Summary of Various Topics in Type Sytem, Linear-Time Temporal Logic and Model Checking

Zhiqiang Ren

Boston University

Table of Contents

1 Summary of "Dependent ML: An Approach to Practical Programming with Dependent Types"[22]

This paper presents an approach to applying dependent types in practical programming. Taking such approach, users write programs in a functional programming language (Dependent ML), which allows for specification and inference of more precise type information than other common programming languages. Such precise type information can then be used to facilitate program error detection and compiler optimization.

In summary, the whole system involves four languages including λ_{pat}, $\mathcal{L}$, $\lambda_{pat}^{\Pi,\Sigma}$ and DML_0. DML_0 is the outmost part of the system, in which users write program. $\lambda_{pat}^{\Pi,\Sigma}$ is an intermediate language with complex syntax, from which precise type information can be extracted. Language $\mathcal{L}$ is for building type index used by both DML_0 and $\lambda_{pat}^{\Pi,\Sigma}$. Language λ_{pat} serves as a bridge between DML_0 and $\lambda_{pat}^{\Pi,\Sigma}$, representing the dynamic semantics of expressions in the later two languages. With the aforementioned settings, the paper gives out elaboration rules to map expression in DML_0 into expression in $\lambda_{pat}^{\Pi,\Sigma}$ which conveys more precise types, while maintaining the dynamic semantics. A detailed illustration of each of these four languages and their relations goes as follows.

Firstly, λ_{pat} is a simply typed language, which extends the simply typed λ-calculus with recursion and general pattern matching. The dynamic semantics of expression in λ_{pat} is assigned through the use of evaluation context and redex. The proof of type soundness of λ_{pat} is given to show the constraint on the result of the evaluation of a well-typed expression. A sensible *operational equivalence* relation between two expressions in λ_{pat} is given based on the concept of general context. Later a reflexive and transitive relation $\leq_{dyn}$ on expressions in λ_{pat} is given which is equivalent to the *operational equivalence* if the right-hand expression of the relation are well-typed. Another way to put it, two expressions in λ_{pat} with relation $\leq_{dyn}$ are operational equivalent if the expression on the right-hand side of the $\leq_{dyn}$ relation is well-typed.

Secondly, the generic type index language $\mathcal{L}$ introduced by the paper is a pure simply typed language, in which constraint relations can be properly defined. Types in $\mathcal{L}$ are called sorts for clarity while expressions in $\mathcal{L}$ is called terms, which are used to build types in $\lambda_{pat}^{\Pi,\Sigma}$.

Thirdly, the explicitly typed language $\lambda_{pat}^{\Pi,\Sigma}$ is an extension of λ_{pat} with both universal and existential dependent types. Similar to that in λ_{pat}, dynamic semantics of expression in $\lambda_{pat}^{\Pi,\Sigma}$ is given based on evaluation context and redex. Formal proof is provided for the type soundness of $\lambda_{pat}^{\Pi,\Sigma}$. The paper defines an eraser operation, which maps expressions and types in $\lambda_{pat}^{\Pi,\Sigma}$ into the their counterparts in λ_{pat}. It also illustrates the relation between the dynamic semantics of a well-typed expression in $\lambda_{pat}^{\Pi,\Sigma}$ and the dynamic semantics of its erasure in λ_{pat}, which makes it reasonable to view these two dynamic semantics as the same.

Finally, the paper presents an external language DML_0 together with a mapping from DML_0 to the internal language language $\lambda_{pat}^{\Pi,\Sigma}$. The process of such mapping from an expression in DML_0 to an expression in $\lambda_{pat}^{\Pi,\Sigma}$ along with its

type is called elaboration. Elaboration rules are given to illustrate this process. Erasure is defined to map an expression in DML_0 into λ_{pat}. The dynamic semantics of the later can be viewed as the dynamic semantics of the former. The paper proves that if an expression in DML_0 can be elaborated into an expression in $\lambda_{pat}^{\Pi,\Sigma}$, then their erasures in λ_{pat} are operational equivalent. Simply put, by the elaboration, we get an an expression with more precise type while maintaining the same dynamic semantics.

The design of the whole system introduced in the paper benefits both builders of programming language and programmers using the language in the following aspects.

The form of dependent types studied in this paper is called a restricted form of dependent types, which is substantially different from the usual form of dependent types in Martin-Löf's development of constructive type theory. One character of such restricted form of dependent types is that the type index language $\mathcal{L}$ used in such dependent type system doesn't contain any side effect (e.g. recursion), which makes it practical for programmers to reason about the types they want to use. Also it's practical to build the type checker which is able to compare types within such system.

Though $\lambda_{pat}^{\Pi,\Sigma}$ provides a way to exploit the dependent type system, which can facilitate program error detection and compiler optimization, programmer may be quickly overwhelmed with the demanding work to write a well-typed expression in $\lambda_{pat}^{\Pi,\Sigma}$ as well as the amount of effort for manual type annotation of the program. The introduction of DML_0 offers a solution to such problem. In DML_0 programmer can write the program with relatively simpler syntax and only need to provide a reasonably small amount of type annotation in practical programming. The compiler can then translate the program into $\lambda_{pat}^{\Pi,\Sigma}$ while maintaining the dynamic semantics. The illustration of such translation process and the formal proof of the soundness of such translation constitute the main contribution of the paper.

Going still further, the paper also extends $\lambda_{pat}^{\Pi,\Sigma}$ with parametric polymorphism, exception and references, attesting the adaptability and practicality of such approach to supporting the use of dependent types in the presence of realistic programming features.

2 Summary of "Linear types can change the world!" [20]

In traditional functional programming languages, objects can be created explicitly and discarded implicitly. And as such, the "update" of an object is conveyed by creating a new object and discarding the old. In such programming paradigm, objects of the same value are treated as of the same identity, which doesn't actually match the computation model provided by real physical machines in which objects are distinguished by their innate identities. On one hand, such programming paradigm releases the programmers from the heavy duty of tracking all the objects within the computation system, which is mostly cumbersome and error-prone. On the other hand, it also deprives the programmers of the power to manipulate objects directly and the general mechanism for tracking "real" objects in the system (e.g. reference counting and garbage collection) often leads to an inefficient implementation.

This paper presents a type system, in which types are divided into two families. Objects of linear type have exactly one reference to them, and so require no garbage collection. Objects of nonlinear type may have many pointers to them, or none, and do require garbage collection. Such system not only offers the explicit manipulation of objects with guaratantee of correctness but also retains the same ability of implicit objects management as normal functional programming language.

Linear type system applied in the paper is based on the linear logic of J.-Y. Girard. The key idea in such system is that each assumption in the context of a type judgement must be used exactly once. It cannot be duplicated or discarded. Another way to put it, the typing rule of the system guarantees that a well typed program can have only one reference to any object of linear type and cannot discard such object implicitly. With such constraints, it's sensible in such linear system that each operation that allocates an object is paired with exactly one operation that deallocates that object. The allocation operations create objects to which a single reference exists. This reference may never be duplicated or discarded, so the deallocation operations act on objects to which they hold the sole reference. In particular, after an application the storage occupied by the function (closure) may be reclaimed, which leads to the fact that a linear function (closure) in such system can and must be applied only once.

