Decidability and Proof Systems for Language-Based Noninterference Relations [*]

Mads Dam

Dept. of Microelectronics and Information Technology, KTH, Electrum 229, SE-164 40 Kista, Sweden.
mfd@kth.se

Abstract

Noninterference is the basic semantical condition used to account for confidentiality and integrity-related properties in programming languages. There appears to be an at least implicit belief in the programming languages community that partial approaches based on type systems or other static analysis techniques are necessary for noninterference analyses to be tractable. In this paper we show that this belief is not necessarily true. We focus on the notion of strong low bisimulation proposed by Sabelfeld and Sands. We show that, relative to a decidable expression theory, strong low bisimulation is decidable for a simple parallel while-language, and we give a sound and relatively complete proof system for deriving noninterference assertions. The completeness proof provides an effective proof search strategy. Moreover, we show that common alternative noninterference relations based on traces or input-output relations are undecidable. The first part of the paper is cast in terms of multi-level security. In the second part of the paper we generalize the setting to accommodate a form of intransitive interference. We discuss the model and show how the decidability and proof system results generalize to this richer setting.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—mechanical verification; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Security, Theory, Verification.

Keywords Multi-Level Security, Information Flow, Noninterference, Language-Based Security, Intransitive Noninterference.

1. Introduction

Noninterference is the basic semantical condition used to account for confidentiality and integrity-related properties in programming languages. Most often the concept is studied in the setting of multi-level security [4, 11] with data assigned levels in a security lattice, such that levels higher in the lattice correspond to data of higher sensitivity. The question which noninterference aims to answer is one of *information flow*: A flow of information from a higher level in the security lattice to a lower one could breach confidentiality, and a flow from a lower level to a higher one might indicate a breach of integrity. In the area of language-based security, noninterference is of basic importance, since it provides the key semantic criterion by which the soundness of analysis methods must be evaluated.

A number of noninterference conditions have been proposed in the literature (cf. [28] for a survey). A common trait is that noninterference should capture the idea that variability of data at levels higher in the security lattice should not cause variability of behaviour which is noticeable at lower levels in the lattice. A large number of attempts have been made to capture this idea, depending on parameters such as the system/execution model, observers power of observation, and the richness of the security model. For instance:

- The trace-based model, starting with Goguen-Meseguer noninterference [18]), see also [21], requires that the set of system traces is closed under permutation of higher level actions with lower level ones.
- Various relational models, with Volpano, Smith, and Irvines type-based reconstruction of Denning and Denning's classical information flow analysis [36, 12] as the canonical example, characterizes information flow in terms of equivalence relations: If a program P is started with different initial stores σ_1 and σ_2, and if σ_1 and σ_2 contain the same low-level visible data, $\sigma_1 \approx_L \sigma_2$, then the final states σ_1', σ_2' after execution of P must also be low-level equivalent $\sigma_1' \approx_L \sigma_2'$.
- A natural generalization of the relational model in the context of concurrency is to introduce a relation of low-level equivalent behaviour on intermediate states, and characterize absence of information flow as preservation of this relation under execution, in the style of bisimulation equivalence. This idea is used in a large number of recent papers (cf. [31, 30, 5, 17, 13, 14, 23]), and it is related to the use of so-called unwinding conditions in the context of Goguen-Meseguer-style noninterference.

Our focus is on the decidability and axiomatizability properties of these noninterference relations. Specifically we show that, for a simple parallel while-language, relative to a decidable first-order theory of expressions, the Sabelfeld-Sands notion of strong low noninterference [31] is decidable. This result is of interest since there has been an at least implicitly stated belief in the programming languages community that partial approaches based on type systems or other static analysis techniques are necessary for noninterference to be tractable. Our results show that this is not necessarily so. We also give a sound and relatively complete proof system for proving noninterference assertions, and we show as part

[*] Work supported by the Swedish Research Council

of the relative completeness proof that proof search is decidable. Complementing this result we also show that other noninterference relations are undecidable. Specifically we show this for Boudol and Castellani's noninterference relation, but the argument applies equally to other flow-sensitive noninterference relations such as Volpano-Smith-Irvine (VSI) noninterference.

The notion of strong low bisimulation is very fine-grained. The idea is that two control states P_1 and P_2 can be regarded as behaving identically to a low-level observer, if whenever $\sigma_1 \approx_L \sigma_2$ and P_1 in state σ_1 can perform a single computation step to the configuration (P_1', σ_1') then P_2 should be able to mimick this step from σ_2 to a configuration, say, (P_2', σ_2') such that P_1' and P_2' continues to behave identically to a low level observer, and so that σ_1' and σ_2' have the same low-level content, $\sigma_1' \approx_L \sigma_2'$ (and vice versa). This stepwise quantification over low-level equivalent stores means that programs become distinguishable even if there are no sequential traces that motivate this, i.e. the definition is flow-insensitive. An example is the program

$$l := 0; \textbf{if } l = 1 \textbf{ then } l := h \textbf{ else noop}$$

which in a sequential setting is perfectly secure, since the positive arm of the conditional will never be taken. In a concurrent setting, however, the program is insecure, since an attacking thread might interfere with the assignment to l.

Besides decidability, this security condition has a number of attractive properties. As shown in [30] the relation is compositional with respect to a range of simple program constructs, including parallel composition. This does not apply in the case of most other noninterference relations, such as that of Boudol and Castellani. Moreover, strong links have recently been established to more systems-oriented models of noninterference. Specifically it has been shown that strong low bisimulation and a timing-sensitive version of the relation P-BNDC of Focardi and Rossi [14] coincides [15].

We add to the language a mechanism for dynamically changing the security level of identifiers. Such mechanisms are crucial for noninterference to be useful in many practical situations, as has been pointed out in a number of recent papers (cf. [32, 7, 9, 37, 20, 22, 23, 26]). The main challenges posed by dynamically changing levels are first to capture precisely the conditions under which a specific level changing operation should be allowed to go through, and secondly to determine the end-to-end security properties which a given level-changing policy delivers. In a recent paper [32] Sabelfeld and Sands have classified mechanisms for dynamic information regrading along several dimensions: "What" information is regraded, "who" is authorized to perform the regrading, "where" (in terms of program points and actions) the regrading is allowed to take place, and "when" information is allowed to be regraded. In this paper we concentrate on the "where" dimension. The aim is to ensure that information regrading actions are confined to program points and operations for which those actions are specifically allowed. To illustrate the range of concerns involved we give some examples:

- Explicit downgrading actions such as the downgrading assignment $[l := h]$ of Mantel and Sands [23]. In this case the intention is that the assignment represents an explicitly permitted flow from a higher level to a lower level in the security lattice. This is an example of an *intransitive* flow: It may that $\mathit{dom}(x_1) \leq \mathit{dom}(x_2) \leq \mathit{dom}(x_3)$ in the security lattice, also that $\mathit{dom}(x_2) \rightsquigarrow \mathit{dom}(x_1)$ (direct flows from $\mathit{dom}(x_2)$ to $\mathit{dom}(x_1)$ are permitted) and $\mathit{dom}(x_3) \rightsquigarrow \mathit{dom}(x_2)$, but *not* $\mathit{dom}(x_3) \rightsquigarrow \mathit{dom}(x_1)$.
- Explicit level changing operations such as operations to downgrade an identifier from high to low, or to upgrade an identifier from low to high. This type of operation is useful when security of the level change operation is supported by some external mechanism such as an access control system, a reference monitor, or a trusted downgrading agent. As an example it might be safe to upgrade parts of an APDU (Application Protocol Data Unit) received by a smart card during the personalization phase by relying on an external access control mechanism to guarantee that the smart card is attached to an authorized card reader.
- Operations that allow the security lattice to change under execution, by dynamically introducing new security levels and removing them again.

The approach we propose in this paper is designed to handle the former two of these three types of dynamic regrading. Dynamically changing the security lattice is outside the scope of this work. The latter problem has been examined recently by Boudol and Matos [6].

The paper is organized as follows: We start by introducing object automata and use these to give semantics to a simple parallel while language. In section 3 and 4 we recall the notion of strong low bisimilarity from [31], and in section 5 we prove decidability. In section 6 we prove undecidability of the noninterference notion of Boudol and Castellani (here called "flat bisimulation" due to its non-lifted quantification over stores). In sections 7 and 8 the proof system is introduced, and soundness and relative completeness is proved. Then in section 9 we turn to dynamically changing security levels, and present our generalization of strong low bisimulation. This generalization is closely related, though subtly different from the corresponding notion by Mantel and Sands [23]. In section 10 the decidability, proof system, soundness, and relative completeness results of sections 5, 7 and 8 are extended to strong dynamic low bisimulation. Finally, in section 11 we conclude, discuss related work, and give pointers to future work.

2. Object Automata

We use a standard state-based model of program execution based on transition systems with a mutable store.

Identifiers, Values, Control States, and Stores The following denumerable sets are primitive: *Identifiers* $x, y, z \in \mathit{Ide}$, *values* $v \in \mathit{Val}$, and *control states*, $s \in S$. A *store* is an assignment $\sigma : \mathit{Ide} \to \mathit{Val}$ of values to identifiers; $\sigma[x \mapsto v]$ is the result of updating σ by assigning v to x.

States and Transitions An object automaton $\mathcal{A}$ consists of a set of *states* of the form (s, σ), a *transition relation* $\to$, and an *initial state* (s_0, σ_0). As usual we write $(s, \sigma) \to (s', \sigma')$ whenever $((s, \sigma), (s', \sigma')) \in \to$; in that case (s, σ) is the *pre-state*, and (s', σ') is the *post-state*. If the set S is finite we say that the corresponding object automaton is *finite control*.

Parallel While Programs Standard examples of object automata are based on parallel while programs $P \in \mathcal{P}$ using the syntax

$$\begin{array}{lcl} \alpha & ::= & x := E \\ P & ::= & \alpha \mid \textbf{stop} \mid P_1; P_2 \mid \textbf{if } E \textbf{ then } P_1 \textbf{ else } P_2 \\ & & \mid \textbf{while } E \textbf{ do } P \mid P \parallel P \, . \end{array}$$

Expressions E range over some fixed first-order theory $\mathcal{E}$, and α is a primitive command. At this point the only primitive commands are assignments. Later we add primitive commands that dynamically control the level assignment. We use **noop** as abbreviation of the assignment $x := x$ for some fixed, arbitrary x. This is justified since the transition semantics assumed below makes assignments atomic. The set $\mathit{Ide}(E)$ is the finite set of identifiers occurring in E. For a store σ, $\sigma(E)$ is the value of E in σ, and $E_1 \approx E_2$ is

$$(\textsc{Cong})\quad \frac{P \equiv P' \quad (P',\sigma) \to (Q',\sigma') \quad Q' \equiv Q}{(P,\sigma) \to (Q,\sigma')}$$

$$(\textsc{Assign})\quad \frac{-}{(x := E,\sigma) \to (\mathbf{stop}, \sigma[\sigma(E)/x])}$$

$$(\textsc{Seq-1})\quad \frac{(P,\sigma) \to (P',\sigma')}{(P;Q,\sigma) \to (P';Q,\sigma')}$$

$$(\textsc{Cond-1})\quad \frac{\sigma(E) \neq 0}{(\mathbf{if}\ E\ \mathbf{then}\ P\ \mathbf{else}\ Q,\sigma) \to (P,\sigma)}$$

$$(\textsc{Cond-2})\quad \frac{\sigma(E) = 0}{(\mathbf{if}\ E\ \mathbf{then}\ P\ \mathbf{else}\ Q,\sigma) \to (Q,\sigma)}$$

$$(\textsc{While-1})\quad \frac{\sigma(E) \neq 0}{(\mathbf{while}\ E\ \mathbf{do}\ P,\sigma) \to (P;\mathbf{while}\ E\ \mathbf{do}\ P,\sigma)}$$

$$(\textsc{While-2})\quad \frac{\sigma(E) = 0}{(\mathbf{while}\ E\ \mathbf{do}\ P,\sigma) \to (\mathbf{stop},\sigma)}$$

$$(\textsc{Par-1})\quad \frac{(P,\sigma) \to (P',\sigma')}{(P \parallel Q,\sigma) \to (P' \parallel Q,\sigma')}$$

$$(\textsc{Par-2})\quad \frac{(Q,\sigma) \to (Q',\sigma')}{(P \parallel Q,\sigma) \to (P \parallel Q',\sigma')}$$

Figure 1. Transition semantics for parallel while language

expression equality, i.e. $E_1 \approx E_2$ iff for all σ, $\sigma(E_1) = \sigma(E_2)$. Let $L \subseteq \mathit{Ide}$ and let $L \cap \mathit{Ide}(E) = \{x_1, \ldots, x_n\}$. Define the property

$$\mathit{dep}(E,L) = \\ \exists x_1,\ldots,x_n,y_1,\ldots,y_n.E[y_1/x_1,\ldots,y_n/x_n] \not\approx E\ .$$

The intention is that $\mathit{dep}(E,L)$ iff there is flow of information from the identifiers in L to E.

