7)$.

- For ease of presentation, we set the condition *cond* to be *true*. The results below can be extended for other classes of formulas defining the condition *cond* under which confidentiality is required to hold.

- We will assume that the observer can observe the values of all boolean variables of the program at the end of the program. We define the following equivalence relation $\approx$ on states in G. Two states g_1 and g_2 are equivalent ($g_1 \approx g_2$) iff they agree on values of all boolean variables, i.e. if for all boolean variables `b` we have that $g_1[\texttt{b}] = g_2[\texttt{b}]$. We extend the equivalence relation to runs of the

program. Two runs, r and r' are equivalent ($r \approx r'$) iff their last states are equivalent.

Instantiating Definition 2 with the three parameters defined above leads to the following characterization. The property f is *confidential* in P if and only if the following condition holds: if there exists a state g_1 such that $g_1[\mathtt{A}] \in f$ and there exists a state g_1' such that $(g_1, g_1') \in [\![\mathtt{P}]\!]$, then there exists a state g_2 such that $g_2[\mathtt{A}] \notin f$ and a state g_2' such that $(g_2, g_2') \in [\![\mathtt{P}]\!]$ and $g_1' \approx g_2'$.

`ND1` programs Let f be a property specifiable as an `ND1` program `P1`. We assume that `P1` has a boolean variable `accept` and that `P1` ends in a state g_1 such that $g_1[\mathtt{accept}] = \mathit{true}$ if and only if f holds for the input array.

Proposition 33. *Let f be a property specified by an* `ND1` *program, and let* `P` *be an* `ND1` *program. Confidentiality of f in* `P` *is decidable.*

Proof. Let `P1` be a program corresponding to f as above. Let us suppose that the variable names of `P1` and `P` are disjoint, except for the name of the array variable, which is shared. We construct a program `T` as `P1;P`.

We ask two reachability questions for T for each boolean state m. First, we ask if a state g_1 such that $g_1[\mathtt{accept}] = \mathit{true}$ and $\mathit{bool}_P(g_1) = m$ is reachable. Second, we ask if a state g_2 such that $g_2[\mathtt{accept}] = \mathit{false}$ and $\mathit{bool}_P(g_2) = m$ is reachable. (Given a state g of the program T, we denote the boolean state of the program `P` by $\mathit{bool}_P(g)$.) We have that confidentiality is preserved if and only if for all boolean states m, the first reachability test succeeds if and only if the second reachability test succeeds. This gives rise to an algorithm for deciding confidentiality of f in `P`. □

Restricted-`ND2` programs We now consider the case of Restricted-`ND2` programs and a property f specifiable by an $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ formula.

Proposition 34. *Let f be a property specified by an $EMSO^2(\approx, +1, \oplus 1)$ formula, and let* `P` *be a Restricted-*`ND2` *program. Confidentiality of f in* `P` *is decidable.*

Proof. Let φ be an EMSO$^2(\approx, +1, \oplus 1)$ formula characterizing f. From the proof of Theorem 24, we have that for a Restricted-`ND2` program `P` and its boolean state m, there exists an EMSO$^2(\approx, +1, \oplus 1)$ formula ψ that characterizes the set of data words on which m is reachable. More precisely, there exists a formula ψ such that $w \in L_m$ iff $w \models \psi$.

We can now rewrite the definition of confidentiality as follows :

$$\forall w : w \models \psi \rightarrow \exists w' : w' \models \psi \wedge w \models \varphi \leftrightarrow w' \not\models \varphi$$

Furthermore, let formula F_1 be defined as follows:

$$\forall w : w \models \psi \wedge w \models \varphi \rightarrow \exists w' : w' \models \psi \wedge w' \not\models \varphi$$

Formula F_2 is created similarly.

We can now remark that the formula has the form: $F_1 = \forall w : F_{11}(w) \rightarrow \exists w' : F_{12}(w')$, which is equivalent to $F_1 = \exists w : F_{11}(w) \rightarrow \exists w' : F_{12}(w')$. The subformula F_{11} is $\psi \wedge \varphi$, and the subformula F_{12} is $\psi \wedge \neg\varphi$. The validity of the formula F_1 can therefore be decided by deciding satisfiability of two EMSO$^2(\approx, +1, \oplus 1)$ formulas F_{11} and F_{12}. □

Chapter 5

Abstraction-based Program Analysis for Confidentiality

We focus on methods written in a subset of Java that contains booleans, integers, on which we allow linear arithmetic, as well as data from an unbounded domain D equipped with only equality tests. Furthermore, the programs can have arrays, which are *a priori* unbounded in length and whose elements are from D. For example, in the application domain of interest, J2ME midlets, the data domain D models strings (representing names or phone numbers), and the array might contain a phone book or a list of events. Our technique currently does not handle method calls. (In practice, midlet methods call methods from a small set of APIs. The effect of these methods has been hard-coded into the tool. In a future version, we plan to allow specification of these methods using pre-/post-conditions.)

Our method proceeds in two steps. First, we compute a formula φ that is valid if the conditional confidentiality requirement holds. In order to do so, we need to consider both an over- and an under-approximation of reachable states for every program location. We use user-specified invariants for over-approximation. In all the examples we considered, the invariants that were used are simple enough, and could have been discovered by existing techniques for automatic invariant generation [48,

68, 74]. The under-approximation is specified by a bound on the number of loop iterations and a bound on the size of the array.

The second step consists of consists of deciding the validity of the obtained formulas, which involves both universal and existential quantifiers. We leverage the restrictions on the program expressions, as well as the specific form of the obtained formulas, to devise a decision method based on using an existing SMT solver. The restriction on the program expressions used is the fact that the domain D (over which the universal quantification takes place) has only equality tests. Therefore given a formula φ, it is possible to produce an equivalent formula φ' where the universal quantification takes place over a bounded domain. As φ' can then be seen as a boolean combination of existential formulas with no free variables, its validity can be decided using an SMT solver.

