ISSN 2277-2685
IJESR/April. 2022/ Vol-12/Issue-2/1-11

Interval Specification for Actually Concurrent Logic

Dr.G Manikanta , Mrs.V Hemasree , Mr.A Srinivasan
Associate Professor[1,2] , Assistant Professor[3]
Department of CSE,
Viswam Engineering College (VISM) Madanapalle-517325 Chittoor District, Andhra Pradesh, India

Abstract

We extend the -calculus to allow the expression of attributes that hold true throughout the execution of an action, therefore obtaining a framework for the definition of genuine concurrency in reactive systems. Transition systems are used to represent the ST-semantics, which provide the basis for the interpretation of this logic. We demonstrate the unparalleled expressive capability of this logic and step equivalence. We further demonstrate that the logic describes the ST-bisimulation equivalence for synchronization-free finite process algebra expressions.

Introduction

Reactive systems rely heavily on concurrency. True concurrency, in which the simultaneity of events can be seen, and interleaving, in which parallel executions are turned into nondeterministic sequential ones, are the two approaches to concurrency management. The ability to tell the difference between a.b + b.a and a_b (where a.b represents the sequential execution of a and b, + represents the choice, and _ represents the parallel operator) is a common indicator of a really concurrent observation. In an interleaved method, it is impossible to tell one procedure apart from the other. For example, process algebras [12,15], event structures [23], and petri nets [23] are all forms of formalization that support genuine concurrency. Specification is an area where modal logics shine as a helpful formalism. The -calculus [14] is an example of a language that uses an interleaving technique and so cannot describe genuine concurrency. Changing the action set in the -calculus, such as by allowing multi-sets of actions, which corresponds to step semantics [21], is one way to have a true concurrent logic. However, there are situations when the step observation is insufficient, such as when it is not maintained by action refinement [7]. The logical expression becomes very convoluted and illegible when dealing with more advanced action sets such as pomsets [22], which is inappropriate for a specification language.

In order to develop a comprehensible genuinely concurrent logic that can be automatically tested, we modify the μ-calculus so that attributes that are true throughout the execution of an action may be expressed. In particular, we don't use formulae of the type _a_1, but rather formulations of the form _a 1_2, where 2 refers to a condition that must hold after the execution of action a (i.e., after the beginning and ending of a) and 1 refers to a condition that must hold immediately after the beginning of a. In-between logic (IBlogic for short) is the name given to the interpretation of this new logic over transition systems that correspond to the ST-semantics. The benefits of the IB-logic are that:

• It is a really concurrent logic, e.g., _a − _b − true_true_true indicates that after the start of action a another action b may be initiated, which is only feasible if a and b are concurrent.

• Its semantic foundation is built on the widely accepted concept of transition systems.

• It is a natural and straightforward development of the standard -calculus. Specifically, the -calculus can be easily incorporated into the IB-logic, which means that the IB-logic is more expressive than the -calculus.

• It may be included into the -calculus with an augmented action set. As a consequence, everything discovered about or developed for the -calculus may be used to the IB-logic. In particular, validity of IB-formulas may be automatically tested.
• It's simple to grasp, making it less prone to mistakes than, say, logic systems that handle an expanded action set.
How expressive this reasoning really is is a fascinating open subject. Is it, for instance, more evocative than, say, and step observation? Or how it relates to the ST-approach [8], which was explored to find the least coarse equivalence between interleaved bisimulations that holds up during action refinement. In the ST-approach, a single action is decomposed into its beginning and end, each of which is treated as a separate activity. IB-bisimulation, and demonstrate that two processes are equal in terms of the IB-formulas if and only if they fulfill all of the identical ones. We demonstrate that the IBequivalence's expressive capability much exceeds that of the step bisimulation. We also demonstrate that the ST-bisimulation is strictly more expressive than the IB-equivalence. Contrarily, we show that processes derived from limited process algebra and devoid of action synchronization may be described using the IB-formulas, which are sufficiently expressive. This paper will have the following structure: In Section 2, we present stack-based transition systems. A process algebra coupled with an ST-based semantics in terms of stack-based transition systems is also offered there. Section 3 presents the IB-logic coupled with the embedding with regard to the μ-calculus. In Section 4, we present IB-bisimulation and compare it to ST-bisimulation. In Section 5, we review related work, and in Section 6, we draw conclusions and talk about what comes next.

Systemic Adaptation for Transition

To demonstrate how action splitting semantics may be applied to stack-based transition systems, we introduce process algebra and provide it with an ST-based operational semantics. The parallel operator in our processes algebra is inspired by CSP [12].

Procedure Algebra Syntax

Let's pretend that the collection of things you do is called Act. A function f from Act to Act is called a relabeling function. For all relabeling functions, we use the notation FL. In addition, we assume that Act and VarP are two independent sets of process variables. The following is the BNF-grammar for process algebra expressions, abbreviated as EXP.

