Integrating Specification and Programs for System Modeling and Verification

Jun Sun, Yang Liu, Jin Song Dong and Chunqing Chen
National University of Singapore
{sunj,liuyang,dongjs,chenchun}@comp.nus.edu.sg

Abstract

High level specification languages like CSP use mathematical objects as abstractions to represent systems and processes. System behaviors are described as process expressions combined with compositional operators, which are associated with elegant algebraic laws for system analysis. Nonetheless, modeling systems with non-trivial data and functional aspects using CSP remains difficult. In this work, we propose a modeling language named CSP# (short for communicating sequential programs) which integrates high-level modeling operators with low-level procedural codes, for the purpose of efficient mechanical system verification. We demonstrate that data operations can be modeled as terminating sequential programs, which can be composed using high-level compositional operators. CSP# is supported by the PAT model checker and has been applied to a number of systems.

1. Introduction

System modeling is very important and highly non-trivial. The choice of specification language is an important factor in the success of the entire development. The language should cover several facets of the requirements and the model should reflect exactly (up to abstraction of irrelevant details) an existing system or a system to be built. The language should have a semantic model suitable to study the behaviors of the system and to establish the validity of desired properties. A formal model can be the basis for a variety of system development activities, e.g., system simulation, visualization, verification or prototype synthesis. Many specification languages have been proposed. High-level languages like CSP [8] and CCS [13] use mathematical objects as abstractions to represent systems or processes. System behaviors are described as process expressions combined with a rich set of compositional operators, e.g., deterministic or nondeterministic choice, parallel composition and recursion. The operators are associated with elegant algebraic laws for system analysis.

The original CSP derives its full name from the built-in syntactic constraint that processes belong to the sequential subset of the language. CSP has passed the test of time. It has been widely accepted and influenced the design of many recent programming and specification languages. Nonetheless, modeling systems with non-trivial data structures and functional aspects completely using languages like CSP remains difficult. A characteristic of CSP is that processes have disjoint local variables, which was influenced by Dijkstra's principle of *loose coupling* [4]. CSP supports inter-process communication through message passing but not shared memory, i.e., shared variables. It has long been known (see [8] and [15], for example) that one can model a variable as a process parallel to the one that uses it. The user processes then read from, or write to, the variable by CSP communication. Though feasible, this is painful for systems with non-trivial data structures (e.g., arrays) and operations (e.g., array sorting). Therefore, 'syntactic sugars' like shared variables are mostly welcomed.

In order to solve the problem, many specification languages integrating process algebras like CSP or CCS with state-based specification languages like the Z language or Object-Z have been proposed. The state-based language component is typically used to specify the data states of the system and the associated data operations in a declarative style. Examples include *Circus* [20] (i.e., an integration of CSP and the Z language), CSP-OZ [7] (i.e., an integration of CSP and Object-Z) and TCOZ [11] (i.e., an integration of Timed CSP and Object-Z). However, because declarative specification languages like Z are very expressive and not executable, automated analyzing (in particular, model checking) of systems modeled using the integrated languages is extremely difficult.

In this work, we propose an alternative solution, i.e., instead of specifying data states and operations in declarative languages, they are given as procedural codes. We propose a modeling language named CSP# (short for communicating sequential programs, pronounced as 'CSP sharp') which mixes high-level modeling operators with low-level programs, for the purpose of system modeling and verification. We demonstrate that data operations can be naturally modeled as terminating sequential programs, which

then can be composed using high-level compositional operators. *The idea is to treat sequential terminating programs as atomic events.* CSP# models are executable with complete operational semantics, and therefore subject to system simulation and, more importantly, fully automated system verification techniques like model checking. CSP# is supported by the PAT model checker [18] (available at http://pat.comp.nus.edu.sg) and has been applied to model and verify a number of systems.

This work is related to research on integrated formal methods, in particular, works on integrating state-based specification and event-based specification [20, 7, 11, 10]. Different from previous approaches, our modeling language is designed for automated system analysis. Therefore, it is fully operational and supported by PAT. Two other languages are designed for similar purposes, namely machine readable CSP (which we will refer to as CSP_M) supported by the refinement checker FDR [14] and *Promela* which is supported by the model checker SPIN [9]. Compared to CSP_M, CSP# supports additional language features like shared variables, asynchronous communication channels and event associated programs, which offers users great flexibility in modeling. Furthermore, we give an interpretation of state/event Linear Temporal Logic in CSP# semantics framework, which allows temporal logic based model checking of CSP# models. Compared to *Promela*, CSP# supports more process constructs, i.e., *Promela* is based on a subset of CSP, whereas all CSP models are valid CSP# models. In particular, CSP# inherits the classic trace, stable failures and failures/divergence semantics from CSP, and therefore, allows us to perform a variety of refinement checkings. CSP# is also remotely related to other languages which are designed for model checking [3].