5.1 Language of expressions

We consider methods in a subset of Java that contains boolean variables, integer variables, data variables, and array variables. Data variables are variables ranging over an infinite domain D equipped with equality. The domain D models any domain, but we restrict the programs to use only equality tests on data variables. The length of the arrays is unbounded, and their elements come from the domain D.

We instantiate the framework from Section 2.2 to capture this fragment. We will use three types of expressions: integer expressions, data expressions, and boolean expressions.

Integer expressions `IE` are defined by the following grammar:

```
IE ::= s | i | IE OP IE,
```

where `s` is a constant, `i` is a variable, and `OP` is in $+, -$.

Data expressions `DE` are of the form:

```
DE ::= c | v | A[IE],
```

where `c` is a constant, `v` is a data variable and A is an array. Note that there is no arithmetic on data expressions. The only way to access the data domain is through equality tests.

Boolean expressions are defined by the following grammar:

```
B ::=   true | b | B and B | not B
      | IE = IE | IE < IE
      | DE = DE
```

Furthermore, we restrict observations to be finite sequences of data values d, where d is in D. This restriction implies that observables cannot be produced in an unbounded loop.

5.2 Analysis of programs for conditional confidentiality

We consider Definition 2 of conditional confidentiality and we show that one needs to compute both over- and under- approximation. If only one of these techniques is used, it is not possible to get a sound approximation of the confidentiality property. The reason is, at a high level, that the definition involves both universal and existential quantification over the set of executions of the program. More precisely, recall that as explained in Section 2.2, the definition requires that for all feasible observations h, if there exists a execution t_1 for which the condition *cond* holds, then there exists an equivalent execution t_2 for which *secret* holds, and an equivalent execution t_3 for which $\neg$*secret* holds. If we use only over-approximation, that is, a technique that makes the set of executions larger, we might find an execution t_2 or t_3 as required, even though it is not an execution of the original program. Such analysis is thus unsound. If we use under-approximation, some feasible observations might become infeasible. An analysis on the under-approximation would tell us nothing

about such observations. It is not difficult to construct a concrete example where reasoning only about the under-approximation would be unsound.

We thus need to consider over- and under-approximations of sets R_e. Let R_e^+ be an over-approximation of R_e, that is, $R_e \subseteq R_e^+$. Similarly, let R_e^- be an under-approximation R_e^-, that is, $R_e \supseteq R_e^-$.

Using the sets R_e^+ and R_e^-, we can approximate conditional confidentiality as follows:

$$\begin{aligned} &\forall h(\exists s_0 : s_0 \in R_e^+ \wedge s_0 \models \mathit{cond} \wedge s_0[\mathit{hist}] = h) \Rightarrow \\ &\quad (\exists s_1 : s_1 \in R_e^- \wedge s_1 \models \mathit{secret} \wedge s_1[\mathit{hist}] = h \wedge \\ &\quad \exists s_2 : s_2 \in R_e^- \wedge s_2 \not\models \mathit{secret} \wedge s_2[\mathit{hist}] = h) \end{aligned} \tag{5.1}$$

The formula (5.1) soundly approximates conditional confidentiality, as expressed by the following lemma.

Lemma 35. *If the formula (5.1) holds, then secret is confidential w.r.t cond.*

We will now show how, given a program and the predicates *secret* and *cond* specified as logical formulae, we can derive a logical formula expressing the formula (5.1). We will use the following logic.

Logic $\mathcal{L}$. The formulas of $\mathcal{L}$ will use boolean, integer, data and array variables (similarly to the expressions defined in Section 2.2). The definition of integer and data expressions will be the same as well. The grammar defining the boolean formulas is:

```
BL ::=   true | b | BL | BL | not BL
       | IE = IE | IE < IE | DE = DE
       | ∃ b: BL | ∃ i: BL | ∃ v: BL
```

The difference between the formulas in $\mathcal{L}$ and the boolean expressions in the programs we consider is that in $\mathcal{L}$ we allow quantification in the logic.

Weakest precondition. We will need the notion of the weakest precondition computation (see e.g. [82]). Given a program P and a formula φ, $WP(\mathtt{P}, \varphi)$ is the weakest formula that guarantees that if $\mathtt{P}$ terminates, it terminates in a state in which φ holds.

We will need the weakest-precondition computation for CFGs that are acyclic. For an acyclic CFG G and a formula φ, we define the weakest precondition $WP(\mathtt{P}, \varphi)$ as follows. Given a node k of G, let $Entry(k)$ and $Exit(k)$ be two formulas associated with k. Let us consider the equations:

$$Exit(k) = \bigwedge_{l \in Succ(k)} Entry(l)$$
$$Entry(k) = WP(L(k), Exit(k))$$

where $L(k)$ is the program with which the node k is labeled and where $WP(C, \varphi$ is defined as follows for a command C (as defined in Section 2.2) and a formula φ.

$$WP(\mathtt{v} = \mathtt{E}, \varphi) = \varphi[\mathtt{E}/\mathtt{x}]$$
$$WP(\mathtt{assume B}, \varphi) = (\mathtt{B} \rightarrow \varphi)$$

We can now calculate the weakest precondition of φ w.r.t. G by setting $Exit(e)$ to φ and using the above equations to calculate $Entry(i)$, where i is the entry node of the CFG and e is the exit node of the CFG.

The main property we require for the logic $\mathcal{L}$ is that it should be closed under the weakest precondition of *loop-free* programs, that is, for any $\mathcal{L}$-formula φ and any loop-free program $\mathtt{P}$, $WP(\mathtt{P}, \varphi)$ is in $\mathcal{L}$. Given the restrictions on expressions in the language, it is easy to show that this requirement holds.