FL = f, VarP = x, a = Act, and A = Act. The pair _decl, B_ consists of the declaration decl: VarP EXP and the expression B EXP, and represents a process with regard to EXP. Let's call the total number of processes PA. If the decl component of an expression B EXP is obvious, we will refer to it as a process. The following is a sense of the meaning behind these expressions: The inactive process is represented by the number 0, while the action prefix process is represented by the notation a.B, which may carry out the action a and then transition to the action B. The process a.0 will be denoted as in the following text. The B1+B2 procedure represents the decision between two possible actions. In accordance with convention, the decision is made when acts begin, in contrast to [5], where it is made after they end. When B1 and B2 run in parallel, they must coordinate on actions from A, which is described by the process B1_AB2. If B performs action a, then B[f] will also do action f(a). x's expected behavior is specified by the declaration.
Note 2.1 In order to prevent the introduction of process termination, we only investigate an action prefix operator, rather than the more general sequential operator B1;B2. This is done to boost readability.

Transitional Systems Based on Stacks

In the ST-semantics [8,7] an action is not regarded to have an atomic execution. Specifically, an action is decomposed into two actions, one for its beginning and another for its end; in addition to the pure splitsemantics of [10], the end of an action has a special relationship to its beginning. The ST-semantics may be encoded using a variety of transition system encoding methods. For a summary, see [2]. The benefits of the stack method [2] led us to adopt it.

In addition, it generates finite transition systems for a large category of processes. Therefore, more processes have a decidable STbisimulation equivalence. The stack-based method also reduces the number of states in transition systems derived from a process algebra expression [2].

• It may be deduced from the transition systems of its parts, or "compositional," which means that the transition system of a process can be constructed from the sum of its pieces' value. This is helpful since it makes axiomatization easier to achieve [2].

• It offers a sufficient method for handling the action refinement operator's operational semantics, as shown in [9].

The idea behind the stack technique is as follows, where an active action is one that has been started but has not yet terminated: the beginning of an action and is indicated in the labels of the transition system by a+, and the end of an action and is indicated by an, where n is a natural number indicating that exactly n 1 a-actions that were initiated after the beginning of the a+ action corresponding to the an action are still active.

If an a-action begins at an execution position ts and ends with an an at an execution position tf, then the number of a-actions that began after ts and have not yet ended before tf is precisely n 1. Here's a case study that can provide light on the method

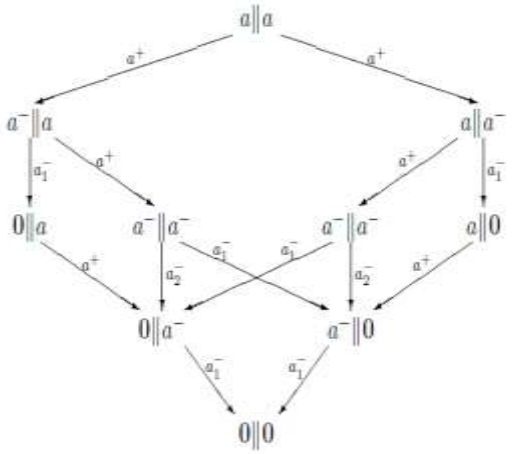

Figure 1 provides an example of the Stack Technique by showing a transition system built using a stack, which is created from the process a_a. The relative active number of the action is n, and it corresponds to the expression an. Hereafter, we will refer to any non-zero natural numbers as N+.

The [Stack-Based Transition System] is defined as follows: A transition system over Act with labels from L = (Act+)(Act N+) is called a stack-based transition system, or a tuple (S,Act,), where • S is a non-empty collection of states and • S LS is a stack-based transition relation.

Instead of writing (s,, s_) , we just need to write s s_. In addition, we write Act + elements as a+ and Act N+ elements as an. It is not sufficient to describe processes using the class of all stack-based transition systems. The following informal definition gives helpful constraints.

Definition: If the beginning of every action execution can be immediately terminated, if every active action can be immediately terminated, if every state can have at most one transition for every termination action, if termination disabling is free, and if action termination deterministic, then a stack-based transition system is atomic sensitive, interrupt free, action termination deterministic, and termination disabling free.

In general, all processes adhere to the deterministic characteristics of atomic sensitivity and action termination. Processes constructed from process algebras often satisfy the termination disabling and start enabling freeness. When there is no disrupt/interrupt operator in the process algebras, we get interruption freedom. Non-active states are those in which no activities are currently being taken. To be more specific, an action is considered to have begun during the execution sequences leading to the state s_ that may end it. Following is a description of how to generate from a stack-based transition system a system using action labels rather than start and termination action labels:

A stack-based transition system is defined as follows: (S,Act,). If there exists s__ such that s a+ s__ a 1 s_, then its un-splitting transition system is the transition system (S,Act, _) with s a _ s_. For transition systems based on stacks, the ST-bisimulation equivalence is defined as follows:

ST-Bisimilarity is defined as: In this example, we'll use a stack-based transition mechanism (S,Act,). For this stack-based transition system, we have an ST-bisimulation if and only if for every (s1, s_) on the stack, R S S.

