Newcastle University

COMPUTING
SCIENCE

Comparing Models of Nondeterministic Expression Evaluation

Ian J. Hayes, Alan Burns, Brijesh Dongol, and Cliff B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1273 September 2011

TECHNICAL REPORT SERIES

No. CS-TR-1273 September, 2011

Comparing Models of Nondeterministic Expression Evaluation

I.J. Hayes, A. Burns, B. Dongol, C.B. Jones

Abstract

Expression evaluation in programming languages is normally deterministic; however, if expres- sions involve variables that are being modified by the environment of the process during their evaluation, the result of the evaluation can be nondeterministic. Two common cases where this occurs are in concur- rent programs where processes share variables and real-time programs that interact to monitor and/or control their environment. In these contexts, while any particular evaluation of an expression gives a single result, there is a range of possible results that could be returned depending on the relative timing of modification of variables by the environment and their access within expression evaluation. Hence to model the semantics of expression evaluation one can use the set of possible values the expression evaluation could return. This paper considers three views of interpreting expressions nondeterministically. The paper formalises the three approaches, highlights different properties satisfied by the approaches, relates the approaches and explores conditions under which they coincide. Furthermore, a link is made to a new notation used in reasoning about interference.

Bibliographical details

HAYES, I.J., BURNS, A., DONGOL, B., JONES, C.B.,

Comparing Models of Nondeterministic Expression Evaluation
[By] I.J. Hayes, A. Burns, B. Dongol, C.B. Jones
Newcastle upon Tyne: Newcastle University: Computing Science, 2011.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1273)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1273

Abstract

Expression evaluation in programming languages is normally deterministic; however, if expres- sions involve variables that are being modified by the environment of the process during their evaluation, the result of the evaluation can be nondeterministic. Two common cases where this occurs are in concur- rent programs where processes share variables and real-time programs that interact to monitor and/or control their environment. In these contexts, while any particular evaluation of an expression gives a single result, there is a range of possible results that could be returned depending on the relative timing of modification of variables by the environment and their access within expression evaluation. Hence to model the semantics of expression evaluation one can use the set of possible values the expression evaluation could return. This paper considers three views of interpreting expressions nondeterministically. The paper formalises the three approaches, highlights different properties satisfied by the approaches, relates the approaches and explores conditions under which they coincide. Furthermore, a link is made to a new notation used in reasoning about interference.

About the authors

Ian Hayes is a Professor in the School of Information Technology and Electrical Engineering at the University of Queensland. His research interests are in the field of formal methods for the specification and development of software, especially for real-time systems. His current research is on the use of time bands and teleo-reactive programming for the development of real-time systems. He is a fellow of the British Computer Society.

Professor Alan Burns a member of the Department of Computer Science, University of York, U.K. His research interests cover a number of aspects of real-time systems including the assessment of languages for use in the real-time domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to real-time applications. Professor Burns has authored/co-authored 450 papers/reports and 15 books. Many of these are in the real-time area. His teaching activities include courses in Operating Systems and Real-time Systems. He is a member of ARTIST - the EU Centre of Excellence in Real-Time and Embedded Systems. In 2009 Professor Burns was elected a Fellow of the Royal Academy of Engineering.

Dr Brijesh Dongol received his PhD from The University of Queensland, Australia in 2009 and is currently working as a post-doctoral researcher with Prof Ian Hayes on combining teleo-reactive programs with time bands to improve dependability of real-time programs. He is mainly interested in developing methods for formally verifying and deriving concurrent programs, lock-free algorithms and real-time systems.

Cliff Jones is a Professor of Computing Science at Newcastle University. He is now applying research on formal methods to wider issues of dependability. Until 2007 his major research involvement was the five university IRC on "Dependability of Computer-Based Systems" of which he was overall Project Director - he is now PI of the follow-on *Platform Grant* "Trustworthy Ambient Systems" (TrAmS) (also EPSRC). He is also PI on an EPSRC-funded project "Splitting (Software) Atoms Safely" and coordinates the "Methodology" strand of the EU-funded RODIN project. As well as his academic career, Cliff has spent over twenty years in industry. His fifteen years in IBM saw among other things the creation -with colleagues in Vienna- of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his doctoral thesis in two years (and enjoyed the family atmosphere of Wolfson College). From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which -among other projects- was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural"(Formal Method) Support Systems theorem proving assistant). Cliff is a *Fellow* of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973 (and was Chair from 1987-96).

Suggested keywords

LOGIC
SEMANTICS
NON-DETERMINISM
TIME BANDS

Comparing Models of Nondeterministic Expression Evaluation

Ian J. Hayes[1], Alan Burns[2], Brijesh Dongol[1], and Cliff B. Jones[3]

[1] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, 4072, Australia.
{Ian.Hayes, Brijesh}@itee.uq.edu.au
[2] Department of Computer Science,
University of York, UK
burns@cs.york.ac.uk
[3] School of Computing Science,
Newcastle University, NE1 7RU, England.
cliff.jones@ncl.ac.uk

Abstract. Expression evaluation in programming languages is normally deterministic; however, if expressions involve variables that are being modified by the environment of the process during their evaluation, the result of the evaluation can be nondeterministic. Two common cases where this occurs are in concurrent programs where processes share variables and real-time programs that interact to monitor and/or control their environment. In these contexts, while any particular evaluation of an expression gives a single result, there is a range of possible results that could be returned depending on the relative timing of modification of variables by the environment and their access within expression evaluation. Hence to model the semantics of expression evaluation one can use the set of possible values the expression evaluation could return. This paper considers three views of interpreting expressions nondeterministically. The paper formalises the three approaches, highlights different properties satisfied by the approaches, relates the approaches and explores conditions under which they coincide. Furthermore, a link is made to a new notation used in reasoning about interference.

1 Introduction

The motivation for this paper comes from two sources: Burns' time band framework [3, 2], and Coleman and Jones' research on fine-grained expression evaluation [4]. The time band framework is intended to allow a system to be viewed in a range of different time bands with different time granularities and precisions. In a time band, events occur within the precision of that band — or perhaps a better term is "imprecision". Due to the timing imprecision, the values of variables may range over a set of possible values and hence evaluation of expressions and predicates is nondeterministic. To handle this imprecision, Burns and Hayes devised a "sampling" logic [3], which allows expression evaluation to be nondeterministic, but assumes that for a single evaluation all occurrences of each variable take on the same (sampled) value (from the set of possible values for that variable).

Coleman and Jones used a fine-grained operational semantics for expression evaluation suitable for reasoning about concurrent processes with shared variables [4]. In their semantics, each occurrence of a variable within an expression may take on a different value (from the set of possible values for that variable) because each occurrence of an identifier within an expression results in a lookup in the state (σ) and σ is subject to change by concurrent threads; furthermore, the order of access is nondeterministic. (This idea has a long history. After McCarthy [20] crystallised the idea of defining programming languages by "abstract interpreters", a key contribution in the early IBM Vienna operational semantics (VDL) approach [19] was to facilitate such nondeterministic expression evaluation. Plotkin's "Structural Operational Semantics" (SOS) was described in his "Aarhus notes" — now conveniently republished as [22]. Here the nondeterminacy was factored out to the meta level by making the choice of which rule to fire nondeterministic. Useful comparative discussions are in [21, 16].)

Both approaches address nondeterministic expression evaluation, but in subtly different ways. It was the realisation that there are different ways to approach nondeterminism in expression evaluation that led to the research reported here, which aims to explore more fully the relationships between the approaches.

In a context in which the environment of a process is modifying variables shared with the process, evaluating expressions involving those variables within the process is nondeterministic: differing relative timings of modifications and accesses can lead to different results. Hence the semantics of expression evaluation is given in terms of the set of all possible results that can be returned. This paper considers three approaches to handling nondeterminism that in general give different sets of possible results.

Sets of states. The first approach considers the set of states that actually occur over the time interval during which the expression is being evaluated. The set of possible values for an expression in this approach is the set of its evaluations, one for each state. For example, if over the expression evaluation time interval u changes from 0 to 1 and then v changes from 0 to 1, the set of states is

$$ssuv = \{\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}\} \; . \tag{1}$$

For *ssuv*, the expression $\mathsf{u} + \mathsf{v}$ has possible values $\{\ 0,\ 1,\ 2\ \}$, and the predicate $\mathsf{u} \le \mathsf{v}$ has possible values $\{\text{false}, \text{true}\}$, but $\mathsf{u} \ge \mathsf{v}$ and $\mathsf{v} = \mathsf{v}$ have only a single possible value, true. Because it assumes that a snapshot of the values of all the variables can be taken instantaneously, this approach does not reflect the imprecision that can occur in an implementation. Most implementations will only be able to access (or sample) the values of variables at slightly different times.

Sets of values. The second approach uses the set of all possible values of each variable over the evaluation interval and all possible evaluations of the expression using those values. For example, the set of states *ssuv* above (1) corresponds to possible values for *u* and *v*, as follows.

$$svuv = \{\mathsf{u} \mapsto \{0,1\}, \mathsf{v} \mapsto \{0,1\}\}. \tag{2}$$

This corresponds to an implementation that samples the value of a variable for each occurrence of the variable within the expression. An implementation evaluating an expression during the same interval as above (corresponding to the set of states *ssuv* above (1)) may sample u before both modifications and get 0 and then sample v after both variables have changed and get 1, and hence evaluate $\mathsf{u} \ge \mathsf{v}$ to false. Note that the state $\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}$ is not one of the actual states in *ssuv*. In evaluating the expression $\mathsf{v} = \mathsf{v}$ in the same context, it may sample the left v first and get 0, and then sample the right v after v has changed from 0 to 1, and get 1, and thus evaluate $\mathsf{v} = \mathsf{v}$ to false.

Sets of apparent states. In the third approach each variable is sampled once during the evaluation interval and this sampled value is used for all occurrences of the variable within the expression. Therefore $\mathsf{v} = \mathsf{v}$ always evaluates to true because a single sampled value of v is used for both occurrences of v. The possible values for v are still all possible values over the interval but all references to v during an evaluation of the expression use just one of those possible values. The set of apparent states for the example above follows.

$$\begin{aligned} apparent(svuv) &= \left\{ \begin{array}{l} \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}, \\ \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\} \end{array} \right\} \\ &= ssuv \cup \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\} \end{aligned} \tag{3}$$

In this approach the possible values of $\mathsf{u} \ge \mathsf{v}$ are $\{\text{false}, \text{true}\}$, but the only possible value of $\mathsf{v} = \mathsf{v}$ is true.

Each of the above forms of expression evaluation is suitable in different contexts. The sets-of-states view corresponds to the states that actually occur. Hence, for example, if one would like to show that some safety property holds for all states of a system, one needs to show it holds in this view. Note that this is an ideal view because in many cases one will not be able to directly observe the actual set of states.

The sets-of-apparent-states view is less deterministic than the sets-of-states view. In it expressions are evaluated in an apparent set of states, where the values of the variables in the sets of apparent states range over their values in the actual sets of states. It corresponds to sampling the state over a time interval—so that the values of the variables may be sampled at slightly different times—but each evaluation of an expression is done with a single sampled value for each variable and hence expressions like $\mathsf{v} = \mathsf{v}$ always evaluate to true. This view corresponds to the approach commonly used in real-time control systems. As is shown below in Section 2.6, the apparent states for a time interval include all the actual states and hence, if a safety property can be shown to hold for all apparent states, it also holds for the actual states. However, it is often harder to show a property holds in the sets of apparent states view because there are typically more apparent states than actual states.