The non-linear part of the type system presented in the paper is similar to that of the simply-typed λ-calculus with the support of data type. Complex types comprising of both linear and non-linear types can be constructed while the type system guarantees that non-linear data structure must not contain any linear components. This is sensible that the non-linear object, which can be duplicated, can cause the linear object embedded inside to be accessed once for each duplication.

Going still further, the system provides a special "let" construct in which multiple references to an object is allowed so long as the object is being "read" as indicated explicitly by the programmers. The type system and the evaluation rule guarantee that all those multiple references would be relinquished till only one left for the coming "write" access.

Though the type system used in this paper is monomorphic, it is not difficult to extend it to a polymorphic language with explicit type applications. Besides, the

language provided by the paper is in essence the traditional λ-calculus extended by pattern matching and data type declaration, which makes it a good candidate to be incorporated into existing languages with more advanced types.

3 Summary of "Safe Programming with Pointers through Stateful Views" [25]

ATS is a programming language with a type system rooted in a framework *Applied Type System*, which is derived from *DML*[1]. It supports a restricted form of dependent types, by which sophisticated structures can be encoded with more confidence of correctness. This paper illustrates another novel and desirable feature of it, the support for safe programming with pointers through a notion of *stateful views*.

The linear type system in section 2 provides a solution to control the access to individual objects within the system. The paper takes advantage of intuitionistic linear logic from another perspective. Instead of introducing linear types for concrete objects in memory, it introduces linear "views" for abstract "proof", which depicts not only the linearity of the objects but also the locations of them. Such feature is made possible by the introduction of a special *sort addr* in the statics of ATS, which represents the address within the memory. Through dependent types, a view is dependent on multiple addresses within the memory, which in turn states the layout of the memory. Therefore, such method is more flexible than that in section 2.

The type system of ATS is divided into two families: nonlinear types and views. From the perspective of isomorphism, a view corresponds to a logic proposition, its instance is an abstract object corresponding to the proof of the proposition. Hence, we just call the instance of a *view* a *proof*. Such proofs are only used at compile-time for performing type-checking and they are neither needed nor available at run-time.

Formally, two forms of constraints including $\Sigma; \overline{P} \models P$ (persistent) and $\Sigma; \overline{P}; \overline{V} \models V$ (ephemeral) are formalized in ATS, which are needed to define type equality. The rules for proving such constraints are given based on intuitionistic logic and intuitionistic linear logic. Besides, for each user-defined view constructor, a couple of constructor-related rules are introduced. With such rules as well as those rules related to type judgement and subtype relation, reasoning about the changes of the states of the memory is turned into a type checking issue.

To sum up, the usage of *stateful views* provides a light-weighted formal method for reasoning about pointers and memory. Though extra effort is needed to encode such reasoning explicitly in program, the manifist of such reasoning makes it practical to apply the method to general sophisticated programs where fully-automatic verification is infeasible.

[1] Please refer to section 1 for details.

4 Summary of "Proofs and Types" [5]

4.1 Normalization

A term in λ-calculus is called a normal form if it contains no redex. A normal form is the termination of the reduction path. Starting from a term, there are many reduction paths which may or may not lead to a normal form. Typed λ-calculus behaves well computationally in the sense that each term can be reduced to a unique normal form along any reduction path. Such property guarantees that the program written in typed λ-calculus shall terminate no matter which evaluation strategy is chosen.

The proof of such property consists of two parts: the uniqueness and existence of the normal form.

The uniqueness comes from that β reduction is *Church-Rosser*, which means that if $t \rightsquigarrow u, v$, then $\exists w.u, v \rightsquigarrow w$. Its proof can be found in chapter 3.2 of [1].

The existence of the normal form can be interpreted in two ways. One way is that there exists a reduction path leading to the normal form. The other is that all the reduction paths lead to the normal form. The former property is called weak normalization, while the later is called strong normalization.

The proof of weak normalization is kind of construction proof in the sense that it provides the algorithm for choosing at each step along the reduction path an appropriate redex which leads us to the normal form. Since the algorithm is defined recursively, an appropriate index has to be chosen to prove that the algorithm terminates, which in this case is the degree of redex. At each step, we locate in the term the redex with the maximal degree n and we also require that the subterm of the redex we pick cannot contain redex of the same degree. It is then proved that after the reduction of the redex we pick, no new redex with degree $\geq n$ would be created and also the number of redex with degree n is decreased by 1. It's easy to reason that such algorithm would lead us to a term without redex (normal form).

The proof theoretic technique for the strong normalization of simply typed λ-calculus can be used to prove the strong normalization of System T and System F, which include a type of integers and hence codes Peano Arithmetic. Therefore, we have to use a strong induction hypothesis, for which an abstract notion called *reducibility* is introduced. For simply typed λ-calculus, a set RED_T ("reducible terms for type T") is defined by induction on the *type* T.

1. For t of atomic type T, t is reducible if it is strongly normalisable.
2. For t of the type $U \to V$, t is reducible if, for all reducible u of type U, tu is reducible of type V.

The following three properties of reducibility are proved together by induction.

1. (CR 1) If $t \in \mathrm{RED}_T$, then t is strongly normalisable.
2. (CR 2, Forward Property) If $t \in \mathrm{RED}_T$, and $t \rightsquigarrow t'$, then $t' \in \mathrm{RED}_T$.
3. (CR 3, Backward Property) If t is neutral (substituting t for any free variable in any term p doesn't create extra redex besides those which already exist in t and p), and whenever we convert a redex of t we obtain a term $t' \in \mathrm{RED}_T$, then $t \in \mathrm{RED}_T$.

Finally we prove that all terms are reducible, which leads to the fact that all terms are strongly normalisable. To prove this by induction, we strengthen the induction hypothesis to handle the case of abstraction and come up with the following proposition, from which we can get the ultimate goal by putting $u_i = x_i$. Let t be any term, and suppose all the free variables of t are among $x_1, \ldots, x_n$ of types $U_1, \ldots, U_n$. If $u_1, \ldots, u_n$ are reducible terms for types $U_1, \ldots, U_n$ then $t[u_1/x_1, \ldots, u_n/x_n]$ is reducible.

4.2 System F

System F is a typed λ-calculus that differs from the simply typed λ-calculus by the introduction of a mechanism of universal quantification over types. Formally, system F contains the following scheme for forming terms: if v is a term of type V, then we can form $\Lambda X.v$ of type $\Pi X.V$, as long as the variable X is not free in the type of any free variable of v. As such, the types in system F correspond to propositions quantified at the second order.