If there exists an ST-bisimulation R such that (s, s_) R, then the two items s, s_ S are ST-bisimilar (or ST-equivalent), represented as s ST s_.

In the sense that two states are dissimilar iff they fulfill the identical formulae, bisimulation over labelled transition systems may be characterized by the Hennessy-Milner logic [11] and the -calculus [14]. Therefore, the ST-equivalence may be characterized by applying these logics with the action set (Act +) (Act N+). It's not easy to tell in a formula which starts action a termination action relates to, especially when there are several indices expressing the relative active numbers. Furthermore, the stack-based transition systems that occur naturally are ignored by these logics.

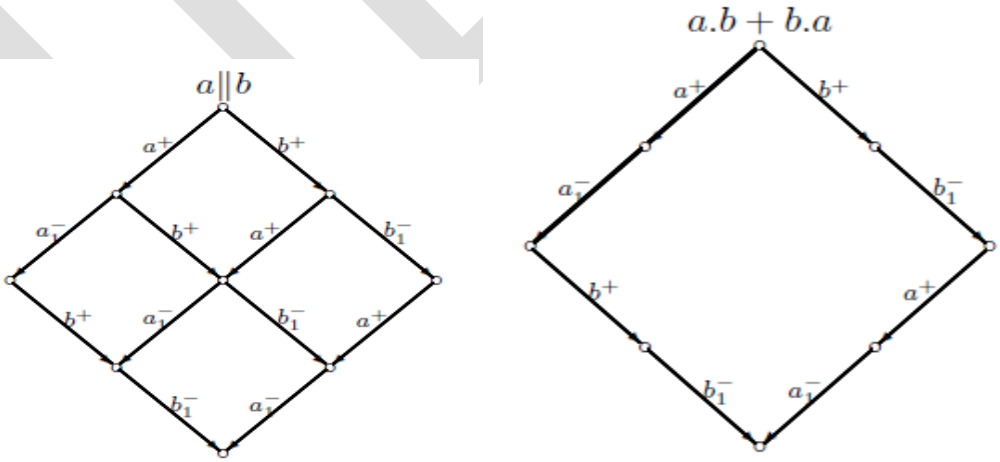

Fig. 2. Some Stack-Based Transition Systems
possess unique qualities (compare with Definition 2.3). Therefore, these logics may be too expressive.

The Process Algebra's Operational Semantics in a Set Theory Context

The operational semantics are defined as described in [2]. This section contains only examples illustrating how an expression in process algebra may be translated into operational semantics. In Figure 1 we see the derived stack-based transition system of process a_a (where the parallel operator has to know which side the distinct relative active number refers to). Assuming that a and b are distinct action names, the resulting stack-based transition system for processes a_b and a.b + b.a is shown in Figure 2. Any transition system built on a stack that is the result of EXP's evaluation will adhere to the guidelines laid forth in Definition 2.3. Furthermore, the unsplitting transition system generated from the stack-based operational semantics of this process is identical to the transition system produced from an EXP process with pure action execution. (restricted to the reachable states). Additionally, if the transition system acquired through pure action execution has finitely many states, then the transition system derived from an expression of EXP has finitely many states as well.

Structured Change in Steps

Using finite multi-sets of actions as its labels, the step transition system is a kind of transition system. Formally:
Defined: [System for Changing Levels] any transition system (S, Act, _) in which the labels range from

$$\mathcal{L}^{\text{step}} = \{\omega : \mathcal{A}ct \to \mathbb{N} \mid \infty > \sum_{a \in \mathcal{A}ct} \omega(a)\}, \text{ and so } \rightsquigarrow \subseteq S \times \mathcal{L}^{\text{step}} \times S.$$

Step-equivalence is similarly defined as ST- equivalence, except that step instead of stack-based transition systems are used. A corresponding step transition system is derived from a stack-based transition system as follows:
Definition: Suppose (S, Act, −→) is a stack-based transition system. Then its step transition system is the transition system $(S, \mathcal{A}ct, \rightsquigarrow)$ with $s \overset{\omega}{\rightsquigarrow} s'$ if $\forall a \in \mathcal{A}ct : \omega(a) > 0 \Rightarrow \exists s_1, s_2 : s \xrightarrow{a^1} s_1 \overset{\omega[a \mapsto \omega(a)-1]}{\rightsquigarrow} s_2 \xrightarrow{a_1} s'$,
Where $\omega[a \longmapsto (\omega(a) - 1)$

Is the function whose value is lowered by 1 on a, but which else agrees with everywhere. For every given EXP process, the step transition system derived from the process's stack-based operational semantics is identical to the system obtained from the process itself. (Restricted to the reachable states).