The sets-of-values view is the least deterministic view. In it expressions are evaluated with each occurrence of a variable taken from a set of possible values for that variable. It corresponds to expression evaluation in which a variable's value is read/sampled whenever its value is needed and hence different occurrences of a variable within an expression may have different values. For example, for integer v, $\mathsf{v}+\mathsf{v}$ may evaluate to an odd number, whereas $2*\mathsf{v}$ will always evaluate to an even number. This view corresponds to evaluating an expression in a context in which a concurrent process may be modifying the values of variables that are shared between processes and each access to a variable in an expression reads the shared variable. As is shown below, if a safety property holds for all possible evaluations in a sets-of-values view, then it holds for all corresponding apparent states and hence for all actual states. Again it is harder to show the property holds because the evaluation in the sets-of-values view has potentially more possible values for the expression.

Section 2 formalises all three forms of expression evaluation, the relationships between them, and the conditions under which the different forms of evaluation are equivalent. Section 3 extends the approach to consider modal predicates corresponding to the three evaluation schemes, and Section 4 relates the approach to expression evaluation over timed traces.

2 Expression evaluation

Section 2.1 introduces the syntax of expressions and Section 2.2 gives their standard (deterministic) evaluation semantics in a single state. Section 2.3 gives the first of the three nondeterministic semantics, all of which give the set of possible values of an expression given a set of states. The first evaluates an expression using the standard semantics for each state in a set of states to give a set of values. A less deterministic evaluation strategy is covered in Section 2.4 using a sets-of-values view and Section 2.5 extends this to sets-of-values views of sets of states (the second evaluation strategy). The third approach, that uses evaluation in sets of apparent states, is covered in Section 2.6; as part of this a Galois connection relating sets of (apparent) states with sets-of-values views is developed. The three strategies are compared in Section 2.7, which shows that sets-of-values evaluation is less deterministic than evaluation over a set of states, with evaluation over a set of apparent states falling between these two.

2.1 Syntax of expressions

For the purposes of this paper a simplified syntax for expressions that highlights the issues involved is used. The formal parts of our model are presented using the mathematical notation of Z [1, 23, 5]. For the syntax $\oplus$ is used to represent a binary operator. As unary operators are treated in the same manner by all evaluation strategies, they are omitted here to simplify the presentation.

Definition 1 (Syntax of expressions). *Expressions consist of constants, variables and binary operators,*

$$\mathsf{E} ::= \mathsf{C} \mid \mathsf{Var} \mid \mathsf{E} \oplus \mathsf{E}\ ,$$

where C *and* Var *give the syntax for constants and variables (not given in detail here).*

The following notational conventions are used: c and d are constants; u, v, and w are variables; and e and f are expressions.

Definition 2 (free variables). *The set of variables that occur free within* e *is represented by vars*(e).

$$\begin{array}{|l}
\mathit{vars} : \mathsf{E} \rightarrow \mathsf{Var} \\
\hline
\forall \mathsf{c} : \mathsf{C};\ \mathsf{v} : \mathsf{Var};\ \mathsf{e}, \mathsf{f} : \mathsf{E} \bullet \\
\qquad \mathit{vars}(\mathsf{c}) = \{\} \\
\qquad \mathit{vars}(\mathsf{v}) = \{\mathsf{v}\} \\
\quad \mathit{vars}(\mathsf{e} \oplus \mathsf{f}) = \mathit{vars}(\mathsf{e}) \cup \mathit{vars}(\mathsf{f})
\end{array}$$

Definition 3 (expressions over a set of variables). E_V *is the set of all expressions over a set of variables* V.

$$\mathsf{E}_\mathsf{V} \mathrel{\widehat{=}} \{\mathsf{e} : \mathsf{E} \mid \mathit{vars}(\mathsf{e}) \subseteq \mathsf{V}\}$$

Note that the set of free variables of an expression in E_V is not required to be the whole of V.

2.2 Expression evaluation in a single state

The state of a system consists of the values of the system's variables; for example, for a system with variables u and v having values taken from the set $\{0,1\}$, a state in which u has the value 0 and v has the value 1 is represented by the mapping $\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}$.

Definition 4 (State). *A state space, Σ_{V}, over a set of variable names, $\mathsf{V} \subseteq \mathsf{Var}$, is represented by a mapping from V to values, represented by the set X.*

$$\Sigma_{\mathsf{V}} \mathrel{\widehat{=}} \mathsf{V} \rightarrow X$$

A state, $\sigma \in \Sigma_{\mathsf{V}}$, maps each variable name in its domain, V, to its value in σ. For simplicity the universal set X is used for all values, rather than each variable having values of a particular type. It is assumed that X contains the booleans and integers, as well as any other values required for a particular application.

Definition 5 (Expression evaluation). *The following defines the evaluation of an expression, $\mathsf{e} \in \mathsf{E}_{\mathsf{V}}$, in a state, $\sigma \in \Sigma_{\mathsf{V}}$, both over the variables, V.*

$$\begin{array}{|l}
eval : \mathsf{E}_{\mathsf{V}} \rightarrow (\Sigma_{\mathsf{V}} \rightarrow X) \\
\hline
\forall \sigma : \Sigma_{\mathsf{V}};\ \mathsf{c} : \mathsf{C};\ \mathsf{v} : \mathsf{V};\ \mathsf{e}, \mathsf{f} : \mathsf{E}_{\mathsf{V}} \bullet \\
\qquad eval(\mathsf{c})(\sigma) = \mathsf{c} \\
\qquad eval(\mathsf{v})(\sigma) = \sigma(\mathsf{v}) \\
\quad eval(\mathsf{e} \oplus \mathsf{f})(\sigma) = eval(\mathsf{e})(\sigma) [\![\oplus]\!] eval(\mathsf{f})(\sigma)
\end{array}$$

Note that the occurrence of $\oplus$ to the left of the "$=$" is syntax, but on the right $[\![\oplus]\!]$ is the semantic interpretation of the operator. For non-well-typed expressions it is assumed that there is some undefined value, which is used as the result; below it is assumed that expressions are well-typed.

The notation, $(\mathsf{V} \lhd \sigma)$, for $\mathsf{V} \subseteq \mathsf{Var}$ and state σ, stands for the subset of σ with its domain restricted to the variables in V.

Lemma 1. *For an expression e and states $\sigma_0 \in \Sigma_{\mathsf{V}_0}$ and $\sigma_1 \in \Sigma_{\mathsf{V}_1}$, where $vars(\mathsf{e}) \subseteq \mathsf{V}_0$ and $vars(\mathsf{e}) \subseteq \mathsf{V}_1$, if $(vars(\mathsf{e}) \lhd \sigma_0) = (vars(\mathsf{e}) \lhd \sigma_1)$, then*

$$eval(\mathsf{e})(\sigma_0) = eval(\mathsf{e})(\sigma_1) .$$

2.3 Expression evaluation over a set of states

Over a time interval, the state may evolve and hence during the interval there may be a set of actual states of the system variables, one state for each time. One may take time to be real numbers in order to handle continuously evolving environmental variables and hence the set of states for a time interval is potentially uncountably infinite. If the values of the variables are changing over time then there will be multiple possible evaluations of an expression involving those variables. This section considers an idealised scheme for handling such nondeterminism, in which the set of values of an expression is formed by evaluating the expression in each of the states.

Definition 6 (sets of states evaluation). *Given an expression, $\mathsf{e} \in \mathsf{E}_{\mathsf{V}}$, and a set of states, $ss \in \mathbb{P}\,\Sigma_{\mathsf{V}}$, the function eval_ss returns the set of values of the expression in every state.*

$$\begin{array}{|l}
eval\_ss : \mathsf{E}_{\mathsf{V}} \rightarrow (\mathbb{P}\,\Sigma_{\mathsf{V}} \rightarrow \mathbb{P}\,X) \\
\hline
eval\_ss(\mathsf{e})(ss) = \{\sigma : ss \bullet eval(\mathsf{e})(\sigma)\}
\end{array}$$

The notation $\{\sigma : ss \bullet f\}$ stands for the set of all values of f for σ ranging over states in ss.[4]

[4] This is more commonly written $\{f \mid \sigma \in ss\}$ but it is preferable not to use this notation because it is unclear whether σ is bound within the set comprehension or a free variable being tested for membership of ss.

Lemma 2 (eval-ss monotonicity). *For an expression,* $\mathsf{e} \in \mathsf{E}_\mathsf{V}$*, and sets of states,* ss_0 *and* ss_1*, both over* V*, if* $ss_0 \subseteq ss_1$*,*

$$eval\_ss(\mathsf{e})(ss_0) \subseteq eval\_ss(\mathsf{e})(ss_1)\ .$$

For a set of states, *ss*, the notation, $\mathsf{V} \overline{\lhd}\, ss$, stands for $\{\sigma : ss \bullet \mathsf{V} \lhd \sigma\}$.

Lemma 3. *For an expression* e *and sets of states* $ss_0 \in \Sigma_{\mathsf{V}_0}$ *and* $ss_1 \in \Sigma_{\mathsf{V}_1}$*, where* $vars(\mathsf{e}) \subseteq \mathsf{V}_0$ *and* $vars(\mathsf{e}) \subseteq \mathsf{V}_1$*, if* $vars(\mathsf{e}) \overline{\lhd}\, ss_0 = vars(\mathsf{e}) \overline{\lhd}\, ss_1$*, then*

$$eval\_ss(\mathsf{e})(ss_0) = eval\_ss(\mathsf{e})(ss_1)\ .$$

Evaluating a binary expression over a set of states gives a subset of the results of evaluating each operand over the same set of states and combining the results according to the binary operator. If "$\oplus$" is a binary operator, the operator "$\overline{\oplus}$" is "$\oplus$" lifted to arguments that are sets, so that for sets of values, *sx* and *sy*,

$$sx\ \overline{\oplus}\ sy \mathrel{\widehat{=}} \{x : sx;\ y : sy \bullet x \oplus y\}\ .$$

Theorem 1 (eval-ss subdistribution). *For any expressions* e *and* f *over* V*, and set of states,* $ss \in \mathbb{P}\,\Sigma_\mathsf{V}$*,*

$$eval\_ss(\mathsf{e} \oplus \mathsf{f})(ss) \subseteq eval\_ss(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_ss(\mathsf{f})(ss)\ .$$

Proof.

$$\begin{array}{l} \quad eval\_ss(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_ss(\mathsf{f})(ss) \\ = \{x : eval\_ss(\mathsf{e})(ss);\ y : eval\_ss(\mathsf{f})(ss) \bullet x[\![\oplus]\!]y\} \\ = \{\sigma_0 : ss;\ \sigma_1 : ss \bullet eval(\mathsf{e})(\sigma_0)[\![\oplus]\!]eval(\mathsf{f})(\sigma_1)\} \\ \supseteq \text{constraining so that } \sigma_0 = \sigma_1 \\ \quad \{\sigma : ss \bullet eval(\mathsf{e})(\sigma)[\![\oplus]\!]eval(\mathsf{f})(\sigma)\} \\ = \{\sigma : ss \bullet eval(\mathsf{e} \oplus \mathsf{f})(\sigma)\} \\ = eval\_ss(\mathsf{e} \oplus \mathsf{f})(ss) \end{array}$$

□

As an example of why Theorem 1 specifies subset rather than equality, consider a set of states

$$ssxy = \{\{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 0\}, \{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 1\}\},$$

then $eval\_ss(\mathsf{x} + \mathsf{y})(ssxy) = \{0, 2\}$ but $eval\_ss(\mathsf{x})(ssxy)\ \overline{[\![+]\!]}\ eval\_ss(\mathsf{y})(ssxy) = \{0, 1, 2\}$.

