Under consideration for publication in Formal Aspects of Computing

Slicing Communicating Automata Specifications: Polynomial Algorithms for Model Reduction

Sébastien Labbé and Jean-Pierre Gallois

CEA-List, Laboratoire Logiciels pour la Sûreté des Procédés
F-91191 Gif-sur-Yvette, France

Abstract. Slicing is a program analysis technique that was originally introduced to improve program debugging and understanding. The purpose of a slicing algorithm is to remove the statements whose execution is not necessary with respect to some point(s) of interest in the program (called slicing criterion). Nowadays, slicing is becoming more important on the specification level, in particular for model reduction. In this article we propose a dependence-based solution to the problem of slicing communicating automata formal specifications. Dependence relations in specifications are formally defined. Efficient algorithms are provided to compute the dependence relations, and use this information to automatically extract slices. All the algorithms have been implemented in a slicing prototype tool, that has shown to be operational in specification debugging and understanding. The model reduction results obtained with our slicer are promising, notably in the area of formal validation and verification methods, e.g. model checking, test case generation.

Keywords: Slicing, specifications, communicating automata, static analysis, model reduction.

1. Introduction

Almost thirty years ago, the concept of slicing arose from research on dataflow analysis and static program analysis [Wei81]. The initial purpose was to facilitate the debugging and understanding of programs expressed in sequential imperative languages. Informally, a slice of a program P with respect to a criterion C (usually a program point) is a set of statements of P, which *are relevant to*[1] the computations performed in C. Since then, slicing has been extended in various ways to deal with more complex program constructs, e.g.

Correspondence and offprint requests to: Sébastien Labbé, CEA Saclay, DRT/LIST/DTSI/SOL/LLSP, F-91191 Gif-sur-Yvette, France. e-mail: sebastien.labbe@cea.fr

[1] If we consider *backward* slicing, then "are relevant to" means "may affect", while if we consider *forward* slicing, then it means "may be affected by".

procedures [HRB88], arbitrary control flow [BH93], arrays and pointers [Kri03], and concurrency [HCD$^+$99, Kri98, NR00, MT00]. Slicing was also extended to more modern formalisms, including Z specifications [CR94, BW05], synchronous languages [GR02], and hierarchical state machines [HW97, WDQ03]. A notion of dynamic slicing has been introduced, too [Tip95]. At present, the slicing literature has become extensive, therefore some useful surveys have been published [Tip95, XQZ$^+$05].

Embedded systems are mainly intended to take part in safety-critical systems, of which a failure could cause heavy damages (e.g. in power plants or transportation systems). For that purpose, tools, formal methods and specification languages are being developed to ensure the construction of safe systems. We claim more effort should be done in transferring to specifications some static analyses (such as slicing), that have been shown to be useful and efficient on programs. We therefore propose in this article to address the problem of slicing formal specifications based on communicating extended automata (here, we are exclusively concerned with static slicing).

Finding a solution is not straightforward, because some significant differences lie between the features of communicating automata and those of imperative programs, preventing the usual program slicing definitions to be directly applicable on automata (notably, the unique end node property satisfied by programs, and the occurrence of communications via channels in automata).

In [LG06], we stated the new definitions that form the cornerstone of our approach to slicing communicating automata specifications. This approach was presented in [LG07], however many points remained to be thoroughly characterised and investigated. The present article continues the work in [LG06] and [LG07], providing three main contributions: first, a thorough description of our algorithms for the computation of dependence relations and specification slices, in section 4; second, state-of-the-art information, focused on program slicing in section 2, and focused on control, data and communication dependencies in sections 4.1, 4.2 and 4.3; and third, an evaluation of the correction and precision of our definitions, in terms of direct, indirect and transitive dependencies, in section 4.4. Prospects of our work appear in several fields of formal methods, including simulation, test-case generation and model checking. The original goals of slicing can be achieved too, through specification debugging and understanding.

The following section 2 is a brief overview of some of the main existing program slicing approaches. In section 3, a theoretical framework for specifications based on communicating automata, called IOSTSs, is introduced. Then, a description of our approach to slicing IOSTS specifications is provided in section 4. In the last sections we conclude with some words on related works, and an overview of ongoing and future work.

2. Program Slicing

Program slicing was first introduced by Weiser [Wei81] as a way to abstract a program with respect to chosen points of interest in the program – these special program locations form a *slicing criterion*. Specifically, a program is abstracted by a selection, in the program itself, of all the statements that may influence the values computed by variables at the slicing criterion; the selected program statements form a *backward slice* of the program. A slice is intended to be smaller and easier to understand than the whole program: informally, a slicer is expected to remove a maximum of statements that are not relevant to the criterion. In an experiment, Weiser claimed that programmers mentally build slices when debugging a program, and therefore a tool automatically computing program slices – a slicer – could help in improving the safety and efficiency of the debugging process. This experiment provided the first main motivation for the development of a slicer. Since then, slicing has been shown to be beneficial in other challenging application areas, e.g. model checking [DHH$^+$06]. Up to now, there have been two main classes of approaches for calculating program slices.

2.1. Weiser-style Slicing

The first approach may be referred to as *Weiser-style* slicing, as Krinke suggested in [Kri03]. It consists of solving equations defining sets of variables and statements that are relevant to a slicing criterion. A fixed point for the set of relevant statements is incrementally computed, leading to the desired slice.

2.2. Dependence-based Slicing

The second main approach, we will refer to as *dependence-based* slicing, involves the computation of dependence relations between the program statements. An additional program representation – a *dependence graph* – is usually constructed from the dependence relations. In a dependence graph, nodes represent program statements, and edges represent the dependence relations: there is an edge from two nodes in the dependence graph if and only if the corresponding statements are dependence-related. Using dependence relations information, slicing can be reduced to a simple reachability problem in the dependence graph [Kri03, FOW87]. Slices resulting of this method are more accurate than those resulting of previous methods [FOW87]. There exists several definitions of dependence graphs, mostly deriving from Ottenstein's Program Dependence Graph [OO84, FOW87], among which we cite the System Dependence Graph [HRB88], the Threaded Program Dependence Graph [Kri98], and the Parallel Program Graph [Sar97].

The adaptability of dependence-based slicing allows one to define appropriate dependence relations in regard to the intended language, then build the corresponding dependence graph, and compute slices by solving graph reachability problems. The precision of a dependence-based slicing approach highly depends on two factors: the definitions of the dependence relations, and the features of the intended language. For instance, in the context of slicing concurrent programs, Krinke [Kri98] introduced the notion of *interference dependence*, to denote dependencies induced by concurrent reads and writes of shared variables, and the notion of *transitive* dependence, to evaluate the precision of dependence relations.

Considering its adaptability, accuracy and relative easiness of understanding and implementation, dependence-based slicing has become the most widespread slicing technique, especially when complex constructs have to be handled by the slicing algorithm [Kri03, NR00, BH93, HRB88, RAB$^+$05].

3. Theoretical Framework

In this section is defined the specification framework, on which is founded our approach to slicing communicating automata specifications (cf. section 4).

3.1. Communicating Automata (IOSTS)

Input/Output Symbolic Transition Systems (IOSTSs) [GGRT06] are automata used to specify the behaviour of reactive systems. When designing complex or infinite-state systems, IOSTSs compare advantageously to classical labelled transition systems. Our slicing method is intended to operate on specifications that are based on IOSTSs; this formalism is defined in the following subsections.

3.1.1. Data Types

Data types are defined within a *typed equational* specification framework.

Syntax A *data type signature* is a couple $\Omega = (\Theta, \Phi)$, where Θ is a set of type names, and each element of Φ is formed of an operation name, together with a profile $\theta_1 \dots \theta_{n-1} \rightarrow \theta_n$, where $n \geq 1$ and for all $i \leq n$, $\theta_i \in \Theta$.

Let $V = \bigcup_{\theta \in \Theta} V_\theta$ be a set of typed variable names. The set of Ω-*terms* whose variables are elements of V is denoted by $\mathscr{T}_\Omega(V)$ and is inductively defined as follows: an Ω-term of type θ_n is either a variable in V_{θ_n}; or an expression formed from an operation $\phi : \theta_1 \dots \theta_{n-1} \rightarrow \theta_n$ in Φ, followed by a parenthesised list $[t_1 \dots t_{n-1}]$ of Ω-terms, such that for all i, t_i is of type θ_i.

An Ω-*substitution* is a function $\sigma : V \rightarrow \mathscr{T}_\Omega(V)$ that preserves types, i.e. $\sigma(v)$ is of type θ whenever $v \in V_\theta$. In the following, we note $\mathscr{T}_\Omega(V)^V$ the set of all Ω-substitutions mapping variables in V to terms in $\mathscr{T}_\Omega(V)$.

The set of Ω-*formulae* whose variables are elements of V is denoted by $\mathscr{F}_\Omega(V)$ and is inductively defined as follows: an Ω-formula is either a truth value in $\{true, false\}$; an expression of the form $t_1 = t_2$ where $t_1, t_2 \in \mathscr{T}_\Omega(V)$ are of the same type; or an expression formed from Ω-formulae and Boolean operators in $\{\neg, \vee, \wedge\}$, with the usual syntax.

Semantics An Ω-*model* is a family $M = \{M_\theta\}_{\theta\in\Theta}$, with a function space defined as follows: $\{\phi_M : M_{\theta_1} \times \cdots \times M_{\theta_n} \to M_\theta \mid (\phi : \theta_1 \ldots \theta_{n-1} \to \theta_n) \in \Phi\}$. We define Ω-*interpretations* as applications ν from V to M, extended to terms in $T_\Omega(V)$, and preserving types. M^V is the set of all Ω-interpretations of V in M. Given a model M and a formula f, f is said *satisfiable* in M, if there exists an interpretation ν such that $M \models_\nu f$.

3.1.2. Input/Output Symbolic Transition Systems

An IOSTS-*signature* Σ is a triple (Ω, V, C), where Ω is a data type signature, $V = \bigcup_{\theta\in\Theta} V_\theta$ is a set of typed variable names, called *attribute variables*, and C is a set of *communication channels* names.

An IOSTS denotes the state variations of the specified system, by describing modifications of the values associated to attribute variables. These values may be modified via interactions with the environment, called *communication actions*, or via internal operations (denoted by variable substitutions, cf. definition 2). Communication actions denote emissions and receptions of messages through communication channels.

Definition 1 (Communication actions). The set of *communication actions* over an IOSTS-signature $\Sigma = (\Omega, V, C)$ is denoted by $Act(\Sigma)$, such that:

$$\begin{aligned} Act(\Sigma) &= \{\tau\} \cup Input(\Sigma) \cup Output(\Sigma) \\ Input(\Sigma) &= \{c? \mid c \in C\} \cup \{c?x \mid c \in C \wedge x \in V\} \\ Output(\Sigma) &= \{c! \mid c \in C\} \cup \{c!t \mid c \in C \wedge t \in \mathscr{T}_\Omega(V)\} \end{aligned}$$

The specified system is stimulated by its environment via actions in $Input(\Sigma)$. In particular, when the action $c?$ is performed, the system waits for a signal to occur on channel c, and when the action $c?x$ is performed, the system waits on channel c for the reception of a value to be assigned to the attribute variable x. Actions in $Output(\Sigma)$ denote responses of the system to its environment. The action $c!t$ (resp. $c!$) is performed for the system to emit a message, having t as argument (resp. having no arguments), on the channel c.

Definition 2 (Input/Output Symbolic Transition System (IOSTS)). An IOSTS over $\Sigma = (\Omega, V, C)$ is a triple $(S, s_0, T)_\Sigma$ where S is a set of state names, $s_0 \in S$ is the *initial state*, and $T \subseteq S \times Act(\Sigma) \times \mathscr{F}_\Omega(V) \times \mathscr{T}_\Omega(V)^V \times S$ is a transition relation. A transition in T is a tuple (s, a, f, σ, s') where states s and s' are respectively called *source state* and *target state*, f is a formula called *guard* of the transition, a is an action and σ a variable substitution.

3.2. Semantics

Definition 3 formally states the semantics of an IOSTS $\mathscr{A}$ in a model M, under the form of an extended labelled transition system.

Notations In the sequel, $(x \mapsto m)$ denotes a variable substitution, that associates m to x, and is the identity for all other variables, $Id_{V\setminus\{x\}}$. For the ease of reading, interpretations are extended to terms, without renaming: $\nu : V \to M$ is extended by $\nu : \mathscr{T}_\Omega(V) \to M$. Finally we note Λ the set of labels such that: $\Lambda = \{\tau\} \cup \{c! \mid c \in C\} \cup \{c? \mid c \in C\} \cup \{c!m \mid c \in C \wedge m \in M\} \cup \{c?m \mid c \in C \wedge m \in M\}$.