Transitional Reasoning

To define attributes that must hold while an action is being executed, we provide a variant of the -calculus [14].

The Logical Syntax

Consider the set of variables in logic to be VarL. The following syntax is used to construct IB-formulas:

$$\phi ::= false \mid true \mid X \mid \langle a - \phi \rangle \phi \mid [a - \phi] \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mu X.\phi \mid \nu X.\phi,$$

Where X ∈ VarL and a ∈ Act. The set of all formulas is denoted by F. A formula is closed if every occurrences of variable X appears inside a formula
of form μX.φ or νX.φ. The intuition of most of these formulas is the same as for the μ-calculus, where, e.g., μX.φ denotes the least and νX.φ denotes the greatest fix-point formula. The intuition of _a − φ1_φ2 is that there is an execution of action a such that φ2 has to hold after the complete execution of this a whereas φ1 has to hold after the start of this a. In particular, the formula states that there is an action a that can be started and that can be immediately

terminated. Please note that φ2 only has to hold immediately after the termination of a, i.e., there is no statement which has to hold after the termination of a

when further executions took place during the execution of a. The intuition of [a − φ1]φ2 is that whenever an action a is started which can be terminated immediately, then φ1 holds after the start of this a or φ2 holds immediately after the termination of this a. This definition is on the first sight strange, since one would expect a definition like 'after every start φ1 has to hold and immediately after every termination φ2 has to hold'. The presented interpretation of the box formula is chosen, since the box formula yields the dual of the diamond formula in our interpretation. The above described formula can be described within our interpretation by ([a−φ1]false)∧([a−false]φ2). We do not introduce a negation formula explicitly, since negation can be modeled by a duality operator, which will be shown later.

Semantics of the Logic

Suppose T = (S,Act,−→) is a stack-based transition system. Then the semantics [[ ]]T : F ×(VarL → P(S))→ P(S) is defined as follows, where P(S) denotes the power set of S.

X = VarL + a = Act. F stands for "the set of all formulas." When all uses of variable X are contained inside a formula of the type X. or vX., we say that the formula is closed. The rationale behind these formulae is similar to that of the -calculus, where symbols like X. signify the least and X. represent the maximum fix-point formula, respectively. The rationale behind _a 1_2 is that there exists an execution of action a such that 2 must hold after the completion of this a, while 1 must hold after the beginning of this a. The formula specifies that there is a certain action "a" that may be initiated and stopped instantly. It is important to remember that if more executions occurred while a was being executed, 2 only needs to hold shortly after a's termination. The rationale behind [a 1]2 is that if an action an is begun that may be ended instantly, then either 1 holds after the beginning of this an or 2 holds immediately after the end of this a. A more intuitive definition would read something like "after every start 1 has to hold and immediately after every termination 2 has to hold." We choose for the stated interpretation of the box formula since, in this reading, the box formula produces the diamond formula's dual. Using our understanding, the above formula may be written as ([a1] false) ([afalse] 2). Since negation may be characterized by a duality operator, which will be shown later, we avoid introducing an explicit negation formula.

Logic's Semantics

Let's assume that the transition system T = (S,Act,) relies on a stack. Here we define the power set of S, denoted by P(S), and the semantics [[]]T: F (VarL P(S)) P(S).

$$[\![false]\!]^S_T = \emptyset \qquad [\![true]\!]^S_T = S \qquad [\![X]\!]^S_T = \varsigma(X)$$

$$[\![\langle a - \phi_1\rangle\phi_2]\!]^S_T = \{s \in S \mid \exists s' \in [\![\phi_1]\!]^S_T, s'' \in [\![\phi_2]\!]^S_T : s \xrightarrow{a^+} s' \xrightarrow{a^-} s''\}$$

$$[\![[a - \phi_1]\phi_2]\!]^S_T = \{s \in S \mid \forall s', s'' \in S : (s \xrightarrow{a^+} s' \wedge s' \xrightarrow{a^-} s'') \Rightarrow (s' \in [\![\phi_1]\!]^S_T \vee s'' \in [\![\phi_2]\!]^S_T)\}$$

$$[\![\phi_1 \wedge \phi_2]\!]^S_T = [\![\phi_1]\!]^S_T \cap [\![\phi_2]\!]^S_T \qquad [\![\phi_1 \vee \phi_2]\!]^S_T = [\![\phi_1]\!]^S_T \cup [\![\phi_2]\!]^S_T$$

$$[\![\mu X.\phi]\!]^S_T = \bigcap \{M \in \mathcal{P}(S) \mid [\![\phi]\!]^{\varsigma[X \leftarrow M]}_T \subseteq M\}$$

$$[\![\nu X.\phi]\!]^S_T = \bigcup \{M \in \mathcal{P}(S) \mid [\![\phi]\!]^{\varsigma[X \leftarrow M]}_T \supseteq M\},$$

where [X _ M] represents the function which is equal to everywhere save X, where it is equal to M. If for some, there exists a state s S such that s |=, then s represents a closed IB-formula. Throughout the remainder of the article, the notation (S,Act,) will refer to a transition stack. In addition, if it is obvious from the surrounding text, the index T is omitted from [[]]T.