2.4 Expression evaluation over a sets-of-values view

Unfortunately, it is not always possible to observe accurately a set of states, especially when observing multiple variables all of which are evolving over time, because a snapshot of an entire state typically cannot be taken. An implementation sampling over an interval in which u changes from 0 to 1 and then v changes from 0 to 1 (corresponding to the set of states *ssuv* above (1)) may sample u before both modifications and get 0 and then sample v after both variables have changed and get 1, but the state $\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}$ is not one of the actual states in *ssuv*. To accommodate this sampling anomaly, an abstraction of the set of states over an interval, which we call the *sets-of-values view*, is used. The sets-of-values view, *svuv*, corresponding to the set of states *ssuv* in (1) above only indicates that both u and v take on the values 0 and 1 during the interval:

$$svuv = \{\mathsf{u} \mapsto \{0, 1\}, \mathsf{v} \mapsto \{0, 1\}\}.$$

The sets-of-values view of a set of states has less information than the set of states, in the sense that a single sets-of-values view may correspond to multiple sets of states.

Definition 7 (sets-of-values view). *A sets-of-values view over a set of variable names,* V*, is represented by a mapping from* V *to sets of values.*

$$VState_\mathsf{V} \mathrel{\widehat{=}} \mathsf{V} \rightarrow \mathbb{P}\,X$$

Expression evaluation in the sets-of-values view is less deterministic than expression evaluation over a set of states because a single variable may take on multiple values within a single evaluation. For example, if a variable, v, has multiple values, an expression like $\mathsf{v} = \mathsf{v}$ may evaluate to either true or false.

Definition 8 (sets-of-values evaluation). *For an expression,* e*, and a sets-of-values view, sv, both over* V*, the function eval_sv returns the set of all values of the expression with respect to sv.*

$$
\begin{array}{|l}
eval\_sv : \mathsf{E_V} \rightarrow (VState_{\mathsf{V}} \rightarrow \mathbb{P}\, X) \\
\hline
\forall\, sv : VState_{\mathsf{V}};\ \mathsf{c} : \mathsf{C};\ \mathsf{v} : \mathsf{V};\ \mathsf{e}, \mathsf{f} : \mathsf{E_V} \bullet \\
\quad eval\_sv(\mathsf{c})(sv) = \{\mathsf{c}\} \\
\quad eval\_sv(\mathsf{v})(sv) = sv(\mathsf{v}) \\
eval\_sv(\mathsf{e} \oplus \mathsf{f})(sv) = eval\_sv(\mathsf{e})(sv)\ \overline{[\![\oplus]\!]}\ eval\_sv(\mathsf{f})(sv)
\end{array}
$$

Again note that $\oplus$ to the left of the "=" is just syntax, but on the right $\overline{[\![\oplus]\!]}$ is the lifted version of the semantic interpretation of the operator. As with *eval*, for non-well-typed expressions it is assumed that there is some undefined value, which is used as the result; below its is assumed that expressions are well-typed.

This form of expression evaluation corresponds to that given by Coleman and Jones [4], although they use an operational semantics to define expression evaluation.

A sets-of-values view, sv_0, is subsumed by another, sv_1, written $sv_0 \Subset sv_1$, iff for every variable, the set of possible values in sv_0 is contained in that of sv_1.

Definition 9 (subsumption). *For* sv_0 *and* sv_1 *both in* $VState_{\mathsf{V}}$*,*

$$sv_0 \Subset sv_1 \mathrel{\widehat{=}} (\forall\, v : \mathsf{V} \bullet sv_0(v) \subseteq sv_1(v))\ .$$

Lemma 4 (eval-sv monotonicity). *For two sets-of-values views,* sv_0 *and* sv_1*, if* $sv_0 \Subset sv_1$*,*

$$eval\_sv(\mathsf{e})(sv_0) \subseteq eval\_sv(\mathsf{e})(sv_1)\ .$$

Lemma 5. *For an expression* $\mathsf{e} \in \mathsf{E_V}$ *and sets-of-values views* $sv_0 \in VState_{\mathsf{V_0}}$ *and* $sv_1 \in VState_{\mathsf{V_1}}$*, where* $\mathsf{V} \subseteq \mathsf{V_0}$ *and* $\mathsf{V} \in \mathsf{V_1}$*, if* $(vars(\mathsf{e}) \lhd sv_0) = (vars(\mathsf{e}) \lhd sv_1)$*, then*

$$eval\_sv(\mathsf{e})(sv_0) = eval\_sv(\mathsf{e})(sv_1)\ .$$

2.5 Relating a set of states to a sets-of-values view

The set of states *ssuv* given above in (1) corresponds to the sets-of-values view *svuv* given in (2). The function *values* represents this relationship, so that $values(ssuv) = svuv$.

Definition 10 (values).

$$
\begin{array}{|l}
values : \mathbb{P}\, \Sigma_{\mathsf{V}} \rightarrow VState_{\mathsf{V}} \\
\hline
\forall\, ss : \mathbb{P}\, \Sigma_{\mathsf{V}} \bullet values(ss) = (\lambda\, v : \mathsf{V} \bullet \{\sigma : ss \bullet \sigma(v)\})
\end{array}
$$

Lemma 6 (values monotonic). *If* $ss_0 \subseteq ss_1$*, then* $values(ss_0) \Subset values(ss_1)$*.*

If a set of states, *ss*, is nonempty then there is at least one value for each variable. The notation $\mathbb{P}_1\, X$ stands for the set of all non-empty subsets of X.

Lemma 7. *For any nonempty set of states,* $ss \in \mathbb{P}_1\, \Sigma_{\mathsf{V}}$*, then* $values(ss) \in \mathsf{V} \rightarrow \mathbb{P}_1\, X$*.*

Definition 11 (deterministic). *A variable* v *is deterministic in a sets-of-values view sv iff* $sv(\mathsf{v})$ *is a singleton set, and a set of variables* W *is deterministic in sv iff all variables in* W *are deterministic in sv.*

If a sets-of-values view only has one value for each variable, it corresponds to a single (standard) state. The function *det_values* extracts a sets-of-values view from a single state. Each variable in the sets-of-values view is mapped to a singleton set containing the value of the variable in the state.

Definition 12 (deterministic values).

$$\begin{array}{|l} \mathit{det\_values} : \Sigma_{\mathsf{V}} \rightarrow \mathit{VState}_{\mathsf{V}} \\ \hline \mathit{det\_values}(\sigma) = \mathit{values}(\{\sigma\}) \end{array}$$

Note that $\mathit{det\_values}(\sigma) = (\lambda\, v : \mathsf{V} \bullet \{\sigma(v)\})$.

For a state, σ, evaluating an expression deterministically (using *eval*) corresponds to using the sets-of-values evaluation for the corresponding deterministic sets-of-values view.

Theorem 2 (deterministic evaluation). *Given a state $\sigma \in \Sigma_{\mathsf{V}}$,*

$$\mathit{eval\_sv}(\mathsf{e})(\mathit{det\_values}(\sigma)) = \{\mathit{eval}(\mathsf{e})(\sigma)\} \;.$$

Proof. The proof is via structural induction over expressions. For a constant c, the argument is as follows.

$$\mathit{eval\_sv}(\mathsf{c})(\mathit{det\_values}(\sigma)) = \{\mathsf{c}\} = \{\mathit{eval}(\mathsf{c})(\sigma)\}$$

The case for variables uses the state.

$$\begin{aligned} & \mathit{eval\_sv}(\mathsf{v})(\mathit{det\_values}(\sigma)) \\ = {} & \text{by Definition 8 (eval-sv)} \\ & \mathit{det\_values}(\sigma)(\mathsf{v}) \\ = {} & \text{by Definition 12 (deterministic values) as } \mathit{det\_values}(\sigma) = (\lambda\, \mathsf{v} : \mathsf{V} \bullet \{\sigma(\mathsf{v})\}) \\ & \{\sigma(\mathsf{v})\} \\ = {} & \text{by Definition 5 (eval)} \\ & \{\mathit{eval}(\mathsf{v})(\sigma)\} \end{aligned}$$

The case for a binary expression follows.

$$\begin{aligned} & \mathit{eval\_sv}(\mathsf{e} \oplus \mathsf{f})(\mathit{det\_value}(\sigma)) \\ = {} & \mathit{eval\_sv}(\mathsf{e})(\mathit{det\_values}(\sigma))\ \overline{[\![\oplus]\!]}\ \mathit{eval\_sv}(\mathsf{f})(\mathit{det\_values}(\sigma)) \\ = {} & \text{by the induction hypothesis (twice)} \\ & \{\mathit{eval}(\mathsf{e})(\sigma)\}\ \overline{[\![\oplus]\!]}\ \{\mathit{eval}(\mathsf{f})(\sigma)\} \\ = {} & \{\mathit{eval}(\mathsf{e})(\sigma) [\![\oplus]\!] \mathit{eval}(\mathsf{f})(\sigma)\} \\ = {} & \{\mathit{eval}(\mathsf{e} \oplus \mathsf{f})(\sigma)\} \end{aligned}$$

□

A sets-of-values view $\mathit{values}(ss)$ can be extracted from a set of states, ss, and hence expression evaluation for a set of states can be defined in terms of the sets-of-values view of the set of states.

Definition 13 (sets-of-values view evaluation from a set of states). *Expression evaluation over the sets-of-values view of a set of states is defined as follows.*

$$\begin{array}{|l} \mathit{eval\_vss} : \mathsf{E}_{\mathsf{V}} \rightarrow (\mathbb{P}\, \Sigma_{\mathsf{V}} \rightarrow \mathbb{P}\, X) \\ \hline \mathit{eval\_vss}(\mathsf{e})(ss) = \mathit{eval\_sv}(\mathsf{e})(\mathit{values}(ss)) \end{array}$$

Lemma 8 (eval-vss monotonic). *If $ss_0 \subseteq ss_1$, then $\mathit{eval\_vss}(\mathsf{e})(ss_0) \subseteq \mathit{eval\_vss}(\mathsf{e})(ss_1)$.*

Lemma 9. *For an expression, e, and sets of states, $ss_0 \in \mathbb{P}\, \Sigma_{\mathsf{V}_0}$ and $ss_1 \in \mathbb{P}\, \Sigma_{\mathsf{V}_1}$, where $\mathit{vars}(\mathsf{e}) \subseteq \mathsf{V}_0$ and $\mathit{vars}(\mathsf{e}) \subseteq \mathsf{V}_1$, if $\mathit{vars}(\mathsf{e}) \overline{\lhd}\, ss_0 = \mathit{vars}(\mathsf{e}) \overline{\lhd}\, ss_1$, then*

$$\mathit{eval\_vss}(\mathsf{e})(ss_0) = \mathit{eval\_vss}(\mathsf{e})(ss_1) \;.$$

Unlike sets-of-states evaluation, evaluation of a binary expression over a set-of-values view, distributes.

Theorem 3 (eval-vss distribution). *For all expressions e and f, and sets of states, ss, all over V,*

$$\mathit{eval\_vss}(\mathsf{e} \oplus \mathsf{f})(ss) = \mathit{eval\_vss}(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ \mathit{eval\_vss}(\mathsf{f})(ss) \;.$$

Proof.

$$\begin{aligned}
& \mathit{eval\_vss}(\mathsf{e} \oplus \mathsf{f})(ss) \\
= {} & \mathit{eval\_sv}(\mathsf{e} \oplus \mathsf{f})(\mathit{values}(ss)) \\
= {} & \mathit{eval\_sv}(\mathsf{e})(\mathit{values}(ss))\ \overline{[\![\oplus]\!]}\ \mathit{eval\_sv}(\mathsf{f})(\mathit{values}(ss)) \\
= {} & \mathit{eval\_vss}(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ \mathit{eval\_vss}(\mathsf{f})(ss)
\end{aligned}$$

□

2.6 Expression evaluation over apparent states

If one considers a mapping in the opposite direction to *values*, i.e., from a sets-of-values view to a set of states, the set of states that may be *apparent* in a sets-of-values view can be extracted by considering all possible states such that each variable maps to an element of its set of possible values. For the sets of values *svuv* given in (2), the corresponding set of apparent states is

$$\mathit{apparent}(\mathit{svuv}) = \left\{ \begin{array}{l} \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}, \\ \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\} \end{array} \right\} .$$

The function, *apparent*, determines the set of apparent states for any sets-of-values view.