Over-approximation R_e^+. Let us consider the antecedent of the formula (5.1), i.e. $(\exists s_0 : s_0 \in R_e^+ \wedge s_0 \models cond \wedge s_0[hist] = h)$. We need to obtain an $\mathcal{L}$ formula characterizing this requirement, given that *cond* is an $\mathcal{L}$ formula. Given an $\mathcal{L}$ formula

ψ that characterizes R_e^+, we obtain the desired characterization as $\varphi^+ \equiv \exists pv : \psi \wedge cond \wedge hist = h$. Note that the free variables in ψ and *cond* are the program variables, and the notation $\exists pv : F$ (for a formula F) is a shorthand for saying that all program variables are existentially quantified.

The formula ψ that characterizes the set of reachable states at a program location can be either provided by the user or computed by standard methods of abstract interpretation [30], using a standard abstract domain (e.g. octagons [61], polyhedra [31]) Recently, such techniques have been extended for discovering disjunctive invariants (see [48, 68, 74]). These latter techniques would be needed to discover the invariants needed for the examples we present in Section 6.2.

Under-approximation R_e^-. The under-approximation is obtained by unrolling the loops in the CFG G. More precisely, all loops are unrolled a fixed number of times (k) and the CFG is thus transformed to a loop-free CFG G'.

Recall from Section 2.2 that we consider only reducible CFGs. A reducible CFG is one where it is possible to identify a unique loop header.

A node l_1 *dominates* a node l_2 when all the paths from i (the entry node) to l_2 pass through l_1. An edge in the CFG is a *back edge* when its head (target of the edge) dominates its tail (source of the edge). A loop is uniquely identified by its header and a back edge.

Given a loop, let h be its header and let b be the origin of the back edge. Note that h is the only entry point to the loop. The k unrolling of a loop is obtained removing the back edge and by copying the nodes in the loop k times (and keeping the edges outgoing from the loop at every copy). Thus all executions with up to k iterations are kept by the new CFG.

Let G' be the CFG obtained by this transformation. Let R'_e be the set of reachable states obtained for the exit node of G'. It is straightforward to prove that $R'_e \subseteq R_e$.

We are interested in characterizing the requirement (from the consequent of formula (5.1)): $\exists s_1 : s_1 \in R_e^- \wedge s_1 \models \mathit{secret} \wedge s_1[\mathit{hist}] = h$ by a $\mathcal{L}$-formula φ_1^-. It is computed using the weakest precondition computation on the CFG G' as follows: $\varphi_1^- \equiv \exists \mathit{pv} : \mathit{WP}(G', \mathit{hist} = h \wedge \mathit{secret})$. Similarly, φ_2^- is defined as $\exists \mathit{pv} : \mathit{WP}(G', \mathit{hist} = h \wedge \neg \mathit{secret})$.

Computing confidentiality We can now check if confidentiality holds using the following formula:

$$\begin{aligned} \forall h :& (\exists \mathit{pv} : \psi \wedge \mathit{cond} \wedge \mathit{hist} = h) \Rightarrow \\ & (\exists \mathit{pv} : \mathit{WP}(G', \mathit{hist} = h \wedge \mathit{secret}) \wedge \\ & \exists \mathit{pv} : \mathit{WP}(G', \mathit{hist} = h \wedge \neg \mathit{secret})) \end{aligned} \tag{5.2}$$

As formulas (2) and (3) are equivalent, we can use Lemma 35 to prove the following:

Lemma 36. *If the formula (5.2) holds, then secret is confidential w.r.t cond.*

5.3 Deciding validity of the confidentiality formula

In this section, we describe a method for deciding the confidentiality formula (5.2). The method is based on satisfiability modulo theories (SMT) solving.

Restrictions on *cond* and *secret* First we identify some restrictions on the predicates *cond* and *secret*. The restriction on *cond* is that we will consider only existential formulas. The predicate *secret* appears in the formula (5.2) also under negation, therefore we restrict it not to use quantification. Note that for some examples, the property *secret* contains a quantification on the array indices. This is the case for the `ArraySearch` example discussed in Section 3. In such cases, the under-approximation uses also a bound on the size of the array, thus making the quantification to be effectively over a bounded set.

The $\mathcal{L}$-formula (5.2) has one quantifier alternation (taking into account the restrictions above). Here we show how such a formula can be decided using an SMT solver. In order to simplify the presentation, in this section we will suppose the observation h in the confidentiality formula consists of only one data value d (and not of a sequence of values from D). The results in this section, as well as their proofs, can be easily extended to the general case.

Let us first suppose that we have an existential formula $\varrho(h)$ (in the logic $\mathcal{L}$) with one free data variable h. Let D be an infinite set, let C be the finite set of values interpreting in D the constants that appear in $\varrho(h)$. For an element d of D, we write $d \models \varrho$ if ϱ holds when h is interpreted as d.

We show that the formula ϱ cannot distinguish between two values d and d', if d and d' are not in C.

Lemma 37. *For all $d, d' \in D$, if $d \notin C$ and $d' \notin C$, then $d \models \varrho \leftrightarrow d' \models \varrho$.*

Intuitively, the lemma holds, because the formula ϱ can only compare the value of h to constants in C or to other existentially quantified data variables. The proof proceeds by structural induction on the formula ϱ.

Lemma 37 suggests a method for deciding whether $\forall h : \varrho$ holds: First, check whether $\varrho(c)$ holds for all constants in C, and second, check whether $\varrho(c)$ and for one value not in C. Note that as C is finite and D infinite, there must exist an element of D not in C.

The following lemma shows that this method can be extended to the confidentiality formula (3). Let C' be $C \cup \{d\}$, where d is in D, but not in C.

Lemma 38. *Let ψ be a formula: $\forall h : \varphi_0(h) \rightarrow (\varphi_1(h) \wedge \varphi_2(h))$, where φ_0, φ_1, φ_2 are existential formulas with one free data variable h. Then ψ is equivalent to $\bigwedge_{c \in C'} \psi_c$, where ψ_c is $\varphi_0(c) \rightarrow (\varphi_1(c) \wedge \varphi_2(c))$.*

The proof uses Lemma 37 for all of φ_0, φ_1 and φ_2.