Take the IBL-formula = _a_b truetruetrue as an example. Therefore, the stack-based transition system shown on the left side of Figure 2 fulfills, but the stack-based transition system depicted on the right side of Figure 2 does not. In

other words, an IB-formula can tell the difference between a_b and a.b + b.a. That's why it's fair to call the IBL logic a genuine concurrent one.
Step-bisimilar processes, a_b and (a_b) + a.b, may be differentiated from one another using the IB-formula _a [b false].false_true.
Note that the diamond (and box) operator's meanings are not fixed; for instance, take

$$\{s \in S \mid \exists s' \in [\![\phi_1]\!]^{\varsigma} : s \xrightarrow{a'} s' \wedge \forall s'' \in S :$$

$s' \xrightarrow{a_1^-} s'' \Rightarrow s'' \in [\![\phi_2]\!]^{\varsigma}\}$ as the interpretation of _a − φ1_φ2. Nevertheless, this definition is equivalent to the presented one if atomic sensitive and action termination deterministic stack-based transition systems are considered. In the following, we present the duality operator, which models negation.

Definition: The duality operator D: F → F is inductively defined as follows:

$$\mathcal{D}(false) = true \qquad \mathcal{D}(true) = false \qquad \mathcal{D}(X) = X$$
$$\mathcal{D}(\langle a - \phi_1\rangle\phi_2) = [a - \mathcal{D}(\phi_1)]\mathcal{D}(\phi_2) \qquad \mathcal{D}([a - \phi_1]\phi_2) = \langle a - \mathcal{D}(\phi_1)\rangle\mathcal{D}(\phi_2)$$
$$\mathcal{D}(\phi_1 \wedge \phi_2) = \mathcal{D}(\phi_1) \vee \mathcal{D}(\phi_2) \qquad \mathcal{D}(\phi_1 \vee \phi_2) = \mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)$$
$$\mathcal{D}(\mu X.\phi) = \nu X.\mathcal{D}(\phi) \qquad \mathcal{D}(\nu X.\phi) = \mu X.\mathcal{D}(\phi)$$

Proposition: The duality operator corresponds to negation, i.e., for any stack-based transition system T = (S, Act, −→) and for any IB-formula φ we have S \ [[φ]] ς T = [[D (φ)]] S\ς T, where S \M = {s ∈ S | s /∈ M} and S \ ς denotes the function with (S \ ς) (X) = S \ ς(X).

Correspondence to the μ-Calculus

We describe how the μ-calculus can be embedded in the IB-logic. The syntax of the μ-calculus is similar to the IB-logic except that the first formulas in the
Diamond and box expressions are omitted. The semantics function {[ ]} ς of the μ-calculus is defined over a transition system (S, Act, _−→). Its interpretation is similar to the IB-logic interpretation except that {[_a]} ς = {s ∈ S | ∃ s_ ∈ {[ϕ]} ς: s a _−→ s_} and {[[a]ϕ]} ς = {s ∈ S | ∀ s_ ∈ S: (s a _−→ s_ ⇒ s_ ∈ [[ϕ]] ς)}. The transformation that maps μ-calculus formulas to IB-logic formulas is given as follows:

$$\Psi(false) = false \qquad \Psi(true) = true \qquad \Psi(X) = X$$
$$\Psi(\langle a\rangle\varphi) = \langle a - true\rangle\Psi(\varphi) \qquad \Psi([a]\varphi) = [a - false]\Psi(\varphi)$$
$$\Psi(\varphi_1 \wedge \varphi_2) = \Psi(\varphi_1) \wedge \Psi(\varphi_2) \qquad \Psi(\varphi_1 \vee \varphi_2) = \Psi(\varphi_1) \vee \Psi(\varphi_2)$$
$$\Psi(\mu X.\varphi) = \mu X.\Psi(\varphi) \qquad \Psi(\nu X.\varphi) = \nu X.\Psi(\varphi)$$

Theorem: The IB logic encodes the -calculus through. A more accurate formula is [] e T = [[()]] T, where _ T indicates the un-splitting transition system of T, for any stack-based transition system T = (S, Act,) and any -calculus formula.