Transition Semantics The transition semantics, shown in Figure 1, uses a structural congruence relation $\equiv$ for ease of presentation. A *congruence* is any equivalence relation which is preserved by the constructs of $\mathcal{P}$. The specific relation $\equiv$ is the least congruence such that

$$\mathbf{stop}; P \equiv P; \mathbf{stop} \equiv \mathbf{stop} \parallel P \equiv P \parallel \mathbf{stop} \equiv P\ .$$

3. Strong Λ-Security

We assume that identifiers are partitioned into classes *high* and *low*, corresponding to their confidentiality levels. The results are easily generalized to richer security lattices. The letter $\Lambda \subseteq \mathit{Ide}$ represents the set of low identifiers. For now the partitioning into high and low identifiers is assumed to be fixed, but later it will be allowed to change. Two stores, σ_1 and σ_2, are Λ-equivalent, $\sigma_1 \approx_\Lambda \sigma_2$, if for all $x \in \Lambda$, $\sigma_1(x) = \sigma_2(x)$.

Noninterference is a reflection of the idea that to be secure in a multi-level setting, there should be no way that variability at the level of high data can influence behaviour at the low level. There are a number of ways in which this idea can be formalized in a language-based setting. In this paper our starting point is Sabelfeld and Sands notion of strong low bisimulation [31, 30].

DEFINITION 1 (Strong Λ-Bisimulation). *A relation R on control states is a* strong Λ-bisimulation *if R is symmetric, and whenever $s_1 R s_2$ then for all $\sigma_1 \approx_\Lambda \sigma_2$, if $(s_1,\sigma_1) \to (s_1',\sigma_1')$ then there are s_2', σ_2' such that $(s_2,\sigma_2) \to (s_2',\sigma_2')$, $s_1' R s_2'$, and $\sigma_1' \approx_\Lambda \sigma_2'$. The relation $\cong_\Lambda$ of* strong Λ-bisimulation equivalence *is the largest strong Λ-bisimulation relation.*

We leave the check that $\cong_\Lambda$ exists and is identical to

$$\bigcup\{R \mid R \text{ is a strong } \Lambda\text{-bisimulation}\}$$

to the reader. Below we often refer to $\cong_\Lambda$ as just Λ-bisimilarity, for short, or strong low bisimilarity, depending on context. The standard property of bisimulation-like security definitions, that $\cong_\Lambda$ is a partial equivalence relation (a "per", a binary relation which is symmetric and transitive, but not necessarily reflexive), is easily verified (cf. [31]).

The usual self-bisimilarity condition is applied on the reflexive elements, by defining:

DEFINITION 2 (Strong Λ-Security). *The state s is* strongly Λ-secure, *if $s \cong_\Lambda s$.*

This security condition is very strong. If $s_1 \cong_\Lambda s_2$ then no variation in the high parts of a store can cause a low observer to distinguish s_1 from s_2, even when running in an arbitrary, possibly hostile, concurrent environment which:

- can read and write arbitrary low identifiers,
- has knowledge of the code under execution, and
- can count execution steps.

Thus, for instance,

$$l = 0\ ;\ \mathbf{if}\ l = 1\ \mathbf{then}\ l := h\ \mathbf{else}\ l := l$$

will be insecure, due to the quantification over stores in def. (1), even if in a sequential setting only the harmless **else** branch will be executed. In a concurrent context, however, this quantification over intermediate stores is in fact quite reasonable, since execution of other threads may cause arbitrary low interference.

The version of strong Λ-security studied in [30, 27] strengthens def. 1 by also requiring preservation of the the number of active threads. This is motivated by an attack model where the attacker has the additional capability of controlling the scheduler. The resulting notion of strong Λ-bisimulation is shown in [27] to be the most inclusive indistinguishability-based noninterference relation which is scheduler-independent and preserved under parallel composition. Below we concentrate on the weaker definition 1 above, and point out how the constructions can be adapted to accomodate the stronger definition of [27].

4. Examples

In this section we give a series of small program examples to illustrate the relationship between strong Λ-security and other noninterference relations in the literature, and to demonstrate the added scope of a semantical analysis of noninterference in relation to a type-based one.

EXAMPLE 1. *We give a few examples of while-programs that are secure, respectively insecure, according to def. 2.*

1. **if** $h = 0$ **then** $l := 0$ **else** $l := 1$
   This program is insecure. We may have $\sigma_1 \approx_\Lambda \sigma_2$ even if $\sigma_1(h) = 0 \neq \sigma_2(h)$. But the post-states of the transitions will then be incomparable with respect to $\approx_\Lambda$.
2. **while** $h > 0$ **do** $h := h - 1$
   The program is insecure since the number of transitions executable depends on the initial value of h.
3. $l := h; l := 0$
   This program is insecure, since stores $\sigma_1 \approx_\Lambda \sigma_2$ can be found such that $\sigma_1[\sigma_1(h)/l] \not\approx_\Lambda \sigma_2[\sigma_2(h)/l]$.
4. **if** $h = 0$ **then** $(l; h)$ **else** $(l; h')$
   In this example and elsewhere we use l, l', l_i (h, h', h_i) etc. as generic examples of low (high) assignments of the form, e.g. $l := l'$ ($h := h'$). The program is secure, since $l; h \cong_\Lambda l; h'$ for all high assignments h, h'.
5. (**if** $h = 0$ **then** $h_1 := 1$ **else stop**); **loop stop**
   In this example we use **loop** P *as an abbreviation of the com-*

mand **while** $x = x$ **do** P. *The program is secure since both branches of the conditional are followed by an infinite sequence of transitions that do not affect the low part of the store.*

6. (**if** $h = 0$ **then** h_1 **else stop**) $\parallel$ **loop** h_2
The program is secure, essentially since $h_1 \parallel$ **loop** $h_2 \cong_\Lambda$ **loop** h_2.
7. (**if** $h = 0$ **then** $(h_1; l)$ **else** $(l; h_3)$) $\parallel$
(**if** $h \neq 0$ **then** $(l; h_4)$ **else** $(h_5; l)$)
The program is secure. Consider two initial stores σ_1 *and* σ_2. *If* σ_1 *and* σ_2 *agree on the assignment to* h *then the bisimulation check is trivial. If not, the positive arm of the first conditional is matched to the negative arm of the second conditional, and vice versa.*

The examples highlight some important distinctions both between different approches to noninterference in the literature, and between secure typable and secure untypable programs.

Examples 1.1 and 2 are standard examples of indirect information leaks.

Example 1.3 illustrates the flow insensitivity of strong Λ-security, contrasting to the flow-sensitivity of the more standard, sequential accounts such as VSI noninterference.

Example 1.4 is secure but untypable in the type system of VSI since the latter prohibits low commands in the context of high branching. It is, however, typable in the type system of Agat [1]. There, an external check is used to ensure that the two branches of the conditional are Λ-bisimilar in the context of high branching. This renders the high branching harmless with respect to low observers. In an example based on RSA, Agat has demonstrated how his approach allows timing leaks to be effectively prevented [1]. One consequence of the results reported in this paper, is the decidability of Agat's type system for the while-language considered in this paper (which, up to minor details, is an extension of Agat's language).

Finally, 1.5-7 are examples of programs that are secure by the global nature of the Λ-security condition, but which will be rejected by most local analyses, including type-based ones. Observe that the stronger security definition of [27] rejects the equivalence $h_1 \parallel$ **loop** $h_2 \cong_\Lambda$ **loop** h_2 since the latter definition is sensitive to the number of threads.

If we extend the language by jumps it is easy to come up with examples of unstructured programs that will in general require semantical methods to prove Λ-secure. One somewhat artificial example is the following variation of example 1.4:

if $h = 0$ **then goto** PC_2 **else**
loop $(l := 0; h := 1; PC_2 : l := 0; h := 2)$

5. Decidability

In this section we show decidability of strong Λ-bisimilarity, first for general object automata, and then for parallel while-programs.

DEFINITION 3 (Effective Separability). *An object automaton* $\mathcal{A}$ *is* effectively separable, *if given control states* $s_1, s_1', s_2, s_2' \in S$ *and set* $\Lambda \subseteq$ *Ide it is decidable if there are stores* $\sigma_1, \sigma_1', \sigma_2$ *such that*

1. $(s_1, \sigma_1) \rightarrow (s_1', \sigma_1')$
2. $\sigma_1 \approx_\Lambda \sigma_2$
3. *whenever* $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$ *then* $\sigma_1' \not\approx_\Lambda \sigma_2'$.

THEOREM 1. *For finite-control, effectively separable object automata, strong* Λ*-bisimilarity is decidable.*

PROOF We construct an effective, strictly decreasing, maximal sequence of relations $R_0 \supseteq R_1 \supseteq \cdots \supseteq R_n$ from $R_0 = S \times S$ such that

1. R_n is a strong Λ-bisimulation.
2. If R is any other strong Λ-bisimulation, then $R \subseteq R_n$.

Since n is bounded by $|R_0| = |S|^2$, this is sufficient to etablish the theorem. Suppose we have constructed R_i. Suppose we can find $s_1, s_2 \in S$ such that $s_1 R_i s_2$, and such that the following condition holds:

(*) There is $s_1' \in S$ and stores $\sigma_1, \sigma_1', \sigma_2$ such that
- $(s_1, \sigma_1) \rightarrow (s_1', \sigma_1')$,
- $\sigma_2 \approx_\Lambda \sigma_1$, and
- for all s_2', σ_2', if $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$ then either $\sigma_2' \not\approx_\Lambda \sigma_1'$ or $\neg(s_1' R_i s_2')$.

We then let

$$R_{i+1} = R_i \setminus \{(s_1, s_2), (s_2, s_1)\}\ .$$

If no such pair (s_1, s_2) can be found then the sequence is complete.

We need to verify that condition (*) is decidable. Note first that, by finite control, it suffices to check (*) for each choice of control states s_1, s_1', s_2, s_2'. But then the condition $\neg(s_1' R_i s_2')$ is independent of the choice of σ_2' in (*) so effective separability can be applied. We then need to verify conditions 1. and 2. above.