Let us now consider the resulting formula $\bigwedge_{c \in C'} \psi_c$. Each ψ_c has the form $\varphi_0(c) \rightarrow (\varphi_1(c) \wedge \varphi_2(c))$, where $\varphi_0(c)$, $\varphi_1(c)$, and $\varphi_2(c)$ are existential formulas without free variables. Therefore we can check satisfiability of each of these formulas separately, and then combine the results appropriately (i.e. if $\varphi_0(c)$ is satisfiable, then both $\varphi_1(c)$ and $\varphi_2(c)$ have to be satisfiable).

We have thus leveraged the fact that the only operation on the data domain is equality to devise a decision method based on SMT checking for the confidentiality formula (5.1).

Example 14. Let us consider the `ArraySearch` example presented in Section 2.2. Recall that we considered the predicate *secret* to be $\exists i : A[i] = 7$ and the condition *cond* to be $\texttt{key} \neq 7$. Recall also that the observer might either see an empty observation, or a observation containing a single number, the final value of `result`.

For the over-approximation, we will need an invariant asserting that (`result = key` or $\texttt{result} = -1$). The formula φ^+ will thus be: $\varphi^+ \equiv ((\texttt{result} = -1) \vee (\texttt{result} = \texttt{key})) \wedge (\texttt{key} \neq 7) \wedge \texttt{result} = h$.

The under-approximation will be specified by a number of unrollings and the size of the array. We choose 2 in both cases. We then compute φ_1^- using the weakest precondition computation $\exists s_1 : WP(P, hist = h \wedge \exists i : \texttt{A}[\texttt{i}] = 7)$ and φ_2^- as $\exists s_2 : WP(P, hist = h \wedge \neg \exists i : \texttt{A}[\texttt{i}] = 7)$.

The formula characterizing confidentiality becomes:

$$\forall h : (\exists s : ((\texttt{result} = -1) \vee (\texttt{result} = \texttt{key})) \wedge (\texttt{result} = h) \wedge (\texttt{key} \neq 7)) \Rightarrow (\varphi_1^- \wedge \varphi_2^-)$$

The formulas contains two constants from the data domain -1 (appeared in the program) and the value 7 (appeared in *cond* and *secret*). We also need to consider one value that is different from these constants. We can pick for example the value 1. For -1 (1) the formula says that if the observer sees the value, he or she cannot infer whether 7 is in the array and are easily proven. For 7, the antecedent of the formula

is false (as the purpose of the condition was to exclude 7 from consideration), thus the formula is proven.

Chapter 6

Implementation and Experimental Evaluation

6.1 Implementation

We have performed experiments in order to confirm that the proposed method is feasible in the sense that the formulas produced can be decided by existing tools in reasonable time. The experiments were performed on methods of J2ME classes and classes from the core Java library on a computer with a 2.8Ghz processor and 2GB of RAM.

We have implemented a prototype tool called ConAn (for CONfidentiality ANalysis). It takes as input a program in Java bytecode, a secret, a condition, and parameters specifying the over- and under- approximation to be used.

The complete toolchain is shown in Figure 6.1. The WALA [2] library is used to process the bytecode. The ConAn tool then performs the analysis on an intermediate representation called WALA IR and produces formulas whose satisfiability is then checked by an SMT (satisfiability modulo theories) solver Yices [37].

The WALA IR represents a method's instructions in a language close to JVM bytecode, but in an SSA-based language which eliminates the stack abstraction. The

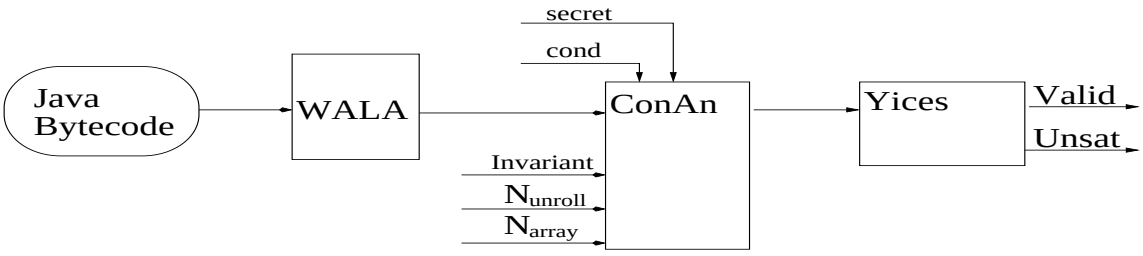

Figure 6.1: Toolchain

IR organizes instructions in a control flow graph of basic blocks. The tool analyzes a fragment of the IR subject to the same restrictions on expressions as described in Section 5.1. Furthermore, the treatment of method calls and dynamic memory allocation is incomplete. The methods call methods from a small set of APIs such as the PIM API mentioned in Section 1.1. The effect of these methods has been hard-coded into the tool. In a future version, we plan to allow specification of these methods using pre-/post-conditions. Furthermore, the programs we examined use iterators (with operations such as `hasNext` and `Next` to iterate over data structures). The effect of these methods was also hard-coded using iteration over arrays. Dynamic memory allocation is represented with the `new` instruction in the WALA IR. As the weakest precondition computation is applied to loop-free CFGs, we treat each allocation as a separate variable. The current version of the tool also assumes there is no aliasing. In the future versions, we plan to use the results of the alias analysis implemented in WALA.

As shown in Figure 6.1, the ConAn tool takes as input a specification for the over-approximation (in the form of the invariant) and the specification of the under-approximation (in the form of the number of loop unrollings to consider and a bound on the size of the array). The tool Yices [37] is used for deciding satisfiability of the resulting formulas.