Definition 14 (apparent).

$$\begin{array}{|l}
\mathit{apparent} : \mathit{VState}_{\mathsf{V}} \to \mathbb{P}\, \Sigma_{\mathsf{V}} \\
\hline
\forall\, sv : \mathit{VState}_{\mathsf{V}} \bullet \mathit{apparent}(sv) = \{\sigma : \Sigma_{\mathsf{V}} \mid (\forall\, v : \mathsf{V} \bullet \sigma(v) \in sv(v))\}
\end{array}$$

Lemma 10 (apparent monotonic). *If $sv_0 \sqsubseteq sv_1$, then $\mathit{apparent}(sv_0) \subseteq \mathit{apparent}(sv_1)$.*

The following examples illustrate the relationship between *apparent* and *values*.

$$\begin{aligned}
\mathit{ssuv} &= \{\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}\} \\
\mathit{svuv} &= \mathit{values}(\mathit{ssuv}) \\
&= \{\mathsf{u} \mapsto \{0,1\}, \mathsf{v} \mapsto \{0,1\}\} \\
\mathit{apparent}(\mathit{svuv}) &= \left\{ \begin{array}{l} \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}, \\ \{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 1\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\} \end{array} \right\} \\
&\supset \mathit{ssuv} \\
\mathit{values}(\mathit{apparent}(\mathit{svuv})) &= \{\mathsf{u} \mapsto \{0,1\}, \mathsf{v} \mapsto \{0,1\}\} \\
&= \mathit{svuv}
\end{aligned}$$

A set of states has potentially finer information than the corresponding sets-of-values view, i.e., the function *values* may map different sets of states, ss_0 and ss_1, to the same set-of-values view, *sv*. Hence *values* does not have a unique inverse, however, the function *apparent* is a pseudo-inverse of *values* in the sense that for all sets-of-values views, *sv*,

$$\mathit{apparent}(\mathit{values}(\mathit{apparent}(sv))) = \mathit{apparent}(sv)$$

or equivalently using function composition,

$$\mathit{apparent} \circ \mathit{values} \circ \mathit{apparent} = \mathit{apparent} . \tag{4}$$

Similarly,

$$\mathit{values} \circ \mathit{apparent} \circ \mathit{values} = \mathit{values} . \tag{5}$$

These properties suggest that the pair of functions $(\mathit{values}, \mathit{apparent})$ forms a Galois connection.

Theorem 4 (Galois connection). *The pair of functions $(\mathit{values}, \mathit{apparent})$ forms a Galois connection, between $\mathbb{P}\, \Sigma_{\mathsf{V}}$ with subset ordering and the space $\mathit{VState}_{\mathsf{V}}$ with the subsumption ordering. That is, for any set of states $ss \in \mathbb{P}\, \Sigma_{\mathsf{V}}$ and set-of-values view $sv \in \mathit{VState}_{\mathsf{V}}$,*

$$\mathit{values}(ss) \sqsubseteq sv \quad \Leftrightarrow \quad ss \subseteq \mathit{apparent}(sv) .$$

Proof.

$$
\begin{aligned}
&\quad ss \subseteq apparent(sv)\\
&\Leftrightarrow ss \subseteq \{\sigma : \Sigma_{\mathsf{V}} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v))\}\\
&\Leftrightarrow \forall \sigma : \Sigma_{\mathsf{V}} \bullet \sigma \in ss \Rightarrow (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v))\\
&\Leftrightarrow \forall v : \mathsf{V} \bullet \forall \sigma : \Sigma_{\mathsf{V}} \bullet \sigma \in ss \Rightarrow \sigma(v) \in sv(v)\\
&\Leftrightarrow \forall v : \mathsf{V} \bullet \forall x : X \bullet x \in \{\sigma : ss \bullet \sigma(v)\} \Rightarrow x \in sv(v)\\
&\Leftrightarrow \forall v : \mathsf{V} \bullet \{\sigma : ss \bullet \sigma(v)\} \subseteq sv(v)\\
&\Leftrightarrow \forall v : \mathsf{V} \bullet values(ss)(v) \subseteq sv(v)\\
&\Leftrightarrow values(ss) \mathrel{\dot\subseteq} sv
\end{aligned}
$$

□

By the theory of Galois connections, both *values* and *apparent* are monotonic and that they satisfy properties (4) and (5). Furthermore, the definition of the function *values* uniquely determines the function *apparent* in order to satisfy the Galois connection property and vice versa. Because $values(ss) \mathrel{\dot\subseteq} values(ss)$ and $apparent(sv) \subseteq apparent(sv)$, the following corollary holds.

Corollary 1. *For any set of states $ss \in \mathbb{P}\,\Sigma_{\mathsf{V}}$ and sets-of-values view $sv \in VState_{\mathsf{V}}$,*

$$ss \subseteq apparent(values(ss)) \tag{6}$$

$$values(apparent(sv)) \mathrel{\dot\subseteq} sv\ . \tag{7}$$

Provided every variable in a sets-of-values view has at least one value, the function *apparent* is one-to-one (i.e., distinct sv_0 and sv_1 are mapped to distinct sets of states) and the pair of functions satisfy the property that $values \circ apparent$ is the identity function. This property strengthens property (5).

Theorem 5. *If every variable in a sets-of-values view, sv, has at least one value, i.e., $sv \in \mathsf{V} \rightarrow \mathbb{P}_1\, X$,*

$$sv = values(apparent(sv))\ .$$

In the proof the notation $\{x : T \mid p \bullet e\}$ stands for the set of all the values of the expression e for x ranging over the set T such that p holds.

Proof. For all sets-of-values views, $sv \in \mathsf{V} \rightarrow \mathbb{P}_1\, X$ the following holds.

$$
\begin{aligned}
&\quad values(apparent(sv))\\
&= values(\{\sigma : \Sigma_{\mathsf{V}} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v))\})\\
&= (\lambda\, v : \mathsf{V} \bullet \{\sigma : \{\sigma : \Sigma_{\mathsf{V}} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v))\} \bullet \sigma(v))\})\\
&= (\lambda\, v : \mathsf{V} \bullet \{\sigma : \Sigma_{\mathsf{V}} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v)) \bullet \sigma(v))\})\\
&= \text{as } sv(v) \text{ is nonempty for all } \mathsf{v} \in V\\
&\quad (\lambda\, v : \mathsf{V} \bullet \{\sigma : \Sigma_{\mathsf{V}} \mid \sigma(v) \in sv(v) \bullet \sigma(v)\})\\
&= (\lambda\, v : \mathsf{V} \bullet sv(v))\\
&= sv
\end{aligned}
$$

□

Definition 15 (deterministic variable). *A variable, v, is deterministic in a set of states, $ss \in \mathbb{P}\,\Sigma_{\mathsf{V}}$, where $\mathsf{v} \in \mathsf{V}$, iff it has the same value in all those states, i.e.,*

$$\forall \sigma : ss \bullet values(ss)(\mathsf{v}) = \{\sigma(\mathsf{v})\}\ .$$

Theorem 6 (single nondeterministic variable). *If in a set of states, $ss \in \mathbb{P}\,\Sigma_{\mathsf{V}}$, all variables, with the possible exception of a single variable, are deterministic,*

$$ss = apparent(values(ss))\ .$$

Proof. Note that for a variable, v, that is deterministic within ss,

$$\forall \sigma' : ss \bullet values(ss)(v) = \{\sigma'(v)\}. \tag{8}$$

Assume the single possibly nondeterministic variable is w. For any σ, the following holds.

$\sigma \in apparent(values(ss))$
$\Leftrightarrow (\forall v : \mathsf{V} \bullet \sigma(v) \in values(ss)(v))$
$\Leftrightarrow (\forall v : \mathsf{V} \bullet v \neq w \Rightarrow \sigma(v) \in values(ss)(v)) \wedge (\sigma(w) \in values(ss)(w))$
$\Leftrightarrow (\forall v : \mathsf{V} \bullet v \neq w \Rightarrow \sigma(v) \in values(ss)(v)) \wedge (\exists \sigma' : ss \bullet \sigma(w) = \sigma'(w))$
$\Leftrightarrow$ no free occurrences of σ' in the universally quantified formula
$\exists \sigma' : ss \bullet (\forall v : \mathsf{V} \bullet v \neq w \Rightarrow \sigma(v) \in values(ss)(v)) \wedge \sigma(w) = \sigma'(w)$
$\Leftrightarrow$ as all v other than w are deterministic in ss, $values(ss)(v)$ is a singleton set
$\exists \sigma' : ss \bullet (\forall v : \mathsf{V} \bullet v \neq w \Rightarrow \{\sigma(v)\} = values(ss)(v)) \wedge \sigma(w) = \sigma'(w)$
$\Leftrightarrow$ as all v other than w are deterministic $values(ss)(v) = \{\sigma'(v)\}$ for any $\sigma' \in ss$ by (8)
$\exists \sigma' : ss \bullet (\forall v : \mathsf{V} \bullet v \neq w \Rightarrow \{\sigma(v)\} = \{\sigma'(v)\}) \wedge \sigma(w) = \sigma'(w)$
$\Leftrightarrow \exists \sigma' : ss \bullet (\forall v : \mathsf{V} \bullet \sigma(v) = \sigma'(v))$
$\Leftrightarrow \exists \sigma' : ss \bullet \sigma = \sigma'$
$\Leftrightarrow \sigma \in ss$

□

Note that if there are two nondeterministic variables within a set of states, ss, this theorem does not hold because, for example, if $sswv = \{\{\mathsf{w} \mapsto 0, \mathsf{v} \mapsto 1\}, \{\mathsf{w} \mapsto 1, \mathsf{v} \mapsto 0\}\}$, in which both w and v are nondeterministic, $apparent(values(sswv))$ also contains the states $\{\mathsf{w} \mapsto 0, \mathsf{v} \mapsto 0\}$ and $\{\mathsf{w} \mapsto 1, \mathsf{v} \mapsto 1\}$, as well as those in $sswv$.

If a state is formed by combining parts of two apparent states, the result is an apparent state, provided any variables in common are deterministic.

Theorem 7 (partitioned apparent state). *For any set-of-values view, sv, over variables* V*, and sets of variables,* $\mathsf{W_0}$ *and* $\mathsf{W_1}$*, such that* $\mathsf{W_0} \cup \mathsf{W_1} = \mathsf{V}$*, then provided sv is deterministic over* $\mathsf{W_0} \cap \mathsf{W_1}$*,*

$$\{\sigma_0, \sigma_1 : apparent(sv) \bullet (\mathsf{W_0} \lhd \sigma_0) \cup (\mathsf{W_1} \lhd \sigma_1)\} = apparent(sv) \ .$$

Proof. Because sv is deterministic over $\mathsf{W_0} \cap \mathsf{W_1}$, for any states $\sigma_0, \sigma_1 \in apparent(sv)$, $(\mathsf{W_0} \lhd \sigma_0) \cup (\mathsf{W_1} \lhd \sigma_1)$ is also a state over V.