Definition 3 (IOSTS Semantics). Given an IOSTS $\mathscr{A} = (S, s_0, T)_\Sigma$, where $\Sigma = (\Omega, V, C)$, and a model M, the semantics of $\mathscr{A}$ in M is a labelled transition system $[\![\mathscr{A}]\!] = ([\![S]\!], (s_0, \nu_0), [\![T]\!])$, such that: $[\![S]\!] \subseteq S \times M^V$, $\nu_0 \in M^V$ and $[\![T]\!] \subseteq [\![S]\!] \times \Lambda \times [\![S]\!]$. The set of extended states $[\![S]\!]$ contains elements of the form (s, ν), where s is a state of $\mathscr{A}$, and ν is an Ω-interpretation. The set of transitions $[\![T]\!]$ is inductively built as follows:

- Initially, $[\![T]\!]$ is the empty set $\emptyset$;
- Apply base rules whenever possible;
- Apply induction rules whenever possible.

$$\begin{aligned}
&\text{Base Rules:}\\
&\text{For all } (s,a,f,\sigma,s_1)\in T, \text{ and } \nu\in M^V, \text{ such that } s=s_0\wedge M\models_\nu f \ :\\
&\text{If } a\in(\Lambda\cap \mathit{Act}(\Sigma)), \text{ then insert } ((s_0,\nu),a,(s_1,\nu\circ\sigma)) \text{ in } [\![T]\!] && (1)\\
&\text{If } a=c!t, \text{ then insert } ((s_0,\nu),c!\nu(t),(s_1,\nu\circ\sigma)) \text{ in } [\![T]\!] && (2)\\
&\text{If } a=c?x, \text{ then for all } m\in M, \text{ insert } ((s_0,\nu),c?m,(s_1,\nu\circ(x\mapsto m)\circ\sigma)) \text{ in } [\![T]\!] && (3)
\end{aligned}$$

$$\begin{aligned}
&\text{Induction Rules:}\\
&\text{For all } (s_1,a,f,\sigma,s_2)\in T, \text{ and } \nu\in M^V, \text{ such that}\\
&M\models_\nu f\wedge(\exists(e,l,e')\in[\![T]\!], e'=(s_1,\nu)) \ :\\
&\text{If } a\in(\Lambda\cap \mathit{Act}(\Sigma)), \text{ then insert } ((s_1,\nu),a,(s_2,\nu\circ\sigma)) \text{ in } [\![T]\!] && (4)\\
&\text{If } a=c!t, \text{ then insert } ((s_1,\nu),c!\nu(t),(s_2,\nu\circ\sigma)) \text{ in } [\![T]\!] && (5)\\
&\text{If } a=c?x, \text{ then for all } m\in M, \text{ insert } ((s_1,\nu),c?m,(s_2,\nu\circ(x\mapsto m)\circ\sigma)) \text{ in } [\![T]\!] && (6)
\end{aligned}$$

In the following, the transitions that are inserted in $[\![T]\!]$ following the rule (N) will be denoted by *(N)-transitions.* In definition 6, (1), (2) and (3)-transitions denote the semantics of the transitions of $\mathscr{A}$ whose source state is the initial state of $\mathscr{A}$. Specifically, (1)-transitions denote the semantics of the transitions that contain a silent communication action (τ), or a signal communication[2] (of the form $c!$ or $c?$). (2)-transitions, resp. (3)-transitions, denote the semantics of the transitions that contain an output action of the form $c!t$, resp. an input action of the form $c?x$. (4), (5) and (6)-transitions denote the semantics of transitions whose source state is any state in $S\backslash\{s_0\}$, with the additional condition that this state is reachable in $[\![\mathscr{A}]\!]$; (4), (5) and (6) are respectively analogous to (1), (2) and (3).

Consequently, definition 3 is in accordance to the intuition of a fireable transition: for a transition to be fired, first its source state must be reached, and then its guard must be satisfied.

3.3. Specifications

Reactive systems are usually specified by synchronising subsystems together. In our framework, a specification is considered as the parallel composition of communicating automata (IOSTSs).

Definition 4 (Specification). A *specification* $\mathscr{S}$ is a non-empty set of IOSTSs, $\mathscr{S}=\{(S_i,s_{0i},T_i)_{\Sigma_i} \mid 0\le i<k \ \wedge \ \Sigma_i=(\Omega_i,V_i,C_i)\}$, for some $k\in\mathbb{N}^*$, such that for all i the sets of attribute variables V_i are pairwise disjoint. The parallel composition of automata in a specification is denoted by interleaving semantics (cf. definition 6).

In a specification, if two different IOSTSs communicate on a channel c (i.e. $c\in C_i\cap C_j$, and $i\neq j$), then c is used for internal communications, otherwise c is used for communications with the environment of the system.

Example 1. Figure 1 shows a graphic representation of a cash machine specification, formed of two concurrent IOSTSs (initial states are indicated by small dots). Two kinds of credit cards can be handled by the specified machine: gold and normal cards. In the former case, overdraft is allowed, while in the latter case, it is not. Basically, the left automaton is an interface with the environment, and the right automaton performs internal operations with the bank. The left automaton acquires the necessary data through channels *nameCh* (the customer's name), *typeCh* (the type of card that has been introduced in the machine), and *amountCh* (the amount the customer requested); it sends data that is needed for the right automaton to operate, through internal channels *intNameCh*, *normalCh*, *goldCh*, and finally it sends data back to the environment through channel *amountCh* (the amount of money to be delivered to the customer). The right automaton acquires the customer's name and the type of his credit card, and gets the customer's account, using the *account* access function. Then, depending on the account overdraft facilities, the balance after

[2] Indeed, $(\Lambda\cap \mathit{Act}(\Sigma))$ contains τ, and all the signal emissions and receptions.

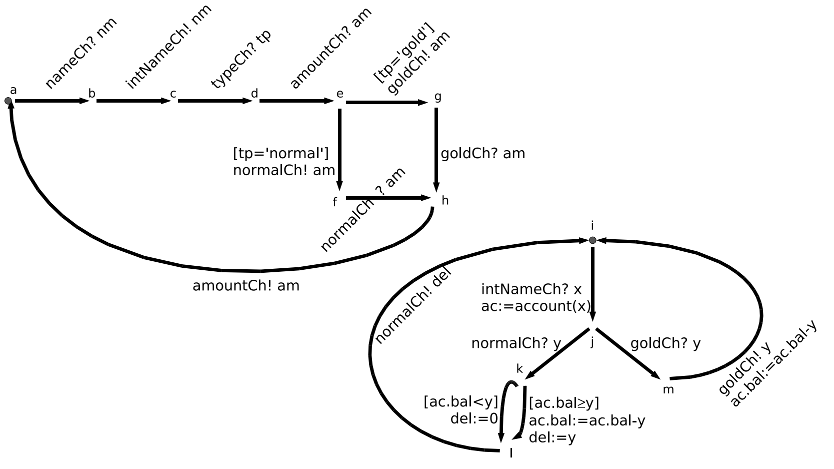

Fig. 1. IOSTS specification example.

withdrawal is checked to be greater than zero, or not (here, an account *ac* is a record that contains at least a variable, *bal*, which indicates the balance of *ac*). Finally, the account is debited only if the withdrawal is allowed. Then, the debited amount is transmitted through channel *normalCh* or *goldCh* (0 in the case the withdrawal is not allowed).

Throughout this paper, a transition of figure 1 will be noted $\alpha \rightharpoonup \beta$, where α is the source state and β is the target state of the transition. Since this notation is ambiguous for two transitions of the right automaton, $(k, \tau, ac.bal < y, (del \mapsto 0), l)$ will be noted $k \rightharpoonup_1 l$, and $(k, \tau, ac.bal \geq y, (ac.bal \mapsto ac.bal - y, del \mapsto y), l)$ will be noted $k \rightharpoonup_2 l$.

As we saw, our definition of a specification vitally depends on the notion of parallel composition of IOSTSs. In the following, we therefore need to formally define this notion. For that purpose, we define a Boolean function *rdv*, that takes a transition and an IOSTS as argument, such that $rdv(tr, \mathscr{A})$ is true if and only if there is a transition in $\mathscr{A}$ that may have a rendezvous with transition *tr*.

Definition 5 (Function *rdv*). The Boolean function *rdv* is formally defined by:

$$\begin{gathered} rdv((s_1, a_1, f_1, \sigma_1, s_1'), (S, s_0, T)_\Sigma) = a_1 \neq \tau \wedge \exists (s_2, a_2, f_2, \sigma_2, s_2') \in T, \\ (a_1 = c! \implies a_2 = c?) \wedge (a_1 = c? \implies a_2 = c!) \\ \wedge (a_1 = c!t \implies a_2 = c?x) \wedge (a_1 = c?x \implies a_2 = c!t) \end{gathered}$$

Example 2. Let us note $\mathscr{A}$ the right automaton in figure 1 then $rdv(e \rightharpoonup f, \mathscr{A})$ is true because a rendezvous is possible between $e \rightharpoonup f$ and a transition of $\mathscr{A}$, namely $j \rightharpoonup k$; $rdv(a \rightharpoonup b, \mathscr{A})$ is false because no transition in $\mathscr{A}$ performs an input action on channel *nameCh*.

Definition 6 formally defines the notion of the parallel composition of two IOSTSs. Two IOSTSs are synchronised using a binary rendezvous mechanism when possible, otherwise all interleavings are considered. Synchronised communication actions result in internal actions in the composed system.

Definition 6 (Parallel composition of IOSTS). Let $\mathscr{A}_1$ and $\mathscr{A}_2$ be two IOSTSs such that $\forall i \in \{1, 2\}$, $\mathscr{A}_i = (S_i, s_{0_i}, T_i)_{\Sigma_i}$, and $\Sigma_i = ((\Theta_i, \Phi_i), V_i, C_i)$. The parallel composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ is an IOSTS $\mathscr{A} = (S, s_0, T)_\Sigma$, where $\Sigma = (\Omega, V, C)$, such that:

$S = S_1 \times S_2$, $s_0 = (s_{0_1}, s_{0_2})$, $\Omega = (\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2)$, $V = V_1 \cup V_2$, $C = C_1 \cup C_2$, and $T \subseteq S \times Act(\Sigma) \times \mathscr{F}_\Omega(V) \times \mathscr{T}_\Omega(V)^V \times S$ is defined as the following set of transitions:

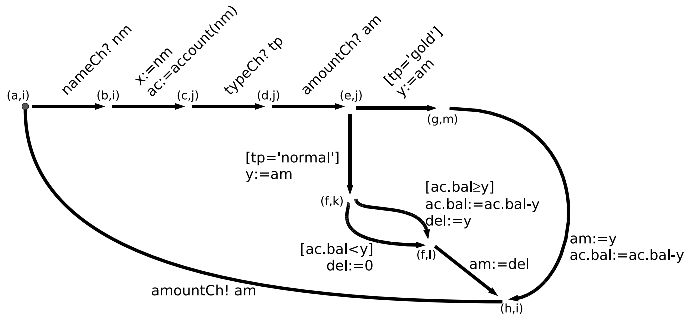

Fig. 2. Parallel composition of specification in figure 1.

$$
\begin{aligned}
&\{\ ((s_1,s_2),a_1,f_1,\sigma_1,(s'_1,s_2))\ | \\
&\exists tr \in T_1, tr = (s_1,a_1,f_1,\sigma_1,s'_1) \wedge \neg rdv(tr,\mathscr{A}_2)\ \} && (7)\\
\cup\ &\{\ ((s_1,s_2),a_2,f_2,\sigma_2,(s_1,s'_2))\ | \\
&\exists tr \in T_2, tr = (s_2,a_2,f_2,\sigma_2,s'_2) \wedge \neg rdv(tr,\mathscr{A}_1)\ \} && (8)\\
\cup\ &\{\ ((s_1,s_2),\tau,f_1 \wedge f_2,\sigma_1 \cup \sigma_2,(s'_1,s'_2))\ | \\
&(s_1,c?,f_1,\sigma_1,s'_1) \in T_1 \wedge (s_2,c!,f_2,\sigma_2,s'_2) \in T_2\ \} && (9)\\
\cup\ &\{\ ((s_1,s_2),\tau,f_1 \wedge f_2,\sigma_1 \cup \sigma_2,(s'_1,s'_2))\ | \\
&(s_1,c!,f_1,\sigma_1,s'_1) \in T_1 \wedge (s_2,c?,f_2,\sigma_2,s'_2) \in T_2\ \} && (10)\\
\cup\ &\{\ ((s_1,s_2),\tau,f_1 \wedge f_2,(x \mapsto t)\circ\sigma_1 \cup \sigma_2,(s'_1,s'_2))\ | \\
&(s_1,c?x,f_1,\sigma_1,s'_1) \in T_1 \wedge (s_2,c!t,f_2,\sigma_2,s'_2) \in T_2\ \} && (11)\\
\cup\ &\{\ ((s_1,s_2),\tau,f_1 \wedge f_2,\sigma_1 \cup (x \mapsto t)\circ\sigma_2,(s'_1,s'_2))\ | \\
&(s_1,c!t,f_1,\sigma_1,s'_1) \in T_1 \wedge (s_2,c?x,f_2,\sigma_2,s'_2) \in T_2\} && (12)
\end{aligned}
$$

In definition 6, the sets at lines (7) and (8) denote the set of transitions that can be executed independently; these transitions potentially create interleavings in the parallel composition. The set at line (7) denotes the set of transitions in $\mathscr{A}$ that result from firing a transition in $\mathscr{A}_1$ while staying in the same state in $\mathscr{A}_2$; the set at line (8) is symmetric to the set at line (7). Sets at lines (9), (10), (11) and (12) denote synchronisations of $\mathscr{A}_1$ and $\mathscr{A}_2$: firing a transition in one of these sets means firing one transition in each automaton. More precisely, the sets at lines (9) and (11) denote the sets of transitions that result from a synchronisation of an output action performed in $\mathscr{A}_2$, with an input action performed in $\mathscr{A}_1$, on the same channel; the set at line (10), respectively (12), is symmetric to the set at line (9), respectively (11).

Notice that, at lines (11) and (12), an expression of the form $\sigma_1 \cup \sigma_2$ is a valid expression for a function, since the domains of σ_1 and σ_2 are disjoint.

Example 3. Figure 2 represents an automaton that results from the parallel composition of our running example, in figure 1. This example system is highly synchronised: the absence of interleavings results in a relatively compact parallel composition of the specification. Another consequence is that many new transitions are created in order to build the parallel composition IOSTS.

For instance, the transition $(b,i) \rightharpoonup (c,j)$ results from the parallel composition of transitions $b \rightharpoonup c$ and $i \rightharpoonup j$. Formally, using the line (11) (or (12) in a symmetric way) of definition 6, the parallel composition of $(b, intNameCh!nm, true, Id_{\{nm,tp,am\}}, c)$ and $(i, intNameCh?x, true, (ac \mapsto account(x)), j)$ is denoted by $((b,i),\tau,true,\sigma,(c,j))$, where $\sigma = (x \mapsto nm, ac \mapsto account(nm))$. As expected, the variable x at the assignments of $i \rightharpoonup j$ refers to nm (via channel $intNameCh$).

Now, let us illustrate the case where a transition is labelled with an input action on a variable, and also an assignment to the same variable. Specifically, let us insert an assignment to x at $i \rightharpoonup j$, i.e. $i \rightharpoonup j$ is replaced by $i \rightharpoonup_1 j = (i, intNameCh?x, true, (x \mapsto \text{“smith”}, ac \mapsto account(x)), j)$ Then, according to definition 6, the composition of $b \rightharpoonup c$ and $i \rightharpoonup j$ is $((b,i),\tau,true,\sigma',(c,j))$, where $\sigma' = (x \mapsto \text{“smith”}, ac \mapsto account(nm))$. It is worth noticing that according to the semantics of definition 3, the term nm that is received in variable x via channel $intNameCh$, is overwritten by the assignment to x at $i \rightharpoonup j$ (i.e. $\sigma'(x) =$ "smith"); but uses of x at the assignments of $i \rightharpoonup j$ are still mapped to nm, indeed, $\sigma'(ac) \neq account($"smith"$)$.

Finally, some transition labels stay unchanged, as no synchronisation happened, cf. transition $(a, i) \rightharpoonup (b, i)$, that is built using the line (7) (or symmetrically, (8)) of definition 6.