6.2 Experiments

We briefly describe the examples we considered and report on the performance of the tool. Table 6.1 contains, for each example, the number of lines of code, the running time of the tool, and the result, i.e. whether the formula was satisfiable (and confidentiality preserved) or unsatisfiable (i.e. no conclusion possible). Note that the running times presented in the table do not include the running time of the translation from bytecode to the WALA IR format. It includes only the running time of the analysis in ConAn, and the time taken by the Yices tool to decide the satisfiability of the formulas.

In all cases the secret is a fact about the array. We used the predicate $\exists i : A[i] = 7$ as the secret. The condition *cond* is specified for each example separately. The over-approximation was specified via an invariant, and the under-approximation was specified via the number of loop unrollings (as shown in Table 6.1) and the bound on the size of the array (chosen to be 2 in all of the examples). The observation visible to the observer is defined by either the message(s) the program send out, or the values the functions return. The latter is useful for modular verification of programs that access a data structure via a call to the analyzed functions and subsequently send messages depending on the returned value.

Example 1 is from the class Vector, whose method elementAt returns an element at the specified position of the array. Examples 2 and 3 are from a J2ME example called EventSharingMidlet. This is the example described in Section 1.1. We considered both the correct version and a version with an artificially introduced bug as in Example 1.2. This example is taken from [1].

Examples 4 and 5 are versions of the `ArraySearch` example from Section 2.2. For Example 4, we used only one unrolling of the loop. The tool did not prove that the secret is not leaked. Increasing the number of unrollings to two (Example 5) helped; the confidentiality was proved in this case.

	project / class	Method Name	# of lines in Java	unroll	running time (s)	result
1	Vector	elementAt	6	1	0.18	valid
2	EventSharing	SendEvent	122	2	1.83	valid
3	EventSharing	SendEvent (bug)	126	2	1.80	unsat
4		find	9	1	0.31	unsat
5		find	9	2	0.34	valid
6	Funambol/Contact	getContact	13	2	0.32	valid
7	Blackchat/ ICQContact	getContact-ByReference	23	2	0.24	valid
8	password	check	9	2	0.22	valid

Table 6.1: Experimental evaluation

Example 6 from the class Contact found in the Funambol library scans the phonebook obtained via a call to PIM API to find an element corresponding to a key. Example 7 is similar to Example 6.

Example 8 is a version of the classical password checking example - an array is scanned and if the name/password pair matches, the function returns 1. The results show that no password is leaked. Example 8 is taken from [51].

Discussion. All Java methods we considered are small in size. For these programs, the running times were all lower than two seconds. The experiments succeeded in showing that our approach is feasible for relatively short Java methods. We argue that this shows that our methods is suitable for the intended application, certification of J2ME midlets. Firstly, J2ME midlets are rather small in size. We surveyed 20 of the most popular[1] midlet applications. We used the tool LOCC[2] to calculate for each of this midlets the average number N_a as well as maximal number N_m of lines of code per method. Over all of these programs, the average of N_a numbers was 15, the maximum of the N_a numbers was 25. The average of the N_m numbers was 206, the maximum of the N_m numbers was 857. These data confirm

[1]The criterion was the number of downloads from sourceforge.net
[2]http://csdl.ics.hawaii.edu/Tools/LOCC/

that the size of methods in J2ME midlets is small, and our methods are directly applicable to average-sized method. Secondly, for each midlet we reported on in Table 6.1 we analyzed the methods that are key from the point of view of preserving secrecy, i.e. the methods that access the data structure for which the secret should hold, or methods that send messages. Therefore we believe that a pre-processing phase using program slicing followed by our techniques would enable our tool to analyze most of the methods of midlets.

Chapter 7

Related Work

Defining confidentiality

Language-based security Noninterference is a security property often used to ensure confidentiality. Informally, it can be described as follows: "if two input states share the same values of low variables then the behaviors of the program executed from these states are indistinguishable by the observer". See [70] for a survey of the research on noninterference and [63] for a Java-based programming language with a type systems that supports information flow control based on noninterference.

The definition of confidentiality we presented can be seen as a relaxation of noninterference. It is relaxed by allowing the user to specify which property(-ies) should stay secret; noninterference requires that *all* properties of high variables stay secret. Furthermore, we show how noninterference can be captured in our framework — this is formally shown by Lemma 1.

It is well-known that the noninterference requirement needs to be relaxed in various contexts. See [72] for a survey of methods for defining such relaxations via declassification. In this context, the main benefit of our approach is automation, as our method allows verification of existing programs without requiring annotations by the programmer.

To illustrate the difference between our approach (conditional confidentiality) and declassification in language-based security, we turn again to the EventSharingMidlet (Figure 1.1). As before, we will consider the property to be kept secret for the example is whether a particular number, say "555-55" is in the phone book. Let us now assume that we want to capture this requirement by noninterference. We consider the variable `phoneBook` as high (i.e. secret), and the variable `message` as low (i.e. public). Noninterference is too strong for the specification of confidentiality for the example in Figure 1.1. The reason is, briefly, that the variable `message` depends on the variable `phoneBook` via control flow. Therefore this program would be rejected. On the other hand, we have seen in Section 1.1 that the requirement can be specified as conditional confidentiality, and that it holds for the program in Figure 1.1.

Let us now show how to relax the noninterference requirement via declassification. We will use the notion of *delimited release* [71]. This policy requires that information is leaked only through *escape hatch* expressions. More precisely, a program satisfies delimited release if it has the following property: for any initial memory state s and any state t obtained by varying the secret part of s, if the value of escape hatch expressions is the same in both s and t, then the publicly observable effect of running the program in state s and t will be the same. The escape hatch expressions are marked by the keyword `declassify` in the code. Returning to the example, we can specify the permissible information release by enclosing the expression in the if conditional in a call to `declassify`. Concretely, in the example in Figure 1.1, the if test would now be: (`if declassify((number == null) || (number == ""))` ...). The delimited release property now holds for the program, thus the program can be deemed secure. Note however again that this approach relies on the annotations by the programmer. It is thus suitable for making programs that are secure by construction, but it is unsuitable for checking existing, possibly malicious, programs.