The IB-logic is more expressive than the -calculus with Act labels, as shown by Proposition 3.6 and the fact that a_b cannot be differentiated by the -calculus. Embedding the IB-logic into the -calculus with L-labeled terms is shown below.

$$\Phi(false) = false \qquad \Phi(true) = true \qquad \Phi(X) = X$$
$$\Phi(\langle a - \phi_1\rangle\phi_2) = \langle a^+\rangle(\Phi(\phi_1) \wedge \langle a_1^-\rangle\Phi(\phi_2)) \qquad \Phi(\phi_1 \wedge \phi_2) = \Phi(\phi_1) \wedge \Phi(\phi_2)$$
$$\Phi([a - \phi_1]\phi_2) = [a^+](\Phi(\phi_1) \vee [a_1^-]\Phi(\phi_2)) \qquad \Phi(\phi_1 \vee \phi_2) = \Phi(\phi_1) \vee \Phi(\phi_2)$$
$$\Phi(\mu X.\phi) = \mu X.\Phi(\phi) \qquad \Phi(\nu X.\phi) = \nu X.\Phi(\phi)$$

The IB-logic, using labels from L through, is represented in the -calculus, it is proposed. To be more specific, [[]] (S, Act,) = [()] (S, L,) holds for any stack-based transition system (S, Act,) and any IB-formula. Proposition 3.7 states that the IB-logic may be extended to all known findings of the -calculus, including the decidability of finite state systems. In addition, the model-checking instruments already developed for the -calculus may be used to the IB-logic. As mentioned in Remark 2.6, the -calculus with labels from L is not a very practical specification language.

A Descriptive Analysis of IB-Logic

IB-Bisimulation

In this part, we provide a bisimulation method for describing the IB-logic-based equivalence.

Definition: [IB-Dissimilarity] in this example, we'll use a stack-based transition mechanism (S, Act,). A symmetric relation R S S satisfying for any (s1, s_ 1) R and a Act yields an IB-bisimulation of this stack-based transition system.

$$\forall s_2, s_3 : (s_1 \xrightarrow{a^+} s_2 \wedge s_2 \xrightarrow{a^-} s_3) \Rightarrow \big(\exists s_2', s_3' : s_1' \xrightarrow{a^+} s_2' \wedge (s_2, s_2') \in R \wedge s_2' \xrightarrow{a^-} s_3' \wedge (s_3, s_3') \in R\big).$$

If there exists an IB-bisimulation R such that (s, s_) R, then the two items s, s_ S are IB-bisimilar (or IB-equivalent), represented as s IB s_.

Lemma:The ST-equivalence implies the IB-equivalence, or ST IB.

To prove this, suppose that s, s_ S.

• For any closed IB-formulas, if s IB s_, then s |= s_ |=.

If (S,Act,) is finitely branching, then s IB s_, and vice versa for all closed IB-formulas.

Corollary:Let's assume that (S,Act,) is a transition stack and that s, s_ S are ST-bisimilar. Then, s |= s_ |= holds for all closed IB-formulas. Proof. The IB equivalence of s and s_ is shown by Lemma 4.2. Everything else follows directly from Theorem 4.3. Subsequently, it will be demonstrated that the converse of Corollary 4.4, namely, "Do the IB-formulas characterize the ST-equivalence?" does not hold. However, in Section 4.3 we demonstrate that for an appropriate subset of stack-based transition systems, the converse of Corollary 4.4 is true.

Assuming IB-equivalence does not automatically imply Step-equivalence

However, if universal stack-based transition systems are permitted, the ST-equivalence is not a necessary consequence of the IB-equivalence. This happens because the IB-semantics is unable to recognize ongoing processes. Take the transition system (s, s_, Act, (s_, a), s_) as an example; in this case, state s is unable to carry out any actions, whereas state s_ may carry out just action a+ and evolve back to s_. Then, s and s_ are comparable in IB but not in ST. This is not a concern since our focus is only on contrasting situations in which no actions are being performed. The IB-equivalence can only take into account the beginning of activities that may end instantly, therefore the categorization fails here as well. To make this new counterexample, we need just swap out a with a+ in the preceding one: (s, s_, Act, (s_, a+, s_)). On the other hand, s and s_ are comparable in IB but not in ST.

Because the IB-equivalence cannot identify disruption or stoppage of active activities, the characterisation also fails for stack-based transition systems that are atomically sensitive. Example: in Figure 3's stack-based transition system, the initiation of action b interrupts the progress of action a+ that was already in progress. This transition system's

states are identical to those of the transition system obtained by solving for a_b, as seen in Figure 2. However, they are not comparable to ST. What about, however, states that map to processes in our process algebra and whose corresponding transition systems, based on stacks, are, say, interrupt-free? That situation is particularly difficult to characterize. Also, unlike STequivalence, which does necessitate step-equivalence, IB-equivalence does not. Think about the idioms

$$\bar{B}_1 = (((a.b.c + d.e)\|_{\{c,d,e\}}(b.d.e + c))\ \|_{\{b\}} b)\ [d \mapsto a][e \mapsto c]$$
$$\bar{B}_2 = (a.(b + c)\|_{\{c\}} b.(a.c + c))\|_{\{a,b\}}(a\|_{\emptyset} b)$$