3.4. Paths in IOSTS

Assuming $\mathscr{A} = (S, s_0, T)_\Sigma$ is an IOSTS, the two following definitions state what we will call *successors* and *predecessors* of a transition in $\mathscr{A}$, and *paths* and *maximal paths* in $\mathscr{A}$.

Definition 7 (Successor/predecessor of a transition). Let tr_i and tr_j be two transitions in T, tr_j is called a *successor* of tr_i in $\mathscr{A}$ if the source state of tr_j is the target state of tr_i; in this case, tr_i is called a *predecessor* of tr_j.

Notation The number of successors of a transition tr is called the outer degree of tr, and is noted $d(tr)$.

Definition 8 (Path in an automaton). A path p in $\mathscr{A}$ is a (possibly partial) function $\mathbb{N} \to T$, such that:

- for all $i \in \mathbb{N}$, if p is defined at i, then p is defined at j for all $j \leq i$;
- and for all $i \in \mathbb{N}$, if p is defined at $i + 1$, then the source state of $p(i + 1)$ is the target state of $p(i)$.

Notice that if a path is a partial function, then it is a finite path. In this paper, we note $\langle tr_i, \dots, tr_j \rangle$ a finite path p such that k is the greatest integer at which p is defined, $p(0) = tr_i$ and $p(k) = tr_j$ (in this case we say p *starts with* tr_i).

Example 4. In figure 1, transitions $e \rightharpoonup f$ and $e \rightharpoonup g$ are both successors of transition $d \rightharpoonup e$, and $\langle c \rightharpoonup d, d \rightharpoonup e, e \rightharpoonup g \rangle$ is a finite path starting with $c \rightharpoonup d$ in the left automaton.

The following definition recalls the graph theory notion of maximal path and extends it to IOSTSs. Intuitively, a maximal path in $\mathscr{A}$ is a path that cannot be continued by any transition in T.

Definition 9 (Maximal path). A path p in $\mathscr{A}$ is called a *maximal* path, if either p is a total function (i.e. p is an infinite path), or a partial function such that: let k be the greatest integer at which p is defined, then there exists no path p' in $\mathscr{A}$ such that $p'(i) = p(i)$ for all $i \leq k$, and $p'(k + 1)$ is defined.

Example 5. The path that consists in the infinite sequence of transitions in $(i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i)^*$ is a maximal path in the right automaton of figure 1. Still in figure 1, suppose that we insert a new node n and a transition $i \rightharpoonup n$ in the right automaton. Then $i \rightharpoonup n$ has no successor in the right automaton, and the path $\langle i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i, i \rightharpoonup n \rangle$ is a maximal path starting with $i \rightharpoonup j$ in the right automaton.

4. Slicing IOSTS Specifications

Referring to section 2, program slicing could be coarsely summarised as finding a subset of a program (called *slice*) with respect to another subset of the program (called *criterion*). We aim for a specification slicing algorithm that closely conforms to this initial concept. Hence, the slicing criteria our algorithm operates on, and the slices it produces, are both formed from elements of the specification. Our approach to slicing formal specifications based on communicating automata is inspired by previous works on dependence-based program slicing (cf. section 2). In this section, we introduce dependence relations defined on specifications, that may be understood as extensions of previous dependence relations defined on programs. These new dependencies will enable the construction of a dependence graph, from which slices can be efficiently extracted.

Control dependence and data dependence are binary relations, extensively used in the area of compiler construction and program analysis. These dependence relations are involved in numerous transformation and optimising techniques, such as vectorisation and parallelization [AK02], instruction scheduling and data-cache optimisation [Muc97], node splitting, code motion and loop fusion [FOW87]; and in program analysis techniques such as program slicing [Kri03, OO84, Tip95].

In accordance to our idea of adopting a dependence-based approach for slicing formal specifications, we need to define dependence relations on specifications. Since information appear on transitions, the dependence relations are defined between transitions (in the following we set the conditions to be met for a transition to be dependent on another one). In sections 4.1 and 4.2, the notions of data and control dependencies are extended to IOSTSs. Then in section 4.3, an extra dependence relation – called communication dependence – is defined, in order to denote the dependencies arising from data and control flows related to communication actions. The three aforementioned sections come with complete descriptions of our algorithms for computing the dependence relations. As we will see in sequel, having calculated the three dependence relations, the construction of the dependence graph is straightforward (cf. section 4.5). Our method for slicing a specification with respect to a criterion will then be presented as finding a solution to a reachability problem – parametrised by the slicing criterion – in the dependence graph. Prior to that, the discussion in section 4.4 provides key elements to evaluate the precision of our dependence-based approach. Throughout the sections 4.1, 4.2, and 4.4, we will assume $\mathscr{A} = (S, s_0, T)_\Sigma$ is an IOSTS over signature $\Sigma = (\Omega, V, C)$.

4.1. Control Dependence

Intuitively, a program statement n is control dependent on a statement m, if whenever an execution of the program reaches m, a choice can be made that determines whether or not n will be executed in the rest of the execution. This section is structured as follows: at first the traditional definition of control dependence is described, then we will see why this definition is inapplicable to IOSTSs. Finally we show how this difficulties may be overcome, and give a detailed description of our algorithm.

4.1.1. Traditional Definition

Control dependence is traditionally defined in terms of a *postdominance relation* in a *control flow graph* (CFG) [AK02, Muc97]. A CFG is a graph that represents a sequential imperative program[3], and notably has a single end node. Intuitively, a node n_i is postdominated by a node n_j in a CFG if in every execution of the program, n_j always occurs after n_i. Stating a formal definition for this *always occurs after* relation is enabled by the fact CFGs satisfy the unique end node property. All definitions of the postdominance relation we are aware of, indeed require the unique end node property to be satisfied. Following is the usual definition: a node n_i is postdominated by a node n_j if every path from n_i to the end node contains n_j. Having calculated the postdominance relation in a CFG, the control dependence relation can be obtained according to the traditional definition of control dependence [AK02, Muc97]: *a statement n_j is said to be* control dependent *on a statement n_i if there exists a nontrivial path p from n_i to n_j such that every statement $n_k \neq n_i$ in p is postdominated by n_j, and n_i is not postdominated by n_j.*

The definition above meets the intuitive concept of control dependence, as it allows the identification of statements that may affect the execution of other statements. Since the statements in IOSTSs are attached to transitions, the concept of control dependence could at first glance be adapted to IOSTSs by redefining the postdominance relation between IOSTS transitions instead of CFG nodes. However, *this definition assumes the structure under analysis satisfies the unique end node property, and thus is not directly applicable to automata or modern program structures, that may have multiple end nodes or no end node at all.* Addressing the issue of multiple end nodes is usually done by inserting an additional end node into the structure, along with edges from each original end node to the new one [Muc97, RAB$^+$05]. The main purpose of IOSTSs is to design and study reactive systems, which are intended to run indefinitely, and therefore have usually no specific end node(s). In this case, it is most of the time unclear whether an additional end node can be automatically inserted, without disrupting the control dependence calculation. We indeed observe that inserting an irrelevant end node in an automaton, together with irrelevant transitions towards this node, will yield undesirable additional control dependencies, causing an inadequate over-approximation of the control dependence relation.

4.1.2. New Definition

In a recent work, Ranganath *et al.* [RAB$^+$05] state new definitions for calculating control dependencies, without any restriction about the existence of end node(s) in the program structure. The key observation

[3] In a CFG, nodes represent the program statements, and edges represent the program control flow.

enabling this achievement is that, *regarding control dependence, reaching again a start node in a reactive program is analogous to reaching an end node in a program.* In [RAB+05], the definition of n_j being control dependent on n_i checks for occurrences of n_j in maximal paths from n_i, instead of paths from n_i to the assumed unique end node, as in previous definitions. Furthermore, the general idea of control dependence remains the same: executing one branch of n_i always leads to n_j, whereas executing another branch of n_i may cause n_j to be bypassed. Two fundamental new definitions for control dependence are stated in [RAB+05]; one is sensitive to potentially non-terminating loops, while the other is not.

In an IOSTS, an infinite execution of a loop may prevent some transitions from being executed[4], creating a control dependence on these transitions (according to the intuitive notion of a control dependence outlined above). *As it is known to be undecidable whether a loop always terminates in general, and as a correct static analysis has to take every execution of the system into account, we make the assumption that each loop in an* IOSTS *is potentially infinite.* Otherwise, the control dependence relation would omit actual dependencies, arising in the case of an infinite loop, consequently making the slicing approach incorrect. Moreover, the non-termination sensitiveness of the control dependence relation involved in slicing is essential for the slices computed to preserve dynamic properties of the original specification, with respect to the criterion (e.g. for model checking). Thus, we base our definition of control dependence on Ranganath *et al.*'s definition of non-termination sensitive control dependence.

Definition 10 (Control dependence ($tr_i \xrightarrow{cd} tr_j$)). A transition $tr_j \in T$ is *control dependent* on a transition $tr_i \in T$ if tr_i has at least two successors, tr_i' and tr_i'', such that the following are true.

- for all maximal paths starting with tr_i' in $\mathscr{A}$, tr_j always occurs;
- there exists a maximal path starting with tr_i'' in $\mathscr{A}$, on which tr_j does not occur.

Example 6. In figure 1, $k \rightharpoonup_1 l$ is control dependent on $j \rightharpoonup k$: first, $j \rightharpoonup k$ has two successors $k \rightharpoonup_1 l$ and $k \rightharpoonup_2 l$; second, $k \rightharpoonup_1 l$ occurs on all maximal paths from $k \rightharpoonup_1 l$ (this is rather obvious); and third, there exists a maximal path from $k \rightharpoonup_2 l$, on which $k \rightharpoonup_1 l$ does not occur – for instance, the path that consists of an infinite sequence of transitions in $(k \rightharpoonup_2 l, l \rightharpoonup i, i \rightharpoonup j, j \rightharpoonup k)^*$. Besides, $l \rightharpoonup i$ is not control dependent on $j \rightharpoonup k$, since $l \rightharpoonup i$ occurs on all maximal paths from $j \rightharpoonup k$.

4.1.3. Algorithm

From Ranganath *et al.*'s algorithm for computing non-termination sensitive control dependence on control flow graphs [RAB+05], we derive our algorithm for computing control dependencies on IOSTSs (algorithm 1 in figure 3), according to definition 10.

Control-flow Analysis In algorithm 1, the set of the successors of a transition tr in $\mathscr{A}$ is denoted by $succs(tr, \mathscr{A})$, and the set of transitions that have at least 2 successors in $\mathscr{A}$ is denoted by $conds(\mathscr{A})$. Algorithm 1 performs a symbolic control-flow analysis on an automaton $\mathscr{A}$: symbolic values are propagated along the transitions of $\mathscr{A}$, to collect control-flow information, and store this information in sets. *The aim of algorithm 1 is to iteratively search for a fixpoint for all the sets of symbolic values, then indicating relevant information about the control structure of $\mathscr{A}$, for the purpose of control dependence calculation.* The symbolic values are called p_{tr_i,tr_j}, denoting all the maximal paths in $\mathscr{A}$ starting with tr_i immediately followed by tr_j. A set S_{tr,tr_c} is attached to each couple of transitions (tr, tr_c) such that $tr_c \in conds(\mathscr{A})$. More precisely, S_{tr,tr_c} denotes the set of all the maximal paths in $\mathscr{A}$ that start with tr_c and contain tr.

In phase (1), successors of conditional transitions are inserted in a worklist. In phase (2), the symbolic values are propagated along transitions, and inserted in sets S_{tr,tr_c} when necessary: typically, p_{tr_c,tr_j} is inserted in S_{tr,tr_c} only if all the maximal paths starting with tr_c immediately followed by tr_j contain tr. Two cases are distinguished when processing a transition tr_l from the worklist. Phase (2.1) deals with the case where tr_l has only one successor tr_s: the idea is that, if a maximal path starting with a transition tr_c contain tr_l, then this path also contains tr_s. Phase (2.2) deals with the case where tr_l has at least 2 successors: the idea is that, if all the maximal paths starting with tr_l contain a transition tr_m, then all the maximal paths starting with a transition tr_c and containing tr_l, contain also tr_m.

[4] For instance, a transition that is not in a loop l, but is a successor of a transition in l, will be prevented from execution when l executes indefinitely.

```
    **Input**: $\mathscr{A} = (S, s_0, T)_\Sigma$ : an IOSTS
    **Data**:
      • $S[|T|, |T|]$ : a matrix of sets of symbolic values, such that $S[tr_i, tr_j]$ represents $S_{tr_i,tr_j}$
      • $worklist$ : a set of transitions
    **Output**: $CD[|T|]$ : an array of sets of transitions
    // At the end of algorithm 1, for each $tr \in T$, $CD[tr]$ contains all the transitions,
    // on which $tr$ is control dependent.
    /* (1) Initialisation                                                          */
 1  $worklist \leftarrow \emptyset$
 2  foreach $tr \in T$ do
 3      $CD[tr] \leftarrow \emptyset$
 4      foreach $tr_c \in conds(\mathscr{A})$ do $S[tr, tr_c] \leftarrow \emptyset$
 5  foreach $tr_c \in conds(\mathscr{A})$ do
 6      foreach $tr_s \in succs(tr_c, \mathscr{A})$ do
 7          $S[tr_s, tr_c] \leftarrow \{p_{tr_c,tr_s}\}$
 8          $worklist \leftarrow worklist \cup \{tr_s\}$
    /* (2) Symbolic control-flow analysis                                          */
 9  while $worklist \neq \emptyset$ do
10      $tr_l \leftarrow$ first($worklist$)
        /* (2.1) One successor case                                                */
11      if $|succs(tr_l, \mathscr{A})| = 1$ and $tr_l \notin succs(tr_l, \mathscr{A})$ then
12          $tr_s \leftarrow$ element($succs(tr_l, \mathscr{A})$)
            // Since $tr_s$ is the only successor of $tr_l$,
            // maximal paths that contain $tr_l$ also contain $tr_s$.
13          foreach $tr_c \in conds(\mathscr{A})$ do
14              if $S[tr_l, tr_c] \backslash S[tr_s, tr_c] \neq \emptyset$ then
15                  $S[tr_s, tr_c] \leftarrow S[tr_s, tr_c] \cup S[tr_l, tr_c]$
16                  $worklist \leftarrow worklist \cup \{tr_s\}$
        /* (2.2) Multiple successors case                                          */
17      if $|succs(tr_l, \mathscr{A})| > 1$ then
18          foreach $tr_m \in T$ do
19              if $|S[tr_m, tr_l]| = |succs(tr_l, \mathscr{A})|$ then
                    // Since all the maximal paths from $tr_l$ contain $tr_m$, maximal paths from $tr_m$
                    // contain the transitions that appear in maximal paths from $tr_l$.
20                  foreach $tr_c \in conds(\mathscr{A}) \backslash \{tr_l\}$ do
21                      if $S[tr_l, tr_c] \backslash S[tr_m, tr_c] \neq \emptyset$ then
22                          $S[tr_m, tr_c] \leftarrow S[tr_m, tr_c] \cup S[tr_l, tr_c]$
23                          $worklist \leftarrow worklist \cup \{tr_m\}$
24      $worklist \leftarrow worklist \backslash \{tr_l\}$
    /* (3) Compute control dependencies                                            */
25  foreach $tr \in T$ do
26      foreach $tr_c \in conds(\mathscr{A})$ do
            // If $S[tr, tr_c]$ contains strictly less elements than $succs(tr_c, \mathscr{A})$, then there is
            // at least one path that induces a control dependence of $tr_c$ on $tr$.
27          if $0 < |S[tr, tr_c]| < |succs(tr_c, \mathscr{A})|$ then
28              $CD[tr] \leftarrow CD[tr] \cup \{tr_c\}$
29  return $CD$
```

Fig. 3. Algorithm 1: Compute control dependencies.