$\{\sigma_0, \sigma_1 : apparent(sv) \bullet (\mathsf{W_0} \lhd \sigma_0) \cup (\mathsf{W_1} \lhd \sigma_1)\}$
$= \{\sigma_0, \sigma_1 : \Sigma_\mathsf{V} \mid (\forall v : \mathsf{V} \bullet \sigma_0(v) \in sv(v)) \wedge (\forall v : \mathsf{V} \bullet \sigma_1(v) \in sv(v)) \bullet (\mathsf{W_0} \lhd \sigma_0) \cup (\mathsf{W_1} \lhd \sigma_1)\}$
$=$ Note this step is valid even if $sv(v) = \{\}$, for some $v \in \mathsf{V}$
$\{\sigma_0 : \Sigma_{\mathsf{W_0}};\ \sigma_1 : \Sigma_{\mathsf{W_1}} \mid (\forall v : \mathsf{W_0} \bullet \sigma_0(v) \in sv(v)) \wedge (\forall v : \mathsf{W_1} \bullet \sigma_1(v) \in sv(v)) \bullet \sigma_0 \cup \sigma_1\}$
$=$ as for all $v \in \mathsf{W_0} \cap \mathsf{W_1}$, $\sigma_0(v) = \sigma_1(v)$
$\{\sigma : \Sigma_\mathsf{V} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in sv(v))\}$
$= apparent(sv)$

□

If one considers a set of states, $ss \in \mathbb{P}\,\Sigma_\mathsf{V}$, the corresponding sets-of-values view, $sv = values(ss)$, and the set of apparent states corresponding to that, $as = apparent(sv)$, then (nondeterministic) expression evaluation over as is less deterministic than over ss, but more deterministic than over sv. The apparent or sampled expression evaluation function, *eval_samp*, is defined over a set of states, ss, such that it is equivalent to evaluation of the expression over the apparent states extracted from the sets-of-values view of ss.

Definition 16 (evaluation over sets of apparent states).

$$eval\_samp : \mathsf{E_V} \to (\mathbb{P}\,\Sigma_\mathsf{V} \to \mathbb{P}\,X)$$
$$eval\_samp(\mathsf{e})(ss) = eval\_ss(\mathsf{e})(apparent(values(ss)))$$

Lemma 11 (eval-samp monotonic). *If* $ss_0 \subseteq ss_1$*, then* $eval\_samp(\mathsf{e})(ss_0) \subseteq eval\_samp(\mathsf{e})(ss_1)$.

Lemma 12. *For an expression,* e*, and sets of states,* $ss_0 \in \mathbb{P}\,\Sigma_{\mathsf{V}_0}$ *and* $ss_1 \in \mathbb{P}\,\Sigma_{\mathsf{V}_1}$*, where* $vars(\mathsf{e}) \subseteq \mathsf{V}_0$ *and* $vars(\mathsf{e}) \subseteq \mathsf{V}_1$*, if* $vars(\mathsf{e}) \mathrel{\overline{\lhd}} ss_0 = vars(\mathsf{e}) \mathrel{\overline{\lhd}} ss_1$*, then*

$$eval\_samp(\mathsf{e})(ss_0) = eval\_samp(\mathsf{e})(ss_1)\ .$$

Theorem 8 (eval-samp subdistribution). *For expressions,* e *and* f*, and a set of states, ss, all over* V*,*

$$eval\_samp(\mathsf{e} \oplus \mathsf{f})(ss) \subseteq eval\_samp(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_samp(\mathsf{f})(ss)\ .$$

Proof.

$$\begin{array}{l}
\quad eval\_samp(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_samp(\mathsf{f})(ss) \\
= eval\_ss(\mathsf{e})(apparent(values(ss)))\ \overline{[\![\oplus]\!]}\ eval\_ss(\mathsf{f})(apparent(values(ss))) \\
\supseteq \text{by Theorem 1} \\
\quad eval\_ss(\mathsf{e} \oplus \mathsf{f})(apparent(values(ss))) \\
= eval\_samp(\mathsf{e} \oplus \mathsf{f})(ss)
\end{array}$$

□

For example, for the set of states, $ssv = \{\{\mathsf{v} \mapsto 0\}, \{\mathsf{v} \mapsto 1\}\}$, $eval\_samp(\mathsf{v} \leqslant \mathsf{v})(ssv) = \{true\}$, whereas $eval\_samp(\mathsf{v})(ssv)\ \overline{[\![\leqslant]\!]}\ eval\_samp(\mathsf{v})(ssv) = \{true, false\}$.

Theorem 9 (eval-samp partition). *Provided each variable in* $vars(\mathsf{e}) \cap vars(\mathsf{f})$ *is deterministic within a set of states, ss, over* V*, where* $vars(\mathsf{e} \oplus \mathsf{f}) \subseteq \mathsf{V}$*,*

$$eval\_samp(\mathsf{e} \oplus \mathsf{f})(ss) = eval\_samp(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_samp(\mathsf{f})(ss)\ .$$

Proof. Let $\mathsf{W_e} = vars(\mathsf{e})$ and $\mathsf{W_f} = vars(\mathsf{f})$.

$$\begin{array}{l}
\quad eval\_samp(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_samp(\mathsf{f})(ss) \\
= \{x : eval\_samp(\mathsf{e})(ss);\ y : eval\_samp(\mathsf{f})(ss) \bullet x[\![\oplus]\!]y\} \\
= \{\sigma_0, \sigma_1 : apparent(values(ss)) \bullet eval(\mathsf{e})(\sigma_0)[\![\oplus]\!]eval(\mathsf{f})(\sigma_1)\} \\
= \text{letting } sv = (\mathsf{W_e} \cup \mathsf{W_f}) \lhd values(ss)\text{; Lemma 1} \\
\quad \{\sigma_0, \sigma_1 : apparent(sv) \bullet eval(\mathsf{e})(\sigma_0)[\![\oplus]\!]eval(\mathsf{f})(\sigma_1)\} \\
= \text{by assumption } (\mathsf{W_e} \cap \mathsf{W_f}) \lhd \sigma_0 = (\mathsf{W_e} \cap \mathsf{W_f}) \lhd \sigma_1\text{; Lemma 1} \\
\quad \{\sigma_0, \sigma_1 : apparent(sv) \bullet eval(\mathsf{e})((\mathsf{W_e} \lhd \sigma_0) \cup (\mathsf{W_f} \lhd \sigma_1))[\![\oplus]\!]eval(\mathsf{f})((\mathsf{W_e} \lhd \sigma_0) \cup (\mathsf{W_f} \lhd \sigma_1))\} \\
= \{\sigma_0, \sigma_1 : apparent(sv) \bullet eval(\mathsf{e} \oplus \mathsf{f})((\mathsf{W_e} \lhd \sigma_0) \cup (\mathsf{W_f} \lhd \sigma_1))\} \\
= \{\sigma : \{\sigma_0, \sigma_1 : apparent(sv) \bullet (\mathsf{W_e} \lhd \sigma_0) \cup (\mathsf{W_f} \lhd \sigma_1)\} \bullet eval(\mathsf{e} \oplus \mathsf{f})(\sigma)\} \\
= \text{by Theorem 7 as } sv \text{ is deterministic over } \mathsf{W_e} \cap \mathsf{W_f} \\
\quad \{\sigma : apparent(sv) \bullet eval(\mathsf{e} \oplus \mathsf{f})(\sigma)\} \\
= \{\sigma : apparent(values(ss)) \bullet eval(\mathsf{e} \oplus \mathsf{f})(\sigma)\} \\
= eval\_samp(\mathsf{e} \oplus \mathsf{f})(ss)
\end{array}$$

□

Corollary 2 (eval-samp disjoint variables). *Provided the free variables of expressions* e *and* f *are disjoint and contained in* V*, then for all sets of states, ss, over* V*,*

$$eval\_samp(\mathsf{e} \oplus \mathsf{f})(ss) = eval\_samp(\mathsf{e})(ss)\ \overline{[\![\oplus]\!]}\ eval\_samp(\mathsf{f})(ss)\ .$$

Proof. Because $vars(\mathsf{e}) \cap vars(\mathsf{f})$ is empty, the proviso for Theorem 9 is satisfied for any set of states, *ss*, over variables including $vars(\mathsf{e} \oplus \mathsf{f})$. □

2.7 Relation between (nondeterministic) expression evaluations

Expression evaluation over a set of states is more deterministic than over the corresponding apparent states.

Theorem 10 (eval-ss in eval-samp). *For any expression,* e*, and set of states, ss, both over* V*,*

$$eval\_ss(\mathsf{e})(ss) \subseteq eval\_samp(\mathsf{e})(ss)\ .$$

Proof. From Corollary 1, $ss \subseteq apparent(values(ss))$, and the result follows using Lemma 2.

$$eval\_ss(\mathsf{e})(ss) \subseteq eval\_ss(\mathsf{e})(apparent(values(ss))) = eval\_samp(\mathsf{e})(ss)$$

□

Nondeterministic expression evaluation results in a set of possible values of the expression in all three evaluation schemes considered. As usual, an implementation of the evaluation of an expression may result in a smaller set of possible values, that is, refinement is the reverse of set inclusion. Theorem 12 (below) tells us that a sets-of-values evaluation may be implemented by a set-of-apparent-states (single) sampled evaluation, and Theorem 10 tells us that this may in turn be implemented by a set-of-states evaluation, although the latter (ideal) evaluation may be hard or impossible to implement in practice.

Definition 17 (refinement of evaluation). *One expression evaluation scheme, $eval_0$, is refined by another $eval_1$, written $eval_0 \sqsubseteq eval_1$, provided*

$$\begin{array}{l} \forall \mathsf{V} : \mathbb{P}\,\mathsf{Var} \bullet \forall \mathsf{e} : \mathsf{E}_\mathsf{V};\ ss : \mathbb{P}\,\Sigma_V \bullet \\ \quad eval_1(\mathsf{e})(ss) \subseteq eval_0(\mathsf{e})(ss)\ . \end{array}$$

The following corollary follows directly from Theorem 10.

Corollary 3 (eval-samp is refined by eval-ss). $eval\_samp \sqsubseteq eval\_ss$.

Note that in practice one has to make use of a sampling implementation (i.e., *eval_samp*) but would like to prove properties for all states (i.e., *eval_ss*), and hence the refinement relationship is the opposite of what is needed to make proofs of properties easy. In the special case where there is a single variable that is nondeterministic, the relationship becomes an equality and a sampling implementation is valid.

Theorem 11 (single nondeterministic variable eval). *If in a set of states, $ss \in \mathbb{P}\,\Sigma_\mathsf{V}$, all variables, with the possible exception of a single variable, are deterministic, then for any expression e, over V,*

$$eval\_ss(\mathsf{e})(ss) = eval\_samp(\mathsf{e})(ss)\ .$$

Proof. The proof follows from Theorem 6 because, under the assumptions, $ss = apparent(values(ss))$.

$$eval\_ss(\mathsf{e})(ss) = eval\_ss(\mathsf{e})(apparent(values(ss))) = eval\_samp(\mathsf{e})(ss)\ .$$

□

For a set of states, *ss*, evaluating an expression in its corresponding set of apparent states, $apparent(values(ss))$, is more deterministic than evaluating the expression in the corresponding sets-of-values view.

Theorem 12. *For any expression, e, and set of states, ss, both over V,*

$$eval\_samp(\mathsf{e})(ss) \subseteq eval\_vss(\mathsf{e})(ss)\ .$$

Proof. The proof relies on Lemma 4, which states that *eval_sv* is monotonic in sets-of-values views with respect to the subsumption ordering and Theorem 2.

$$\begin{array}{l} \quad eval\_samp(\mathsf{e})(ss) \\ = eval\_ss(\mathsf{e})(apparent(values(ss))) \\ = \{\sigma : apparent(values(ss)) \bullet eval(\mathsf{e})(\sigma)\} \\ = \{\sigma : \Sigma_\mathsf{V} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in values(ss)(v)) \bullet eval(\mathsf{e})(\sigma)\} \\ = \text{by Theorem 2} \\ \quad \bigcup\{\sigma : \Sigma_\mathsf{V} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in values(ss)(v)) \bullet eval\_sv(\mathsf{e})(det\_values(\sigma))\} \\ \subseteq \text{as } det\_values(\sigma) \sqsubseteq values(ss) \text{ for any } \sigma \text{ such that } (\forall v : \mathsf{V} \bullet \sigma(v) \in values(ss)(v)) \\ \quad \bigcup\{\sigma : \Sigma_\mathsf{V} \mid (\forall v : \mathsf{V} \bullet \sigma(v) \in values(ss)(v)) \bullet eval\_sv(\mathsf{e})(values(ss))\} \\ = \text{because } eval\_sv(\mathsf{e})(values(ss)) \text{ is independent of } \sigma \\ \quad eval\_sv(\mathsf{e})(values(ss)) \\ = eval\_vss(\mathsf{e})(ss) \end{array}$$

□

Evaluation of an expression using a single sampled value for each variable used in the expression is more deterministic than using a separate sample for each occurrence of a variable.