CLAIM 1. *R_n is a strong Λ-bisimulation*

PROOF (of claim) Suppose that $s_1 R_n s_2$, $\sigma_1 \approx_\Lambda \sigma_2$, and that $(s_1, \sigma_1) \rightarrow (s_2, \sigma_2)$. By condition (*) it must be the case that s_2', σ_2' can be found such that $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$, $s_1' R_n s_2'$ and $\sigma_1' \approx_\Lambda \sigma_2'$, since n is maximal. □

CLAIM 2. *If $R \subseteq S \times S$ is a strong Λ-bisimulation, then $R \subseteq R_n$.*

PROOF (of claim) Suppose $s_1 R s_2$. We prove $R \subseteq R_i$ for all $i : 0 \leq i \leq n$. The base case is trivial. For the induction step, if $\neg(s_1 R_{i+1} s_2)$ then we find $s_1' \in S$ and $\sigma_1, \sigma_1', \sigma_2$ such that $(s_1, \sigma_1) \rightarrow (s_1', \sigma_1')$ and $\sigma_2 \approx_\Lambda \sigma_1$, but whenever $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$ then either $\sigma_2' \not\approx_\Lambda \sigma_1'$ or $\neg(s_1' R_i s_2')$. Since R is a strong bisimulation, we know that some s_2' and σ_2' can be found such that $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$, $\sigma_1 \approx_\Lambda \sigma_2'$, and $s_1' R s_2'$. But this contradicts the induction hypothesis. □

This concludes the proof of theorem 1. □

To prove decidability for parallel while-programs, define the relation $\sim_\Lambda$ on assignments as the symmetric closure of the following clauses (where $E_1 \approx_\Lambda E_2$ abbreviates the condition: for all $\sigma_1 \approx_\Lambda \sigma_2$, $\sigma_1(E_1) = \sigma_2(E_2)$):

1. If α_1 is the assignment $x := E_1$, α_2 is the assignment $x := E_2$, and $E_1 \approx_\Lambda E_2$ then $\alpha_1 \sim_\Lambda \alpha_2$
2. If α_1 is the assignment $x_1 := E_1$, α_2 is the assignment $x_2 := E_2$, $x_1 \in \Lambda$, $x_2 \notin \Lambda$, and $E_1 \approx x_1$ then $\alpha_1 \sim_\Lambda \alpha_2$
3. If α_1 is the assignment $x_1 := E_1$ and α_2 is the assignment $x_2 := E_2$, $x_1, x_2 \notin \Lambda$, then $\alpha_1 \sim_\Lambda \alpha_2$.

Notice that $\sim_\Lambda$ is an effective relation when the expression theory $\mathcal{E}$ is decidable, since the relation $\approx_\Lambda$ is expressible in $\mathcal{E}$.

LEMMA 1. $\alpha_1 \cong_\Lambda \alpha_2$ *iff* $\alpha_1 \sim_\Lambda \alpha_2$.

PROOF Observe first that if either of the three conditions hold, then $\alpha_1 \cong_\Lambda \alpha_2$. Conversely let $\alpha_1 = x_1 := E_1$ be given, and assume that $x_1 \in \Lambda$. If $\alpha_2 = x_2 := E_2$ and $x_2 \in \Lambda$, if $E_1 \not\approx_\Lambda E_2$ then $\alpha_1 \ncong_\Lambda \alpha_2$. If $x_2 \notin \Lambda$ and $E_1 \not\approx x_1$ then we find a store σ such that $\sigma(E_1) \neq \sigma(x_1)$, but then $\sigma[\sigma(E_1)/x_1] \not\approx_\Lambda \sigma[\sigma(E_2)/x_2]$, so $\alpha_1 \ncong_\Lambda \alpha_2$. This covers the cases and the proof. □

THEOREM 2. *For parallel while-programs, if $\mathcal{E}$ is decidable then so is strong Λ-bisimilarity.*

PROOF We need to show that object automata for parallel while programs are finite control, and that they are effectively separable. For finite control first, let $P \rhd P'$ iff there are σ, σ', α such that $(P, \sigma) \rightarrow (P', \sigma')$. It suffices to show that the set $\{P' \mid P \rhd^* P'\}$ is finite up to structural congruence $\equiv$, and that we can effectively find a set of canonical representatives of each congruence class. To see that it is finite, define the measure $|P|$ on $\mathcal{P}$ in the following way:

- $|x := E| = 2$
- $|\textbf{stop}| = 1$
- $|P; Q| = |P| + |Q|$
- $|\textbf{if } E \textbf{ then } P \textbf{ else } Q| = |P| + |Q| + 1$
- $|\textbf{while } E \textbf{ do } P| = |P| + 2$
- $|P \parallel Q| = |P| \cdot |Q|$

If $[P]_\equiv$ is the congruence class of P under $\equiv$ it is sufficient to show that

$$|\{[Q]_\equiv \mid P \rhd^* Q\}| \leq |P| \,. \qquad (1)$$

The proof of (1) is by induction on the structure of P. For instance, to show (1) for the case $P = \textbf{while } E \textbf{ do } P_1$ it is sufficient to observe that $P \rhd^* Q$ iff Q has the form either **stop**, or P, or $P_1'; P$ where $P_1 \rhd^* P_1'$.

For effective separability note first that each pair P, P' such that $P \rhd P'$ determines exactly one of the rules ASSIGN, COND-1, COND-2, WHILE-1, or WHILE-2 which is used in the derivation of a transition $(P, \sigma) \rightarrow (P', \sigma')$, for any σ, σ'. This is easily seen by induction on the structure of P. In case the rule is ASSIGN, let *label*(P, P') be the reduced assignment $x := E$. In the other four cases let *label*(P, P') be $x := x$ for some arbitrary x. Assume then given the program terms P_1, P_1', P_2, P_2' such that $P_1 \rhd P_1'$ and $P_2 \rhd P_2'$, and let $\Lambda \subseteq$ *Ide*. Let $\alpha_1 =$ *label*(P_1, P_1') and $\alpha_2 =$ *label*(P_2, P_2'). The result then follows by Lemma 1. □

Observe that the proofs of theorems 1 and 2 are quite generic and could be adapted without much trouble, e.g. to bytecode languages. They are also easy to adapt to the security condition of [27] by requiring preservation of the number of threads. For time complexity, if n is the cost of checking condition (*) for given s_1, s_1', s_2, s_2', and m is the number of control states, the overall time complexity for checking strong λ-bisimilarity is $\mathcal{O}(m^4 n)$. For the case of parallel while-programs, m is exponential in the size of the input program, due to state space explosion, and n will normally be at least single exponential, cf. the case of boolean expressions.

6. Flat Bisimulation

It is instructive to compare strong Λ-bisimulation to the notion of $(\Gamma, \mathcal{L})$-bisimulation introduced by Boudol and Castellani [5]. The difference between strong Λ-bisimulation and Boudol and Castellani's bisimulation is that the unwinding condition in the latter is cast in terms of configurations whereas the former is lifted to control states. For this reason we also in this paper refer to Boudol-Castellani bisimulation as "flat" bisimulation.

DEFINITION 4 (Flat Bisimulation). *The relation R on configurations is a* strong flat Λ-bisimulation *if R is symmetric, and whenever $(s_1, \sigma_1) R (s_2, \sigma_2)$ then*

1. $\sigma_1 \approx_\Lambda \sigma_2$
2. *if $(s_1, \sigma_1) \rightarrow (s_1', \sigma_1')$ then $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$ for some s_2', σ_2' such that $(s_1', \sigma_1')\ R (s_2', \sigma_2')$.*

The relation $\simeq_\Lambda$, strong flat Λ-bisimulation equivalence, *is the largest strong flat Λ-bisimulation relation. The relation $\simeq_\Lambda$ is lifted to control states by $s_1 \simeq_\Lambda s_2$ iff for all σ_1, σ_2, if $\sigma_1 \approx_\Lambda \sigma_2$ then $(s_1, \sigma_1) \simeq_\Lambda (s_2, \sigma_2)$.*

Again it is easy to see that $\simeq_\Lambda$ exists. We first note that strong Λ-equivalence is strictly finer than strong flat equivalence.

PROPOSITION 1. $\cong_\Lambda \subsetneq \simeq_\Lambda$

PROOF $\subseteq$: We show that if R is a strong Λ-bisimulation then the relation

$$R'((s_1, \sigma_1), (s_2, \sigma_2)) \text{ iff } s_1 R s_2 \text{ and } \sigma_1 \approx_\Lambda \sigma_2$$

is a strong flat bisimulation. To show this let $(s_1, \sigma_1) \rightarrow (s_1', \sigma_1')$. We obtain directly that $(s_2, \sigma_2) \rightarrow (s_2', \sigma_2')$ such that $s_1' R s_2'$ and $\sigma_1' \approx_\Lambda \sigma_2'$, hence $(s_1', \sigma_1') R' (s_2', \sigma_2')$ which is what needs to be shown.

$\subsetneq$: Let s_1 be the program $x := y; y := x$ and s_2 the "residual" $y := x$. Let $\Lambda = \{y\}$ and let $(s, \sigma_1) R'\ (s', \sigma_2)$ iff $s = s' \in \{s_1, s_2\}$ and $\sigma_1 \approx_\Lambda \sigma_2$. The relation R' is a strong flat bisimulation, since if $(s, \sigma_1) \rightarrow (s', \sigma_1')$ then $(s, \sigma_2) \rightarrow (s', \sigma_2')$ such that $s', \sigma_1') R'\ (s', \sigma_2')$. Thus, $s \simeq_\Lambda s$. On the other hand, $s \not\cong_\Lambda s$. To see this, assume that $s_1\ R\ s_1$ and that R is a Λ-bisimulation. Then we would need to obtain that $s_2 R\ s_2$ as well. But this is impossible. To see this, let $\sigma_1(y) = \sigma_2(y)$ and $\sigma_1(x) \neq \sigma_2(x)$. Then $\sigma_1 \approx_\Lambda \sigma_2$. If $(s_2, \sigma_1) \rightarrow (s_2', \sigma_1')$ then $\sigma_1'(y) = \sigma_1(x)$. But whenever $(s_2, \sigma_2) \rightarrow (s_2'', \sigma_2')$ then $\sigma_2'(y) = \sigma_2(x) \neq \sigma_1'(y)$. Hence $s_2 R\ s_2$ is impossible, and so is $sR\ s$. □

The problem with flat bisimulation, as pointed out in [5], and in contrast to strong Λ-bisimulation, is that flat bisimulation is not preserved under parallel composition. This is due to the possibility of interference (in the classical concurrent programming sense). An example is the program $(x := y; y := x) \parallel x := z$. Each of the component processes $x := y; y := x$ and $x := z$ are flat bisimulation secure for $\Lambda = \{y\}$, but their parallel composition is not, as is easily checked. This phenomenon does not arise, however, in the case of strong Λ-bisimulation, due to the lifted unwinding condition. Another major distinction between the two bisimulation notions is the decidability properties.