Probabilistic notions of confidentiality We have presented a possibilistic definition of confidentiality. Probabilistic definitions have been examined in the literature (see e.g. [46, 81]). We chose a possibilistic one for two reasons: first, a probabilistic definition could not be applied without making (artificial) assumptions about the probability distribution on inputs, and second, common midlets do not use randomization (so a security measure might be to reject programs that use randomization). However, there are settings where a probabilistic definition would be appropriate, and the question on how to extend the analysis method to a probabilistic definition is left for future work.

Opacity The definition of confidentiality we use is related to opacity [22]. In [22], several variants of opacity are considered. The possibility of specifying the conditions under which confidentiality should be preserved is limited. In particular, the conditional confidentiality (with the secret specified by a property f and the condition specified by a property g), is not specifiable in the framework of [22]. If we set g to be true, then the confidentiality notion considered in this thesis corresponds exactly to the property f being final-opaque under a static observation function, in the terminology of [22].

Cryptographic protocols Preservation of confidentiality is an important part of correctness of cryptographic protocols. Cortier et al. [29] compare two styles of secrecy definitions: the first based on reachability (which implies that a secret term should never be directly exposed), the second based on observational equivalence of traces. The latter is conceptually close to the definition presented in this thesis, but the details are significantly different — the definition is given for a process calculus as a congruence on terms. The tool ProVerif [18] implements a sound (but not complete) algorithm for checking equivalence-based confidentiality.

Specification frameworks

In some aspects, tree logics with path equivalences are related to logics of knowledge [39]. The main semantic difference is that logics of knowledge are concerned about what an agent knows, whereas in the temporal logics presented in this thesis we are concerned about what an agent has revealed. From an intuitive point of view, it might be possible to capture what an agent a reveals by adding one "observer agent", who would observe a and record its observations (e.g. outputs and inputs of a) and then ask about the knowledge of this observer agent. However, in a finite state setting under the standard semantics for knowledge operators (the semantics is defined in terms of equivalence relations on states of the Kripke structure, not the paths), this is not possible.

The idea of introducing an observer agent would work in the case of *perfect recall* semantics [39], i.e. when an agent remembers the sequence of its past states. In this case, our equivalence operator $EI_a\varphi$ can be translated as $\neg K_{Obs_a}\neg\varphi$, where Obs_a is the agent introduced to record the observable actions of a. Note however, that the nonequivalence operator $EI_{\bar{a}}\varphi$ cannot be expressed in logic of knowledge with perfect recall, because this logic can express properties that some or all equivalent nodes have and there is no way to refer to nonequivalent nodes. In the setting of perfect recall semantics, van der Meyden and Shilov [79] have considered model checking of LTL with knowledge operators and Shilov and Garanina [76] consider model checking of CTL and μ-calculus with knowledge operators. Note also that the construction of our finite-state model is similar to "k-trees" used in these papers. However, note that the notions of nesting depths are different, and that our notion yields better complexity bounds. We argue that our logics are more suitable for specifying secrecy and information flow properties than logics of knowledge. First, we showed that it is possible to specify information flow properties using standard tree logics (CTL, μ-calculus), provided that we enrich the tree model with path equivalences. This approach can be readily extended to other tree logics, such as ATL [9]. Second,

we are also able to model information flow properties directly, without the need to introduce an observer agent for each agent in the original system. Third, some information flow properties can be expressed naturally using the $EI_{\bar{a}}$ operator. This is not possible in logic of knowledge. For μ-calculus, we have identified a decidable fragment (EXPTIME-complete), in which it is possible to specify partial-information adaptive games.

For simplicity, we presented our approach using Kripke structures as a basic model. However, there are other models, such as alternating transition systems (see [9]), which are better suited for modeling games. We believe our results can be easily lifted to ATSs. Note that partial information games have also been studied in the context of ATL, but were proven undecidable for multiple players.

Confidentiality analysis for programs

Software model checking Traditional software verification is not directly applicable to checking confidentiality. The reason is that conditional confidentiality cannot be expressed in branching-time temporal logics, such as μ-calculus. Furthermore, abstractions based solely on over-approximations or solely on under-approximations are not sufficient for checking conditional confidentiality. Frameworks for three-valued abstractions of modal transition systems ([34],[42]) combine over- and under-approximations, but the logics studied in this context (μ-calculus or less expressive logics) cannot express the conditional confidentiality requirement.

Program analysis Program analysis for (variants of) noninterference has been examined in literature. The approaches that have been considered include slicing [77] or using a logic for information flow [10]. These methods conservatively approximate noninterference, and thus would not certify valid midlets. It is possible to relax these requirements by using e.g. escape-hatch expressions [10]. It would be interesting to see if these ideas can be used to develop a specification-driven automated method

for checking confidentiality.

For finite state systems, noninterference has been shown decidable in [80]. Decidability of some of the variants of noninterference for WHILE-programs is shown in [32].

Dam and Giambagi [33] introduce a notion of *admissible* information flow, allowing a finer grained control. Admissible information flow is a relaxation of noninterference, where the programmer can specify which specific data dependencies are allowed. The information required from the programmer are quite complex however (a set of relabellings) and it is not straightforward to see how this method can be automated.

Other approaches for checking confidentiality are also based on noninterference variants. A weaker notion, when an observer is allowed to see only the observable part of final states lends itself to checking a safety property on self-composition, a composition of a program with itself. This approach is explored in [14, 78].

Malacaria [59] provides information-theoretic semantics of programs, and develops a method for quantifying information leakage in a program. However, the approach is not automated, and it is unclear whether it can be automated.