System F allows recursive constructions to be embedded in a natural manner. For a structure Θ described by a finite number of constructors $f_1, \ldots, f_n$ respectively of type $S_1, \ldots, S_n$, its type in system F is $T = \Pi X.S_1 \to S_2 \to \cdots \to S_n \to X$. Recursivity is manifested by the fact that X may appear within one of the types S_i. The type S_i must itself be of the particular form $S_i = T_1^i \to T_2^i \to \ldots T_{k_i}^i \to X$. Then the constructor f_i can be encoded in system F as follows
$f_i = \lambda x_1^{T_1^i} \ldots . \lambda x_{k_i}^{T_{k_i}^i} . \Lambda X . \lambda y_1^{S_1} \ldots . \lambda y_n^{S_n} . y_i t_1 \ldots t_{k_i}$ where

- $t_j = x_j$, X doesn't occur in T_j^i
- $t_j = x_j X y_1^{S_1} \ldots y_n^{S_n}$, if X occurs positively in T_j^i.

Based on such scheme, both simple types (e.g. booleans, product type, and sum type) and inductive types (e.g. integer, list, and tree) can be encoded in system F. The intuition behind such scheme goes as follows. Terms of inductive types are encoded as high order functions with "constructors functions" as the input arguments. For example, the integer type is encoded as follows
$Int \stackrel{def}{=} \Pi X.X \to (X \to X) \to X$
and the integer n is represented by
$\bar{n} = \Lambda X.\lambda x^X.\lambda y^{X \to X}.\underbrace{y(y(y \ldots (y}_{\text{n occurrences}} x) \ldots))$
These "constructors" are applied inductively based on the definition of the structure corresponding to the type. Such form makes it easy to define *iterator* on terms of induction types in system F. However it's quite difficult to define recursion, for which generally, a full iteration has to be taken starting from the most "atomic", then reconstructs a new term (usually a tuple) iteratively and uniformly. Starting from recursion, all kinds of common operations related to those inductive types can be defined in system F, though the encoding is unacceptably tedious with very low efficiency in evaluation.

Though it's not practical to program directly in system F due to low efficiency in both coding and execution, system F serves as a formalization of the notion

of parametric polymorphism in programming languages, and forms a theoretical basis for languages such as Haskell and ML.

5 Summary of "The Temporal Logic of Programs" [14]

This paper provides a unified approach to program verification, which applies to both sequential and parallel programs.

To verify a program, first we need to define what a program is. For that, this paper presents a general framework to support the modelling of both sequential and concurrent programs. Informally, a program can be viewed as multiprocessor computer with shared memory, in which each processor runs its statements independently. But each time only one processor can execute one statement. The scheduling choice of the next processor to be stepped is nondeterministic.

More formally, a dynamic discrete system including both sequential and concurrent program consists of $< S, R, S_0 >$ where

- S: the set of states the system may assume;
- R: the transition relation holding between a state and its possible successors, $R \subseteq S \times S$;
- S0: the initial state.

The state of a system is represented by $s =< \pi_1, \ldots, \pi_n; u >$. Each π_i may be considered as the value of the program counter for the i-th processor, while u is the shared data component.

In such model, the execution of statements on one processor can only influence the other processor though the usage of shared data. But the user can form up appropriate atomic statements based upon which program is built. Therefore its still feasible to model in the framework the execution of a concrete program in which interference between different phases of concurrent statements execution may exist.

Based on the aforementioned framework, this paper illustrates how to specify properties of systems and their development in time. First, relations on states $q(s)$, expressed in a suitable language as well as explicit time variables t are introduced. Then the properties of a program is formed as the development of the properties $q(s)$ in time t (a.k.a. time functional $H(t, q) \equiv q(s_t)$), which can be interpreted as at time instance t, the state of the system (denoted by s_t) satisfies the predicate $q(s)$.

Though any arbitrary complex time specification can be expressed in this way, the paper focuses on two major categories of them, invariance and eventuality (temporal implication). The former one is a statement with only one time variable which is universally quantified. It is in the form $\forall t.H(t, q)$. The later one is a statement of the form $\forall t_1.\exists t_2.(t_2 \geq t_1) \wedge (H(t_1, \varphi) \rightarrow H(t_2, \psi))$, and we use $\varphi \rightsquigarrow \psi$ to denote such form.

Three proof approaches (invariance, well founded set, and temporal reasoning) are presented in the paper for the reasoning of both invariance and eventuality properties. The basic idea of the third approach, focused by the paper, is that one derives simple time dependency relations (e.g. eventuality) directly from the system transition rules and then use combination rules, and general logic reasoning to derive more complex properties.

Focusing on establishing simple properties including invariance and eventuality, the paper chooses to find a minimal basis for temporal reasoning without

taking the brute force approach of installing an explicit real time clock variable. Based on such design principle, two formalizations of the temporal reasoning are presented in the paper. First one is an axiomatic system (ER) with the eventuality connective $\rightsquigarrow$. Without time variable t, the statement $\varphi \rightsquigarrow \psi$ is interpreted as that for all executions, if φ is valid at certain moment, then ψ would be valid as well later but eventually.

Though system ER is proved to be sound and complete, its very difficult to express natural intuitive arguments for the behavior of concurrent programs in (ER). Therefore, the paper presents a second formalization, which actually adopts a fragment of the tense logic K_b. Two basic tense operators, F and G are introduced. Denoting the present by n, they can be interpreted as follows:

- F(p): $\exists t.t \geq n \wedge H(t, p)$
- G(p): $\forall t.t \geq n \rightarrow H(t, p)$

The K_b fragment has the following properties. Firstly, since $G(p \rightarrow F(q))$ matches the notion of eventuality, the K_b fragment is at least as strong as (ER). Secondly, as a fragment of K_b, the system is proved to be complete. Finally, the K_b fragment is isomorphic to the modal logic system S_4 with G standing for $\Box$ and F standing for $\Diamond$. Therefore the K_b fragment presented in the paper is decidable.

A scheme of a proof in such logic system consists of two separate phases. In the first phase, one reasons about states, immediate successors and their properties to translate all the relevant properties of the program into basic tense-logic statements. This is done via the application of domain dependent axioms, which restricts the future to only those developments which are consistent with the transition mechanism of the system. The provision of such axioms in the logic system makes it distinguished from the pure tense logic K_b. In the second phase, one uses the pure logic rules in tense logic and manipulates those tense logical statements into the final result.

Going still further, the paper points out that the validity of any arbitrary tense formula on a finite state system is decidable and can be proved in an extension to the K_b fragment presented in the paper.

In summary, the K_b fragment presented in the paper can be viewed as a simplified version (with less operators) of Linear-Time Temporal Logic (LTL) illustrated in [7]. Users' natural intuitive reasoning of programs can be encoded in such systems in a formal and practical way. As a trade off, the expressive power of such systems is limited due to the lack of time variable. However, quite a lot of properties of time in practice can already be captured in such system, which leads to the growing application of LTL in the field of model checking.

The approach taken in the paper can be classified as endogenous approach, in which we immerse ourselves in a single program which we regard as the universe, and concentrate on possible developments within that universe. With such special universe, it's natural trying to find a semantical proof (by model checking), which seems to be a more practical and economic way than trying to find a syntactical proof. I think that's another reason why LTL is so widely used in the field of model checking.