Control Dependencies When a fixpoint is found for all the sets of symbolic values S_{tr_i,tr_j}, phase (2) of the algorithm terminates and the non-termination sensitive control dependence relation can be deduced from a walk into the sets S_{tr_i,tr_j}, in phase (3). Namely, if the cardinality of the set S_{tr,tr_c} is strictly greater than 0, then it means that there is a successor of tr_c from which all the maximal paths include tr; if the cardinality of the set S_{tr,tr_c} is also lesser than the number of successors of tr_c, then it means that there exists a maximal path from a successor of tr_c, on which tr does not occur, and thus tr is control dependent on tr_c, by definition 10. Finally, the algorithm stores the control dependence relation in CD, which is an array of sets of transitions, indexed by the set of transitions in $\mathscr{A}$, and such that for all tr in $\mathscr{A}$, $CD[tr]$ is the set of transitions of $\mathscr{A}$ on which tr is control dependent.

4.1.4. Complexity Analysis

In this section, is evaluated the time complexity of algorithm 1. We did not detail the computation of *succs* and *conds*, which are assumed to be in the data structure – if not, these data can be computed in the worst case in $O(|T|^2)$.

Initialisation In phase (1), the loop at line 2 initialises the sets S_{tr,tr_c} for each transition tr in T, and each transition tr_c in $conds(\mathscr{A})$, in time $O(|T|.|conds(\mathscr{A})|)$. The loop at line 5 processes each successor of each conditional transition, adding, at each iteration, one element in the worklist, and one element in a set S_{tr_s,tr_c} (where tr_s is a successor of a conditional transition tr_c). Hence, each iteration of the loop at line 5 has a constant time cost, and this loop completes in time $O(\sum_{tr_c \in conds(\mathscr{A})} d(tr_c))$, where $d(tr_c)$ is the outer degree of tr_c – note that the following equality holds: $d(tr) = |succs(tr)|$.

Control-flow Analysis In phase (2), the termination condition of the main loop, at line 9, is that all the sets S_{tr,tr_c} stabilise: in this case, none of the conditionals at lines 14 and 21 can be true, and consequently no more transition can be added in the worklist. By construction, each set S_{tr,tr_c} contains at most $d(tr_c)$ elements (cf. section 4.1.3). In each iteration of the loop at line 9, either all the sets S_{tr,tr_c} remain the same, or there is at least one of these sets whose size is increased (at line 15 or 22). In the latter case only, a transition is added in the worklist, contributing one iteration of the main `while` loop (line 9). Consequently, given $tr \in T$ and $tr_c \in conds(\mathscr{A})$, the set S_{tr,tr_c} stabilises after at most $d(tr_c)$ iterations. Furthermore, given a transition tr in T, all the sets S_{tr,tr_c} stabilise after at most $\sum_{tr_c \in conds(\mathscr{A})} d(tr_c)$ iterations of the main loop. Finally, for all $tr \in T$ and $tr_c \in conds(\mathscr{A})$, the sets S_{tr,tr_c} stabilise after $O(|T|. \sum_{tr_c \in conds(\mathscr{A})} d(tr_c))$ iterations; this is thus the maximum number of iterations of the main loop.

In each iteration of the main loop, the loop at line 13 processes at most all the transitions in $conds(\mathscr{A})$, and the loop at line 20 processes at most all the transitions in $conds(\mathscr{A})$, for each transition in T. As a consequence, phase (2.1) iterates $O(|conds(\mathscr{A})|)$ times, and phase (2.2) iterates $O(|T|.|conds(\mathscr{A})|)$ times, both in the worst case. Hence, phase (2) has a total time complexity of $O(\sum_{tr_c \in conds(\mathscr{A})} d(tr_c).|T|^2.|conds(\mathscr{A})|.lg(|T|))$. The factor $lg(|T|)$ denotes the cost of operations on sets S_{tr,tr_c} (at lines 14, 15, 21 and 22): each of these sets contains at most $|T|$ elements.

Control Dependencies In phase (3), the algorithm computes the control dependence relation, in traversing the set $conds(\mathscr{A})$ for each transition in T (cf. section 4.1.3). At each iteration of the loop at line 25, at most one element is added in one of the sets $CD[tr]$; this can be done in constant time (indeed, no ordering of the sets $CD[tr]$ is required). Hence, the cost of phase (3) is in $O(|T|.|conds(\mathscr{A})|)$.

In conclusion, considering that the complexity of phase (2) dominates the complexity of phases (1) and (3), the overall complexity of algorithm 1 is $O(\sum_{tr_c \in conds(\mathscr{A})} d(tr_c).|conds(\mathscr{A})|.|T|^2.lg(|T|))$.

4.2. Data Dependence

Data dependencies are constraints arising from flows of data between statements. As we mentioned in the introduction of section 4, data dependencies have been widely studied in the areas of compiler construction and static analysis, leading to numerous optimisation and transformation techniques. This section first states a preliminary definition of variable *definitions* and *uses*, and then recalls that several intuitive ideas may underlie the notion of data dependence; therefore the notion of data dependence we use in the setting of slicing IOSTSs will be made clear. We conclude this section by describing our algorithm for the calculation of data dependence.

4.2.1. Variable Definitions and Uses

Definitions for data dependencies are usually stated in terms of variable definitions and uses. *In a program, a variable x is said to be* defined *at a statement s if a value is assigned to x at s; x is said to be* used *at s if either s is an assignment and x appears on the right-hand side of s, or s is a conditional and x appears in s.* The following definition is our proposition for extending these notions to IOSTSs. In IOSTSs, variables can be used in transition guards and can be used or defined in variable substitutions, in a similar fashion to conditionals and assignments in programs. Additionally, our extension to IOSTSs has to handle communication actions, where definitions and uses of variables are also involved.

Definition 11 (Variable definitions/uses). Let $tr = (s, a, f, \sigma, s')$ be a transition in T, and x be a variable in V.

- x is *defined* at tr if either $\sigma(x) \neq x$, or $a = c?x$ for some $c \in C$.
- x is *used* at tr if either x is a variable of formula f, or there exists a term $t \in \mathscr{T}_\Omega(V)$ such that x is a variable of term t, and one of the following is true: either $\sigma(y) = t$ for some $y \in V$, or $a = c!t$ for some $c \in C$.

Example 7. In figure 1, x is defined at $i \rightharpoonup j$ because x appears on the right hand side of an input action, ac is defined at $i \rightharpoonup j$ because ac appears on the left hand side of an assignment, x is used at $i \rightharpoonup j$ because x appears on the right hand side of an assignment, and tp is used at $e \rightharpoonup f$ because tp appears in the guard of $e \rightharpoonup f$.

4.2.2. Traditional Definitions

Depending on the aimed application, several definitions of data dependence have been stated. In the most general setting, the data dependence relation is introduced as a joint relation, composed of four varieties of data dependencies [Muc97]. Let n_i and n_j be two statements, such that n_i precedes n_j in their given execution order. A *flow dependence*, or *true dependence*, holds between n_i and n_j if n_i sets the value of a variable, and n_j uses this variable value; an *antidependence* holds between n_i and n_j if n_i uses a variable value and n_j sets it; an *output dependence* holds between n_i and n_j if both statements set the value of some variable; and finally an *input dependence* holds between n_i and n_j if both statements read the value of some variable. Data dependencies can be further classified as *loop-carried* or *loop-independent* [FOW87]. A data dependence between two statements is called loop-carried if the dependence arises when the two statements occur in two different instances of a loop[5], otherwise this dependence is called loop-independent.

Only flow dependence is relevant for the purpose of slicing, as it identifies statements that are directly involved in the computations performed at another statement. *In this work, the data dependence relation will therefore be restricted to flow dependencies.*

Further in this section, is described our algorithm for computing loop-carried *and* loop-independent data dependencies in an IOSTS. Our slicing algorithm indeed deals with the two kinds of data dependencies in the same manner, i.e. whenever a transition tr_j in the slicing criterion is data dependent on another transition tr_i, tr_i is included in the slice, no matter whether the data dependence between tr_i and tr_j is loop-carried or loop-independent. Therefore, the distinction will not be made, for the purpose of slicing IOSTS specifications; although Nanda [Nan01] found this distinction useful for the purpose of slicing multi-threaded programs, in the presence of threads nested in loops.

4.2.3. New Definition

A definition of a variable x at transition tr_i, such that there is no redefinition of x on a path to tr_j, is said to reach tr_j. A straightforward adaptation of the traditional definition of data dependencies to the IOSTS framework would state: a transition tr_j is data dependent on a transition tr_i in $\mathscr{A}$ if there exists a variable x that is defined at tr_i and used at tr_j, and the definition of x at tr_i reaches tr_j. However, this is not sufficient

[5] For instance, a definition in a loop iteration that is used in a subsequent iteration of the same loop.

according to the semantics of IOSTSs (cf. definition 3): it may indeed happen that x is redefined at tr_j by the mean of an input action. In this case, the definition of x at tr_i cannot be used at tr_j, unless x is also used in the guard of tr_j.

Definition 12 (Data dependence ($tr_i \xrightarrow{dd} tr_j$)). A transition $tr_j \in T$ is *data dependent* on a transition $tr_i = (s, a, f, \sigma, s') \in T$ if there exists a variable $x \in V$ and a path $p = \langle tr_i, \ldots, tr_j \rangle$ in $\mathscr{A}$ such that:

- x is defined at tr_i;
- and for all $tr \in \langle p(1), \ldots, tr_j \rangle$, x is not defined at tr;
- and one of the following is true:
  1. x is used at f,
  2. or x is not defined at a and x is used at tr.

Example 8. In figure 1, $m \rightharpoonup i$ is data dependent on $i \rightharpoonup j$ because there exists a variable (namely, ac) that is defined at $i \rightharpoonup j$, used at $m \rightharpoonup i$, not defined by any input action at $m \rightharpoonup i$, and such that there is a path from $i \rightharpoonup j$ to $m \rightharpoonup i$, on which ac is not redefined – namely, $\langle i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i \rangle$.

Now, let us insert a use of y at $j \rightharpoonup m$: there are two possibilities, either in the guard or in assignments. On one hand, if $j \rightharpoonup m$ is replaced by $j \rightharpoonup_1 m = (j, goldCh?y, y > 0, Id_{\{ac,x,y,del\}}, m)$, then the definition of y at $j \rightharpoonup k$ can be used at the guard of $j \rightharpoonup_1 m$, for instance, following the path $\langle j \rightharpoonup k, k \rightharpoonup_1 l, l \rightharpoonup i, i \rightharpoonup j, j \rightharpoonup_1 m \rangle$, and thus $j \rightharpoonup_1 m$ is data dependent on $j \rightharpoonup k$. On the other hand, if $j \rightharpoonup m$ is replaced by $j \rightharpoonup_2 m = (j, goldCh?y, true, (x \mapsto y + 1), m)$, then the definition of y at $j \rightharpoonup k$ cannot be used at $j \rightharpoonup_2 m$ because the input action on y at $j \rightharpoonup_2 m$ kills that definition; then $j \rightharpoonup_2 m$ is not data dependent on $j \rightharpoonup k$.

A definition d of a variable x at transition tr_i, that reaches tr_j, is called a *reaching definition* of tr_j. Considering definition 12, and assuming that we are able to calculate the reaching definitions on an IOSTS, then *the data dependence relation is given by marking transitions tr_j as data dependent on transitions tr_i, whenever there is a use at tr_j of a reaching definition from tr_i, that satisfies condition 1. or 2. of definition 12.* Specifically, the problem of finding all the reaching definitions in an IOSTS is for each transition tr to find all the definitions that may reach tr when executing the automaton.[6]

The problem of reaching definitions is a well-known application of dataflow analysis theory and algorithms. A dataflow analysis is generally defined as the analysis of a CFG in a theoretical framework that consists of a complete lattice and a set of monotone transfer functions [KSV96, Muc97]. Defining a dataflow analysis in such a framework is advantageous in proving its correctness and termination. Dataflow analyses of IOSTSs may be defined in a similar way.

4.2.4. Dataflow Analysis

A framework for dataflow analysis of IOSTSs consists of a complete lattice and a set of transfer functions. The lattice L denotes the partially ordered set of values – called dataflow *facts* – that are relevant to the analysis; values in L are intended to be associated to each transition of the automaton. The set of transfer functions contains a transfer function f_{tr} for each transition tr in the automaton. f_{tr} calculates the dataflow facts to be transferred to the successors of tr when tr is encountered during analysis. Notice that transfer functions should be monotone, and preferably distributive, for the solutions of dataflow analyses to be more precise [KSV96].