Backes et al [11] provide an automated method to calculate the how much observation reduces the uncertainty the attacker has about inputs. It would be interesting to see if the method can be used to determine how the uncertainty of the attacker about a property of the inputs decreases.

Analyzing array-accessing programs Our results establish connections between verification of programs accessing arrays and logics and automata on data words. Kaminski and Francez [53] initiated the study of finite-memory automata on infinite alphabets. They introduced register automata, that is automata that in addition to finite state have a fixed number of registers that can store data values. The results of Kaminski and Francez were recently extended in [64, 19, 17, 16]. Data

automata introduced in this line of research were shown to be more expressive than register automata. Furthermore, the logic $\text{EMSO}^2(\approx, +1, \oplus 1)$ was introduced, and Bojanczyk et al. ([19]) show that $\text{EMSO}^2(\approx, +1, \oplus 1)$ and data automata are equally expressive. The reduction from $\text{EMSO}^2(\approx, +1, \oplus 1)$ to data automata and the fact that emptiness is decidable for data automata imply that satisfiability is decidable for $\text{EMSO}^2(\approx, +1, \oplus 1)$. We show that Restricted-`ND2` programs can be encoded in $\text{EMSO}^2(\approx, +1, \oplus 1)$.

However, adding a third variable to the logic or allowing access to order on data variable makes satisfiability undecidable for the resulting logic, even for the first order fragment. We show, perhaps somewhat surprisingly, that the undecidability does not translate into undecidability of reachability for `ND1` programs that access order on the data domain and have an arbitrary number of index and data variables. The results on automata and logics on data words model were applied in the context of XML reasoning [64] and extended temporal logics [35]. The connection to verification of programs with unbounded data structures is the first to the best of our knowledge.

Deutsch et al. [36] consider a model of database-driven systems similar in some aspects to our model of programs. The key difference is that they consider a dense order. They specifically note that the model-checking problem they consider is open for the case of a discrete order. It would be interesting to see if our result on programs on structures with discrete order can be extended to the setting of database-driven systems.

Fragments of first order logic on arrays have been shown decidable in [21, 50, 12, 20]. These fragments do not restrict the number of variables (as was the case with $\text{EMSO}^2(\approx, +1, \oplus 1)$), but restrict the number of quantifier alternations. These papers focus on theory of arrays, rather than on analysis of array-accessing programs. In particular, reachability in programs (that contain loops) is not reducible to these first-order fragments.

Static analysis of programs that access arrays is an active research area, with

recent results including [44, 49, 12]. The approach consists in finding inductive invariants for loops using abstraction methods, such as abstract domains that can represent universally quantified facts [49] and a predicate abstraction approach to shape analysis [12]. In contrast, our results yield decision procedures for array-accessing programs, with an interesting feature in the context of previous work being that our method does not need to discover the loop invariants explicitly. However, the methods based on abstraction are applicable to a richer class of programs.

Chapter 8

Conclusion

In this thesis, we show how it is possible to extend software model-checking to confidentiality properties. Software model checking is a specification-driven approach to program analysis. The central question here is whether (an abstraction of) a program satisfies a property specified by the user. Automated software model checkers have become efficient and widely deployed in industry for both verification and bug finding.

We start by observing that confidentiality is not specifiable in standard temporal logics. We extend the specification language traditionally used, namely the temporal logics, in order to be able to express confidentiality properties. We then develop a model checking algorithm for systems with finite number of states. Turning to software systems, we present a decision procedure for confidentiality for classes of array-accessing programs. The programs in these classes have boolean variables as well as variables from an infinite domain D, and can access a single array, which is of unbounded size. The domain of the elements of the array is D. The classes include programs that find a particular value in the array, programs that find a minimal value, checking that all values in the array are within specific bounds, or checking for duplicate data values. The decision procedures for confidentiality is based on algorithms for reachability. We believe that the algorithms for reachability are of

independent interest, both theoretically due to its connections with logics on data words and practically, due to a potential use as a back-end decision procedure in software model checkers. Finally, we develop abstraction-based program analysis methods for confidentiality for a more general class of programs.

There are many possible directions for future work. First, we believe it would be interesting to investigate the extensions of our decidability results to classes of programs that access a single array. These extensions include (1) programs accessing data structures other than the array, (2) programs that modify the data structure, (3) programs accessing more than one data structure, and finally, (4) programs with procedures. The extension to linked lists seems straightforward. We will study the extension to data structures with more successors, such as trees. The proof of decidability of the reachability problem for programs with non-nested loops can be extended to programs that modify the contents of the array. However, the proofs of the other two main decidability results cannot be extended in a straightforward way, and the question of decidability remains open. Second, one possible approach to improving practical performance of our solution would be to combine it with a coarser-grained approach such as taint-analysis style dataflow analysis. Third, we have shown that both over- and under- approximation are necessary for sound analysis of confidentiality requirements, therefore an interesting question for future research is how to develop a counter-example guided abstraction refinement for checking confidentiality. Fourth, one of the most important current challenges in the field of software verification is the area of concurrent and parallel programs. Devising scalable methods for checking confidentiality for concurrent programs is a challenging goal with many potential applications.

Bibliography

[1] JavaTM ME Developer's Library 2.0. Available on: http://www.forum.nokia.com.

[2] WALA - Watson libraries for analyses. Available on: http://wala.sourceforge.net.

[3] Health insurance portability and accountability act of 1996. Public Law 104-191, 1996.

[4] JSR 118 Expert Group. Mobile Information Device Profile for J2ME 2.1, 2007.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[6] R. Alur, P. Černý, and S. Chaudhuri. Model checking on trees with path equivalences. In *Proc. of TACAS '07*, pages 664–678, 2007.

[7] R. Alur, P. Černý, and S. Weinstein. Algorithmic analysis of array-accessing programs. In *CSL (to appear)*, 2009.