THEOREM 3. *Strong flat Λ-bisimulation equivalence on parallel while programs is undecidable.*

PROOF (Sketch) Let P be an arbitrary while program, let $x, y, z \in$ *Ide* be variables not mentioned in P, and let $\Lambda = \{z\}$. We construct the program $P' = P_1 \parallel P_2 \parallel P_3$ such that $P_1 = x := 1; (\textbf{while } x \neq 0 \textbf{ do stop}); z := y$, $P_2 = \textbf{while } 0 = 0 \textbf{ do stop}$, $P_3 = P; x := 0$. We claim that $P' \simeq_\Lambda P'$ iff P does not terminate on any initial assignment (which is an undecidable problem). To see this, if P does not terminate on any initial assignment then the relation

$$\{(Q, \sigma), (Q', \sigma') \in \mathcal{R}(P') \mid \sigma \approx_\Lambda \sigma'\}$$

is a strong flat bisimulation where

$$\mathcal{R}(P) = \{(Q, \sigma) \mid \exists \sigma'.(P, \sigma') \rightarrow^* (Q, \sigma)\} \,.$$

Conversely if P does terminate on some initial assignment then a strong flat Λ-bisimulation relation will have to match the assignment $z := y$ of P_1 for arbitrary initial assignments to y which is impossible. □

The same construction can be used to show undecidability of other noninterference conditions, including various variants of Volpano-Smith style noninterference [36, 33], in both strong and weak versions, using the terminology of Milner [25].

7. Proof System

In this section we give a sound and relatively complete proof system for strong Λ-bisimilarity for parallel while programs. Existing information flow type systems (c.f. [36, 19, 31, 30, 5, 33, 35, 1]) use type assertions of the shape $P : \Lambda$ where P is a single program term and Λ represents the level, here the set of low identifiers. This formulation has well-known shortcomings in the case of high branching. As an example it is not possible to reduce type correctness for a conditional such as **if** $h = 0$ **then** P **else** Q to type correctness of P and Q, since low assignments in P or Q, even if P and Q are type correct by themselves, can be used to leak the branch condition $h = 0$. The normal, quite restrictive, approach is to completely prohibit low observable actions in the scope of a high branch (cf. [36]). Various proposals have been made to allow more programs to be typed, including Agat's suggestion to require bisimilarity of both arms of the conditional [1].

Judgments Our proposal is to replace type assertions of the form $P : \Lambda$ with more general judgments of the form $\Gamma : \Lambda$ where Γ is a finite set of program terms. The intuition is that Γ is the set of control states that are possible at some given point of execution, given the initial uncertainty in the values of high identifiers. We use standard sequent-type notation for judgments and so write, e.g., $\Gamma, P : \Lambda$ for $\Gamma \cup \{P\} : \Lambda$. If a judgment $\Gamma : \Lambda$ is valid, it should not be possible for a low observer to tell any two (possibly identical) members of Γ apart, since that might indicate unlicensed information flow. Thus, a judgment $\Gamma : \Lambda$ will be *valid*, $\models \Gamma : \Lambda$, if $P \cong_\Lambda Q$ whenever $P, Q \in \Lambda$. In particular, if $\models \Gamma : \Lambda$ and $P \in \Gamma$ then P is strongly Λ-secure. This representation of goal states using sets allows us to reduce a judgment such as **if** $h = 0$ **then** P **else** $Q : \Lambda$ (where $h \notin \Lambda$) to the judgment $P, Q : \Lambda$.

Approach The proof system we present is essentially a tableau system with a coinductive termination condition. Each proof goal $\Gamma : \Lambda$ is reduced, using the tableau rules or the structural congruence relation, until it is in a special *prefix form*, where each member of Γ has the form $(\alpha; P)$ **or** Q. If the proof goal is valid, such a reduction should be possible in such a way that all the prefixes α are well-behaved with respect to the level assignment α, in the sense that either all the α's are assignments to high identifiers, or else they are all *identical* assignments of a low expression to a low identifier. If either of these two cases hold the transition can be taken for each member of Γ, and the tableau elaboration process continues. For termination, the tableau system is, besides the local reduction rules which reduce a tableau node $\Gamma : \Lambda$ to a (small, finite) set of subsidiary tableau nodes, equipped with a rule of discharge. This rule of discharge is a simple loop detection condition: If a leaf node is seen to have been already elaborated at an earlier node of the tableau construction process, the leaf node is closed by introducing a back arc.

A first issue is how to allow nodes to be reduced to prefix form. Most cases, except parallel composition, are straightforward. For parallel composition, the issue is how to reduce a proof goal of the form $\Gamma, P \parallel Q : \Lambda$. A partial solution would be to exploit the congruential properties of $\cong_\Lambda$ under parallel composition to at least allow the reduction of $\{P \parallel Q\} : \Lambda$ to subgoals $\{P\} : \Lambda$ and $\{Q\} : \Lambda$. First, however, this approach is not easily generalized to sequents $\Gamma : \Lambda$ where $|\Gamma| > 1$, and secondly the approach would be essentially incomplete, as shown e.g. by example 1.6 which does not lend itself as easily to a compositional analysis.

Our solution uses a construct $⫫$ (left merge) in addition to nondeterministic choice (**or**) to incrementally eliminate $\parallel$, in a way which is reminiscent of the interleaving law of CCS. The transition rules for these constructs are shown in figure 2. The advantage of the left merge operator in this context is that it allows the reduction to prefix form to proceed more easily, since for program terms of the form $P ⫫ Q$, P is forced to perform the first transition. It also allows an easy reduction of the general parallel composition,

$$\text{OR-1} \quad \frac{(P,\sigma) \to (P',\sigma')}{(P \textbf{ or } Q,\sigma) \to (P',\sigma')}$$

$$\text{OR-2} \quad \frac{(Q,\sigma) \to (Q',\sigma')}{(P \textbf{ or } Q,\sigma) \to (Q',\sigma')}$$

$$\text{LMERGE} \quad \frac{(P,\sigma) \to (P',\sigma')}{(P ⫫ Q,\sigma) \to (P' \parallel Q,\sigma')}$$

Figure 2. Transition rules for choice and left merge

$$\frac{E_1 \approx E_2}{x := E_1 \equiv x := E_2} \qquad \frac{-}{x_1 := x_1 \equiv x_2 := x_2}$$

$$\frac{-}{P_1; (P_2; P_3) \equiv (P_1; P_2); P_3}$$

$$\frac{-}{P_1 \textbf{ or } (P_2 \textbf{ or } P_3) \equiv (P_1 \textbf{ or } P_2) \textbf{ or } P_3}$$

$$\frac{-}{P_1 \textbf{ or } P_2 \equiv P_2 \textbf{ or } P_1} \qquad \frac{-}{P \textbf{ or } P \equiv P}$$

$$\frac{-}{\textbf{stop or } P \equiv P} \qquad \frac{-}{\textbf{stop} ⫫ P \equiv \textbf{stop}} \qquad \frac{-}{P ⫫ \textbf{stop} \equiv P}$$

Figure 3. Extended structural congruence rules

by validating the reduction of $\Gamma, P \parallel Q : \Lambda$ to the subgoal $\Gamma, (P ⫫ Q)$ **or** $(Q ⫫ P) : \Lambda$.

Reduction Contexts and Structural Congruence The full tableau system is somewhat complicated by the need to nest sequential composition and left merge. This problem is addressed by first refining the structural congruence relation, and, secondly, by adding the concept of a reduction context, to allow the tableau rules to consider program terms placed in the "active position" of arbitrary sequential or left merge compositions.

For the structural congruence relation we extend $\equiv$ by the rules shown in fig. 3. This extension is mainly a convenience to allow a more compact presentation of the tableau system below. The rules in figure 3 are safe in the sense that if $P \equiv Q$ then $P \cong_\Lambda Q$, for any set Λ, where $\cong_\Lambda$ is defined using the original rules of section 2. We state the following easy decidability result without proof.

PROPOSITION 2. *Structural congruence on $\mathcal{P}$ is decidable.* □

A *reduction context* is an expression of the form

$$C[\cdot] ::= [\cdot] \mid (C[\cdot]; P) ⫫ Q .$$

An *extended reduction context*, $C^+[\cdot]$, allows choice at the outermost level:

$$C^+[\cdot] ::= C[\cdot] \textbf{ or } Q .$$

For each $P \in \mathcal{P}$ define $C^{\parallel}[P]$ inductively by:

$$\begin{aligned} [P]^{\parallel} &= P \\ ((C[P']; P) ⫫ Q)^{\parallel} &= (C^{\parallel}[P']; P) \parallel Q \end{aligned}$$

The important property of reduction contexts is the following:

LEMMA 2. *If $(C[P],\sigma) \to (Q,\sigma')$ and $P \not\equiv$ **stop** then there is a P' such that $(P,\sigma) \to (P',\sigma')$ and $Q = C^{\parallel}[P']$.*

PROOF Induction on the structure of $C[\cdot]$. □

$$\textsc{HighAssnC}\quad \frac{C_1^{\|}[\textbf{stop}],\dots,C_n^{\|}[\textbf{stop}]:\Lambda \qquad Q_1,\dots,Q_n:\Lambda}{C_1[x_1:=E_1]\ \textbf{or}\ Q_1,\dots,C_n[x_n:=E_n]\ \textbf{or}\ Q_n:\Lambda}\quad (\{x_1,\dots,x_n\}\cap\Lambda=\emptyset)$$

$$\textsc{LowAssnC}\quad \frac{C_1^{\|}[\textbf{stop}],\dots,C_n^{\|}[\textbf{stop}]:\Lambda \qquad Q_1,\dots,Q_n:\Lambda}{C_1[x:=E]\ \textbf{or}\ Q_1,\dots,C_n[x:=E]\ \textbf{or}\ Q_n:\Lambda}\quad (x\in\Lambda \text{ and } \neg \mathit{dep}(E,\overline{\Lambda}))$$

$$\textsc{CongC}\quad \frac{\Gamma,Q:\Lambda}{\Gamma,P:\Lambda}\quad (P\equiv Q)\qquad \textsc{StopC}\quad \frac{-}{\textbf{stop}:\Lambda}\qquad \textsc{HighCondC}\quad \frac{\Gamma,C^+[\textbf{noop};P_1],C^+[\textbf{noop};P_2]:\Lambda}{\Gamma,C^+[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2]:\Lambda}$$

$$\textsc{LowCondC}\quad \frac{\Gamma,C^+[\textbf{noop};P_1]:\Lambda \quad \Gamma,C^+[\textbf{noop};P_2]:\Lambda}{\Gamma,C^+[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2]:\Lambda}\quad (\neg \mathit{dep}(E,\overline{\Lambda}))$$

$$\textsc{TrueCondC}\quad \frac{\Gamma,C^+[\textbf{noop};P_1]:\Lambda}{\Gamma,C^+[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2]:\Lambda}\quad (E\not\approx 0)\qquad \textsc{FalseCondC}\quad \frac{\Gamma,C^+[\textbf{noop};P_2]:\Lambda}{\Gamma,C^+[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2]:\Lambda}\quad (E\approx 0)$$

$$\textsc{HighWhileC}\quad \frac{\Gamma,C^+[\textbf{noop};P;\textbf{while}\ E\ \textbf{do}\ P],C^+[\textbf{noop}]:\Lambda}{\Gamma,C^+[\textbf{while}\ E\ \textbf{do}\ P]:\Lambda}$$

$$\textsc{LowWhileC}\quad \frac{\Gamma,C^+[\textbf{noop};P;\textbf{while}\ E\ \textbf{do}\ P]:\Lambda \quad \Gamma,C^+[\textbf{noop}]:\Lambda}{\Gamma,C^+[\textbf{while}\ E\ \textbf{do}\ P]:\Lambda}\quad (\neg \mathit{dep}(E,\overline{\Lambda}))$$

$$\textsc{TrueWhileC}\quad \frac{\Gamma,C^+[\textbf{noop};P;\textbf{while}\ E\ \textbf{do}\ P]:\Lambda}{\Gamma,C^+[\textbf{while}\ E\ \textbf{do}\ P]:\Lambda}\quad (E\not\approx 0)\qquad \textsc{FalseWhileC}\quad \frac{\Gamma,C^+[\textbf{noop}]:\Lambda}{\Gamma,C^+[\textbf{while}\ E\ \textbf{do}\ P]:\Lambda}\quad (E\approx 0)$$

$$\textsc{ParC}\quad \frac{\Gamma,C^+[P_1 \mathbin{\|\!\lfloor} P_2]\ \textbf{or}\ C^+[P_2 \mathbin{\|\!\lfloor} P_1]:\Lambda}{\Gamma,C^+[P_1 \parallel P_2]:\Lambda}\quad (P_1,P_2\not\equiv \textbf{stop})\qquad \textsc{OrC}\quad \frac{\Gamma,C^+[P_1]\ \textbf{or}\ C^+[P_2]:\Lambda}{\Gamma,C^+[P_1\ \textbf{or}\ P_2]:\Lambda}\quad (P_1,P_2\not\equiv \textbf{stop})$$