Corollary 4 (eval-vss is refined by eval-samp). $\mathit{eval\_vss} \sqsubseteq \mathit{eval\_samp}$.

Combining Corollary 3 and Corollary 4 gives

$$\mathit{eval\_vss} \sqsubseteq \mathit{eval\_samp} \sqsubseteq \mathit{eval\_ss} \ .$$

Theorem 13 (single variable reference eval). *If an expression, e, over V has only a single reference to each nondeterministic variable in a set of states, ss, over V, then*

$$\mathit{eval\_samp}(\mathsf{e})(\mathit{ss}) = \mathit{eval\_vss}(\mathsf{e})(\mathit{ss}) \ .$$

Proof. The proof uses induction over the structure of expressions. For a constant, c, the proof follows.

$$\mathit{eval\_samp}(\mathsf{c})(\mathit{ss}) = \{\sigma : \mathit{ss} \bullet \mathit{eval}(\mathsf{c})(\sigma)\} = \{\mathsf{c}\} = \mathit{eval\_sv}(\mathsf{c})(\mathit{values}(\mathit{ss})) = \mathit{eval\_vss}(\mathsf{c})(\mathit{ss})$$

For a variable the proof follows.

$$\begin{aligned}
&\mathit{eval\_samp}(\mathsf{v})(\mathit{ss})\\
=\ &\{\sigma : \mathit{apparent}(\mathit{values}(\mathit{ss})) \bullet \sigma(\mathsf{v})\}\\
=\ &\mathit{values}(\mathit{ss})(\mathsf{v})\\
=\ &\mathit{eval\_sv}(\mathsf{v})(\mathit{values}(\mathit{ss}))\\
=\ &\mathit{eval\_vss}(\mathsf{v})(\mathit{ss})
\end{aligned}$$

For a binary operator, if $\mathsf{e} \oplus \mathsf{f}$ has only a single reference to each nondeterministic variable, then the same property holds for both e and f and furthermore the references to nondeterministic variables in e and f are disjoint.

$$\begin{aligned}
&\mathit{eval\_vss}(\mathsf{e} \oplus \mathsf{f})(\mathit{ss})\\
=\ &\text{by Theorem 3}\\
&\mathit{eval\_vss}(\mathsf{e})(\mathit{ss}) \ \overline{[\![\oplus]\!]}\ \mathit{eval\_vss}(\mathsf{f})(\mathit{ss})\\
=\ &\text{by induction hypothesis (twice)}\\
&\mathit{eval\_samp}(\mathsf{e})(\mathit{ss}) \ \overline{[\![\oplus]\!]}\ \mathit{eval\_samp}(\mathsf{f})(\mathit{ss})\\
=\ &\text{by Corollary 2}\\
&\mathit{eval\_samp}(\mathsf{e} \oplus \mathsf{f})(\mathit{ss})
\end{aligned}$$

□

3 Predicates in the different evaluation approaches

The previous section considered three nondeterministic expression evaluation schemes. This section considers properties of predicates using these different schemes. For each scheme modal predicates are introduced that correspond to a predicate, p, holding for all possible evaluations in the scheme and for p holding for some evaluation in the scheme.

Section 3.1 first defines predicates on a single state, then Section 3.2 promotes these to predicates on sets of states such that the predicate holds for all states ($\boxast\,\mathsf{p}$) or for some state ($\boxdot\,\mathsf{p}$). Section 3.3 considers similar predicates (written $\circledast\,\mathsf{p}$ and $\odot\,\mathsf{p}$) on apparent (sampled) states and Section 3.4 considers similar predicates (written $\Diamondstar\,\mathsf{p}$ and $\Diamonddot\,\mathsf{p}$) on sets-of-values views.[5] Sections 3.5 and 3.6 explore the relationship between the different predicates. Finally, Section 3.7 offers a link to a notion used in rely/guarantee reasoning.

3.1 Predicates

A predicate over a state space, Σ, is represented as a boolean-valued expression over Σ.

Definition 18 (Predicate). *For a state space Σ, a predicate is represented as follows.*

$$\mathit{Pred}[\Sigma] == \Sigma \rightarrow \mathbb{B}$$

The conventional notations, "$\wedge$", "$\vee$", "$\neg$" and "$\Rightarrow$" are used for conjunction, disjunction, negation and implication. These operators are lifted point-wise to states, i.e., $(p \wedge q)(\sigma) = p(\sigma) \wedge q(\sigma)$. Universal implication, denoted $p \Rrightarrow q$, is defined as $(\forall \sigma \bullet p(\sigma) \Rightarrow q(\sigma))$. Universal equivalence is denoted $p \equiv q$.

[5] As a mnemonic, the universal versions of the operators include a "$*$", while the existential versions include a "$\cdot$".

3.2 Predicates on sets of states

Given a state predicate, p, there are two obvious ways to promote it to a set of states (as in modal logics [10]): p holds for *all states* in the set, written $\boxast \mathsf{p}$, and p holds for *some state* in the set, written $\boxdot \mathsf{p}$. For example, for the set of states

$$\{\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\}\}$$

satisfies the predicates $\boxast(\mathsf{u} \geq \mathsf{v})$ and $\boxdot(\mathsf{u} = \mathsf{v})$.

$\mathsf{P_V}$ stands for the subset of expression syntax $\mathsf{E_V}$ that are boolean valued.

Definition 19 (All states and some states).

$$\begin{array}{|l}
\boxast : \mathsf{P_V} \rightarrow \mathit{Pred}[\mathbb{P}\,\Sigma_\mathsf{V}] \\
\boxdot : \mathsf{P_V} \rightarrow \mathit{Pred}[\mathbb{P}\,\Sigma_\mathsf{V}] \\
\hline
\forall \mathsf{p} : \mathsf{P_V};\ ss : \mathbb{P}\,\Sigma_\mathsf{V} \bullet \\
\qquad (\boxast \mathsf{p})(ss) \Leftrightarrow (\forall b : \mathit{eval\_ss}(\mathsf{p})(ss) \bullet b) \land \\
\qquad (\boxdot \mathsf{p})(ss) \Leftrightarrow (\exists b : \mathit{eval\_ss}(\mathsf{p})(ss) \bullet b)
\end{array}$$

For example, in a real-time system there may be two boolean sensors, t and b. One can state that it is always the case that one or the other of the sensors is true $\boxast(t \lor b)$, which is a weaker requirement than one or the other of the sensors always being true, $(\boxast t) \lor (\boxast b)$. If one samples the sensors and both are true, one can deduce $(\boxdot t) \land (\boxdot b)$, which does not allow one to deduce the stronger property that both sensors are on together, i.e., $\boxdot(t \land b)$, because in the first case the state in which t is true may differ from the state in which b is true.

From Lemma 2 one can deduce the following lemma.

Lemma 13. *For all predicates,* p*, and sets of states,* ss_0 *and* ss_1 *such that* $ss_0 \subseteq ss_1$,

$$(\boxast \mathsf{p})(ss_1) \Rightarrow (\boxast \mathsf{p})(ss_0) \tag{9}$$
$$(\boxdot \mathsf{p})(ss_0) \Rightarrow (\boxdot \mathsf{p})(ss_1) \tag{10}$$

The boolean operators are promoted to predicates on sets of states in the obvious way (because they are defined as predicates, but over sets of states rather than states). The following properties of "all states" and "some state" hold when combined with logical operators. Note that (14) and (15) are not equivalences. These properties follow from Theorem 1.

Theorem 14. *For any predicates,* p *and* q,

$$\neg \boxast \mathsf{p} \equiv \boxdot(\neg \mathsf{p}) \tag{11}$$
$$\boxast \mathsf{p} \land \boxast \mathsf{q} \equiv \boxast(\mathsf{p} \land \mathsf{q}) \tag{12}$$
$$\boxdot \mathsf{p} \lor \boxdot \mathsf{q} \equiv \boxdot(\mathsf{p} \lor \mathsf{q}) \tag{13}$$
$$\boxast \mathsf{p} \lor \boxast \mathsf{q} \Rrightarrow \boxast(\mathsf{p} \lor \mathsf{q}) \tag{14}$$
$$\boxdot(\mathsf{p} \land \mathsf{q}) \Rrightarrow \boxdot \mathsf{p} \land \boxdot \mathsf{q} \tag{15}$$

Theorem 15. *Given predicates* p *and* q,

$$(\boxdot \mathsf{p} \Rightarrow \boxast \mathsf{q}) \Rrightarrow \boxast(\mathsf{p} \Rightarrow \mathsf{q}) \tag{16}$$

Proof. By (11) and (14) and the definition of implication. □

3.3 Predicates on apparent states

We promote a predicate, p, on a single state, to a predicate on sets of apparent states in two ways: if p holds for all apparent states one says p *definitely* holds, abbreviated $\circledast \mathsf{p}$, and if p holds for at least one apparent state, one says p *possibly* holds, abbreviated $\odot \mathsf{p}$.

Definition 20 (Definitely and possibly).

$$\begin{array}{|l}
\circledast : \mathsf{P_V} \to \mathit{Pred}[\mathbb{P}\,\Sigma_{\mathsf{V}}] \\
\odot : \mathsf{P_V} \to \mathit{Pred}[\mathbb{P}\,\Sigma_{\mathsf{V}}] \\
\hline
\forall\, \mathsf{p} : \mathsf{P_V};\ ss : \mathbb{P}\,\Sigma_{\mathsf{V}} \bullet \\
\qquad (\circledast\, \mathsf{p})(ss) \Leftrightarrow (\forall\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b) \wedge \\
\qquad (\odot\, \mathsf{p})(ss) \Leftrightarrow (\exists\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b)
\end{array}$$

From Lemma 11 one can deduce the following lemma.

Lemma 14. *For all predicates,* p*, and sets of states,* ss_0 *and* ss_1 *such that* $ss_0 \subseteq ss_1$*,*

$$(\circledast\, \mathsf{p})(ss_1) \Rightarrow (\circledast\, \mathsf{p})(ss_0) \tag{17}$$
$$(\odot\, \mathsf{p})(ss_0) \Rightarrow (\odot\, \mathsf{p})(ss_1) \tag{18}$$

The following properties are directly derivable from the properties of predicates on sets of states (11)–(15).

Theorem 16. *For any predicates,* p *and* q*,*

$$\neg \circledast \mathsf{p} \equiv \odot(\neg\, \mathsf{p}) \tag{19}$$
$$\circledast \mathsf{p} \wedge \circledast \mathsf{q} \equiv \circledast(\mathsf{p} \wedge \mathsf{q}) \tag{20}$$
$$\odot \mathsf{p} \vee \odot \mathsf{q} \equiv \odot(\mathsf{p} \vee \mathsf{q}) \tag{21}$$
$$\circledast \mathsf{p} \vee \circledast \mathsf{q} \Rrightarrow \circledast(\mathsf{p} \vee \mathsf{q}) \tag{22}$$
$$\odot(\mathsf{p} \wedge \mathsf{q}) \Rrightarrow \odot \mathsf{p} \wedge \odot \mathsf{q} \tag{23}$$

We can represent the fact that a variable is deterministic at the predicate level via a predicate *stable*(v) that states that v only takes on a single value.