[8] R. Alur, P. Černý, and S. Zdancewic. Preserving secrecy under refinement. In *Proc. of ICALP '06*, pages 107–118, 2006.

[9] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.

[10] T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *Proc. of FMSE'07*, pages 2–11, 2007.

[11] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, 2009.

[12] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Proc. of VMCAI'05*, pages 164–180, 2005.

[13] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL'02*, pages 1–3, 2002.

[14] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In *Proc. of CSFW'04*, pages 100–114, 2004.

[15] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Proc. of SP'06*, 2006.

[16] H. Björklund and M. Bojanczyk. Shuffle expressions and words with nested data. In *Proc. of MFCS'07*, pages 750–761, 2007.

[17] H. Björklund and T. Schwentick. On notions of regularity for data languages. In *Proc. of FCT'07*, pages 88–99, 2007.

[18] Bruno Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

[19] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proc. of LICS'06*, pages 7–16, 2006.

[20] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *Proc. of FCT'07*, pages 1–22, 2007.

[21] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *Proc. of VMCAI'06*, pages 427–442, 2006.

[22] J. Bryans, M. Koutny, L. Mazaré, and P. Ryan. Opacity generalised to transition systems. *Int. J. Inf. Sec.*, 7(6):421–435, 2008.

[23] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[24] Pavol Černý and Rajeev Alur. Automated analysis of java methods for confidentiality. In *CAV*, pages 173–187. Springer, 2009.

[25] S. Chong and A. Myers. Decentralized robustness. In *Proc. of CSFW'02*, pages 242–256, 2006.

[26] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of CAV'02*, pages 359–364, 2002.

[27] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71, 1981.

[28] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on http://www.grappa.univ-lille3.fr/tata, 1997.

[29] V. Cortier, M. Rusinowitch, and E. Zălinescu. Relating two standard notions of secrecy. *Logical Methods in Computer Science*, 3(3), 2007.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252, Los Angeles, California, 1977.

[31] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL'78*, pages 84–97, 1978.

[32] M. Dam. Decidability and proof systems for language-based noninterference relations. In *POPL'06*, pages 67–78, 2006.

[33] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. of CSFW'00*, pages 233–244, 2000.

[34] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[35] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. In *Proc. of LICS '06*, pages 17–26, 2006.

[36] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.

[37] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV'06*, pages 81–94, 2006.

[38] E. Emerson and C Jutla. Tree automata, mu-calculus and determinacy. In *FOCS*, pages 368–377, 1991.

[39] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[40] J. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Commun. ACM*, 24(9):597–605, 1981.

[41] N. Globerman and D. Harel. Complexity results for two way and multipebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.

[42] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. of CONCUR'01*, pages 426–440, 2001.

[43] S. Goodman and H. Lin, editors. *Toward a Safer and More Secure Cyberspace*. National Academy Press, 2007.

[44] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL '05*, pages 338–350, 2008.

[45] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proc. of CAV'07*, pages 72–83, 1997.

[46] J. Gray. Probabilistic interference. In *Proc. of SP'90*, pages 170–179, 1990.

[47] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. Synergy: a new algorithm for property checking. In *FSE'06*, pages 117–127, 2006.

[48] B. Gulavani and S. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proc. of TACAS'06*, pages 474–488, 2006.

[49] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proc. of POPL '08*, pages 235–246, 2008.

[50] P. Habermehl, R. Iosif, and T. Vojnař. What else is decidable about integer arrays? In *Proc. of FoSSaCS'08*, pages 474–489, 2008.

[51] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *Proc. of ISoLA'06*, pages 136–145, 2006.

[52] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of CAV'02*, pages 526–538, 2002.

[53] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[54] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[55] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[56] J. Lambek. How to program an infinite abacus. *Canadian Mathematical Bulletin*, 4:295–302, 1961.

[57] H. Lauchli and Ch. Savioz. Monadic second order definable relations on the binary tree. *J. Symb. Log.*, 52(1):219–226, 1987.

[58] R. Lipton. The reachability problem requires exponential space. Technical Report Dept. of Computer Science, Research report 62, Yale University, 1976.

[59] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL*, pages 225–235, 2007.

[60] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of SP'94*, pages 79–93, 1994.

[61] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[62] M. Minski. Recursive unsolvability of Post's problem of 'tag' and other topics in theory of Turing machines. *Annals of Mathematics*, 74:437–455, 1962.

[63] A. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of POPL'99*, pages 228–241, 1999.

[64] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.

[65] J. O'Connor. Attack surface analysis of Blackberry devices. White Paper: Symantec security response, 2007.

[66] Ch. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing, Reading, MA, USA, 1994.

[67] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[68] C. Popeea and W. Chin. Inferring disjunctive postconditions. In *Proc. of ASIAN'06*, 2006.

[69] J. Reif. Universal games of incomplete information. In *Proc. of STOC '79*, pages 288–308, 1979.

[70] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[71] A. Sabelfeld and A. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.

[72] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05*, pages 255–269, 2005.

[73] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[74] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Proc. of SAS'06*, pages 3–17, 2006.

[75] F. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.

[76] N. Shilov and N. Garanina. Model checking knowledge and fixpoints. In *Proc. of FICS'02*, pages 25–39, 2002.

[77] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[78] T. Terauchi and A. Aiken. Secure Information Flow as a Safety Problem. In *Proc. of SAS 2005*, pages 352–367, 2005.

[79] R. van der Meyden and N. Shilov. Model checking knowledge and time in systems with perfect recall. In *Proc. of FSTTCS'99*, pages 432–445, 1999.

[80] R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electr. Notes Theor. Comput. Sci.*, 168:61–75, 2007.

[81] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.

[82] G. Winskel. *The formal semantics of programming languages: An Introduction.* MIT Press, 1993.

[83] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proc. of SP'97*, pages 94–102, 1997.