The remainder of the paper is organized as follows. Section 2 presents the syntax of CSP#. Section 3 defines its semantic model. Section 4 demonstrates a CSP# model of a multi-lift system. Section 5 concludes the paper.

2. Syntax

Integrating a highly abstract language like CSP with programming codes leads to many complications. Our design principle is to maximally keep the original CSP as a sublanguage of CSP#, whilst offering a connection to the data states and executable data operations.

A motivating example We use a mult-lift system as a running example. The reason is that it has complicated dynamic behaviors as well as nontrivial data states. Furthermore, the single-lift system has been modeled using many modeling languages including CSP. The system contains multiple components, e.g., the users, the lifts, the floors, the internal button panels, etc. There are non-trivial data components and data operations, e.g., the internal requests and external requests and the operations to add/delete requests. For simplicity, we assume there is no central controller for assigning external requests. Instead, each lift functions on its own to find and serve requests, in the following way. Initially, a lift resides at the ground level ready to travel upwards. Whenever there is a request (from the internal button panel or outside button) for the current residing floor, the lift opens the door and later closes it. Otherwise, if there are requests for a floor on the current traveling direction (e.g., a request for floor 3 when the lift is at floor 1 traveling upwards), then the lift keeps traveling on the current direction. Otherwise, the lift changes its direction. Other constraints on the system include that a user may only enter a lift when the door is open, there could be an internal request if and only if there is a user inside, etc.

2.1. Sequential Programs as Events

Shared variables offer an alternative means of communication among processes (which reside at the same computing device or are connected by wires with negligible transmission delay). They record the global state and make the information available to all processes. In the lift example, the internal/external requests can be naturally modeled as shared arrays. In CSP#, they are declared as follows.

1. **#define** $NoOfFloor$ 3;
2. **#define** $NoOfLift$ 2;
3. **var** $extUpReq[NoOfFloor]$;
4. **var** $extDownReq[NoOfFloor]$;
5. **var** $intRequests[NoOfLift][NoOfFloor]$;
6. **var** $doorOpen[NoOfLift]$;

where $define$ and var are reserved keywords. The former defines a global constant, e.g., $NoOfFloor$ which denotes the number of floors and $NoOfLift$ which denotes the number of lifts. The latter defines a variable, e.g., $extUpReq$ and $extDownReq$ which store external requests, $intRequests$ which store internal requests and $doorOpen$ which captures lift doors' states. CSP# has a weak type system (like JSP) and therefore type information is not necessary for variable declaration. By default, all the above defined are treated as arrays of integers. In particular, elements in $extUpReq$ (or $extDownReq$) are binary: 1 at j-th position means that there is a request for traveling *upwards* (or *downwards*) at j-th floor; 0 means no request. Two dimensional array $intRequests$ stores internal requests from all lifts. In particular, the internal request for the j-th floor from the i-th lift is stored at $intRequests[i][j]$ in the array. Elements in $intRequests$ are binary: 1 means that the floor has been requested and 0 means not requested. Elements in array $doorOpen$ range from -1 to $NoOfFloor - 1$. The i-th element of $doorOpen$ is -1 if and only if the door of i-th lift is closed and it is j such that $j \geq 0$ if and only if the

```
intRequests[i][level] = 0;
if (dir > 0){extUpReq[level] = 0; }
else {extDownReq[level] = 0; }
```

Figure 1. CSP# codes for clearing requests

i-th lift has opened door at j-th floor. We assume that initially all doors are closed. We remark if the Z language is used for specification, specific types for elements in the arrays may be defined to constrain their values. In CSP#, we instead use PAT to verify that the constraints hold given any system behavior.

Associated with the variables are data operations which query or modify the variables. In the lift system, whenever a lift opens its door, the requests must be updated accordingly. For instance, the codes shown in Figure 1 clear the requests when the i-th lift opens the door at $level$-th floor. Let dir be the current traveling direction (1 for upwards and -1 for downwards). The first line clears internal requests, by simply resetting the respective position in array $intRequests$ to 0. The rest clears external requests. Only the request along the lift's traveling direction is cleared. A more complicated operation is to determine whether there are requests along the current traveling direction, so as to determine whether a lift should keep traveling in the same direction or to change direction. This operation may be *implemented* by the codes in Figure 2, where $level$ is a variable recording the floor that the lift is residing at, $index$ is a loop counter and $result[i]$ records the result (0 for no such request and 1 for yes). A while-loop is used to search for a request along the current traveling direction, e.g., if the lift is traveling upwards, we search for a request for (or from) an upper floor. The search stops when the ground or top floor is reached.