Reaching definitions is a forward dataflow analysis, i.e. the analysis starts from the initial node and then walks through the structure under analysis, in the same direction as the control flow; as opposed to postdominance (cf. section 4.1), that can be calculated via backward dataflow analysis from a supposed unique end node. *Reaching definitions are thus calculable on IOSTSs by deriving an algorithm from standard algorithms for solving forward dataflow analyses.*

There are two families of algorithms for solving dataflow analyses: *elimination methods* (e.g. interval, structural analyses) and *iterative algorithms* (e.g. worklist, round robin, node listing algorithms). Elimination methods are significantly harder to implement than iterative methods, and are mainly intended to

[6] Finding the definitions that *actually* reach a point is undecidable in general. It would be the same problem as requiring that p be a *feasible* path in definition 12.

efficiently handle dataflow information updates during a complex optimisation process [Muc97]. Among iterative methods, the worklist algorithm is the most flexible and allows optimisations that spare needless analyses of program statements. Other approaches may be called *dense* analyses [TGL06], in the sense that parts of the system under analysis are needlessly re-analysed. For instance, a round robin algorithm systematically analyses all the statements at each iteration, until a fixpoint is reached.

In the remainder of this section we present our worklist algorithm for computing data dependencies in an IOSTS.

4.2.5. Generic Dataflow Algorithm

As an introduction to our algorithm, we propose a brief description of a generic worklist algorithm for solving forward dataflow analyses on IOSTSs.

A set of transitions to visit is maintained by the algorithm, initialised with the set of all the transitions that have the initial state of the automaton as source state. At each iteration of the algorithm, a transition tr_l from the worklist is processed. This means that a new dataflow fact (i.e. an element of the lattice, e.g. reaching definitions) is computed for tr_l, using the transfer functions associated to tr_p, for each predecessor tr_p of tr_l. Then, transition tr_l is removed from the worklist; if the new data flow fact for tr_l is not included in the previously computed data flow fact for tr_l, then the successors of tr_l are added in the worklist. The algorithm iterates until a fixpoint is reached for all the dataflow facts.

4.2.6. Specific Algorithm

The generic dataflow algorithm, briefly described in section 4.2.5, may be instantiated to solve specific forward dataflow analyses, by providing the relevant lattice L, partially ordered by an order relation $\sqsubseteq$, and relevant transfer function space $\{f_{tr} : L \to L \mid tr \in T\}$. The following explains how algorithm 2 instantiates the generic algorithm to enable the computation of reaching definitions, and consequently data dependencies, in an IOSTS $\mathscr{A}$ (cf. figure 4).

Reaching Definitions A preliminary step (phase (1) of algorithm 2) is to mark all the variable definitions and uses in the automaton, according to definition 11; this marking requires a single-pass through the set of transitions. During this pass, encountered variable definitions and uses are respectively stored in def and ref, which are arrays of sets of variables, indexed by the set of transitions in $\mathscr{A}$. Specifically, $def[tr]$ and $ref[tr]$ denote the sets of variables that are respectively defined and used at transition tr. When initialising def and ref at lines 3 and 4, expressions $vars(t)$ and $vars(f)$, for terms $t \in \mathscr{T}_\Omega(V)$ and formulae $f \in \mathscr{F}_\Omega(V)$, denote the sets of variables that appear in t and f.

For implementation issues, we consider a variable definition as a couple (v, tr), meaning that variable v is defined at transition tr. The set of definitions $\mathcal{D}$ is the set of dataflow facts for the reaching definitions analysis, and the lattice L is the powerset of $\mathcal{D}$. By construction, L is partially ordered by subset inclusion $\subseteq$. For each transition tr we define a transfer function $f_{tr} : L \to L$ such that if D is a set of definitions reaching tr, then $f_{tr}(D)$ is the set of definitions reaching the successors of tr. This is done by the mean of defining gen and $kill$, still in phase (1). These arrays contain sets of definitions. Specifically, $gen[tr]$ denotes the set of definitions that will be propagated from tr, while $kill[tr]$ denotes the set of definitions, whose propagation will be stopped by tr. Then, for each transition tr in $\mathscr{A}$, f_{tr} is defined by $\forall D \in L, f_{tr}(D) = (D \backslash kill[tr]) \cup gen[tr]$. Reaching definitions information is updated in phase (2) each time a transition tr_l is processed, using functions f_{tr_p} for each predecessor tr_p of tr_l. If new reaching definitions have been found, the current knowledge of reaching definitions for tr_l (i.e. $RD[tr_l]$) is updated, and all the successors of tr_l are added in *worklist*. The algorithm iterates until a fixpoint on is found for all the sets $RD[tr]$.

Data Dependencies Once the reaching definitions information is known, the computation of the data dependence relation is performed in phase (3) of the algorithm, according to definition 12: a transition tr_j is data dependent on a transition tr_i, if there is a reaching definition from tr_i, that is used at the guard of tr_j, or is used at tr_j while being not killed by any input action at tr_j. The data dependence relation is stored in DD, which is an array of sets of transitions, indexed by the set of transitions in $\mathscr{A}$, and such that for all tr in $\mathscr{A}$, $DD[tr]$ is the set of transitions of $\mathscr{A}$ on which tr is data dependent.

```
Input: $\mathscr{A} = (S, s_0, T)_\Sigma$ : an IOSTS, where $\Sigma = (\Omega, V, C)$
Data:
  • $def[|T|], ref[|T|]$ : arrays of sets of variables
  • $gen[|T|], kill[|T|], RD[|T|]$ : arrays of sets of definitions
  • $newInfo$ : a set of definitions
  • $worklist$ : a set of transitions
Output: $DD[|T|]$ : an array of sets of transitions
// At the end of algorithm 2, for each $tr \in T$, $DD[tr]$ contains all the transitions,
// on which tr is data dependent.
/* (1) Initialisation                                                         */
1  $worklist \leftarrow \emptyset$
2  foreach $tr = (s, a, f, \sigma, s') \in T$ do
3      $def[tr] \leftarrow \{v \in V \mid \exists c \in C, a = c?v\} \cup \{v \in V \mid \exists x \in V, \sigma(x) \neq x \wedge v \in vars(\sigma(x))\}$
4      $ref[tr] \leftarrow \{v \in V \mid \exists c \in C, \exists t \in \mathscr{T}_\Omega(V), a = c!t \wedge v \in vars(t)\}$
           $\cup \{v \in V \mid v \in vars(f)\} \cup \{v \in V \mid \exists x \in V, \sigma(x) \neq x \wedge v \in vars(\sigma(x))\}$
5      $RD[tr] \leftarrow \emptyset$
6      $DD[tr] \leftarrow \emptyset$
7      $gen[tr] \leftarrow \{(v, tr_k) \in V \times T \mid tr_k = tr \wedge v \in def[tr]\}$
8      $kill[tr] \leftarrow \{(v, tr_k) \in V \times (T \backslash \{tr\}) \mid v \in def[tr]\}$
9      if $source(tr) = s_0$ then $worklist \leftarrow worklist \cup \{tr\}$
/* (2) Compute the sets of reaching definitions                               */
10 while $worklist \neq \emptyset$ do
11     $tr_l \leftarrow$ first$(worklist)$
12     $newInfo \leftarrow \emptyset$
       // Compute the definitions that reach $tr_l$ from all its predecessors,
       // and put the union in newInfo.
13     foreach $tr_p \in preds(tr_l)$ do
14         $newInfo \leftarrow newInfo \cup (RD[tr_p] \backslash kill[tr_p]) \cup gen[tr_p]$
       // If new reaching definitions have been found for $tr_l$,
       // then update $RD[tr_l]$ and insert all the successors of $tr_l$ in the worklist.
15     if $RD[tr_l] \subset newInfo$ then
16         $RD[tr_l] \leftarrow newInfo$
17         foreach $tr_s \in succs(tr_l)$ do
18             $worklist \leftarrow worklist \cup \{tr_s\}$
19     $worklist \leftarrow worklist \backslash \{tr_l\}$
/* (3) Compute data dependencies                                              */
20 foreach $tr = (s, a, f, \sigma, s') \in T$ do
21     foreach $(v, tr_k) \in RD[tr]$ do
           // $(v, tr_k)$ reaches tr.
           // Then tr is data dependent of $tr_k$ if either v appears in f,
           // or v is not defined at a, and in the meantime v is used at tr.
22         if $(v \in vars(f)) \vee ((\nexists c \in C, a = c?v) \wedge (v \in ref[tr]))$ then
23             $DD[tr] \leftarrow DD[tr] \cup \{tr_k\}$
24 return $DD$
```

Fig. 4. Algorithm 2: Compute data dependencies.

4.2.7. Complexity Analysis

This section provides an evaluation of the time complexity of algorithm 2.

Initialisation Phase (1) of the algorithm can be completed in a single walk through the set of transitions T: each transition tr is syntactically analysed to initialise $def[tr]$ and $ref[tr]$, in time proportional to the number of definitions in $\mathscr{A}$. Notice that in our implementation of this algorithm, gen and $kill$ are not explicitly stored in arrays; the reason why is explained in the sequel. Consequently, if we note $\mathcal{D}$ the set of definitions in $\mathscr{A}$, the complexity of phase (1) of the algorithm is in $O(|T|.|\mathcal{D}|)$.

Reaching Definitions $kill$ and gen information is only used at line 14, for computing the reaching definitions from tr_p to tr_l, tr_p being a predecessor of the transition under analysis, tr_l. The purpose of the enclosing loop at line 13 is to compute $\bigcup_{tr_p \in preds(tr_l)}((RD[tr_p] \backslash kill[tr_p]) \cup gen[tr_p])$, which is the new dataflow fact[7] for tr_l. Referring to section 4.2.6, it is equivalent to say that the loop at line 13 computes $\bigcup_{tr_p \in preds(tr_l)} f_{tr_p}(RD[tr_p])$. In our implementation of this loop, computing and storing the whole sets gen and $kill$ is unnecessary, because $(RD[tr_p] \backslash kill[tr_p])$ is deduced on-the-fly from $RD[tr_p]$ and $def[tr_p]$; and $gen[tr_p]$ is a straightforward transcription of $def[tr_p]$.

Each iteration of the main `while` loop (at line 10) may contribute further iterations only if the conditional at line 15 is satisfied. That is, if the new dataflow fact $newInfo$, computed by the loop at line 13 for tr_l, provides more information than the previously known set of reaching definitions for tr_l, i.e. $RD[tr_l]$. In this case, each successor of the processed transition is added in the worklist; this leads to a worst case total of $\sum_{tr \in T} d(tr)$ insertions in the worklist.

As we saw in section 4.2.6, transfer functions f_{tr} are defined on a lattice L. L is the powerset of $\mathcal{D}$, the set of definitions in $\mathscr{A}$, and is partially ordered by subset inclusion $\subseteq$. Let us show that the transfer functions f_{tr}, for all $tr \in T$, are monotone. For all D_1 and D_2 in L, and given k and g in L, we can say:

$$\begin{array}{lll} D_1 \sqsubseteq D_2 & \Longrightarrow & D_1 \subseteq D_2 \\ & \Longrightarrow & (D_1 \backslash k) \subseteq (D_2 \backslash k) \\ & \Longrightarrow & (D_1 \backslash k) \cup g \subseteq (D_2 \backslash k) \cup g \end{array}$$

In particular, let $k = kill[tr]$, and $g = gen[tr]$, then:

$$\begin{array}{lll} (D_1 \backslash k) \cup g \subseteq (D_2 \backslash k) \cup g & \Longrightarrow & f_{tr}(D_1) \subseteq f_{tr}(D_2) \\ & \Longrightarrow & f_{tr}(D_1) \sqsubseteq f_{tr}(D_2) \end{array}$$

As a summary, $\forall D_1, D_2 \in L,\ D_1 \sqsubseteq D_2 \implies f_{tr}(D_1) \sqsubseteq f_{tr}(D_2)$. This shows that f_{tr} are monotone functions.

Now, back to our algorithm: as we saw above, an iteration of the main loop (line 10) contributes $\sum_{tr \in T} d(tr)$ iterations in the worst case, which is possible only if the new dataflow fact computed for tr_l at line 14 strictly includes the previous dataflow fact for tr_l. As f_{tr_p} are monotone, and L is finite, this case can happen only a finite number of times, which is, in the worst case, the length of the longest chain in L. The longest chain in L "leads" from the least element $\emptyset$ to $\mathcal{D}$ at the top of the lattice; the length of this chain is $|\mathcal{D}| + 1$. Hence, the main loop completes after $O(|\mathcal{D}|. \sum_{tr \in T} d(tr))$ iterations. In each iteration of the main loop, the loop at line 13 has a cost in $O(|T|.lg(|\mathcal{D}|))$: the factor $lg(|\mathcal{D}|)$ reflects the cost of set operations at line 14, where each set contains at most all the elements of $\mathcal{D}$. Finally, the complexity of phase (2) is $O(\sum_{tr \in T} d(tr).|T|.|\mathcal{D}|.lg(|\mathcal{D}|))$.

Data Dependencies Algorithm 2 computes the data dependence relation in phase (3), by matching reaching definitions and uses of variables, as explained in section 4.2.6. In the worst case, all the definitions reach each transition, and then the complexity of phase (3) is $O(|T|.|\mathcal{D}|)$ (same as phase (1)).

In conclusion, the overall complexity of algorithm 2 is $O(\sum_{tr \in T} d(tr).|T|.|\mathcal{D}|.lg(|\mathcal{D}|))$, since the complexity of phase (2) dominates the complexity of phases (1) and (3).

[7] in this case, the new set of reaching definitions for tr_l.

4.3. Communication Dependence

An IOSTS communicates with its environment via communication actions (*cf.* section 3.1.2). As we saw in section 4.2, a variable may be defined through an input action, that is, the variable is assigned a value that depends on variables used in the corresponding output actions. This remark shows that communication actions induce inter-automata data dependencies. Communication actions also induce inter-automata control dependencies, as an input action may be executed only if a corresponding output action is performed. Our definition of a *communication dependence* encompasses the different kinds of dependencies induced by communication actions. These dependencies are not handled by our definitions of control and data dependencies (in sections 4.1 and 4.2), since these definitions only consider intra-automata control flows and data flows.

Informally, there is a communication dependence between two transitions in two different IOSTSs if there exists a channel that potentially allows a control flow or a data flow to occur between these two transitions. Let $\mathscr{S}$ be a specification that contains at least 2 IOSTSs; let $(S_i, s_{0i}, T_i)_{\Sigma_i}$ and $(S_j, s_{0j}, T_j)_{\Sigma_j}$ be two distinct IOSTSs in $\mathscr{S}$, where $\Sigma_i = (\Omega_i, V_i, C_i)$, and $\Sigma_j = (\Omega_j, V_j, C_j)$.