Definition 21 (stable).

$$\begin{array}{|l}
\mathit{stable}_{\mathsf{V}} : \mathsf{V} \to \mathit{Pred}[\mathbb{P}\,\Sigma_{\mathsf{V}}] \\
\hline
\forall\, \mathsf{w} : \mathsf{V};\ ss : \mathbb{P}\,\Sigma_{\mathsf{V}} \bullet \\
\qquad \mathit{stable}_{\mathsf{V}}(\mathsf{w})(ss) \Leftrightarrow (\forall\, \sigma : ss \bullet \mathit{values}(ss)(\mathsf{w}) = \{\sigma(\mathsf{w})\})
\end{array}$$

There are two interesting properties of definitely ($\circledast$) and possibly ($\odot$) that do not hold for "all states" ($\boxast$) and "some states" ($\boxdot$).

Theorem 17. *For predicates* p *and* q *over variables* V*,*

$$\mathit{stable}_{\mathsf{V}}(\mathit{vars}(\mathsf{p}) \cap \mathit{vars}(\mathsf{q})) \Rrightarrow (\odot(\mathsf{p} \wedge \mathsf{q}) = \odot\, \mathsf{p} \wedge \odot\, \mathsf{q}) \tag{24}$$
$$\mathit{stable}_{\mathsf{V}}(\mathit{vars}(\mathsf{p}) \cap \mathit{vars}(\mathsf{q})) \Rrightarrow (\circledast\, \mathsf{p} \vee \circledast\, \mathsf{q} = \circledast(\mathsf{p} \vee \mathsf{q})) \tag{25}$$

Proof. We focus on property (24) because (25) can be derived from it because $\circledast\, \mathsf{p} = \neg \odot \neg\, \mathsf{p}$.

$$\begin{array}{l}
\quad \odot(\mathsf{p} \wedge \mathsf{q})(ss) \\
\Leftrightarrow (\exists\, b : \mathit{eval\_samp}(\mathsf{p} \wedge \mathsf{q})(ss) \bullet b) \\
\Leftrightarrow \text{by Theorem 9 as } \mathit{vars}(\mathsf{p}) \cap \mathit{vars}(\mathsf{q}) \text{ are deterministic in } ss \\
\quad (\exists\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \mathbin{\overline{\wedge}} \mathit{eval\_samp}(\mathsf{q})(ss) \bullet b) \\
\Leftrightarrow (\exists\, x : \mathit{eval\_samp}(\mathsf{p})(ss);\ y : \mathit{eval\_samp}(\mathsf{q})(ss) \bullet x \wedge y) \\
\Leftrightarrow (\exists\, x : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet x) \wedge (\exists\, y : \mathit{eval\_samp}(\mathsf{q})(ss) \bullet y) \\
\Leftrightarrow (\odot\, \mathsf{p})(ss) \wedge (\odot\, \mathsf{q})(ss) \\
\Leftrightarrow (\odot\, \mathsf{p} \wedge \odot\, \mathsf{q})(ss)
\end{array}$$

□

Note that (24) holds for $\odot$ but not $\boxdot$. For example, if

$$\mathit{ssuv} = \{\{\mathsf{u} \mapsto 0, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 0\}, \{\mathsf{u} \mapsto 1, \mathsf{v} \mapsto 1\}\}$$

then $\boxdot(\mathsf{u} = 0) \wedge \boxdot(\mathsf{v} = 1)$ holds in *ssuv* but not $\boxdot(\mathsf{u} = 0 \wedge \mathsf{v} = 1)$.

For the example of two boolean sensors, t and b, Theorem 17 allows one to deduce that $(\odot\, t) \wedge (\odot\, b)$ is equivalent to $\odot(t \wedge b)$, because t and b are distinct variables, whereas one cannot deduce $\boxdot(t \wedge b)$.

Theorem 18. *For predicates* p *and* q *over* V,

$$stable_{\mathsf{V}}(vars(\mathsf{p}) \cap vars(\mathsf{q})) \Rightarrow (\circledast(\mathsf{p} \Rightarrow \mathsf{q}) \equiv (\odot\,\mathsf{p} \Rightarrow \circledast\,\mathsf{q}))$$

Proof. The theorem follows from Theorem 17 and the definition of implication. □

3.4 Sets-of-values predicates

A predicate, p, on a single state can be promoted to a predicate on a sets-of-values view in two ways: if p holds for all evaluations in a sets-of-values view one says p *positively* holds, abbreviated $\Diamondast\mathsf{p}$ and if p holds for at least one evaluation in a sets-of-values view, one says p *maybe* holds, abbreviated $\Diamonddot\mathsf{p}$.

Definition 22 (Positively and maybe).

$$\begin{array}{|l}
\Diamondast : \mathsf{P}_{\mathsf{V}} \rightarrow Pred[\mathbb{P}\,\Sigma_{\mathsf{V}}] \\
\Diamonddot : \mathsf{P}_{\mathsf{V}} \rightarrow Pred[\mathbb{P}\,\Sigma_{\mathsf{V}}] \\
\hline
\forall\,\mathsf{p} : \mathsf{P}_{\mathsf{V}};\ ss : \mathbb{P}\,\Sigma_{\mathsf{V}} \bullet \\
\qquad (\Diamondast\mathsf{p})(ss) \Leftrightarrow (\forall\, b : eval\_vss(\mathsf{p})(ss) \bullet b) \wedge \\
\qquad (\Diamonddot\mathsf{p})(ss) \Leftrightarrow (\exists\, b : eval\_vss(\mathsf{p})(ss) \bullet b)
\end{array}$$

From Lemma 8 one can deduce the following lemma.

Theorem 19. *For all predicates,* p, *and sets of states,* ss_0 *and* ss_1 *such that* $ss_0 \subseteq ss_1$,

$$(\Diamondast\mathsf{p})(ss_1) \Rightarrow (\Diamondast\mathsf{p})(ss_0) \tag{26}$$
$$(\Diamonddot\mathsf{p})(ss_0) \Rightarrow (\Diamonddot\mathsf{p})(ss_1) \tag{27}$$

Theorem 20. *For all predicates,* p *and* q,

$$\neg\,\Diamondast\mathsf{p} \equiv \Diamonddot(\neg\,\mathsf{p}) \tag{28}$$
$$\Diamondast\mathsf{p} \wedge \Diamondast\mathsf{q} \equiv \Diamondast(\mathsf{p} \wedge \mathsf{q}) \tag{29}$$
$$\Diamonddot\mathsf{p} \vee \Diamonddot\mathsf{q} \equiv \Diamonddot(\mathsf{p} \vee \mathsf{q}) \tag{30}$$
$$\Diamondast\mathsf{p} \vee \Diamondast\mathsf{q} \equiv \Diamondast(\mathsf{p} \vee \mathsf{q}) \tag{31}$$
$$\Diamonddot(\mathsf{p} \wedge \mathsf{q}) \equiv \Diamonddot\mathsf{p} \wedge \Diamonddot\mathsf{q} \tag{32}$$

Proof. The properties follow from Theorem 3. The interesting cases are (31) and (32) as in the set-of-values view they are equivalences. We show just (32) here.

$$\begin{array}{ll}
& \Diamonddot(\mathsf{p} \wedge \mathsf{q})(ss) \\
\Leftrightarrow & (\exists\, b : eval\_vss(\mathsf{p} \wedge \mathsf{q})(ss) \bullet b) \\
\Leftrightarrow & \text{by Theorem 3} \\
& (\exists\, b : eval\_vss(\mathsf{p})(ss)\ \overline{\wedge}\ eval\_vss(\mathsf{q})(ss) \bullet b) \\
\Leftrightarrow & (\exists\, x : eval\_vss(\mathsf{p})(ss);\ y : eval\_vss(\mathsf{q})(ss) \bullet x \wedge y) \\
\Leftrightarrow & (\exists\, x : eval\_vss(\mathsf{p})(ss) \bullet x) \wedge (\exists\, y : eval\_vss(\mathsf{q})(ss) \bullet y) \\
\Leftrightarrow & (\Diamonddot\mathsf{p})(ss) \wedge (\Diamonddot\mathsf{q})(ss) \\
\Leftrightarrow & (\Diamonddot\mathsf{p} \wedge \Diamonddot\mathsf{q})(ss)
\end{array}$$

□

3.5 Relating the predicates

The refinement relations between the evaluation approaches induce a relationship between the modal predicates as captured by the following theorem.

Theorem 21. *For any predicate,* p,

$$\Diamondast\mathsf{p} \Rightarrow \circledast\,\mathsf{p} \Rightarrow \boxast\,\mathsf{p} \tag{33}$$
$$\boxdot\,\mathsf{p} \Rightarrow \odot\,\mathsf{p} \Rightarrow \Diamonddot\mathsf{p} \tag{34}$$

Proof. We concentrate on (33); the proof of (34) is similar. For all sets of states, *ss*,

$$
\begin{array}{l}
\quad (\diamondast\, \mathsf{p})(ss) \\
\Leftrightarrow \text{ by Definition 22 (positively)} \\
\quad (\forall\, b : \mathit{eval\_vss}(\mathsf{p})(ss) \bullet b) \\
\Rightarrow \text{ by Theorem 12} \\
\quad (\forall\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow \text{ by Definition 20 (definitely)} \\
\quad (\circledast\, \mathsf{p})(ss) \\
\Leftrightarrow \text{ by Definition 20 (definitely)} \\
\quad (\forall\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b) \\
\Rightarrow \text{ by Theorem 10} \\
\quad (\forall\, b : \mathit{eval\_ss}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow \text{ by Definition 19 (allstates)} \\
\quad (\boxast\, \mathsf{p})(ss)
\end{array}
$$

□

A simple case is if there is only one nondeterministic variable, w, i.e., all other variables have only one value in all states, e.g., the predicate is of the form $\circledast(\mathsf{w} \in \mathsf{S})$ or $\odot(\mathsf{w} \in \mathsf{S})$, where S is stable over the observation interval, then $\circledast(\mathsf{w} \in \mathsf{S})$ is equivalent to $\boxast(\mathsf{w} \in \mathsf{S})$, and $\odot(\mathsf{w} \in \mathsf{S})$ is equivalent to $\boxdot(\mathsf{w} \in \mathsf{S})$. Special cases of these predicates are comparisons of a variable with an expression, e, that is stable over the observation interval, e.g., $\mathsf{w} = \mathsf{e}$ and $\mathsf{w} < \mathsf{e}$.

Theorem 22 (single non-stable variable). *For a predicate* p *over* V*,*

$$\mathit{stable}_{\mathsf{V}}(\mathit{vars}(\mathsf{p}) \setminus \mathsf{w}) \Rrightarrow (\circledast\, \mathsf{p} = \boxast\, \mathsf{p}) \tag{35}$$

$$\mathit{stable}_{\mathsf{V}}(\mathit{vars}(\mathsf{p}) \setminus \mathsf{w}) \Rrightarrow (\odot\, \mathsf{p} = \boxdot\, \mathsf{p}) \tag{36}$$

Proof. We focus on the proof of (35). The proof of (36) is similar.

$$
\begin{array}{l}
\quad (\circledast\, \mathsf{p})(ss) \\
\Leftrightarrow (\forall\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow \text{ by assumption and Theorem 11} \\
\quad (\forall\, b : \mathit{eval\_ss}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow (\boxast\, \mathsf{p})(ss)
\end{array}
$$

□

If one samples a variable, v, in the environment and gets the value c, one can deduce $\boxdot(\mathsf{v} = \mathsf{c})$ which is equivalent to $\odot(\mathsf{v} = \mathsf{c})$. Similarly, by sampling w, one may deduce $\boxdot(\mathsf{w} = \mathsf{d})$, which is equivalent to $\odot(\mathsf{w} = \mathsf{d})$. Theorem 17 then allows one to deduce $\odot(\mathsf{v} = \mathsf{c} \wedge \mathsf{w} = \mathsf{d})$ but not the stronger condition $\boxdot(\mathsf{v} = \mathsf{c} \wedge \mathsf{w} = \mathsf{d})$.