However, to provide tools and guidance for constructing a correct system rather than just analyse an existing one, we have to take exogenous approaches,

which usually suggest a uniform formalism dealing with formulas with varying context (program segment). Due to the complexity of such approaches, no dominant system has been provided to fulfill such tasks, especially for the concurrent programs.

6 Summary of "The Theory and Practice of Concurrency"[16]

Communicating Sequential Process (CSP) is a formal language for describing interactions between different components within a system on the level of communication. It was introduced by Hoare in the late 1970s. And since then, it has been widely developed and applied in the specification and verification of reactive and concurrent systems in which synchronization and communication play a key role.

Components with independent inner states are modeled as *processes* in CSP. There is no shared information between the process and its environment. They can only communicate with each other through the synchronization of *events*. Such events are both atomic and instantaneous. The set of events that may be communicated by a process is said to comprise its alphabet Σ. If a process offers an event with which its environment agrees to synchronize, the event is performed. A finite sequence of events that a process may perform is call a *trace* of the process. For (semantic) convenience the alphabet of each process is extended to contain two further events: $\tau \notin \Sigma$ represents an internal event and $\surd \notin \Sigma$ represents termination.

CSP is equipped with a rich set of process operators as well as several atomic processes, based on which complicated processes can be defined recursively. For each process operator, CSP also provides a set of algebraic laws, by which we can reason about the equality of two processes. These laws are chosen in such manner that if two processes are "equal" according to the laws, their communicating behaviours are indistinguishable by the environment. CSP is thus categorized as a member of process algebras. The common process operators in CSP includes prefixing $a \rightarrow P$, external choice $P \,\Box\, Q$, internal choice $P \,\sqcap\, Q$, parallel composition $P \underset{A}{\parallel} Q$, hiding $P \setminus A$, and sequential composition $P; Q$. The process $a \rightarrow P$ offers its environment the opportunity to synchronize on event a in which case it then behaves like P. The process $P \,\Box\, Q$ offers its environment a choice between P and Q based on synchronization with their initial events respectively. The process $P \,\sqcap\, Q$ behaves like either P or Q but the choice is made internally, beyond environmental influence. The process $P; Q$ behaves like P and, if that terminates, then behaves like Q. The process $P \setminus A$ executes the events in the set A internally, without synchronization by its environment; they can be thought of as being replaced by τ events. The parallel composition $P \underset{A}{\parallel} Q$ requires P and Q to synchronize on each event $a \in A$, but performs other events of P or Q as determined by those processes. The atomic processes includes $STOP$, $SKIP$, DIV. $STOP$ models deadlock by offering no events. $SKIP$ models successful termination by offering only $\surd$. DIV models the process that does nothing but diverge, that is to enter into an infinite sequence of consecutive internal actions.

CSP has a range of semantic models (denotational semantics) including the traces models $\mathcal{T}$, the stable failures model $\mathcal{F}$, and the failures-divergences model $\mathcal{N}$. In the traces model of CSP, a process P is represented by $traces(P)$ which is the set of all its possible traces. Such semantics is useful in deciding questions of safety. Going still further, the failures-divergences model is useful in specifying

both safety and liveness properties. In such model, a process P is represented by two sets of behaviours:

- *divergences* are finite traces, each of which can lead the process to diverge, the net effect of which is that the process won't accept anything from the environment [2] ;
- $failures$ are pairs $(s, refusals)$ where s is a finite trace of the process and $refusals$ is a set of events it can refuse after s [3].

The failures-divergences model allows us to assert that a process must eventually accept certain event from a set that is offered to it since refusal and divergence are the two ways a process can avoid doing this, and we can specify in this model that neither of these can happen. The stable failures model can be viewed as a simplified version of failures-divergence model, only containing the information of failures. Though not as powerful as the later, it is sometimes advantageous to use such model since the calculations required to determine if a process diverges are very costly.

With the aforementioned semantic models, *refinement* relation between processes can be defined as subset relation. Process Q is a refinement of process P means that Q's behaviors are contained in those of P. Formally, for any of the three semantic models $\mathcal{M} \in \mathcal{T}, \mathcal{F}, \mathcal{N}$, we define such relation by $P \sqsubseteq_{\mathcal{M}} Q \iff \mathcal{M}[Q] \subseteq \mathcal{M}[P]$ where $\mathcal{M}[P]$ denotes the semantics of process P in semantic model $\mathcal{M}$. Also we have that for each operator $\bigoplus$ in the language, $P \bigoplus Q$ is a refinement of $P_0 \bigoplus Q_0$ as long as P is a refinement of P_0 and Q is a refinement is Q_0 (with obvious modifications for non-binary operators). In this way, CSP provides a theoretic background for refinement development from abstract specification to implementation.

From my perspective, it's difficult to understand the concepts of CSP without referring to its operational semantics as a tuition. The operational semantics of a process is a *labelled transition system* (LTS), which is a directed graph with a label on each edge. Each node can be viewed as a state of the process and the labels on the leaving edges of the node represent the actions the process can possibly take at that state. The set of possible labels is $\Sigma^{\tau,\surd} = \Sigma \cup \{\tau, \surd\}$. For each CSP operators, we have several rules to derive the LTS of the top-level process from the LTS' of its syntactic parts (sub-processes). The operational and denotational semantics of CSP are congruent in the sense that we can extract all three kinds of denotational semantics of a process from its LTS directly. Such congruence makes it possible to build model checkers for refinement checking of two processes practically.

[2] The precise definition of *divergences* should contain any finite sequence with prefix which can lead the process to diverge. Such definition offers a more concise theoretic treatment of refinement relation, yet matches our intuition that what a process can do after divergence is of no concern. But since such precise definition contains no more useful information about the behaviour of the process, most of time the simpler version of definition is used to illustrate the idea of *divergence*.

[3] Due to the same reason as for the *divergence*, precisely, we should use $failures_\perp$ instead of $failures$, whose definition is $failures_\perp(P) = failures(P) \cup \{(s, X) | s \in divergences(P)\}$.

All three refinements are supported by the automatic refinement checker FDR which proves or refutes assertions of the form $P \sqsubseteq_{\mathcal{M}} Q$. FDR inputs processes expressed in CSP_M, which is a standard for machine-readable CSP. CSP_M expresses CSP by a functional language, offering constructs such as *function*, *let* expressions and supporting pattern matching. It also provides a number of predefined data types, including booleans, integers, sequences and sets, and allows user-defined data types to certain extent. In short, with the support of CSP_M, the computation of sub-process objects (e.g. events and process parameters) can be expressed along with the CSP process. Besides FDR, some other tools such as ProB and PAT can do refinement checking as well. Furthermore, these tools can also do LTL model checking of CSP model encoded in languages specific to these tools.

7 Summary of "Model-Checking CSP"[15]

The denotational semantics of processes in CSP equips us with a mathematical foundation for defining a special relation between processes called "refinement" relation. As the co-worker of the inventor of CSP, the author of the paper describes his work of building a model-checker/refinement checker for CSP based on its operational semantics, experience of the application of such tool, and prospects about the improvement of it.