Definition 13 (Communication dependence ($tr_i \overset{com}{\longleftrightarrow} tr_j$)). Two transitions $tr_i = (s_i, a_i, f_i, \sigma_i, s'_i) \in T_i$ and $tr_j = (s_j, a_j, f_j, \sigma_j, s'_j) \in T_j$ are *communication dependent* on each other if there exists a channel $c \in C_i \cap C_j$ such that:
$$(1)\left\{\begin{array}{l}\bullet\ a_j = c?x \text{ for some } x \in V_j;\\ \bullet \text{ and } a_i = c!t \text{ for some } t \in \mathscr{T}_\Omega(V_i).\end{array}\right. \quad\text{or}\quad (2)\left\{\begin{array}{l}\bullet\ a_j = c?\\ \bullet \text{ and } a_i = c!\end{array}\right.$$

The communication dependence relation $\overset{com}{\longleftrightarrow}$ denotes all the dependencies induced by communication actions. In cases (1) and (2), there are two inter-automata control dependencies $tr_i \xrightarrow{com} tr_j$ and $tr_j \xrightarrow{com} tr_i$: by definition of the rendezvous mechanism, tr_i and tr_j control each other's execution.

Furthermore, there is an inter-automata data dependence $tr_i \xrightarrow{com} tr_j$ in case (1), as a variable definition at tr_j uses variable values at tr_i. Note that the notion of inter-automata data dependence is not analogous to the notion of data dependence, defined in section 4.2. As will be seen in the sequel, the inter-automata dependence $tr_i \xrightarrow{com} tr_j$, in conjugation with the data dependence relations, is useful as a link between tr_j and the transitions tr_d, such that variables that are used in the communication action at tr_i may refer to a definition at tr_d (in other terms, such that tr_i is data dependent on tr_d, according to definition 12). These indirect dependencies will be caught by our slicing algorithm (cf. section 4.5); for that reason, the aforementioned indirect dependencies are not explicitly included in the definition of $\xrightarrow{com}$.

Example 9. In figure 1, $j \rightharpoonup k$ and $e \rightharpoonup f$ are mutually communication dependent: there exists a channel (namely, *normalCh*), on which $e \rightharpoonup f$ performs an output action and $j \rightharpoonup k$ performs an input action. Moreover, the inter-automata data dependence $(e \rightharpoonup f) \xrightarrow{com} (j \rightharpoonup k)$ induce an indirect data dependence of $j \rightharpoonup k$ on $d \rightharpoonup e$, since the definition of the variable *am* at $d \rightharpoonup e$ is used at the communication action at $e \rightharpoonup f$ (i.e. $(d \rightharpoonup e) \xrightarrow{dd} (e \rightharpoonup f)$).

4.3.1. Related Work

Related works on communication dependencies may be found mainly in the field of static analysis of multi-threaded programs.

In [Sar97], Sarkar defined a dataflow analysis on parallel graphs (called PPGs), which communicate through events. Dataflow analysis is made more precise by taking into account the synchronisation constraints imposed by the `wait`/`post` communication mechanism. In PPGs, a `wait` statement for an event e has to wait for all its synchronisation predecessors (i.e. the `post` statements for e) to complete execution. This does not work on IOSTSs, where an input action on a channel c has to wait for only one of the corresponding output actions on c to complete execution (cf. 3.2).

Millett *et al.* define in [MT00] a method for slicing Promela (the input language of the Spin model checker). In [MT00], channels are handled as variables, thus no communication dependence relation is defined, but their handling of shared variables is similar to our handling of communication dependencies. The reader is referred to section 5 for more considerations on related work.

```
Input: $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$ :
    a specification formed of $k$ IOSTSs (given $k \in \mathbb{N}^*$)
Data:
  • $emit[|\bigcup_{0 \leq i < k} C_i|]$ : an array of sets of transitions
  • $rcv[|\bigcup_{0 \leq i < k} C_i|]$ : an array of sets of transitions
  • $C$ : a set of channels
Output: $ComD[|\bigcup_{0 \leq i < k} T_i|]$ : an array of sets of transitions
// At the end of algorithm 3, for each tr $\in T$, $ComD[tr]$ contains all the transitions,
// on which tr is communication dependent.
/* (1) Initialisation                                                        */
1  $C \leftarrow \bigcup_{0 \leq i < k} C_i$
2  $T \leftarrow \bigcup_{0 \leq i < k} T_i$
3  foreach $c \in C$ do
4      $emit[c] \leftarrow \emptyset$
5      $rcv[c] \leftarrow \emptyset$
6  foreach $tr = (s, a, f, \sigma, s') \in T$ do
7      $ComD[tr] \leftarrow \emptyset$
8      Let $i$ be such that $tr \in T_i$
9      if a matches with $c!$ or $c!x$, where $c \in C_i$ and $x \in V_i$ then
10         $emit[c] \leftarrow emit[c] \cup tr$
11     if a matches with $c?$ or $c?x$, where $c \in C_i$ and $x \in V_i$ then
12         $rcv[c] \leftarrow rcv[c] \cup tr$
/* (2) Compute communication dependencies                                    */
13 foreach $c \in C$ do
14     foreach $tr_e \in emit[c]$ do
15         Let $i$ be such that $tr_e \in T_i$
16         foreach $tr_r \in rcv[c]$ do
17             Let $j$ be such that $tr_r \in T_j$
18             if $i \neq j$ then
                   // $tr_r$ is inter-automata control dependent and data dependent on $tr_e$.
19                 $ComD[tr_r] \leftarrow ComD[tr_r] \cup \{tr_e\}$
                   // $tr_e$ is inter-automata control dependent on $tr_r$.
20                 $ComD[tr_e] \leftarrow ComD[tr_e] \cup \{tr_r\}$
21 return $ComD$
```

Fig. 5. Algorithm 3: Compute communication dependencies.

4.3.2. Algorithm

Algorithm 3 enables the computation of the communication dependence relation in an IOSTS specification $\mathscr{S}$ (cf. figure 5).

Initialisation Phase (1) of the algorithm fills two arrays of transitions, *emit* and *rcv*, indexed by the set of channels in $\mathscr{S}$. *emit* denotes for each channel c the set of transitions that perform an output action on c, while *rcv* denotes for each channel c the set of transitions that perform an input action on c.

Communication Dependencies Phase (2) processes every transition, marking transitions tr_e and tr_r as communication dependent on each other, whenever tr_r performs an input action on a channel and tr_e performs an output action on the same channel – specifically, this is done by inserting tr_e in $ComD[tr_r]$ and tr_r in $ComD[tr_e]$, where $ComD[tr_r]$ (resp. $ComD[tr_e]$) denotes the set of transitions, on which tr_r (resp. tr_e) is communication dependent. Algorithm 3 is conservative, but efficient (cf. section 4.3.3).

4.3.3. Complexity Analysis

In this section, is evaluated the time complexity of algorithm 3.

Initialisation Loops at lines 3 and 6 complete in a single walk through respectively the set of channels C and the set of transitions T in specification $\mathscr{S}$; then phase (1) completes in $O(|C| + |T|)$.

Communication Dependencies By definition 2, a transition of an IOSTS is labelled by at most one communication action. Thus, once the `for` loop at line 6 has completed, each transition is either in *emit* or in *rcv*, i.e. the total number of elements in "$emit \cup rcv$" is at most the total number of transitions in T. Consequently, the loop at line 14 completes in $O(|T|)$. Phase (2) consists of executing the loop at line 14 for each element of C, hence phase (2) completes in $O(|C|.|T|)$.

Finally, the overall complexity of algorithm 3 is $O(|C|.|T|)$.

4.4. On Correct, Precise and Optimal Slicing

We say a transition tr_j is *directly dependent* on a transition tr_i, denoted as $tr_i \xrightarrow{d} tr_j$, if tr_j is either control dependent, data dependent or communication dependent on tr_i (i.e. $\xrightarrow{d}$ is the union of $\xrightarrow{cd}$, $\xrightarrow{dd}$, and $\xrightarrow{com}$).

Intuitively, tr_j is *indirectly dependent* on tr_i if there is a sequence of dependencies in $\xrightarrow{d}$, leading from tr_i to tr_j. Still intuitively, tr_j is *transitively dependent*[8] on tr_i if tr_j is indirectly dependent on tr_i, with the additional condition that there is a path in the specification that corresponds to the sequence of dependencies that leads from tr_i to tr_j.

Definition 14 (Indirect/Transitive Dependence). A transition tr_j is *transitively dependent* on a transition tr_i in a specification $\mathscr{S}$, if there is a sequence $[tr_1, \ldots, tr_k]$ where $tr_1 = tr_i$ and $tr_k = tr_j$, such that for all $1 \leq m < k$:

1. $tr_m \xrightarrow{d} tr_{m+1}$
2. and there is a path $\langle tr_m, \ldots, tr_{m+1} \rangle$ in the parallel composition of $\mathscr{S}$.

If condition 1. is satisfied, then we say that tr_j is *indirectly dependent* on tr_i.

4.4.1. General Results

The minimum property a slicer should satisfy, is to be *correct*. That is, transitions that actually have an influence on the given slicing criterion should not be sliced away, and thus a specification is a correct slice of itself (it corresponds to the most conservative slicing algorithm, which is analogous to the identity function).

Intuitively, a slicing algorithm is *optimal* if, given a slicing criterion, it produces a slice that contains only the transitions that actually have an influence on the criterion. An optimal slice is then the smallest correct slice, in terms of number of transitions: there exists no better solution, since further slicing any transition away would invalidate the correctness of the slice (and adding transitions would lower the precision).

However, the following undecidability result prevents slicing algorithms from being optimal – it actually concerns most of the static analyses: it is undecidable whether a condition can be satisfied within the possible executions of the system [Göd31]. Weiser [Wei81] showed, as a consequence, that optimal program slicing is undecidable. Even in a decidable arithmetic (e.g. Presburger arithmetic [Pre30]), the undecidability of the termination problem still prevents slicing algorithms from being optimal. In dataflow analyses, a common way to cope with these difficulties, is to define feasible paths and optimality of a solution in a non-deterministic version of the system under analysis, i.e. branching choices are interpreted as non-deterministic choices. Our definition of a transitive dependence (definition 4.4) underlines such an abstraction, i.e. guards are not considered when thinking of feasible paths.

We consider that a dependence-based slicing algorithm is *precise* if it takes into account all the transitive

[8] One could find more appropriate the name of *precise dependence* for this concept. However, our definition can be understood as an extension of Krinke's transitive dependence [Kri98] to IOSTS specifications.

dependencies of the specification, and only transitive dependencies. Loosely speaking, precise slicing is of intermediary precision, between correct slicing and optimal slicing.

4.4.2. Dependence Properties

Now we point out three important properties of the dependence relations, concerning specification slicing.

1. By definition, data dependencies and control dependencies are always transitive. Definitions 10 and 12 imply that whenever there is a data or control dependence between two transitions, there exists a path between these transitions in the automaton, and consequently there exists a path in the parallel composition of the specification, too (this leads us to the result);
2. Referring to the definition of a transition in an IOSTS (definition 2), since there is a unique communication action – possibly silent (τ) – per transition, then whenever $tr_i \xrightarrow{com} tr_j$, there exists no tr_k in the specification, such that $tr_k \xrightarrow{com} tr_i$ or $tr_j \xrightarrow{com} tr_k$;
3. Let $\mathscr{A}_1$ and $\mathscr{A}_2$ be two IOSTSs; let tr_{i_1}, tr_{j_1} be two transitions in $\mathscr{A}_1$, and tr_{i_2}, tr_{j_2} be two transitions in $\mathscr{A}_2$. If $tr_{i_1} \xrightarrow{cd,dd} tr_{j_1}$, $tr_{j_1} \xrightarrow{com} tr_{i_2}$ and $tr_{i_2} \xrightarrow{cd,dd} tr_{j_2}$ hold, then using property 1. and according to the interleaving semantics of a specification (see definition 6): if the rendezvous is feasible, then there exists a path $\langle tr_{i_1}, \ldots, tr_c, \ldots, tr_{j_2} \rangle$ in the parallel composition of $\mathscr{A}_1$ and $\mathscr{A}_2$, where tr_c is the transition resulting from the parallel composition of tr_{j_1} and tr_{i_2}. If the rendezvous is not feasible, then the precision of the resulting slice is lowered: irrelevant transitions may be included in the slice.

4.4.3. Discussion

Property 2. indicates that property 3. can be generalised to the relation $\xrightarrow{d}$. Property 3. indicates a potential loss of precision: the relation $\xrightarrow{d}$ is not transitive in the case where no rendezvous is feasible, in the non-deterministic abstraction of the specification, between two transitions that are related by a communication dependence. However, the information of which rendezvous are feasible is contained in the parallel composition of the specification. This point suggests two possible improvements of the accuracy of our method.

The first improvement is slicing the parallel composition of the specification with respect to data and control dependence only; this solution is not suitable for the purpose of specification understanding, debugging, or even simulation: for these purposes, it is indeed preferable that the slice be under the form of a set of concurrent IOSTSs, and keep track of internal communication actions. Moreover, as the parallel composition of a specification is of exponential size in the worst case, computing it as a first step of a model reduction process seems not a convincing solution.

The second improvement consists of using the parallel composition information to refine our definition of a communication dependence. This solution seems interesting, but only an empirical study will tell whether the accuracy improvement is worth the overhead induced by checks in the parallel composition of the specification (as mentionned above, this overhead might indeed be exponential).

4.4.4. Precision Considerations

Still, in this paper we keep definition 13 for communication dependence, and the sequel explains our reasons.

First of all, every possible rendezvous is taken into account by our communication dependence relation, hence the correction of resulting slices is not compromised.

Second, definition 13 allows the design of a simple and very efficient algorithm for the computation of communication dependencies[9].

Third, property 3. also shows that, in the case where for all the couples of transitions that are related by a communication dependence, the corresponding rendezvous is feasible in the non-deterministic abstraction of the specification, then $\xrightarrow{d}$ is transitive. That is, whenever $tr_i \xrightarrow{d} * tr_j$ holds, tr_j is transitively dependent on tr_i. Now, experience suggests that it is reasonable to make that assumption: in practical specifications, most of couples of transitions, that according to definition 13 are communication dependent, are designed to

[9] Krinke's and Nanda's algorithms for computing transitive dependencies have a worst-case exponential complexity, although they claim their implementations are practical [Kri03, Nan01].

run in parallel. In this case a rendezvous is feasible in the non-deterministic abstraction of the specification; under this assumption, $\xrightarrow{d}$ is transitive.

4.5. Specification Slicing

Having calculated the three dependence relations $\xrightarrow{cd}$, $\xrightarrow{dd}$ and $\xrightarrow{com}$, respectively defined in sections 4.1, 4.2, and 4.3, and given the following definition, the construction of a dependence graph is straightforward.