3.6 On the relationship between sets of apparent states and set-of-values views

Theorem 23 (single variable reference). *If a predicate,* p*, only has a single reference to each variable that is not stable,*

$$\diamondast\, \mathsf{p} \equiv \circledast\, \mathsf{p} \tag{37}$$

$$\Diamonddot\, \mathsf{p} \equiv \odot\, \mathsf{p} \tag{38}$$

Proof. We focus on the proof of (37) — the proof of (38) is similar.

$$
\begin{array}{l}
\quad (\diamondast\, \mathsf{p})(ss) \\
\Leftrightarrow (\forall\, b : \mathit{eval\_vss}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow \text{ by assumption and Theorem 13} \\
\quad (\forall\, b : \mathit{eval\_samp}(\mathsf{p})(ss) \bullet b) \\
\Leftrightarrow (\circledast\, \mathsf{p})(ss)
\end{array}
$$

□

3.7 Possible values

A completely disjoint piece of research has led to the need to reason about what values arise in states when they are being changed by a process other than the one being specified. This section links the two previously separate notations.

Standard "Hoare logic" [9] is well known. VDM [13] uses –in so-called "operation decomposition" proofs– a related set of rules that cope with post-conditions which are relations (of initial and final state). Expressions in such relations distinguish between the initial values of variables, written $\overleftarrow{x}$, and their final values written simply as x. Thus a post condition for a simple assignment statement, $x \leftarrow y$, could be written $x = \overleftarrow{y}$.

The basis of rely/guarantee reasoning [11, 12, 14] is to face the fact that interference is the essence of concurrency and to use rely conditions to record the interference an operation can tolerate and guarantee conditions to warn of interference the operation can inflict. In a recent paper [17], a new element of notation was introduced and its links to the analysis in Section 1 above are intriguing. The notation $\widehat{y}$ is defined as the set of values that the variable y has over the execution of an operation. In the cited paper, the main payoff of this new concept is in rely and guarantee conditions but, for the purposes of the current paper, the use can be illustrated with a simple post condition. If the set of states over the execution interval is ss, then using the concepts developed above $\widehat{y}$ corresponds to $values(ss)(y)$.

Suppose a specification is implemented by some code which includes the assignment $x \leftarrow y$, and that this is executed in an environment that can potentially change the value of y. The *specification* of the statement must reflect the fact that x could acquire the initial or final values of y (i.e. $x = \overleftarrow{y} \lor x = y \lor \cdots$). However, there remains the possibility that the value of y changes multiple times between the start of the operation being specified and its termination. Written in a post condition, $x \in \widehat{y}$ expresses exactly that the final value of x will be some value that y possessed during execution of the operation being specified. Clearly, this notation will be of more use in larger applications but hopefully this simple example illustrates the concept without going into the details of a complex piece of reasoning about concurrency as is contained in [17].

The closest link with the distinctions in Section 1 is with what is called there "sets of apparent states". Writing $\exists u' \in \widehat{u}, v' \in \widehat{v} \bullet u' \geq v'$ and $\forall x' \in \widehat{x} \bullet x' = x'$ gives exactly that meaning, i.e., $\odot(u \geq v)$ and $\circledast(x = x)$. To obtain "sets of values", it is necessary to recognise that references to variables in expressions imply separate accesses. Thus the second expression changes to $\forall x', x'' \in \widehat{x} \bullet x' = x''$, which corresponds to $\diamond\!\!\!\ast(x = x)$.

The "sets of states" interpretation is, as commented upon in Section 1, useful for invariants. Referencing the possible values of a vector of variables, e.g., $\widehat{(u, v)}$, allows the values of u and v to be captured together (at the same time). For a vector, (u, v), its possible values $\widehat{(u, v)}$, for a set of states ss, corresponds to $\{\sigma : ss \bullet (\sigma(u), \sigma(v))\}$. For example $\forall (u', v') \in \widehat{(u, v)} \bullet u' \geq v'$ corresponds to $\boxast(u \geq v)$. It might not be immediately obvious why the vector extension of "possible values" would be useful in the presence of interfering concurrency but it was an extension that had already been considered. It can be realised by some form of locking.

It remains to be seen what this notation adds to the underlying concept of nondeterministic expression evaluation but it is always encouraging when separate directions of research come together.

4 Expression evaluation over time intervals

The motivation for considering models of nondeterministic expression evaluation comes from evaluating expressions in contexts in which the values of variables within an expression may be changing while the expression is being evaluated. We can model such evolving states as a trace over time:

$$Trace_{\mathsf{V}} \mathrel{\widehat{=}} Time \to \Sigma_{\mathsf{V}}$$

where *Time* could be real numbers to handle real time or natural numbers representing abstract (progression of) time. This model is used in the real-time refinement calculus [7, 6] and in specifying systems in the context of their environment [8, 15]. The time interval over which an expression is evaluated can be represented as a set of times, T, and then the corresponding set of states can be extracted from a trace, tr, by the following function.

$$states : \mathbb{P}\, Time \to (Trace_{\mathsf{V}} \to \mathbb{P}\, \Sigma_{\mathsf{V}})$$

$$states(T)(tr) = \{t : T \bullet tr(t)\}$$

Although the theory presented in this paper could be presented directly on timed traces and evaluation time intervals, the essential differences between the evaluation strategies can be handled by considering just the corresponding set of states in the evaluation interval and the theory is more simply elucidated on sets-of-states model than the model based on timed traces and intervals. In fact, if a special auxiliary variable, called τ, is introduced to represent time within a state, a set of states has all the expressive power of a trace over a time interval. For a trace, $tr \in Trace_{\mathsf{V}}$, the corresponding set of states is $\mathrm{ran}(tr)$, where assuming $\tau \in \mathsf{V}$,

$$\forall\, t : \mathrm{dom}(tr) \bullet tr(t)(\tau) = t\;.$$

The use of states with time encoded as the special variable τ allows properties involving time to be expressed in the set-of-states model, although perhaps not as succinctly as in the timed-trace model.

5 Conclusions

We have presented three models of nondeterministic evaluation of an expression, e, over a set of states, *ss*:

– a set-of-states evaluation, in which e is evaluated in each state within *ss*;
– a set-of-apparent-states (or sampling) evaluation, in which e is evaluated in each state within the apparent states corresponding to *ss*;
– a sets-of-values view evaluation, in which the sets-of-values evaluation is applied to the set-of-values view corresponding to *ss*.

These evaluation strategies are progressively less deterministic. Sets-of-states evaluation is desirable for expressing safety properties of systems but is usually not possible to observe in an implementation. Sets-of-values evaluation corresponds to accessing a variable every time it is needed, even if it occurs multiple times in an expression. This strategy has been used by Ward [26], Larsen and Hansen [18], and Coleman and Jones [4], and is suitable for modelling expression evaluation in concurrent processes with shared memory. Set-of-apparent-states evaluation is an intermediate strategy that only samples each variable occurring in an expression once. This strategy was introduced by Burns and Hayes [3] and is suitable for modelling expression evaluation in real-time systems.

The pair of functions $(values, apparent)$ forms a Galois connection between the different views of the sets of states. The function *values* takes a set of states and gives the corresponding set-of-values view and the function *apparent* maps in the reverse direction. The fact that they form a Galois connection encourages us that *apparent* is the appropriate reverse map corresponding to *values*.

We have explored under what conditions the different evaluation strategies coincide. The sets-of-states and sets-of-apparent-states strategies coincide if there is only a single free variable of the expression that is nondeterministic within the set of states. The sets-of-apparent-states and sets-of-values strategies coincide if the expression has only a single reference to each nondeterministic variable.

Our models have assumed that variables can only be accessed and updated atomically. This may not be the case for variables with multiple-word values, such as arrays or records, being accessed/updated as a whole. Within the models presented here, such variables can be represented by treating a composite variable, such as an array, as a set of variables, one for each (atomic) word of the composite, e.g., having one variable for each element (word) of an array. Alternatively, a composite structure can be represented as a single variable, but the update operation on the whole must be represented as a sequence of (atomic) partial updates which overall correspond to the update of the whole, e.g., assigning a new value to the whole of an array is represented by a sequence of updates to the array in which each update modifies a single element (word) in the array, thus making the intermediate states of the array visible.

Conditional operators, such as conditional "and", e.g., $(\mathsf{i} \in \mathrm{dom}(\mathsf{A}))\ \mathsf{cand}\ (\mathsf{A}(\mathsf{i}) = 0)$, only evaluate the second argument conditionally; in the case of "cand" if the first argument evaluates to true. Such operators are handled by our theories, for example, the above evaluation will not lead to an index out of range being used in either the sets-of-states model or the apparent-sets-of-states model, but may in the set-of-values model.[6]

In some cases it may be known that updates to variables will occur in a particular order, which may reduce the number of possible values of an expression if it too evaluates subexpressions in a particular, for example, if it uses the conditional "and" operator described above. Our models do not explicitly take into account ordering

[6] Assuming the position and length of the array A do not change dynamically.

of updates, but these could be encoded via (abstract) time using the auxiliary variable, τ, discussed in Section 4. There is a further complication now being foisted on software developers: in "relaxed memory models" [25], the order of writes into memory becomes a new topic for nondeterminacy! From the point of view of a concurrent expression evaluation this increases the possible nondeterminacy.

When reasoning about systems one would like to state that properties hold for all states of the system or for some state and hence suitable modal predicates have been introduced for each of the models. One can show that a property holds for all states of the system by showing the (stronger) property that it holds for all apparent states, or the (even stronger) property that it holds for the sets-of-values view of the states. Conversely, if a property holds for some actual state, it also holds for some apparent state, and if it holds for an apparent state it holds for a set-of-values view of the state.

Acknowledgements. This research was supported, in part, by Australian Research Council (ARC) Discovery Grants DP0987452 and EPSRC(UK) TrAmS and Tempo grants. We would like to thank Joey Coleman for discussions on his research that influenced the approach taken in this paper.

References

1. J. R. Abrial, S. A. Schuman, and B. Meyer. Specification language and on the construction of programs: An advanced course. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An advanced course*, pages 343–410. Cambridge University Press, 1980.
2. G. Baxter, A. Burns, and K. Tan. Evaluating timebands as a tool for structuring the design of socio-technical systems. In P. Bust, editor, *Contemporary Ergonomics 2007*, pages 55–60. Taylor & Francis, 2007.
3. Alan Burns and Ian J. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems*, 45(1–2):106–142, June 2010.
4. J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.
5. I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall, second edition, 1993.
6. I. J. Hayes. A predicative semantics for real-time refinement. In A. McIver and C. C. Morgan, editors, *Programming Methodology*, pages 109–133. Springer Verlag, 2003.
7. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.
8. I.J. Hayes, M.A. Jackson, and C.B. Jones. Determining the specification of a control system from that of its environment. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 154–169. Springer Verlag, 2003.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
10. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. University Paperbacks. Routledge, 1968.
11. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
12. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
13. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.
14. C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
15. C. B. Jones, I. J. Hayes, and M. A. Jackson. Deriving specifications for systems that are connected to the physical world. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer Verlag, 2007.
16. Cliff B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, 2003.
17. Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.
18. Peter Gorm Larsen and Bo Stig Hansen. Semantics of under-determined expressions. *Formal Aspects of Computing*, 8(1):47–66, 1996.
19. P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6, Part 3 of *Annual Review in Automatic Programming*. Pergamon Press, 1969.
20. J. McCarthy. A formal description of a subset of ALGOL. In *[24]*, pages 1–12, 1966.

21. Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.
22. Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.
23. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.
24. T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.
25. Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 43–54. ACM, 2011.
26. N. Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, The Department of Computer Science, The University of Queensland, February 1994.