The standard model for CSP is the *failures/divergences* model. Such model ($\mathcal{N}_\Sigma$) over a given finite alphabet Σ of communication is the set of pairs of sets (F, D) satisfying certain "healthiness" properties [4] . The denotational semantics of a process P is then an element (F_P, D_P) of $\mathcal{N}_\Sigma$. Based on these, the refinement relation $P \sqsubseteq_{\mathcal{N}} Q$ [5] is defined by $F_Q \subseteq F_P \wedge D_Q \subseteq F_P$. Due to the congruence between the denotational semantics and operational semantics of a process. The decision problem of whether $P \sqsubseteq Q$ can be carried out by their operational semantics.

This paper only tackles the problem in which the operational semantics of the processes on both sides of the relation are of *finite state*. This simply means that, as the operational semantics is unfolded, the resulting LTS contains only finite nodes. Also the paper doesn't take into consideration the termination of process, which means that the LTS' being worked on are finite directed graphs where all edges are labelled with an action, either τ or visible. But it's not difficult to add the support of $\surd$ to the method developed in the paper.

The basic idea of refinement checking is that, given a certain trace s and a reachable state in the implementation process (right-hand of the refinement relation), there is a corresponding state in the specification process (left-hand of the refinement) reachable on the same trace and has the same behaviour.

Typically, the LTS arising from CSP descriptions contains a high degree of nondeterminism, in the sense that after any trace s of visible actions there may be many states of a system which the process might be in. This can happen both because of the existence of invisible actions and because of the branching that occurs when a node has two identically-labelled actions, whether visible or invisible. Any method for deciding refinement between these systems will have to keep track of all the states reachable at the specification side on a given trace s. The method provided in the paper accomplishes this by normalizing the specification process before doing the refinement checking. The normalization is processed in two stages. In first stage, multiple nodes in the LTS, which are reachable after certain trace, are merge into one node in the new LTS. And the new node conveys all the information of the original nodes, namingly all the $refusals$ sets of the original nodes. Also if one of the nodes being merged diverges in the original LTS, then the new node is marked diverging in the new LTS. Such strategy matches our idea that if certain trace can lead the process to diverge, then whatever after is of

[4] Please refer to section 6 for the precise definition of the failures/divergences model.

[5] For the purpose of clarity, we may use the notation of $\sqsubseteq$ instead since we only focus on the failures/divergences model in the paper.

no concern. Second stage is actually a bisimulation checking, in which bisimilar nodes are merged together.

After normalization, the refinement checking is conveyed by enumerating and checking all the possible pairs (v, w) of nodes satisfying the following condition: $\exists$ traces s, such that v is reachable after s in the normalized specification process and w is reachable after s in the implementation process. For the refinement checking in the failures/divergences model, the checking on each pair includes the compatibility of initial actions, divergence, and refusals. If we only check the initial actions of the nodes in the pair, then this method is degraded into a refinement checking in the traces model. Normally the enumeration of pairs is done by DFS (Depth First Search). If the checking on certain pair fails, then the whole refinement relation doesn't hold. The path leading to such pair serves as a counter-example. BFS (Breadth First Search) can then be applied to find the shortest path leading to an error.

A tool called FDR was built to do the refinement checking based on the method shown above with the specification and implementation processes encoded in the CSP_M language. Besides the syntax of CSP_M, FDR also puts some extra syntactic restrictions on the input processes to enforce finite state space. However process with infinite state space can still be created under these restrictions due to the use of recursion and infinite data types. It's the users' responsibility to treat such processes with care.

Though checking failures/divergence refinement of LTS is PSPACE-hard, fortunately "real" process definitions simply do not behave as badly as some pathological examples. But great effort has to be taken when building FDR due to the efficiency concern. The paper discusses the usage of hashmap and sorted list for implementing set in different parts of FDR and justifies such choices. Going still further, the paper discusses the possibilities to improve FDR to handle more practical usage. Three possibilities I think more applicable goes as follows. First one is to build certain logical inference tool based on the algebraic rules of CSP to alleviate the work of the model-checking. Second one is to compress the state-space so that we can prove results of the state space in blocks instead of expanding the state-spaces of processes fully and explicitly. The third one is to include symbolic representations of data in the LTS instead of expanding the LTS for each possible values. Such method is applicable to CSP processes in which data (at least most of the data) does not alter the control-flow of a process. Some work related to this topic has been done in [8].

8 Summary of "How to Make FDR Spin — LTL Model Checking of CSP by Refinement"[10]

In "classical" model checking, systems to be checked are encoded in the specification language used by the model checker. Extracted from these specifications, models of the systems are then checked against the expected correctness properties, which are encoded in certain formal language, by the model checker. Among all those formal languages, Linear-time Temporal Logic (LTL) allows the specification of temporal properties encountered in practice in a natural and succinct manner, which makes LTL model checking a common approach for the verification of hardware and software system.

The concept of refinement relation in Communicating Sequential Process (CSP) offers a new approach for model checking. In such method, system to be checked is specified as a process (implementation process) in CSP, while the correctness properties are also specified as a process (specification process) in CSP. The verification of correctness properties of the system is then turned into the question whether the implementation process (or process derived from it) is a refinement of the specification process, which can be decided by the refinement checker (e.g. FDR). This refinement-based approach suits itself very nicely to the stepwise development of systems. However there is no general way to generate the specification process and the appropriate derivation of the implementation process for any correctness properties, which makes such approach less usable in practice. To tackle this, the paper presents a new approach which is also based on refinement checking but can check the implementation process against the correctness properties specified in LTL directly and mechanically.

In "classical" LTL model checking, that a system satisfies an LTL formula ϕ means that all of its computations satisfy ϕ and a computation is defined as an infinite sequence of states, the transitions of which are possbile in the system. Similar concept is applied to CSP process in this paper. That a process satisfies an LTL formula ϕ means that the process cannot deadlock and all of its infinite traces satisfy ϕ. And an infinite trace of a process is an infinite sequence of events that the process can possibly communicate with. Given an infinite trace $\pi = \pi_0, \pi_1, \ldots$. We define π^i to be suffix of π starting from π_i. $\pi \models \phi$ (a trace π satisfies ϕ) is defined as follows:

- $\pi \not\models false$
- $\pi \models true$
- $\pi \models a$ iff $\pi_0 = a$
- $\pi \models \neg a$ iff $\pi_0 \neq a$
- $\pi \models \phi \wedge \psi$ iff $\pi \models \phi$ and $\phi \models \psi$
- $\pi \models \phi \vee \psi$ iff $\pi \models \phi$ or $\phi \models \psi$
- $\pi \models X\phi$ iff $\pi^1 \models \phi$
- $\pi \models \phi\ U\psi$ iff there exists a $k \geq 0$ such that $\pi^k \models \psi$ and $\pi^i \models \phi$ for all $0 \leq i < k$
- $\pi \models \phi\ R\ \psi$ iff for all $k \geq 0$ such that $\pi^k \models \neg\psi$ there exists an i, $0 \leq i < k$ such that $\pi^i \models \phi$

We denote by $|\phi|_\omega$ the set of infinite traces which satisfy the formula ϕ. It's difficult and sometimes impossible to come up with a process ($Spec_\phi$) which can generate the same set of infinite traces as $|\phi|_\omega$. Even if we find $Spec_\phi$, a traces refinement test $Spec_\phi \sqsubseteq_\mathcal{T} S$ is not adequate to model check $S \models \phi$ since refinement checking in FDR is based on *finite* traces only. Going further, the papers points out that though failures refinement test can guaratantee the preservation of LTL properties as long as $Spec_\phi$ is finite-branching, such condition doesn't hold in most cases in practice.