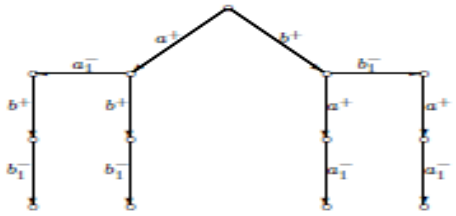

Figure 3: Disruption in a Stack-Based Transition System It is thus clear that B1 and B1 + B2 are IB-equivalent:

How the B2 execution may be matched is the sole non-obvious situation. Let's pretend B2 does anything, say a. Following the initialization of a, the processes (1) (a.(b + c)_cb.(a.c + c))_a,b(a_b) and (2) ((b + c)_cb.(a.c + c))_a,b(0_b) must be matched. (1) Is IB-equivalent to (0_cb? (a.c + c))_a, b (0_b), as IB-equivalence cannot see the expression underlying an active action.

As a result of the synchronization mechanism, both (1) and (2) may be written as "IB-equivalents" of b and b.c + c, respectively. It is easy to observe, however, that the initialization (and that the initialization and termination) of action an in B1 also provide an expression that is IB-equivalent to b (respectively b.c + c). In addition, a process IB-equivalent to an is produced when action b is initiated in B2, and a process IB-equivalent to a + a.c is produced when action b is terminated. In addition, a and _ (a + d.e)_c,d,ed.e _ [d _ a][e _ c], which is IB-equivalent to a + a.c, are produced by the beginning and ending of b in B1. Therefore, IB equivalence holds between B1 and B1 + B2.

However, it is maintained that B1 and B1+ B2 are not step equivalent: 0 is the result of running "a, b" through "B1." The process step equivalent of c is achieved by executing a, b in B1. In the aforementioned counterexample, the ability to synchronize activities is crucial. For non-synchronized expressions in finite process algebra, we show that the IB-equivalence and the ST-equivalence coincide. In the next part, we'll take care of it.

For a non-synchronized finite process algebra, IB describes ST.

Where a Act, the BNF-grammar for the expressions EXPfp of a synchronization-free finite process algebra looks like this: P::= 0 | a.P | P + P | P_P. By inserting them into EXP from Section 2.1, where _ is read as _, we can easily determine their meaning.

We find that the IB-logic defines the STequivalence for this process algebra:

Assume that P, P_ EXPfp. Thus, if P IB P_, then P ST P_.

As a corollary, we have P, P_ EXPfp. This means that P ST P_ iff s |= s_ |= holds for any closed IBformula.

Similar Research

To our knowledge, there does not exist a really concurrent logic that relies on the partitioning of tasks. Consideration of causality and concurrency is included into logics defined over partial orders. The many interleavings of the same partly ordered execution are all treated as equivalent by them. Partial order reductions [18] take use of this characteristic to condense state spaces. One such logic is the Temporal Logic of Concurrency (TLC) [1], which may be interpreted across causal structures. There are other temporal logics, such as local logics, interpreted at the events of a trace [3, 4, 6], or global logics, interpreted at its cuts [24, 25], for which it has been proven that Mazurkiewicz's traces are expressively complete. There are additional methods, such as [16,20], that interpret logics on event structures. The ability to directly describe characteristics concerning causality and concurrency is another another motivation for developing logics on partial orders. We introduce logics that are interpreted over trace systems (or runs), such as ISTL, which is interpreted over partly ordered runs of global states (see, for example, [13,17]). Over the global states of trace systems, the logic CTL is extended by a past operator, CTLP, in [19].

Finally, Future Plans

Here, we introduce IB-logic, an extension of the -calculus capable of specifying certain genuine concurrent features. Its semantics are modeled after the ST-semantics, which in turn are based on stack-based transition systems. By incorporating the IB-logic into the -calculus, which makes use of a start/termination-action label set, tool support for the IB-logic may be obtained. Using IB-bisimulation, we characterize the logical equivalence established by IB-logic. For processes derived from a synchronization-free finite process algebra, we demonstrated that the ST-bisimulation equivalence is characterized by the IB-logic. The unique expressiveness of the IB-logic and step execution was also shown.

To further understand the differences between IB- and ST-semantics in the future, researchers will look at questions like whether or not IB- and ST-bisimulation agree on expressions in generic process algebra that do not need synchronization. The question of whether the IB-logic may benefit from more powerful tools than only the transition into the -calculus is equally intriguing. The most exciting part of the ST-bisimulation project will be developing a clear logic to describe the simulation.

Acknowledgement

Wojciech Penczek's insightful criticism of similar works is well appreciated. In addition, I appreciate the contributions of Willem-Paul de Roever and Mila Majster-Cederbaum.