Figure 4. Tableau system for strong Λ-bisimilarity: Local rules

Tableau System The local tableau rules are shown on figure 4. Other rules such as those for sequential composition are derivable:

$$\textsc{Seq1C}\quad \frac{\Gamma,P_1;P_2 \mathbin{\|\!\lfloor} \textbf{stop}:\Lambda}{\Gamma,P_1;P_2:\Lambda}$$

$$\textsc{Seq2C}\quad \frac{\Gamma,C[P_1;(P_2;P)\mathbin{\|\!\lfloor} Q]:\Lambda}{\Gamma,C[(P_1;P_2);P\mathbin{\|\!\lfloor} Q]:\Lambda}$$

LEMMA 3 (Local Soundness). *If $\Gamma:\Lambda$ is provable using the rules in figure 4 then $\models \Gamma:\Lambda$.*

PROOF We show that each of the rules of figure 4 preserve validity.
CONGC: Since $\equiv\subseteq\cong_\Lambda$.
HIGHASSNC: Suppose $\{x_1,\dots,x_n\}\cap\Lambda=\emptyset$, and assume $C_i^{\|}[\textbf{stop}]\cong_\Lambda C_j^{\|}[\textbf{stop}]$ and $Q_i\cong_\Lambda Q_j$ whenever $i,j\in[1,\dots,n]$. We show that $C_i[x_i:=E_i]$ **or** $Q_i\cong_\Lambda C_j[x_j:=E_j]$ **or** Q_j. So assume that $\sigma_i\approx_\Lambda\sigma_j$, and that $(C_i[x_i:=E_i]\ \textbf{or}\ Q_i,\sigma_i)\to(Q_i',\sigma_i')$. By Lemma 2, either $(Q_i,\sigma_i)\to(Q_i',\sigma_i')$ or $Q_i'\equiv C_i^{\|}[\textbf{stop}]$ and $\sigma_i'=\sigma_i[\sigma_i(E_i)/x_i]$. In either case we find Q_j' and σ_j' such that $Q_i'\cong_\Lambda Q_j'$, $\sigma_i'\approx_\Lambda\sigma_j'$, and $(C_j[x_j:=E_j]\ \textbf{or}\ Q_j,\sigma_j)\to(Q_j',\sigma_j')$.
LOWASSNC: This case is similar to the previous one, except that here the assumptions $x\in\Lambda$ and $\neg\mathit{dep}(E,\overline{\Lambda})$ are used to ensure that $\sigma_i[\sigma_i(E)/x]\approx_\Lambda\sigma_j[\sigma_j(E)/x]$.
HIGHCONDC: It suffices to show that if $P\cong_\Lambda C^+[\textbf{noop};P_1]$ and $P\cong_\lambda C^+[\textbf{noop};P_2]$ then $P\cong_\Lambda C^+[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2]$. The argument is straightforward.
PARC: It is sufficient to show that $C^+[P_1\mathbin{\|\!\lfloor}P_2]$ **or** $C^+[P_2\mathbin{\|\!\lfloor}P_1]\cong_\Lambda C^+[P_1\parallel P_2]$ whenever $P_1\not\equiv\textbf{stop}$ and $P_2\not\equiv\textbf{stop}$. This is straigthforward.
The remaining cases are instances of cases already treated. □

Rule of Discharge The proof system is completed by adding a single coinductive rule of discharge, for loop detection. The rule of discharge works as follows. Let Δ be a multiset of judgments $\Gamma':\Lambda'$ each labelling an undischarged node $\overline{n}'$ in a proof tree with root node $\overline{n}$, labelled $\Gamma:\Lambda$. The assumption occurrence $\Gamma':\Lambda'$ is *dischargeable* for such a proof tree, if $\Gamma'=\Gamma$, $\Lambda'=\Lambda$, and the path from $\overline{n}$ to $\overline{n}'$ in the proof tree passes the left branch of one of the proof rules HIGHASSNC or LOWASSNC. If this is the case we say that $\overline{n}'$ is a *discharged leaf*, and $\overline{n}$ its *companion*. We write this rule in standard natural deduction style as

$$\begin{array}{c}[\Gamma':\Lambda']\\ \vdots\end{array}$$
$$\textsc{Disch}\quad \frac{\Gamma:\Lambda}{\Gamma:\Lambda}\quad (\Gamma':\Lambda' \text{ is dischargeable})$$

The complete tableau system thus consists of the local rules in figure 4 together with the rule DISCH, and $\Delta\vdash\Gamma:\Lambda$ denotes the existence of a proof of $\Gamma:\Lambda$ from the set of undischarged assumption occurrences Δ in this system. In particular, $\vdash\Gamma:\Delta$ denotes derivability where all assumptions are discharged.

EXAMPLE 2. *For the sake of the example let* **if** *abbreviate the conditional* **if** $h=0$ **then** h_1 **else stop**. *We give a proof of the judgment*

$$\textbf{if}\parallel\textbf{loop}\ h_2:\Lambda\ . \tag{2}$$

First, (2) is reduced using PARC *and* CONGC *to*

$$(\textbf{if}\mathbin{\|\!\lfloor}\textbf{loop}\ h_2)\ \textbf{or}\ (\textbf{loop}\ h_2\mathbin{\|\!\lfloor}\textbf{if}):\Lambda, \tag{3}$$

and then using TRUEWHILEC, HIGHCONDC *(and* CONGC*) to*

$$\begin{aligned}&((\textbf{noop};h_1)\mathbin{\|\!\lfloor}\textbf{loop}\ h_2)\ \textbf{or}\ ((\textbf{noop};h_2;\textbf{loop}\ h_2)\mathbin{\|\!\lfloor}\textbf{if}),\\ &(\textbf{noop}\mathbin{\|\!\lfloor}\textbf{loop}\ h_2)\ \textbf{or}\ ((\textbf{noop};h_2;\textbf{loop}\ h_2)\mathbin{\|\!\lfloor}\textbf{if}):\Lambda\ .\end{aligned} \tag{4}$$

By HIGHASSNC, *(4) is reduced to the following two subgoals:*

$$h_1\parallel\textbf{loop}\ h_2,\textbf{stop}\parallel\textbf{loop}\ h_2:\Lambda \tag{5}$$
$$(\textbf{noop};h_2;\textbf{loop}\ h_2)\mathbin{\|\!\lfloor}\textbf{if},(\textbf{noop};h_2;\textbf{loop}\ h_2)\mathbin{\|\!\lfloor}\textbf{if}:\Lambda \tag{6}$$

Subgoal (5) is further reduced by PARC, TRUEWHILEC, CONGC, *and* HIGHASSNC *to the two subgoals:*

$$\textbf{loop}\ h_2, h_2; \textbf{loop}\ h_2 : \Lambda \tag{7}$$

$$(h_2; \textbf{loop}\ h_2) \parallel h_1, h_2; \textbf{loop}\ h_2 : \Lambda \tag{8}$$

We proceed to reduce (8) as before obtaining two further subgoals:

$$\textbf{stop} \parallel (h_2; \textbf{loop}\ h_2), \textbf{loop}\ h_2 : \Lambda \tag{9}$$

$$\textbf{loop}\ h_2 \parallel h_1, \textbf{loop}\ h_2 : \Lambda \tag{10}$$

Note that even if (9) is equal to (7) up to CONGC, *(9) cannot (yet) be discharged, since the two nodes are not connected by a path in the tableau. However, (10) can be rewritten using* CONGC *such that it becomes dischargable against (5), since the two nodes are connected by a path which traverses the left branch of* HIGHASSNC *(in the derivation of (10) from (8)). Proofs of the remaining (easy) proof goals are omitted.*

8. Soundness and Relative Completeness

THEOREM 4 (Soundness). *If* $\vdash \Gamma : \Lambda$ *then* $\models \Gamma : \Lambda$.

PROOF (Sketch) Define the relation R by PRQ just in case $P, Q \in \Gamma$ such that $\Gamma : \Lambda$ labels some node $\overline{n}$ in a proof of some judgment $\Gamma_0 : \Lambda_0$. Since all paths from a companion node to a leaf node must pass the left hand branch of a node labelled LOWASSNC or HIGHASSNC, it may be assumed that the same also applies to all paths from $\overline{n}$ to a discharged leaf. We show that R is a Λ-bisimulation. To this end suppose that $\sigma_1 \approx_\Lambda \sigma_2$ and that $(P, \sigma_1) \to (P', \sigma_1')$. We show that $(Q, \sigma_2) \to (Q', \sigma_2')$ for some Q', σ_2' such that $P'RQ'$. Suppose now that $\overline{n}$ is obtained by an application of LOWASSNC. Then P has the form $C_i[x := E]$ **or** P_i and Q has the form $C_j[x := E]$ **or** Q_i. By lemma 2, either $P' = C_i^{\parallel}[\textbf{stop}]$ or $(P_i, \sigma_1) \to (P', \sigma_1')$. In the first case we obtain that $(Q, \sigma_2) \to (C_j^{\parallel}[\textbf{stop}], \sigma_2')$. Moreover, $C_i^{\parallel}[\textbf{stop}]\ R\ C_j^{\parallel}[\textbf{stop}]$ by construction, and $\sigma_1' \approx_\Lambda \sigma_2'$. The second case is concluded by the induction hypothesis. The case for HIGHASSNC is similar. The remaining cases (DISCH included) are straightforward. One example is sufficient to typify the argument. If $\overline{n}$ is obtained by an application of HIGHCONDC so that $P = C_i[\textbf{if}\ E\ \textbf{then}\ P_{i,1}\ \textbf{else}\ P_{i,2}]$ **or** P_i and $\sigma_1(E) \neq 0$, either $(\textbf{noop}; P_{i,1}, \sigma_1) \to (P', \sigma_1')$ or $(P_i, \sigma_1) \to (P_i', \sigma_1')$. Q is either in Γ, or $Q = P$. In the former case we are immediately done by the induction hypothesis. The latter case is resolved depending on whether $\sigma_2(E) = 0$ or not. The details are left to the reader. □

THEOREM 5 (Relative Completeness). *For decidable expression theories* $\mathcal{E}$, *if* $\models \Gamma : \Lambda$ *then* $\vdash \Gamma : \Lambda$

PROOF (Sketch) Assume given an arbitrary valid judgment $\Gamma : \Lambda$. We first show how to construct a *proof segment* for $\Gamma : \Lambda$, a (non-recursive) proof tree with root $\overline{n}$ labelled by $\Gamma : \Lambda$, possibly involving a set of unproved leaf nodes. Each such leaf node $\overline{n}'$ has the property that the path from $\overline{n}$ to $\overline{n}'$ traverses the left hand branch of one the rules LOWASSNC or HIGHASSNC. Write $\Gamma \Rightarrow \Gamma'$ if such a path exists.