One more problem with the previous definition of satisfaction of LTL formula is that it overlooks the fact that a CSP process which may deadlock is still possible to "satisfy" an LTL formula intuitively. To tackle this, the paper introduces an extra event Δ which indicating deadlock. Any deadlocking trace of the process S is then turned into an infinite trace by appending an infinite number of Δ's. To capture the intuition when a deadlocking trace satisfies an ordinary LTL formula ϕ over Σ, we can translate from an LTL formula ϕ into a formula ϕ_Δ over $\Sigma \cup \{\Delta\}$, e.g., $\phi\ U \psi$ is translated to $(\neg\Delta \wedge \phi)\ U \psi$. A process S satisfies a LTL formula is now defined as follows:
$S \models \phi$ iff $\forall \pi \in |S|_\Delta, \pi \models \phi_\Delta$ where $|S|_\Delta = |S|_\omega \cup \{\gamma\Delta^\omega | (\gamma, \Sigma) \in failures(S)\}$.

By existing automaton theory, we can create a Büchi automaton $\mathcal{B}$ (from $\neg\phi_\Delta$) over $\Sigma \cup \{\Delta\}$ such that $|\mathcal{B}| = |\neg\phi_\Delta|_\omega$, where $|\mathcal{B}|$ is the set of accepting words of $\mathcal{B}$. With the aforementioned setting, the question whether a process S satisfies the LTL formula ϕ ($S \models \phi$) is now turned into the question whether $|S|_\Delta \cap |\mathcal{B}| = \phi$. It's easy to see that if the answer is yes, then $S \models \phi$, otherwise $S \not\models \phi$.

At this stage, we have two possible strategies. First is to create another Büchi automaton $\mathcal{A}$ whose set of accepting words is $|S|_\Delta$ and then use existing algorithm to check the emptiness of the intersection of the sets of accepting words of these two Büchi automata (a.k.a checking the emptiness of $\mathcal{A} \cap \mathcal{B}$). Research following this path for LTL checking CSP can be found in [19].

This paper takes the second strategy in which a process $TESTER$ is generated from the Büchi automaton $\mathcal{B}$ and the decision problem is then turned into several refinement checks based on the parallel composition of $TESTER$ and S. The intuition behind this is that $TESTER$ "represents" the traces accepted by the Büchi automaton $\mathcal{B}$, and we want to check whether the process S has "similar" traces to that of the $TESTER$. If it has, then S has a trace $\pi \models \neg\phi$ which indicates that $S \not\models \phi$. Concretely, the paper describes the method as follows.

Noticing that the Büchi automaton $\mathcal{B}$ may accept certain traces (in which events in Σ appear after Δ) which no CSP process can possibly generate, this paper translates $\mathcal{B}$ into a simplified automaton called Büchi Δ-automaton $\mathcal{B}_\Delta$ whose accepting set of words $|\mathcal{B}_\Delta| = |\mathcal{B}| \cap (\Sigma^\omega \cup \Sigma^*.\Delta^\omega)$. The process $TESTER$ can be generated from $\mathcal{B}_\Delta$ with the introduction of three extra events *success*, *deadlock*, and *ko*. And to check $S \models \phi$, the following two checks shall be performed.

1. $SUC \sqsupseteq_\mathcal{T} (S \underset{\Sigma}{\parallel} TESTER \backslash (\Sigma \cup \{deadlock, ko\})$ where $SUC = success \rightarrow SUC$
2. $deadlock \rightarrow STOP \sqsupseteq_\mathcal{F} (S \underset{\Sigma}{\parallel} TESTER(\Sigma \cup \{success\})$

By detailed discussion, the paper shows that either of two checks succeeds iff $S \not\models \phi$.

This paper uses FDR refinement checker to perform the refinement checking involved in the process, which means that optimisations such as hierarchical compression, data-independence and induction can be applied. But it's unclear whether the overall system has better performance over the mothod developed following the first possible strategy. Another issue is that since FDR doesn't provide any "candidiate" trace when the refinement check succeeds, which indicates that $S \not\models \phi$, there is no way to get the counter-example needed by the user to reason about the cause of the failure.

9 Summary of "Polymorphic CSP Type Checking"[4]

This paper illustrates the design of a type checker for CSP_M, which is based on Milner's type inference algorithm and Robinson's unification algorithm.

To begin with, this paper explains the classical type inference algorithm, which is similar to the one I implemented for the CS 552 Compiler course. Then it introduces two categories of types in addition to types usually associated with functional languages (e.g. *int* and *bool*): unqualified types and qualified types. Unqualified types including *process*, *event* function type, product type, sequence type, set type, union type (I think this one is similar to datatype in ATS.), dot type (unique to CSP_M's dot composition syntax) and yield type (type for channels and tags of datatype). Qualified types are types subject to additional constraints. There are only four constraints, which indicate that certain type must have the following four kinds of properties: 1. the type must supports equality operation; 2. the type supports ordering comparison; 3. the type is actually a well-formed function type; 4. the type must be a process type. (Compared to ATS, such constraints are relatively simple.)

The paper presents an extension of Robinson's unification algorithm so that it can handle both the dot and qualified types. Going still further, a type inference algorithm, which extends Milner's polymorphic type checking algorithm, is presented.

I think I'd better acquire some knowledge about type unification and type inference from textbook so that I can fully understand this paper.

10 Summary of "Java2CSP: A System for Verifying Concurrent Java Programs"[17]

This paper presents the system **Java2CSP** which translates concurrent Java programs into **CSP** processes, which can be then be fed to model-checking tool FDR to verify the synchronization behaviour of the Java program, such as deadlock and livelock.

CSP processes which simulate those mechanisms that are supported by Java but not by **CSP** such as shared variables, threads and monitors are called *process patterns*.

The Java bytecode of a Java program is used as the source code for the translation. The main reason I agree with is that "A Java program can include some auxiliary sources from libraries or from other users, which are sometimes only available in their bytecode representations".

One challenge is to provide CSP specification of corresponding mechanism in Java programs. For this, a set of CSP process patterns have been developed which specify the general behaviour of a set of Java methanisms, like shared variables, threads and monitores. Also the abstract reperessentation of Java object-model, types, and algorithms have been formalized. The author says algorithms are implemented to identify threads which can run concurrently, and to identify variables which influence the behaviour of threads in different ways. But no further illustration in the paper at all.