Definition 15 (Dependence graph). Let $\mathscr{S}$ be a specification, where $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k\}$ for some $k \in \mathbb{N}^*$. A graph $G = (N, E)$ is a dependence graph for $\mathscr{S}$ if and only if:

- There is a bijection between the sets N and $\bigcup_{0\leq i<k} T_i$;
- and for each couple of transitions (tr_i, tr_j) such that $tr_i \xrightarrow{d} tr_j$ holds, if we call n_i and n_j the nodes that respectively represent tr_i and tr_j, then there is an edge from n_i to n_j in the dependence graph.

Referring to the main intuition of slicing (cf. section 2), the information on which the extraction of slices is mainly based, is the data dependence relation; however, as we saw in previous sections, control dependencies and communication dependencies have to be taken into account too, for the slices respectively to preserve some dynamic properties of the original system, and to handle the data flows and control flows occurring when automata communicate in the system.

The following definition is our definition of a *backward* specification slice; it could still be easily adapted to calculate *forward* slices (cf. [Kri03], and footnotes in section 1). A slice of a specification is computed with respect to a *slicing criterion*, which is a set of transitions of the specification. Informally, a *specification slice* is a specification, such that for each transition *tr* in the slice, there is at least one transition in the slicing criterion, that is indirectly dependent on *tr*. The construction of a slice with respect to a slicing criterion *Crit* proceeds by finding the sets of transitions T'_i, from which the transitions in *Crit* are reachable via the transitive closure of the dependence relation $\xrightarrow{d}$, noted $\xrightarrow{d}*$.

Definition 16 (Slice of a specification). Let $\mathscr{S}$ be a specification, where $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$, for some $k \in \mathbb{N}^*$. Let $T = \bigcup_{0\leq i<k} T_i$ be the global set of transitions, and $Crit \subseteq T$ be a slicing criterion.

For each $0 \leq i < k$, let $T'_i = \{tr \in T_i \mid \exists tr' \in Crit,\ tr \xrightarrow{d}* tr'\}$, and $S'_i = \bigcup_{tr\in T'_i}\{source(tr), target(tr)\}$.

For each $0 \leq i < k$, let V'_i and C'_i be respectively the set of variables and the set of channels that appear in T'_i.

Then, the specification $Slice_{Crit}(\mathscr{S})$ is a slice of $\mathscr{S}$, with respect to *Crit*:

$$Slice_{Crit}(\mathscr{S}) = \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid 0 \leq i < k \wedge T'_i \neq \emptyset \wedge \Sigma'_i = (\Omega_i, V'_i, C'_i)\}$$

Remark By definition, a slice may be smaller than the original specification, in terms of transitions, but also in terms of automata: let $k' = |Slice_{Crit}(\mathscr{S})|$, then $k' \leq k$ in general. In the case there is an IOSTS, in which no transition may influence the criterion, then $k' < k$ holds.

Our definition of a specification slice is more intuitive in the perspective of the dependence graph. A slice of a specification, with respect to a criterion, is the set of transitions such that there exists a path from their corresponding node in the dependence graph to a node that corresponds to a transition in the criterion.

4.5.1. The Slicing Algorithm

General Idea An efficient algorithm to extract slices from specifications, according to our definitions, starts with the construction of a *dependence graph* (cf. definition 15). The general idea is to extract slices from a specification by identifying the nodes that are backward reachable in the dependence graph, from the nodes that represent the transitions of the criterion; the desired slice is then formed from the transitions that correspond to the nodes identified in the previous step.

Algorithm Description Based on this general idea, algorithm 4 automatically extracts a slice $\mathscr{S}'$ from a specification $\mathscr{S}$, given a slicing criterion *Crit* (cf. figure 6). However, the dependence graph is not explicitly

```
Input:
  • $\mathscr{S} = \{\mathscr{A}_i = (S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$ :
    a specification formed of $k$ IOSTSs (for a given $k \in \mathbb{N}^*$)
  • $Crit$ : a set of transitions
Data:
  • $CD[|\bigcup_{0\leq i<k} T_i|]$, $DD[|\bigcup_{0\leq i<k} T_i|]$, $ComD[|\bigcup_{0\leq i<k} T_i|]$ : arrays of sets of transitions
  • $worklist$ : a set of transitions
Output: $\mathscr{S}' = \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid 0 \leq i < k \ \wedge \ \Sigma'_i = (\Omega_i, V'_i, C'_i)\}$ : a specification formed of
        $k'$ IOSTSs, where $k' \leq k$
// $\mathscr{S}'$ is built over the sets $T'_i$ of transitions that induce a dependence over the criterion.
/* (1) Compute dependencies                                                    */
$CD \leftarrow \emptyset$ ; $DD \leftarrow \emptyset$ ; $ComD \leftarrow \emptyset$
foreach $i \in \{0 \ldots k-1\}$ do
    $CD \leftarrow CD \cup$ (ComputeControlDependencies($\mathscr{A}_i$))
    $DD \leftarrow DD \cup$ (ComputeDataDependencies($\mathscr{A}_i$))
if $k > 1$ then $ComD \leftarrow$ (ComputeCommunicationDependencies($\mathscr{S}$))
/* (2) Initialisation                                                          */
$worklist \leftarrow \emptyset$
foreach $i \in \{0 \ldots k-1\}$ do
    $S'_i \leftarrow \emptyset$ ; $T'_i \leftarrow \emptyset$ ; $V'_i \leftarrow \emptyset$ ; $C'_i \leftarrow \emptyset$
foreach $tr \in Crit$ do
    Let $i$ be such that $tr \in T_i$
    $T'_i \leftarrow T'_i \cup \{tr\}$
    $worklist \leftarrow worklist \cup \{tr\}$
/* (3) Find the transitions that induce a dependence on the criterion          */
while $worklist \neq \emptyset$ do
    $tr_l \leftarrow$ first($worklist$)
    foreach $tr \in (CD[tr_l] \cup DD[tr_l] \cup ComD[tr_l])$ do
        if $\neg(\exists j,\ tr \in T'_j)$ then
            Let $i$ be such that $tr \in T_i$
            $T'_i \leftarrow T'_i \cup \{tr\}$
            $worklist \leftarrow worklist \cup \{tr\}$
    $worklist \leftarrow worklist \backslash \{tr_l\}$
/* (4) Finalise the slice with respect to the criterion                        */
foreach $i \in \{0 \ldots k-1\}$ do
    RecoverReachability($\mathscr{A}_i$)
    foreach $tr = (s, a, f, \sigma, s') \in T'_i$ do
        $S'_i \leftarrow S'_i \cup \{source(tr), target(tr)\}$
        $V'_i \leftarrow V'_i \cup def[tr] \cup ref[tr]$
        if $a$ matches with $c!$, $c!x$, $c?$, or $c?x$ then
            $C'_i \leftarrow C'_i \cup \{c\}$
$\mathscr{S}' \leftarrow \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid T'_i \neq \emptyset\}$
$k' \leftarrow |\mathscr{S}'|$
return $\mathscr{S}'$
```

Fig. 6. Algorithm 4: Slicing IOSTS specifications.

built; rather, the dependence relations are stored under the form of arrays CD, DD and $ComD$, indexed by the transitions of the specification, such that for each tr in the specification, $CD[tr]$, $DD[tr]$ and $ComD[tr]$ denote the sets of transitions, on which tr is respectively control, data and communication dependent.

In phase (1), the loop at line 2 computes the control and data dependence relations for each IOSTS in $\mathscr{S}$, using respectively algorithms 1 and 2; the resulting control and data dependence information is stored respectively in CD and DD. Here we indulge a slight notation abuse, for saving clarity in algorithm 4: strictly speaking, lines 3 and 4 should not be written as set unions, since CD and DD are arrays. In our implementation, line 3 is replaced by two instructions: first, (Compute control dependencies ($\mathscr{A}_i$)), and second, a loop where, for each $tr \in \mathscr{A}_i$, $CD[tr]$ is set accordingly. Line 4 can be replaced analogously. Finally, if the specification contains at least 2 IOSTS, then the communication dependence relation is computed using algorithm 3, and is stored in $ComD$.

In phase (2) of algorithm 4, the sliced specification $\mathscr{S}'$ is initialised with the transitions of $Crit$, as the criterion is the basis of the final slice. The worklist will be used to find backward reachability information in the dependence graph, from transitions in $Crit$; $worklist$ is therefore initialised with the set of transitions in $Crit$.

Once phase (1) has been completed, $CD[tr_j] \cup DD[tr_j] \cup ComD[tr_j]$ denote for each tr_j the set of transitions tr_i, on which tr_j is dependent, i.e. $\{tr_i \mid tr_i \xrightarrow{d} tr_j\}$. This property is used in phase (3), to deduce backward reachability in the dependence graph, from the sets CD, DD and $ComD$. The loop at line 13 processes each transition in the worklist, exactly once. For each processed transition, each of its predecessors in the dependence graph, that has not been processed yet, is inserted in the worklist (this is done in the loop at line 15). Each processed transition is inserted in the appropriate set T_i'.

Finally, the construction of the slice $\mathscr{S}'$ is achieved in phase (4): the loop at line 21 calls a procedure *RecoverReachability*, for each automaton in the specification (cf. the following reachability remark), and sets the remaining data of the slice $\mathscr{S}'$, according to the transitions that have been inserted in sets T_i' (here we suppose that def and ref information, computed by algorithm 2, is still available). At line 28, all the non-empty automata are inserted in $\mathscr{S}'$, and then the construction of $\mathscr{S}'$ ends with setting the value of k', the definitive size of the slice, in terms of automata.

Reachability Remark Once phases (1), (2) and (3) of algorithm 4 are achieved, the sets of relevant transitions T_i' have been successfully constructed. Then it may happen that transitions that were reachable in the specification are no longer reachable in the slice formed of the sets of transitions T_i'; this problem is solved by processing an additional reduction procedure, called *RecoverReachability*, for each automaton. This procedure call is optional to the user of our prototype tool, and we implemented two different algorithms for reachability recovery, inspired from τ-reduction of labelled transition systems, and ϵ-reduction of finite non-deterministic automata with ϵ-transitions. The algorithms will not be described in this paper; basically, the reduction procedures modify source and target nodes of transitions (cf. example 10).

Example 10. Let us insert a use of nm at $c \rightharpoonup d$, i.e. $c \rightharpoonup d$ is replaced by $c \rightharpoonup_1 d = (c, goldCh?y, true, (nm \mapsto$ "customer "+nm$), d)$. Given the slicing criterion $\{c \rightharpoonup_1 d\}$, algorithm 4 determines that only transitions $a \rightharpoonup b$ and $c \rightharpoonup_1 d$ may influence the criterion. The reachability in the slice is then restored by creating a new state *b_c*, to be both the target state of $a \rightharpoonup b$ and the source state of $c \rightharpoonup_1 d$.

Reuseability Remark Noticing that the dependence graph is a highly reusable data structure, a dependence-based approach to slicing is very efficient for the purpose of computing multiple slices of a given specification. Actually, once the dependence graph has been constructed for a given specification, numerous slices can be extracted from the specification, each of which in nearly linear time (namely, $O(|T|.lg(|T|))$, cf. section 4.5.2). For that purpose, phase (1) is performed exactly once, for a given specification, and then phases (2) and (3) are performed consecutively, for each different criterion.

4.5.2. Complexity Analysis

This section provides an evaluation of the time complexity of algorithm 4.

Dependencies Computation In phase (1), the loop at line 2 computes the data and control dependence relations, for each IOSTS $\mathscr{A}_i$ in the input specification $\mathscr{S}$. Using the results of section 4.1.4, the complexity of line 3 is $O(\sum_{tr_c \in conds(\mathscr{A}_i)} d(tr_c).|conds(\mathscr{A})|.|T_i|^2.lg(|T_i|))$. Using the results of section 4.2.7, if we call $\mathcal{D}_i$

the set of variable definitions in $\mathscr{A}_i$, then the complexity of line 4 is $O(\sum_{tr \in T_i} d(tr).|T_i|.|\mathcal{D}_i|.lg(|\mathcal{D}_i|))$. As $conds(\mathscr{A}_i)$ is the set of conditional transitions in $\mathscr{A}_i$ (cf. section 4.1.3), then by definition $conds(\mathscr{A}_i) \subseteq T_i$. Furthermore, it is reasonable to claim that, in practical specifications, $\mathcal{D}_i$ is proportional to T_i. Consequently, the complexity of an iteration of the loop at line 2, for computing the dependencies of $\mathscr{A}_i$, is dominated by $O(E)$, where $E = \sum_{tr \in T_i} d(tr).|T_i|^3.lg(|T_i|)$. The overall complexity of the loop at line 2 is dominated by the sum, for all i, of expression E. However, as $O(\sum_i |T_i|^3.lg(|T_i|))$ is dominated by $O(|T|^3.lg(|T|))$, the complexity of the loop at line 2 is dominated by $O(\sum_{tr \in T} d(tr).|T|^3.lg(|T|))$.

Using the results of section 4.3.3, the complexity of line 5 is $O(|C|.|T|)$. By definition of a transition of an IOSTS, $|C| \leq |T|$ (cf. definition 2). The complexity of line 5 is thus in $O(|T|^2)$, which is dominated by the complexity of the loop at line 2, as we saw above. In conclusion, the complexity of phase (1) is dominated by $O(\sum_{tr \in T} d(tr).|T|^3.lg(|T|))$.

Initialization In phase (2), data of the output specification is initialised, and all the transitions of the criterion are inserted in the worklist. This phase completes in $O(|T|)$, although there are usually much less transitions in the criterion than in T.

Slice Computation In the loop at line 13, each time a transition is processed, it is added in the slice, and in the worklist (lines 18 and 19). The test at line 16 thus ensures that each transition is processed at most once. The test at line 16, and the processing of a transition, both have a constant-time cost. Consequently, the complexity of the loop at line 13 is $O(|T|)$. The loop at line 21 processes each transition of the specification, exactly once; processing one transition has a cost of $O(lg(|\mathcal{D}_i|))$, due to the set operation at line 25. The complexity of the loop at line 21 is thus dominated by $O(\sum_i |T_i|.lg(|\mathcal{D}_i|))$. Again, in practical specifications, $|\mathcal{D}_i|$ is proportional to $|T_i|$. If we note $\mathcal{D}$ the global set of variable definitions in the specification, then $|\mathcal{D}|$ is proportional to $|T|$, and the complexity of phase (3) is dominated by $O(|T|.lg(|T|))$.