For the construction note first that, using CONGC together with the rules for conditionals, while, parallel, and choice, all members of Γ can be assumed to have the form $C_1[P_1]$ **or** $\cdots$ **or** $C_n[P_n]$ such that the P_i are either α, **stop**, $P_{i,1}; P_{i,2}$, or $P_{i,1} \mathbin{\lfloor\!\lfloor} P_{i,2}$. The case for $P_i =$ **stop** and $C_i[]$ not the identity context is eliminated using $\equiv$, specifically the congruences $\textbf{stop}; P \equiv P; \textbf{stop}$ and $\textbf{stop}; \textbf{stop} \equiv \textbf{stop}$, and $\textbf{stop} \mathbin{\lfloor\!\lfloor} P \equiv \textbf{stop}$. The case for sequential composition is eliminated by ;-associativity, and the case of left merge is eliminated by rewriting $P_{i,1} \mathbin{\lfloor\!\lfloor} P_{i,2} \equiv P_{i,1}; \textbf{stop} \mathbin{\lfloor\!\lfloor} P_{i,2}$. Thus, all members of Γ can be assumed to have the form either $C_1[\alpha_1]$ **or** $\cdots$ **or** $C_n[\alpha_n]$ or **stop**. Note now that these two cases are incompatible, since $\Gamma : \Lambda$ is valid and all transformations using $\equiv$ preserve validity. Thus, either $\Gamma = \{\textbf{stop}\}$ or $\textbf{stop} \notin \Gamma$. In the former case the proof construction is completed using STOPC. In the latter case let Γ have the form $Q_1, \ldots, Q_m$. Since $\Gamma : \Lambda$ is valid, using the semilattice properties of **or** and **stop** under $\equiv$, each Q_i can be written in the form $C_{i,1}[\alpha_{i,1}]$ **or** $\cdots$ **or** $C_{i,n}[\alpha_{i,n}]$ where n is fixed, such that for all $j : 1 \leq j \leq n$, $C_{1,j}^{\parallel}[\textbf{stop}], \ldots, C_{m,j}^{\parallel}[\textbf{stop}] : \Lambda$ is valid. Reflecting this, the case is completed by n sequential applications of either LOWASSNC or HIGHASSNC. This step uses a case analysis similar to the one in the proof of Theorem 2, using the property that if $\neg dep(E, \overline{\Lambda})$ and $E \approx_\Lambda E'$ then $E \approx E'$.

FACT 1. *The segment construction procedure terminates and produces a proof segment.* □

The complete proof search procedure iterates the segment construction procedure, and terminates each branch as soon as the rule of discharge becomes applicable. To prove termination we use a closure construction. Define

$$cl(\Gamma) = \bigcup_{n \in \omega} cl_n(\Gamma)\ ,$$

and let cl_n be determined by the following conditions:

1. $\textbf{stop} \in cl_0(\Gamma)$
2. $C[\alpha] \in cl_n(\Gamma)$ implies $C^{\parallel}[\textbf{stop}] \in cl_{n+1}(\Gamma)$
3. $C[\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2] \in cl_n(\Gamma)$ implies $C[\textbf{noop}; P_1], C[\textbf{noop}; P_2] \in cl_{n+1}(\Gamma)$
4. $C[\textbf{while}\ E\ \textbf{do}\ P] \in cl_n(\Gamma)$ implies $C[\textbf{noop}], C[\textbf{noop}; P; \textbf{while}\ E\ \textbf{do}\ P] \in cl_{n+1}(\Gamma)$
5. $C[P_1 \parallel P_2] \in cl_n(\Gamma)$ implies $C[P_1 \mathbin{\lfloor\!\lfloor} P_2], C[P_2 \mathbin{\lfloor\!\lfloor} P_1], C[P_1], C[P_2] \in cl_{n+1}(\Gamma)$
6. $C[P_1\ \textbf{or}\ P_2] \in cl_n(\Gamma)$ implies $C[P_1], C[P_2] \in cl_{n+1}(\Gamma)$
7. $C[P \mathbin{\lfloor\!\lfloor} Q] \in cl_n(\Gamma)$, $P \neq P_1; P_2$ for any $P_1, P_2 \in \mathcal{P}$ implies $C[P; \textbf{stop} \mathbin{\lfloor\!\lfloor} Q] \in cl_{n+1}(\Gamma)$
8. $C[\textbf{stop}; P \mathbin{\lfloor\!\lfloor} Q] \in cl_n(\Gamma)$ implies $C[P \mathbin{\lfloor\!\lfloor} Q] \in cl_{n+1}(\Gamma)$
9. $C[(P_1; P_2); P_3] \in cl_n(\Gamma)$ implies $C[P_1; (P_2; P_3)] \in cl_{n+1}(\Gamma)$

Say that $\Gamma = P_1, \ldots, P_n$ is *A-generated*, if each P_i, $1 \leq i \leq n$, can be written $P_{i,1}$ **or** $\cdots$ **or** P_{i,m_i} such that $P_{i,j} \in A$ for all $j : 1 \leq j \leq m_i$, and say that the set $A \subseteq \mathcal{P}$ is *closed*, if $A = cl(A)$.

LEMMA 4. *If* Γ *is A-generated for some closed set* A *and* $\Gamma \Rightarrow \Gamma'$ *then* Γ' *is A-generated.*

PROOF By inspecting the segment construction procedure. □

It is thus sufficient to prove that $cl(A)$ is finite when A is. To this end we define a measure $|P|_B$ where B is a set of terms in $\mathcal{P}$ such that $|P|_B = 0$ if $P \in B$, and if $P \notin B$ then:

$$\begin{array}{rcl}
|P \parallel Q|_B &=& (|P|_B + 1) \cdot (|Q|_B + 1) \\
|P \mathbin{\lfloor\!\lfloor} Q|_B &=& |P \parallel Q|_B \\
|\textbf{if}\ E\ \textbf{then}\ P_1\ \textbf{else}\ P_2|_B &=& |P_1|_B + |P_2|_B + 2 \\
|\textbf{while}\ E\ \textbf{do}\ P|_B &=& |P|_B + 2 \\
|\alpha|_B &=& 1 \\
|\textbf{stop}|_B &=& 0 \\
|P_1; P_2|_B &=& |P_1|_B + |P_2|_B \\
|P_1\ \textbf{or}\ P_2|_B &=& |P_1|_B + |P_2|_B + 1
\end{array}$$

Clearly, all closure conditions are non-increasing, in the sense that if $P \in cl_{n+1}(\Gamma) \setminus cl_n(\Gamma)$ then $|P|_{B \cup \{Q\}} \leq |Q|_B$ for some

$Q \in cl_n(\Gamma)$. Moreover, the only non-decreasing conditions are conditions 2, 7, 8 and 9. It is, however, easy to see that these four conditions can only be applied consecutively a finite number of times before one of the other, strictly decreasing, conditions must be applied. This is sufficient to complete the termination argument, and thus the proof of theorem 5. $\square$

Complexity The tableau construction algorithm gives an alternative to the decision procedure of theorem 1. The set $cl(\{P\})$ is single exponential in the size n of P in the worst case. This gives a double exponential bound on the size of judgments and hence a triple exponential bound on the size of the tableau. This is worse than the single exponential bound for the decision procedure of section 5. However, the tableau construction algorithm is likely to explore only a small part of the potential state space in practice. Moreover there is potential for savings. By restricting attention to sequential programs one exponential is cut from the worst case complexity. A more intelligent handling of sharing may cut another exponential. In practice it is thus far from clear that the direct algorithm of section 5 will always outperform the proof search based algorithm of this section, in spite of its asymptotic superiority.

9. Dynamic Security Levels

Strong Λ-bisimulation, along with other notions of noninterference we have been able to identify in the literature, assumes a static assignment of security levels to identifiers. In many situations such a static level assignment is either not available, or it is not meaningful, because some form of dynamic upgrading or downgrading needs to be explicitly supported. In this section we lift the assumption of a static level assignment, and adapt the security definition accordingly. The main challenge in doing so is to identify a suitable mechanism that allows the set Λ of low identifiers in the definition of strong Λ-bisimilarity to change while preserving the desired end-to-end information flow properties.

Localized Level Change Policies There is a wide spectrum of possible approaches (cf. [32] for a recent survey). Here, the aim is to model information release policies in the style of intransitive noninterference [23, 6, 26, 22] where declassification is required to be localized to specific program points and operations, but no constraints are imposed on the nature of information actually released (the "where"-dimension of [32]). In other words, the decision whether a downgrading of x from high level to low at some given program execution point should be permitted or not should not depend on the information actually held by x at that point, but only on whether the operation or principal invoked to perform the downgrading is actually authorized for this. This type of policy is in constrast to more fine-grained information release policies in the style of admissible interference [17], the declassification types of [7] or [20], or the delimited release model of Sabelfeld and Myers [29].

Approach To accomodate level changing operations we augment the syntax of $\mathcal{P}$ with primitive operations a such that now

$$\alpha ::= x := E \mid a \mid$$

Three examples are considered in this paper:

- *down*(x) and *up*(x) are used for identifier downgrading and upgrading, respectively.
- $[x := y]$ is a regrading assignment [23] which acts as a normal assignment, but is always authorized to go through, regardless of the current levels of x and y.

Other examples can easily be conceived, such as a "panic" operation that upgrades all identifiers to high, a "publicize" operation which downgrades all high identifiers to low, or a conditional downgrade operation which lets the level of x be governed by the value of another identifier, say low_x. Another variant would be a timed downgrade operation that automatically downgrades an identifier once a certain time interval has lapsed, or once some (e.g. cryptographic) operation has been performed a sufficient number of times. In the conclusion we comment on our frameworks ability to handle these types of operation.

To each level operation a is associated the pre-post relation $\|a\|(\sigma, \sigma')$. Specifically, for $a = down(x)$ or $a = up(x)$, $\|a\|(\sigma, \sigma')$ iff $\sigma = \sigma'$, and $\|[x := y]\|(\sigma, \sigma') = \|x := y\|(\sigma, \sigma')$. The transition rule is then the obvious one:

$$(\text{LVL}) \quad \frac{\|a\|(\sigma, \sigma')}{(a, \sigma) \xrightarrow{a} (\textbf{stop}, \sigma')}$$

For uniformity of notation we write $(P, \sigma) \xrightarrow{x:=E} (P', \sigma')$ if $(P, \sigma) \to (P', \sigma')$ by elaborating either the assignment $x := E$, or a conditional or while command, and then $x := E = \textbf{noop}$.

In order to capture the effect of level operations on the level assignment we associate to each primitive operation α and each set Λ the following two (effective) operations:

- *post*(Λ, α), the *post-set*, is the set of low identifiers in the post-state which should depend only on the value of low identifiers in the pre-state.
- *upd*(Λ, α), the *level updater*, is the set of low identifiers in the post-state after "administrative" level changes (upgradings or downgradings) have been completed.