Another challenge is to generate CSP model which can be analysized efficiently by tools such as FDR.

10.1 Quote

The output of Java2CSP is again the input for FDR, the improvement of hte quality of the output format with respect to the readability and the recoverability is also an indispensable effort to make the system practical useful for large Java programs.

10.2 Comment

The tool now only supports the verification of deadlock and livelock behaviours of the Java programs. No support for the verification of LTL properties.

11 Summary of "Model-Checking CSP-Z: Strategy, Tool Support and Industrial Application"[11]

12 Summary of various model checkers

12.1 SAL

The fact that SAL is based on the symbolic approach to model checking and that the symbolic technique is not suitable for software analysis where dynamic creation of objects/data loses relevance in the context of applicative RAISE specifications, where no dynamic object creation is possible. Together with this idea are the multiple advantages that the symbolic management of states can offer: sets of states can be handled collectively (better temporal performance in some cases) and the ability to handle much bigger state spaces due to symbolic representation (compared with the explicit state representation approach).[12]

12.2 List of model checker

http://en.wikipedia.org/wiki/List_of_model_checking_tools

12.3 Erlang

Erlang is based on the ACTOR model, not CSP. The main practical differences between Erlang and CSP is that Erlang uses asynchronous dynamically-typed messages sent to a particular address (process id), whereas CSP systems usually deal with synchronous messages sent down a particular, typed channel. But they are both message-passing systems with the idea of removing shared mutable data, as you say. For an implementation of CSP in the pure functional language Haskell, see my library CHP (http://www.cs.kent.ac.uk/projects/ofa/chp/). (From internet.)

13 Summary of "An Analytical and Experimental Comparison of CSP Extensions and Tools"[18]

13.1 summary

This paper compares CSP_M and CSP# from the aspects of syntax, operatioinal semantics as well as their supporting tools such as FDR, PRoB, and PAT. Such comparison can be used to guide users to choose the appropriate CSP extension and verification tool based on the system characteristics.

I didn't find the information I am seeking about the roles of channels in CSP# when dealing with refinement checking. Do the messages passed through channels constitute the events of refinement checking? I don't know.

4.1 of this paper summaries various optimization approaches used by different model checkers. Also it compares the abilities of FDR and PAT to do LTL checking.

This paper includes several benchmarks for the comparison of the efficiency of different model checkers.

13.2 quote

enhancing CSP by taking data and other system aspects into account

CSP# integrates high-level CSP-like process operators with low-level procedure code.

formally model concurrent systems, compositional operators, non-trivial data structures, different verification capabilities

perspectives of modeling and verification needs, reasoning power of their supporting tools, reason techniques and verifiable properties, the semantic differences between CSP_M and CSP# lead to different state spaces and optimizaitons in model checking

expression language

14 Summary of "Combining CSP and B for Specification and Property Verification"[2]

14.1 summary

This paper illustrates how a system could be specified as a combination of a CSP process and a B model, and gives out the operational semantics of such system based on the semantics of B model and CSP process. (I like the description.) An interpreter was implemented in Prolog for CSP, and was later combined with ProB to give out the operational semantics of the whole system. With this tool, we can use CSP to specify some desirable or undesirable behaviours and use PRoB to find traces of the B machine that exhibit those behaviours.

14.2 quote

The state of a B machine is a mapping from variables to values, while the state of a CSP pocess is a syntactic process expression. Whenever possible there is value in applying model checking to a size-restricted version of a B model before attempting semi-automatic deductive proof.

15 Summary of "Probing the Depthes of CSP-M: A new FDR-compliant validation Tool"[9]

15.1 summary

The paper describes the implementation of a parser and an interpreter for CSP using Haskell and Prolog, based on which a new animation and model checking tool for CSP is created and incorperated into ProB. This tool can do refinement checking as well as LTL checking on CSP. Also this tool can be applied to the validation of combined B and CSP specifications; see [2].

16 Summary of "Model Checking RAISE Applicative Specifications"[13]

16.1 quote

these tools are proof langauges, not suitable for temporal logic verification.

Given the fact that FDR's input language is quite expressive, this approach allows a straightforward translation of most Z predicates (sequences and sequence operations for example), yet with considerable loss of abstraction.

17 Summary of "Kratos – A Software Model Checker for SystemC"[3]

17.1 summary

A SystemC design is a complex entity comprising a multi-threaded program where scheduling is cooperative, according to a specific set of rules, and the execution of threads is mutually exclusive. This paper presents KRATOS, a software model checker providing two different analyses for verifying safety properties (in the form of program assertions) of SystemC designs.

First, KRATOS implements a sequential analysis based on lazy predicate abstraction for verifying sequential C programs. and translates from SystemC to a sequential C program by encoding both the threads and scheduler in the SystemC design. Such analysis is a reimplementation of the same analysis performed by existing software model checkers based on the lazy predicate abstraction, like BLAST and CPACHECKER.

Second, KRATOSimplements a novel concurrent analysis, called ESST, that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy predicate abstraction, to deal with the *Threads*. Another way to put it, the authors built a scheduler, which is more concrete than the abstract model used for translating SystemC design into a sequential C program mentioned above, exploring all possible schedules, and then apply similar analysis method as sequential analysis to each sheduled thread.

Translations from SystemC designs into C prgrams used in the aforemented two kinds of analyses are performed by SYSTEMC2C, a new back-end of PINAPA.

KRATOS is built on top of an extended version of NuSMV, which is tightly integrated with the MATHSAT SMT sovler. KRATOS replies on these two to carry out the aforementioned two analyses.

In essence, KRATOS is a software model checker for sequential and threaded C programs. (I think it's a special subset of C since the C programs are translated from SystemC.) KRATOS implements the lazy predicate abstraction and the lazy abstraction with interpolants for analyzing sequential programs, and ESST for analyzing threaded programs. KRATOS also provides a specialization of ESST to SystemC verification. KRATOS is built on top of an extended version of NUSMV, which is tightly integrated with MATHSAT SMT solver.

I like the section for "Novel Functionalities".

17.2 quote

symbolic lazy predicate abstraction technique

SMT-based techniques for program abstractions and refienements

Due to space limit, the results of an experimental evaluation that compares KRATOS with other model checkers on various benchmarks can be found in xx.

cooperative scheduler

conterexample-guided abstraction refinement (CEGAR)

"off-the-shelf" software model checking techniques

hybrid predicate that integrates BDDs and SMT solvers, as described in xx.

Bounded model checking (BMC)

We have observed that different encodings for transitions can affect the preformace of KRATOS.

Depending on the nature of the problem, the availablity of several encodings allow users to choose the most effective one for tackling the problem.

The concurrent analysis, based on the novel ESST algorithm, combines explicit state techniques with lazy predicate abstraction to verify threaded C program that models a SystemC design.