In conclusion, the overall complexity of algorithm 4 is dominated by $O(\sum_{tr \in T} d(tr).|T|^3.lg(|T|))$. Notice that this complexity result was obtained after several over-approximations[10] to simplify the expression. Referring to the reuseability remark in section 4.5.1, once a first slice has been extracted, each subsequent slice can be extracted in $O(|T|.lg(|T|))$.

4.5.3. Application Example

Figure 7 is a representation of the slice of the specification in figure 1, with respect to the slicing criterion $\{m \rightharpoonup i\}$, that is produced by our slicing algorithm (cf. algorithm 4). Shaded transitions in figure 7 are not included in the slice. To obtain this result, the slicing algorithm first computes all the data, control and communication dependencies in the specification, and then it intuitively starts from the slicing criterion, and collects all the transitions that are backward reachable from it, in the dependence graph. Transitions collected during the search form the desired slice.

In figure 7, $m \rightharpoonup i$ is data dependent on $j \rightharpoonup m$ because of the definition and uses of y at these transitions. $j \rightharpoonup m$ is in turn control dependent on $i \rightharpoonup j$, by definition (cf. section 4.1), and communication dependent on $e \rightharpoonup g$, because of the potential rendezvous between these transitions, on channel *goldCh*. Analogously, $b \rightharpoonup c$, $c \rightharpoonup d$ and $d \rightharpoonup e$ are backwards reachable in the dependence graph, from $i \rightharpoonup j$ and $e \rightharpoonup g$, and this process continues until no more transitions can be added to the slice. To summarise, following is a representation of a search through the dependence graph that leads to the desired slice:

$$(m \rightharpoonup i) \overset{dd}{\longleftarrow} (j \rightharpoonup m) \begin{cases} (j \rightharpoonup m) \overset{cd}{\longleftarrow} (i \rightharpoonup j) \left\{ (i \rightharpoonup j) \overset{com}{\longleftarrow} (b \rightharpoonup c) \left\{ (b \rightharpoonup c) \overset{dd}{\longleftarrow} (a \rightharpoonup b) \right.\right. \\ (j \rightharpoonup m) \overset{com}{\longleftarrow} (e \rightharpoonup g) \begin{cases} (e \rightharpoonup g) \overset{dd}{\longleftarrow} (c \rightharpoonup d) \\ (e \rightharpoonup g) \overset{dd}{\longleftarrow} (d \rightharpoonup e) \end{cases} \end{cases}$$

Many other dependencies exist in the dependence graph, but either they cannot be reached backwards from

[10] For instance, the most complex algorithm (algorithm 1), is called on each $\mathscr{A}_i$, and not on the whole specification at once: the asymptotic cost of $\sum_i |T_i|^3.lg(|T_i|)$ can be far less than the cost of approximation $|T|^3.lg(|T|)$; on the example of figure 1, the former is 3274, while the latter is 16384. Furthermore, $\sum_{tr_c \in conds(\mathscr{A})} d(tr_c)$ is usually significantly smaller than $\sum_{tr \in T} d(tr)$; on the example of figure 1, the former is 6, while the latter is 19.

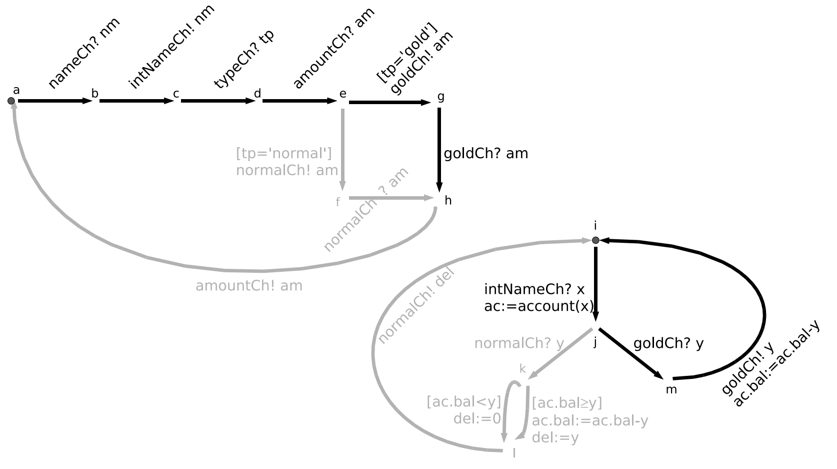

Fig. 7. Slice of the specification in figure 1, *wrt.* $\{T_{24}\}$.

the criterion, or they do not add new transitions once the search described above has been performed. For instance: the control dependence $(j \rightharpoonup k) \xrightarrow{cd} (k \rightharpoonup_1 l)$ is not taken into account, since $j \rightharpoonup k$ is not backward reachable from $m \rightharpoonup i$; although $c \rightharpoonup d$ is in the slice, the data dependence $(c \rightharpoonup d) \xrightarrow{dd} (a \rightharpoonup b)$ is not taken into account, since $a \rightharpoonup b$ is already in the slice after $(a \rightharpoonup b) \xrightarrow{dd} (b \rightharpoonup c)$ has been reached by the algorithm.

In conclusion, the slicing algorithm has identified the parts in the specification, that have no potential influence on the criterion; these parts are consequently not included in the slice. On the example of figure 7, the criterion corresponds to a successful withdrawal with a gold card, and it is interesting to notice that all the parts of the specification that only deal with normal card withdrawals have been sliced out.

5. Related Work

This section is a brief overview of mostly related works, on slicing communicating automata specifications, on precision issues regarding dependence relations, and finally on communicating automata specifications. The reader is kindly referred to previous sections for related works on control dependencies [RAB$^+$05, AK02, Muc97] (in section 4.1), data dependencies [Muc97, FOW87, Nan01, KSV96, TGL06] (in section 4.2) and communication dependencies [Sar97, MT00] (in section 4.3).

5.1. Slicing Communicating Automata Specifications

To the best of our knowledge, the unique previously published work on slicing automata specifications was Bozga *et al.*'s. In [BFG03b], they presented an approach to improve automatic test generation by calculating slices of specifications based on extended automata that communicate using FIFO queues, as opposed to IOSTSs, which communicate using binary rendezvous. The slicing definitions in [BFG03b] are not dependence-based as ours, and thus do not have the convenient properties of dependence-based slicing (cf. sections 2.2 and 4.5). In [BFG03b], slices are calculated with respect to sets of signals (inputs or outputs), while our slicing criteria are sets of transitions in the specification. Finally, Bozga *et al.* give several definitions of a slice, each of which is dedicated to test case generation, and involves external data (*test purposes*, *feeds*). Our definition of a specification slice only involves the specification characteristics, and therefore is intended to be more general and closer to the traditional concepts of the program slicing literature.

5.2. Precision Issues

Regarding dependence precision issues, Krinke [Kri98] pioneered the concept of transitive dependence in a concurrent programming language with `fork/join` mechanisms. In that work, *threaded witnesses* are used to define transitive dependencies, and ensure a good level of precision of the method. In section 4.4, we extended the notion of transitive dependence to IOSTS specifications (definition 14). In such specifications, there are no shared variables, nor procedure calls; thus, the only source of intransitivity may lie in communication actions (cf. the related discussion, in section 4.4.4).

Note that we agree with Millett *et al.* in [MT00], on the point that even imprecise slices may be useful; however, the aforementioned discussion may lead to believe there is a higher potential of imprecise dependencies in their handling of shared variables, than in our handling of communication actions.

5.3. Communicating Automata Specifications

Amongst related works on automata specifications in general, Gaston *et al.* propose in [GGRT06] a method to define test purpose and generate test cases on IOSTS specifications, using symbolic execution techniques. Rapin *et al.* propose in [RGLG03] symbolic execution techniques, for the purpose of exhibiting all the behaviours of IOLTS specifications – the formalism of IOSTSs extends the formalism of IOLTSs, by handling data types (cf. section 3.1).

6. Conclusion

As mentioned in the beginning of this article, slicing has proven to be useful in debugging and program understanding. In current research, slicing techniques are being examined in the context of model reduction for model checking [BW05, DH99, WDQ03], simulation [MT00], test case generation [BFG03b]. The challenge of obtaining the benefits of slicing on formal specifications based on communicating extended automata, namely IOSTSs, is at the origin of the present work.

6.1. Synthesis

In this article, we present definitions and efficient algorithms for the computation of dependence relations in IOSTS specifications – namely, control dependence in section 4.1, data dependence in section 4.2 and communication dependence in section 4.3. Starting from these results, we present in section 4.5 a definition of a slice of a specification with respect to a criterion, together with an algorithm for the automatic extraction of slices in such specifications. In section 4.4, we defined a measure of precision for IOSTS specifications slicing approaches, and indentified the necessary conditions for our approach to be exhaustively precise. We also emphasized a potential lowering of the precision and claimed that it is acceptable, notably for complexity reasons; but also because it may not dominate in practice, assuming that in practical specifications, dual communication actions are usually designed to run in parallel.

Throughout the present article, formal definitions are illustrated on a running example. All these algorithms have been implemented in a slicing prototype tool for IOSTS specifications, that has shown to be effective for specification reduction, debugging and understanding, on examples such as the short one presented in this article (cf. figures 1 and 7).

The model reduction results obtained with our slicing tool give rise to interesting prospects in formal validation methods, e.g. test case generation or model checking; these challenging application areas have become active in very recent years [BFG03b, BW05, DHH+06, WDQ03]. For that purpose, our slicing tool has been successfully connected to the AGATHA tool [BFG+03a, RGLG03]. We are currently working on the next step, of major interest: evaluating the impact of slicing in conjugation with AGATHA, for the purpose of specification validation.

6.2. Ongoing and Future Work

In ongoing work, we think of refining our method, by designing a "fine-grained" IOSTS specifications slicing method, in which transitions are no longer considered as atomic elements, allowing one to "slice out" parts of transitions, and consequently, to compute smaller and more precise slices (this implies a modification of our definition of a specification, but we will show that the dependence definitions would remain correct).

Finally, the two following directions may also be considered for future work, although certainly increasing the overall complexity of the method. First, improving the accuracy of the communication dependence relation, for instance using a *May Happen in Parallel* algorithm [NA98] for IOSTSs: the idea is that if two transitions tr_i and tr_j communicate on the same channel, but never happen in parallel, then we should not observe any communication dependence between tr_i and tr_j. As seen in section 4.4.3, the accuracy of the communication dependence relation could also be improved using parallel composition information. The resulting method should be empirically compared to the one presented in this article, as it is *a priori* unclear whether the precision benefit is worth the induced complexity overhead.

Second, IOSTSs use pairwise disjoint sets of attribute variables in a specification (according to definition 4), i.e. IOSTSs cannot interact using shared variables; this assumption may be removed. As a consequence, an additional kind of inter-automata data dependencies – similar to Krinke's interference dependencies [Kri98] – would have to be handled by the method. It is worth noticing that Müller-Olm and Seidl showed in [MOS01] that slicing in this setting is PSPACE-complete; for the rest, all the methods we are aware of, that deal with such dependencies in a precise way, have worst-case exponential complexity [Kri98, Nan01].

References

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann, 2002.

[BFG$^+$03a] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 591–596, 2003.

[BFG03b] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using static analysis to improve automatic test generation. *Software Tools for Technology Transfer (STTT)*, 4(2):142–152, 2003.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging (AADEBUG)*, pages 206–222, 1993.

[BW05] Ingo Brückner and Heike Wehrheim. Slicing an integrated formal method for verification. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *Lecture Notes in Computer Science*, pages 360–374, 2005.

[CR94] J. Chang and D. J. Richardson. Static and dynamic specification slicing. In *Fourth Irvine Software Symposium*, 1994.

[DH99] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 105–118, 1999.

[DHH$^+$06] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 73–89, 2006.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *IFIP International Conference on Testing of Communicating Systems (TestCom)*, pages 1–18, 2006.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der *Principia Mathematica* und verwandter systeme I. *Monatshefte für mathematik und physik*, 38:173–198, 1931. English translation: *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, translated by Bernard Meltzer. Dover, 1992.

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs. *Electronic Notes in Theoretical Computer Science*, 65(5), 2002.

[HCD$^+$99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Static Analysis Symposium (SAS)*, pages 1–18, 1999.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.

[HW97] Mats Per Erik Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *European Software Engineering Conference (ESEC) / Foundations of Software Engineering (SIGSOFT FSE)*, pages 450–467, 1997.

[Kri98] Jens Krinke. Static slicing of threaded programs. In *Program Analysis For Software Tools and Engineering (PASTE)*, pages 35–42, 1998.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universitat Passau, Germany, april 2003.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):268–299, 1996.

[LG06] Sébastien Labbé and Jean-Pierre Gallois. Towards slicing communicating extended automata, 2006. In the Doctoral Symposium of Formal Methods, available on-line at http://fm06.mcmaster.ca/01SebastienLabbe.pdf.

[LG07] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications for efficient model reduction. In *Australian Software Engineering Conference (ASWEC)*, 2007. To appear (accepted for publication Dec'2006).

[MOS01] Markus Müller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *ACM Symposium on Theory of Computing (STOC)*, pages 647–656, 2001.

[MT00] Lynette I. Millett and Tim Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. *Software Tools for Technology Transfer (STTT)*, 2(4):343–349, 2000.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[NA98] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Software Engineering Notes*, 23(6):24–34, 1998.

[Nan01] Mangala Gowri Nanda. *Slicing Concurrent Java Programs: Issues and Solutions.* PhD thesis, Indian Institute of Technology, 2001.

[NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. *SIGSOFT Software Engineering Notes*, 25(5):180–190, 2000.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Software Development Environments (SDE)*, pages 177–184, 1984.

[Pre30] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich*, pages 92–101, Warszawa, 1930. Annotated English version: Ryan Stansifer. Presburger's article on integer arithmetic: remarks and translation. Technical Report TR84–639, Cornell University Computer Science Department, 1984.

[RAB$^+$05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming (ESOP)*, pages 77–93, 2005.

[RGLG03] Nicolas Rapin, Christophe Gaston, Arnault Lapitre, and Jean-Pierre Gallois. Behavioural unfolding of formal specifications based on communicating extended automata. In *Automated Technology for Verification and Analysis (ATVA)*, 2003.

[Sar97] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 94–113, 1997.

[TGL06] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Compiler Construction (CC)*, pages 17–31, 2006.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[WDQ03] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. *Lecture Notes in Computer Science*, 2495:435–446, 2003.

[Wei81] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.

[XQZ$^+$05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.