The distinction between the *post* and *upd* operations is a little subtle. The post-set is needed to provide a way for high-level data to become available at low level in connection with a permitted downgrade operation. For instance, for the regrading assignment $[x := y]$, even if $\sigma_1 \approx_\Lambda \sigma_2$, when x is low and y is high the stores resulting from the regrading assignment, $\sigma_1[x := \sigma_1(y)]$ and $\sigma_2[x := \sigma_2(y)]$, are Λ-equivalent only when $\sigma_1(y) = \sigma_2(y)$. Thus, to allow the regrading assignment to go through, a suitably adapted version of Λ-bisimulation must, in the post-state, amend the set Λ by removing from it the target x of the regrading assignment. In this case we thus define:

$$\textit{post}(\Lambda, [x := y]) = \begin{cases} \Lambda \setminus \{x\} & \text{if } x \in \Lambda, y \notin \Lambda, \\ \Lambda & \text{otherwise} \end{cases}$$

On the other hand, after executing the regrading assignment the security levels should remain unchanged, whence

$$\textit{upd}(\Lambda, [x := y]) = \Lambda \ .$$

The identifier regrading operations *up*(x) and *down*(x) are simpler. These cases do not involve interferent information flow, only update of the level assignment in the poststate, so that:

$$\begin{array}{rcl} \textit{post}(\Lambda, \textit{up}(x)) & = & \Lambda \\ \textit{upd}(\Lambda, \textit{up}(x)) & = & \Lambda \setminus \{x\} \\ \textit{post}(\Lambda, \textit{down}(x)) & = & \Lambda \\ \textit{upd}(\Lambda, \textit{down}(x)) & = & \Lambda \cup \{x\} \end{array}$$

Finally, when α is an ordinary assignment $x := E$, *post*$(\Lambda, x := E) =$ *upd*$(\Lambda, x := E) = \Lambda$. That is, an assignment $x := E$ is not allowed to directly copy information from high level to low level (*post*$(\Lambda, x := E) = \Lambda$), and after executing the assignment $x := E$ the level assignment remains unchanged (*upd*$(\Lambda, x := E) = \Lambda$).

Strong Dynamic Λ-Bisimulation We next generalize strong Λ-bisimulation to dynamic security levels. Since the primitive transition labels now have observable effects in terms of level changes it now becomes necessary to reflect this in the generalized definition. For this purpose say that the labels α and α' are Λ-*compatible*,

$\alpha \equiv_\Lambda \alpha'$, if the level changing effects of α and α' are the same for Λ, i.e. $\mathit{post}(\Lambda, \alpha) = \mathit{post}(\Lambda, \alpha')$ and $\mathit{upd}(\Lambda, \alpha) = \mathit{upd}(\Lambda, \alpha')$.

DEFINITION 5 (Strong Dynamic Λ-Bisimulation). *Let an indexed family $R = \{R_\Lambda\}_{\Lambda \subseteq \mathit{Ide}}$ of binary relations on control states be given. The family R is a* strong dynamic Λ-bisimulation, *if each relation R_Λ is symmetric, and whenever $s_1\ R_\Lambda\ s_2$ then for all $\sigma_1 \approx_\Lambda \sigma_2$, if $(s_1, \sigma_1) \xrightarrow{\alpha_1} (s_1', \sigma_1')$ then there are s_2', σ_2' such that $(s_2, \sigma_2) \xrightarrow{\alpha_2} (s_2', \sigma_2')$, $\alpha_1 \equiv_\Lambda \alpha_2$, $\sigma_1' \approx_{\mathit{post}(\Lambda,\alpha)} \sigma_2'$, and $s_1'\ R_{\mathit{upd}(\Lambda,T_1)}\ s_2'$. The control states s_1 and s_2 are strongly dynamic Λ-bisimilar, $s_1 \cong_{d,\Lambda} s_2$, if there is a strong dynamic Λ-bisimulation R_Λ such that $s_1\ R_\Lambda\ s_2$, and say that the state s is* strongly dynamic Λ-secure, *if $s \cong_{d,\Lambda} s$.*

Again we normally refer to strong dynamic Λ-bisimulation as just dynamic Λ-bisimulation. As before we easily see that $\cong_{d,\Lambda}$ is itself a dynamic Λ-bisimulation, and that it is a per. Since the definition now is a bit more complex, we prove the latter statement.

PROPOSITION 3. *For each set Λ, the relation $\cong_{d,\Lambda}$ is a per.*

PROOF If R_Λ is a dynamic Λ-bisimulation then trivially so is R_Λ^{-1}. Assume that R_1 and R_2 are both dynamic Λ-bisimulations. We show that the family $\{R_\Lambda \mid R_\Lambda = R_{1,\Lambda} \circ R_{2,\Lambda}\}$ is also a dynamic Λ-bisimulation (where $\circ$ is relational composition). Suppose $s_1 R_\Lambda s_3$, $(s_1, \sigma_1) \xrightarrow{\alpha_1} (s_1', \sigma_1')$ and $\sigma_1 \approx_\Lambda \sigma_2$. By the definition of R we find s_2 such that $s_1 R_{1,\Lambda} s_2 R_{2,\Lambda} s_3$. Since R_1 is a dynamic Λ- bisimulation we find $(s_2, \sigma_2) \xrightarrow{\alpha_2} (s_2', \sigma_2')$ such that $\alpha_1 \equiv_\Lambda \alpha_2$, $\sigma_1' \approx_{\mathit{post}(\Lambda,\alpha_1)} \sigma_2'$ and $s_1'\ R_{\mathit{upd}(\Lambda,\alpha_1)}\ s_2'$. Then, since R_2 is a dynamic Λ-bisimulation, and since $\sigma_2 \approx_\Lambda \sigma_2$ we find $(s_3, \sigma_2) \xrightarrow{\alpha_3} (s_3', \sigma_3')$ such that $\alpha_3 \equiv_\Lambda \alpha_2$, $\sigma_2' \approx_{\mathit{post}(\Lambda,\alpha_2)} \sigma_3'$ and $s_2'\ R_{\mathit{upd}(\Lambda,\alpha_2)}\ s_3'$. By the requirements for *post* and *upd* we obtain that $\mathit{post}(\Lambda, \alpha_1) = \mathit{post}(\Lambda, \alpha_2)$ and $\mathit{upd}(\Lambda, \alpha_1) = \mathit{upd}(\Lambda, \alpha_2)$, from where we can conclude that $\sigma_1' \approx_{\mathit{post}(\Lambda,\alpha)} \sigma_3'$ and $s_1'\ R_{\mathit{upd}(\Lambda,\alpha_1)}\ s_3'$ as desired. $\square$

Trivially we also obtain that dynamic Λ-bisimilarity is a generalization of Λ-bisimilarity, for the special case where Λ is constant (the conservativity property of [32]).

EXAMPLE 3.

1. $[l_1 := h]; l_2 := l_1$
   The program is secure. The assignment to l_1 is harmless since l_1 is removed from the set Λ when evaluating the post-set, and it is reinstated by the level updater.
2. $[l_1 := h]; l_2 := h$
   This program is insecure since the second assignment copies directly into Λ.
3. $\mathit{down}(h); l := h$
   The program is secure since, after downgrading, h becomes low.
4. **if** $h = 0$ **then** $h_1 := 1$ **else** $\mathit{up}(l)$
   The program is insecure since we find a Λ such that $h_1 := 1 \not\equiv_\Lambda \mathit{up}(1)$.
5. $\mathit{up}(l_1); l_2 := l_1$
   The program is insecure since l_1 is high after upgrading (even if, in a purely sequential environment, the value of l_1 immediately after the upgrade is known).
6. **if** $h = 0$ **then** $[l := h_1]$ **else** $[l := h_2]$*:*
   This program is secure since the actions $[l := h_1]$ and $[l := h_2]$ are Λ-compatible for all Λ, and whenever $\sigma_1 \approx_\Lambda \sigma_2$ then $\sigma_1[\sigma_1(h)/l] \approx_{\Lambda\setminus\{l\}} \sigma_2[\sigma_2(h)/l]$.

Example 3.6 shows the key point where our definition differs from that of Mantel and Sands [23]. They argue that the program of ex. 3.6 should be rejected since it leaks not only the value of h_1 or h_2 at the prescribed control points, but also information about h; in this sense the declassifications in ex. 3.6 are not localized. Our account is weaker: At the point of declassification, h_1 or h_2 may or may not contain the bit $h = 0$, the policy does not specify which, neither how that bit was derived. As in Rushby's account of intransitive noninterference [26] we make the implicit worst case assumption that once one high bit is leaked, all bits are, since nothing binds the leaked high bit to any particular piece of information. By this intuition, an indirect flow such as that of 3.6 would be as legitimate as the direct flow in $h_1 := (h = 0); [l := h_1]$ or $h_1 := (h = 0) \parallel [l := h_1]$.

Interestingly, the discrepancy with the Mantel and Sands security condition is localized to just this feature: Other than examples, none of their results are affected by replacing their security definition by our notion of dynamic Λ-security (and adapting the other definitions accordingly).

In this connection it is important, though, to make sure our definition makes sense also when the security lattice is lifted beyond the standard two-point lattice. A concern might be the example

$$\textbf{if } t = 0 \textbf{ then } [l := h_1] \textbf{ else } [l := h_2] \tag{11}$$

where t belongs to a third security level (top secret) above high. In this case the security definition would be amended by requiring dynamic Λ-security not only where Λ is the set of low identifiers, but also where it is the set of identifiers that are high or below. Since in the latter case the post and update functions on downgrading commands $[l := h]$ would leave Λ unchanged, the program (11) would be immediately rejected.

10. Decidability and Relative Completeness

In this section we adapt the decidability and relative completeness results from Λ-bisimilarity to dynamic Λ-bisimilarity. Proofs are adaptations of the corresponding proofs in sections 5 and 8.

DEFINITION 6 (Effective Post-Separability). *The object automaton $\mathcal{A}$ is* effectively post-separable, *if given $s_1, s_1', s_2, s_2' \in S$ and $\Lambda \subseteq \mathit{Ide}$ it is decidable if there is a transition label α_1 and stores $\sigma_1, \sigma_1', \sigma_2$ such that*

1. *$(s_1, \sigma_1) \xrightarrow{\alpha_1} (s_1', \sigma_1')$*
2. *$\sigma_1 \approx_\Lambda \sigma_2$*
3. *whenever $(s_2, \sigma_2) \xrightarrow{\alpha_2} (s_2', \sigma_2')$ then either $\alpha_1 \not\equiv_\Lambda \alpha_2$, or $\sigma_1' \not\approx_{\mathit{post}(\Lambda,\alpha_1)} \sigma_2'$.*

THEOREM 6. *Let an object automaton $\mathcal{A}$ be given. If $\mathcal{A}$ is finite-control and effectively post-separable then strong dynamic Λ-bisimilarity is decidable.*

PROOF We construct a pointwise decreasing chain

$$R = \{R_{i,\Lambda}\}_{1 \le i \le n, \Lambda \subseteq \mathit{Ide}}$$

of families, similar to the chain $R_0 \supseteq \cdots \supseteq R_n$ in the proof of theorem 1. Condition (*) in the proof of theorem 1 is now replaced by the following:

(**) There is $s_1' \in S$, label α_1 and control stores $\sigma_1, \sigma_1', \sigma_2$ such that $(s_1, \sigma_1) \xrightarrow{\alpha_1} (s_1', \sigma_1')$ and $\sigma_2 \approx_\Lambda \sigma_1$, but whenever $(s_2, \sigma_2) \xrightarrow{\alpha_2} (s_2', \sigma_2')$ then either $\alpha_1 \not\approx_\Lambda \alpha_2$, $\sigma_2' \not\approx_{\mathit{post}(\Lambda,\alpha_1)} \sigma_1'$ or $\neg(s_1' R_{i,\mathit{upd}(\Lambda,\alpha_1)} s_2')$.