References

According to [1] R. Alur, D. Peled, and W. Penczek. Verification of causality characteristics in models. 10th IEEE Symposium on Logic in Computing, pages 90–100, Proceedings. 1995, IEEE Computer Society Press. References: [2] M. Bravetti and R. Gorrieri. For a process algebra supporting recursion and action refinement, deciding on and axiomatizing a weak ST bisimulation is a necessary step. ACM Transactions on Computational Logic, Volume 3:2002, Issue 3 (ACMTCL). Diekert, Victor, and Gastin, Paul. For Mazurkiewicz traces, pure future local temporal logics provide a full expressive form. Submission to LNCS. As cited by V. Diekert and P. Gastin [4]. Mazurkiewicz traces have reached expressive completion in LTL. 2003, 64:396-418, Journal of Computer and System Sciences. According to [5] H. Fecher and M. Majster-Cederbaum. Making choices based on outcomes and honing focus on what has to be done at the last minute. REFINE 2002, volume 70 of Electronic

Notes in Theoretical Computer Science, editors J. Derrick, E. Boiten, J. Woodcock, and J. von Wright. Publication year: 2002 by Elsevier Science. P. Gastin; Mukund; Kumar; N. Kumar(6). Mazurkiewicz traces may be expressed fully in local LTL using past constants. In LNCS, volume 2747, pages 429-438, MFCS'03. It was published by Springer-Verlag in 2003. Reference: [7] R. v. Glabbeek and U. Goltz. Improvements made to the concepts of action and equivalence in concurrent systems. 2001's Acta Informatica:37:229-327. According to R. v. Glabbeek and F. Vaandrager [8]. Models of concurrency in the form of Petri nets for algebraic approaches. Parallel Architectures and Languages Europe (Volume II), edited by J. de Bakker, A. Nijman, and P. Treleaven, appears in volume 259 of LNCS, pages 224-242. 1987, Springer-Verlag. Reference: [9] R. Gorrieri and A. Rensink. Modification of action. The article may be found on pages 1047-1147 in the Handbook of Process Algebra, edited by J. A. Bergstra, A. Ponse, and S. A. Smolka. Publication year: 2001, North-Holland. The Tenth M. Hennessy. Finitely occurring concurrent processes are axiomatized. SIAM Journal of Computing, Volume 17 Issue 5 (1988), Pages 997–1017. M. Hennessy and R. Milner [11]. Nondeterminism and concurrency rules in algebra. As published in the Association for Computing Machinery Journal, Volume 32, Issue 1, Pages 137–161. Communications Sequential Processes, by C. A. R. Hoare [12]. Computer Science: An International Series. Originally published in 1985 by Prentice Hall. Source: [13] S. Katz and D. Peled. Logic with sets and timestamps interleaved. TCST 75:263–287 (1990) Theoretical Computer Science.

(See reference 14) D. Kozen. Theoretical findings in propositional -calculus. 27:333–354, 1983, Theoretical Computer Science. R. Milner, "Communication and Concurrent Processing," 15. Computer Science: An International Series. Book published by Prentice Hall in 1989. Mukund and Thiagarajan [16]. Characterization of well-branching event structures from a logical perspective. Theory of Computing Systems, Volume 96, Issue 35-72, 1992. Partial order properties: a proof by D. Peled [17]. Journal of the Foundations of Computing, 126(1):143-182, 1994. Ten years of partial order reductions, by D. Peled [18]. published in CAV'98, pages 17-28 of LNCS volume 1427. Dated 1998 by Springer-Verlag.

W. Penczek, page 19. Automated verification of time-based logics for trace systems. Foundations of Computing Science, an International Journal, 4:31-67 (1993).

Using model checking for a certain category of event structures, [20] by W. Penczek. Tools and Algorithms for the Construction and Analysis of Systems, edited by E. Brinksma, appears in volume 1217 of the Lecture Notes in Computer Science series, pages 145-164. 1998 Springer-Verlag. Some Equivalence Notions for Concurrent Systems [21], by L. Pomello. In Summing Up. Advances in Petri Nets 1985, edited by G. Rozenberg, appears in LNCS 222, pages 381–400.

Publishing house: Springer-Verlag, 1986. Pratt, V., "Modeling Concurrent Processing Using Partial Orders" (22). Pages 33–71 of 1986's issue of the International Journal of Parallel Programming. Petri Nets: An Introduction, by W. Reisig (23). Publications in Theoretical Computer Science from the EATCS. published by Springer-Verlag in 1985. Specifically, P. S. Thiagarajan and I. Walukiewicz [24]. For Mazurkiewicz traces, we provide a fully expressive linear time temporal logic. 230-249 in 2002's issue of Information and Computation. Complexity of LTrL setups [25], I. Walukiewicz. Pages 140-151 of ICALP'98 in volume 1443 of LNCS, edited by K. Larsen, S. Skyum, and G. Winskel. Dated 1998 by Springer-Verlag. [26] "Event Structures" by G. Winskel. Petri Nets: Applications and Relation to Other Models of Concurrency, Part II, Advances in Petri Nets 1986, volume 255 of LNCS, editors W. Brauer, W. Reisig, and G. Rozenberg, pages 325-392. 1987, Springer-Verlag.