18 Summary of "Yet Another Model Checker for PROMELA – the Transformation Approach"[24]

18.1 summary

This paper illustrates the way of translating PROMELA code into CSP# code. A very detailed comparison of all parts of PROMELA and CSP# is given. Some

techniques for writing programs in CSP# is given to support encoding features not yet supported in CSP#.

18.2 quote

Simulation and Verification are critical for system design. In system verification, liveness is always a property with concern. Fairness is important to prove liveness properties. Without fairness, verification of liveness properties may produce unrealistic loops during which one process or event is unfairly favoured. Nonetheless, there are few established tools supporting verification under strong fairness.

The design and verification of concurrent and real-time systems are notoriously difficult problems. In particular, the interaction of concurrent processes in large systems often leads to subtle bugs that are extremely difficult to discover using the conventional techniques of simulation and testing. Automated verification based on model checking promises a more effective way of discovering design errors, which has been used successfully in practice to verify complex software systems.

SPIN is deficient as having no supoort for strong fairness. The only way to perform verification under strong fairness is to encode the fairness constraints as part of the property.

To achieve good performance, advanced techniques are implemented in PAT, e.g. partial order reduction [x], process counter abstraction [x], bounded model checking [x], parallel model checking [x], etc.

In PROMELA, channels can be used as parameters for the purpose of communication between processes. PAT doesnt support this feature for performance reason. Passing local declared channel between processes or checking data types, considering the case when channel is used as message type, will definitely slow down the speed of verification, especially when the verification needs to search all possible system status exhaustively.

19 Summary of "Linearizability: A Correctness Condition for Concurrent Objects"[?]

19.1 summary

The paper gives out a formal definition of linearizability based on the concept of history. It also compares linearizability with other correctness properties of concurrent systems, such as sequential consistency and serializability.

The notion of "equivalence" between two histories is important.

19.2 quote

To decide whether a concurrent history is acceptable, it is necessary to take into account the objects intended semantics. For example, acceptable concurrent behaviors for FIFO queues would not be acceptable for stacks, sets, directories, etc.

Informally, a concurrent system consists of a collection of sequential threads of control called processes that communicate through shared data structures called objects.

Formally, an execution of a concurrent system is modeled by a history, which is a finite sequence of operation invocation and response events.

Angle brackets for events and square brackets for operations are omitted where they would otherwise be unnecessarily confusing; object and process names are omitted where they are clear from context.

Restricting attention to complete(H) captures the notion that the remaining pending invocations have not yet had an effect.

L2 states that this apparent sequential interleaving respects the real-time precedence ordering of operations.

A linearizable object is one whose concur- rent histories are linearizable with respect to some sequential specification.

A concurrent system based on a nonlocal correctness property must either rely on a centralized scheduler for all objects, or else satisfy additional constraints placed on objects to ensure that they follow compatible scheduling protocols.

Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object.

An immediate practical consequence is that con- currency control mechanisms appropriate for serializability are typically inap- propriate for linearizability because they introduce unnecessary overhead and place unnecessary restrictions on concurrency.

20 Summary of "Operational Semantics for Model Checking Circus"[21]

20.1 summary

todo

20.2 quote

21 Summary of "Compositional Network Mobility"[23]

21.1 summary

This paper gives out a "geomorphic view of networks in which the layer is the fundamental module of network software, and is defined constructively in terms of its components. Such view is relatively simple and clear compared to the real networking infrastructure. Based on this view, the authors specify the vertical networking stack in Promela and model check it against properties such as deadlock. The authors also give out the definition of mobility and specify it formally using Alloy.

According to the "geomorphic" view of networks, a level consists of multiple layers, which can be thought of as certain subnet.

Two types of mobility (attachment, location) are similar. The difference lies in the level currently under concern in the whole implementation of mobility.

Hard to tell the contribution of the paper. The stack model may be referred when we want to implement our own networking stack instead of imitating existing Socket implementation.

21.2 quote

22 Summary of "Why Functional Programming Matters"[6]

22.1 summary

In this paper, the authors argue that modularity is the key to successful programming. Languages which aim to impromve productivity must support modular programming well.

22.2 quote

todo

23 Summary of "xxxx"[?]

23.1 summary

todo

23.2 quote

todo

References

1. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, chapter 1-7, 11. Elsevier, revised edition, November 1984.
2. Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In *Proceedings of the 2005 International Conference on Formal Methods*, FM'05, pages 221–236, Berlin, Heidelberg, 2005. Springer-Verlag.
3. Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. KRATOS A Software Model Checker for SystemC. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 310–316. Springer Berlin Heidelberg, 2011.
4. Ping Gao and Robert Esser. Polymorphic CSP type checking. In *Proceedings of the 24th Australasian conference on Computer science*, ACSC '01, pages 156–162, Washington, DC, USA, 2001. IEEE Computer Society.
5. Jean-Yves Girard, Yves Lafon, and Paul Taylor. *Proofs and Types*, chapter 2, 3, 4, 6, 11. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, April 1989.
6. John Hughes. Why Functional Programming Matters. In *The Computer Journal*, volume 32, pages 98–107, 1984.
7. Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 0 edition, December 1999.
8. Ranko Lazic. *A Semantic Study of Data Independence with Application to Model Checking*. PhD thesis, 1999.
9. Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Formal Methods and Software Engineering*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297. Springer Berlin / Heidelberg, 2008.
10. Michael Leuschel, Thierry Massart, and Andrew Currie. How to Make FDR Spin LTL Model Checking of CSP by Refinement. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 99–118, London, UK, UK, 2001. Springer-Verlag.
11. Alexandre Mota and Augusto Sampaio. Model-Checking CSP-Z: Strategy, Tool Support and Industrial Application. In *Science of Computer Programming*, 2001.
12. J. I. Perna and C. George. Model checking RAISE specifications. Technical report, December 2005.
13. J. I. Perna and C. George. Model Checking RAISE Applicative Specifications. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 257–268. IEEE, 2007.

14. Amir Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, volume 0 of *SFCS '77*, pages 46–57, Washington, DC, USA, September 1977. IEEE.
15. A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
17. Hui Shi. Java2CSP: A System for Verifying Concurrent Java Programs. In *FM-TOOLS*, pages 111–115, 2000.
18. Ling Shi, Yang Liu, Jun Sun, Jin S. Dong, and Gustavo Carvalho. An Analytical and Experimental Comparison of CSP Extensions and Tools. In *The 14th International Conference on Formal Engineering Methods (ICFEM 2012)*, November 2012.
19. Jun Sun, Yang Liu, and Jin S. Dong. *Model Checking CSP Revisited: Introducing a Process Analysis Toolkit*, volume 17 of *Communications in Computer and Information Science*, chapter 22, pages 307–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
20. Philip Wadler. Linear Types Can Change the World! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.
21. Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Operational Semantics for Model Checking Circus. In *of Lecture Notes in Computer Science*, pages 237–252, 2005.
22. Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.
23. Pamela Zave and Jennifer Rexford. Compositional Network Mobility.
24. WenDi Zhao. Yet Another Model Checker for PROMELA - The Transformation Approach. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 137–142. IEEE, June 2010.
25. Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *In Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, 2005.