The result now follows from effective post-separability as in the proof of theorem 1. $\square$

To extend the proof of decidability for parallel while-programs to strong dynamic Λ-bisimilarity we extend the relation $\sim_\Lambda$ of section 5 by the symmetric closure of the following clauses:

4. If α_1 and α_2 are both signals and $\alpha_1 \equiv_\Lambda \alpha_2$ then $\alpha_1 \sim_\Lambda \alpha_2$.

5. If α_1 is a signal, *post*$(\Lambda, \alpha_1) =$ *upd*$(\Lambda, \alpha_1) = \Lambda$, α_2 is a regrading assignment $[x := y]$, $x \notin \Lambda$ then $\alpha_1 \sim_\Lambda \alpha_2$.
6. If α_1 is a signal, *post*$(\Lambda, \alpha_1) =$ *upd*$(\Lambda, \alpha_1) = \Lambda$, α_2 is an assignment $x := E$, $x \in \Lambda$, and $E \approx x$ then $\alpha_1 \sim_\Lambda \alpha_2$.
7. If α_1 is a signal, *post*$(\Lambda, \alpha_1) =$ *upd*$(\Lambda, \alpha_1) = \Lambda$, α_2 is an assignment $x := E$, $x \notin \Lambda$ then $\alpha_1 \sim_\Lambda \alpha_2$.
8. If α_1, α_2 are identical regrading assignments $[x := y]$ then $\alpha_1 \sim_\Lambda \alpha_2$.
9. If α_1, α_2 are regrading assignments, respectively $[x_1 := E_1]$ and $[x_2 := E_2]$, and $x_1, x_2 \notin \Lambda$, then $\alpha_1 \sim_\Lambda \alpha_2$.
10. If α_1 is a regrading assignment $[x := y]$, $x, y \in \Lambda$, α_2 is an assignment $x := E$ and $y \approx E$ then $\alpha_1 \sim_\Lambda \alpha_2$
11. If α_1 is a regrading assignment $[x_1 := y]$, $x \notin \Lambda$, and α_2 is an assignment $x_2 := E$, $x_2 \notin \Lambda$, then $\alpha_1 \sim_\Lambda \alpha_2$.

Observe that $\sim_\Lambda$ continues to be effective with these extensions. The proof of the correctness lemma for $\sim_\Lambda$ is a straightforward case analysis and left out.

LEMMA 5. *$\alpha_1 \cong_{d,\Lambda} \alpha_2$ iff $\alpha_1 \sim_\Lambda \alpha_2$ with the extended definition of $\sim_\Lambda$.* □

THEOREM 7. *For parallel while-programs, if $\mathcal{E}$ is decidable then so is strong dynamic Λ-bisimilarity.*

PROOF The non-trivial task is to show effective post-separability. As in the proof of theorem 2, assume given program terms P_1, P_1', P_2, P_2' such that $P_1 \rhd P_1'$ and $P_2 \rhd P_2'$, and set Λ. Let $\alpha_1 =$ *label*(P_1, P_1') and $\alpha_2 =$ *label*(P_2, P_2') and the result follows by Lemma 5. □

For the tableau system it is sufficient to add the following two rules:

$$\textsc{ActC} \quad \frac{\Gamma, C^+[\alpha_1] : \Lambda}{\Gamma, C^+[\alpha_2] : \Lambda} \quad \alpha_1 \sim_\Lambda \alpha_2$$

$$\textsc{CmdC} \quad \frac{C_1^{\|}[\textbf{stop}], \ldots, C_n^{\|}[\textbf{stop}] : \textit{upd}(\Lambda, a) \quad Q_1, \ldots, Q_n : \Lambda}{C_1[a] \textbf{ or } Q_1, \ldots, C_n[a] \textbf{ or } Q_n : \Lambda}$$

Let $\vdash_d \Gamma : \Lambda$ if $\Gamma : \Lambda$ is provable in the extended tableau system, and let $\models_d \Gamma : \Lambda$ is $\Gamma : \Lambda$ is correspondingly valid. Soundness and relative completeness is proved by minor adaptations of the proofs of theorems 4 and 5. We leave out the details.

THEOREM 8 (Soundness and relative completeness). *If $\vdash_d \Gamma : \Lambda$ then $\models_d \Gamma : \Lambda$, and if for decidable expression theories $\mathcal{E}$, if $\models_d \Gamma : \Lambda$ then $\vdash \Gamma : \Lambda$.* □

11. Discussion

We have demonstrated that decidability results and sound and (relative) complete axiomatizations are possible in the area of language-based security. We have also specifically provided support to Sabelfeld and Sands notion of strong low bisimulation as an interesting and tractable model for noninterference, contrasting with the undecidability results that apply in the case of other, flow-sensitive noninterference relations based on execution traces, trees, or pre-post relations such as [36, 5]. Moreover we have proposed a generalization of strong low noninterference which can accomodate dynamically changing security levels in a spirit similar to intransitive noninterference [23, 6, 26, 22].

An interesting application of our results is likely to be security analysis of low-level code. Current work in this area (cf. [3, 24, 16]) is based on the idea of reconstructing a structured control flow in order to apply a typed-based analysis. Such a reconstruction will often not be possible, however, and an approach such as ours based on bisimulation checking is likely to be simpler and more robust.

Our results are constrained by two key assumptions, finite control, and effective separability. The finite control assumption is tantamount to assuming that the set of global control points is finite up to structural congruence, and effectively computable. This holds for the parallel while language considered here, but it does not hold for languages where the number of spawned processes is not bounded, or for languages with richer control structures, e.g. lambda calculus. It is of interest to identify decidable classes that go beyond finite control, and to identify safe approximations for the case when decidability is lost. The issue of decidable classes is likely to be linked to corresponding decidability results for process algebra, e.g. BPP decidability [8] for the case of unbounded process creation.

The other assumption, effective separability, concerns decidability of the underlying expression theory. This seems very reasonable in practice. In fact, to be meaningful, timing-sensitive security properties such as strong low security require that primitive state transitions (which includes expression evaluation steps) are executable in constant time: Any discrepancy can immediately be used by an attacker to create a timing channel.

A couple of other recent papers go beyond typability for checking noninterference. Terauchi and Aiken [34] use the self-composition approach introduced by Darvas, Hähnle and Sands [10] to reduce noninterference to a safety problem, potentially checkable by an automated safety analysis tool. Amtoft et al [2] introduce a separation-like logic to specify flow properties of sequential pointer programs. They also give an algorithm which under some conditions compute strongest postconditions in their logic, thus obtaining a completeness result. Both papers address sequential programs only, under flow-sensitive definitions of information flow, and it is unclear if and how they scale to threaded programs.

Turning to frameworks for dynamically changing security levels, the paper [32] point out four common semantic requirements which declassification mechanisms in their opinion should support. These are:

- Semantic consistency: If two programs are semantically equivalent then they should satisfy the same security properties.
- Conservativity: Security definitions should be weakenings of noninterference.
- Monotonicity of release: Adding declassifications should not make a secure program insecure.
- Occlusion: The presence of declassification should not mask other information leaks.

Of these properties, monotonicity of release fails for our model if the term "declassification" is taken to include upgradings as well, since most secure programs can be made insecure by somewhere upgrading some low identifier. The remaining three requirements are all validated by our definition of strong dynamic Λ-bisimulation.

The model for dynamically changing security levels is not yet as rich as we would want. For instance it is perfectly possible to conceive of more general settings in which level changes may depend on, and affect, the state. In this case the *post* and *upd* operations will depend on an entire transition, rather than just the transition label. As an example consider a specialized downgrading command where another identifier, say low_x, is used to determine the level of the identifier x. We would then get:

$$\textit{post}(\Lambda, (s_1, \sigma_1) \xrightarrow{(\alpha)} (s_2, \sigma_2)) =$$
$$\begin{cases} \Lambda \setminus \{x\} & \text{if } \sigma_1(\textit{low}_x) = 0 \textit{ and } \sigma_2(\textit{low}_x) = 1, \\ \Lambda \setminus \{x\} & \text{if } \sigma_2(\textit{low}_x) = 0, \\ \Lambda & \text{otherwise} \end{cases}$$

$$upd(\Lambda, (s_1, \sigma_1) \xrightarrow{(\alpha)} (s_2, \sigma_2)) = \begin{cases} \Lambda \cup \{x\} & \text{if } \sigma_2(low_x) = 1 \\ \Lambda \setminus \{x\} & \text{otherwise} \end{cases}$$

If x is high in the prestate ($low_x = 0$) and low in the post-state, x should be removed from the post-set to allow high variability to filter through to the low level. On the other hand, if x is high in the post-state, x should also be removed from the post-set, now to allow high variability to filter through to x as a high identifier.

In the present framework where the integrity of low_x cannot be protected such an extension appears to be of marginal interest. Also we have not yet been able to extend our decidability and completeness results to this richer setting. For practicality, however, this kind of extension would appear to be important.

Further ahead we plan to use a similar approach as the one we have used for dynamic levels to support more fine-grained information flow control in the style of admissible interference [9, 17].

Acknowledgments

Thanks to Dave Sands for helpful comments.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000. ACM.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow analysis of pointer programs. In *Proc. POPL'06*. ACM, 2006.

[3] Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 103–112. ACM, 2005.

[4] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report MTR-2997, Mitre Corp., Bedford, Mass., USA, June 1976.

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1–2):109–130, 2002.

[6] G. Boudol and A. Matos. on declassification and the non-disclosure policy. In *Proc. Computer Security Foundations Workshop*, pages 226–240, 2005.

[7] S. Chong and A. C. Myers. Security policies for downgrading. In *Proc. ACM Conference on Computer and Communications Security*, pages 198–209, 2004.

[8] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Bisimulation equivalence is decidable for basic parallel processes. In *Proc. CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 1993.

[9] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. Computer Security Foundations Workshop*, pages 233–244, 2000.

[10] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proc. Second International Conference on Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.

[11] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[13] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[14] R. Focardi and S. Rossi. Information flow security in dynamic contexts. In *Proc. 15th Computer Security Foundations Workshop*, pages 307–319, 2002.

[15] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Proc. FoSSaCS*, pages 299–315, 2005.

[16] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proc. VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer, 2005.

[17] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Sci.Comput. Program.*, 50(1–3):73–99, 2004.

[18] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[19] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Proc. POPL'98*, pages 365–377, 1998.

[20] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. POPL'05*, pages 158–170, 2005.

[21] H. Mantel. Possibilistic definitions of security – an assembly kit –. In *Proc. Computer Security Foundations Workshop*, pages 185–199, 2000.

[22] H. Mantel. Information flow control and applications – bridging a gap. In *Proc. FME*, pages 153–172, 2001.

[23] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *Proc. APLAS*, pages 129–145, 2004.

[24] Ricardo Medel, Adriana B. Compagnoni, and Eduardo Bonelli. A typed assembly language for non-interference. In *Proc. Italian Conference on Theoretical Computer Science*, volume 3701 of *Lecture Notes in Computer Science*, pages 360–374. Springer, 2005.

[25] Robin Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[26] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, Stanford Research Institute, 1992.

[27] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. A. Ershov 5th International Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2003.

[28] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):1–15, January 2003.

[29] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symposium on Software Security*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[30] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. Computer Security Foundations Workshop*, pages 200–214, 2000.

[31] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[32] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. 18th Computer Security Foundations Workshop*, pages 255–269, 2005.

[33] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. POPL'98*, pages 355–364, 1998.

[34] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Proc. SAS'05*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[35] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, MA, June 1998.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[37] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*, pages 27–40, 2004.