A system may contain multiple data operations, each of which is *terminating* and is assumed to be executed *atomically*. They can be implemented using the CSP# syntax as shown above, or they can be implemented using existing programming languages. For instance, we offer the keyword **call** in PAT to allow invocation of data operations (as atomic events) implemented externally as C# static methods in CSP# models. Data operations may be invoked alternatively or in parallel. *In CSP#, data operations can be treated as atomic events and composed using compositional operators.* From another point of view, we allow an event to be associated with an optional sequential terminating program. For instance, the program in Figure 2 may be labeled as event $checkIfToMove.i.level$, which then can be used to constitute CSP process expressions, e.g., see Figure 3 and 5. Data races are prevented by not allowing synchronization of events containing procedural code.

```
index = level + dir; result[i] = 0;
while (index >= 0 && index < NoOfFloor
      && result[i] == 0){
  if (extUpReq[index]! =  ||
      extDownReq[index]! = 0  ||
      intRequests[i][index]! = 0){
    result[i] = 1;
  }
  else {index = index + dir; }
}
```

Figure 2. CSP# codes for searching requests

2.2. Composing Programs

The high-level compositional operators in CSP capture common system behavior patterns. They are very useful in system modeling. Furthermore, process equivalence can be proved or disproved by appealing to algebraic laws which are defined for the operators. In CSP#, we reuse most of the operators and integrate them with our extensions in a rigorous way so as to maximally preserve the algebraic laws.

A CSP# specification may contain multiple process definitions. A process definition gives a process expression a name, which can be referenced in process expressions. The following is a BNF description of the process expression[1].

$$
\begin{aligned}
P ::= {} & Stop \mid Skip \mid e\{prog\} \rightarrow P \mid \\
& ch!exp \rightarrow P \mid ch?x \rightarrow P \mid P \setminus X \mid P;\ Q \mid \\
& P \,[]\, Q \mid P \sqcap Q \mid \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\} \mid \\
& [b]P \mid P \parallel Q \mid P \,|||\, Q \mid P \triangle Q \mid ref(Q)
\end{aligned}
$$

where P, Q are processes, e is a name representing an event with an optional sequential program $prog$, X is a set of event names (e.g., $\{e_1, e_2\}$), b is a Boolean expression, ch is a channel, exp is an expression, and x is a variable. Besides events attached with programs, the most noticeable extension to CSP is the use of asynchronous channels, which again can be supported in CSP by explicitly modeling the communication buffer. Nonetheless, explicitly supporting them in CSP# is not only for users' convenience but also for possibly more efficient mechanical system exploration (refer to Section **??**). Given a channel ch with pre-defined buffer size, process $ch!exp \rightarrow P$ evaluates the expression exp (with the current valuation of the variables) and puts the value into the tail of the respective buffer and behaves as P. Process $ch?x \rightarrow P$ gets the top element in the respective buffer, assigns it to variable x and then behaves as P. Sending/receiving multiple messages at once are supported.

1 Refer to PAT's user manual for ASCII version of the symbols.

$Stop$ is the process that does nothing. $Skip = \checkmark \rightarrow Stop$, where $\checkmark$ is the special event of termination. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program, the program is executed atomically together with the occurrence of the event. Process $P \setminus X$ hides all occurrences of events in X. Sequential composition, $P;\ Q$, behaves as P until its termination and then behaves as Q. External choice $P \ \Box \ Q$ is solved only by the occurrence of a visible event. On the contrast, internal choice $P \sqcap Q$ is solved non-deterministically. Conditional choice $if\ b\ \{P\}\ else\ \{Q\}$ behaves as P if b evaluates to true, and behaves as Q otherwise. Process $[b]P$ waits until condition b becomes true and then behaves as P. Notice that it is different from $if\ b\ \{P\}\ else\ \{Q\}$. One distinguishing feature of CSP is alphabetized multi-processes parallel composition. Let P's alphabet, written as αP, be the events in P excluding the special invisible event τ. Process $P \parallel Q$ synchronizes common events in the alphabets of P and Q. In contrast, process $P \ ||| \ Q$ runs all processes independently (except for communication through shared variables). Process $P \ \triangle \ Q$ behaves as P until the first occurrence of an visible event from Q. A process expression may be given a name for referencing. Recursion is supported by process referencing.

In CSP#, we support global variables which are globally accessible, process parameters which are accessible in the respective process expression and local variables which are accessible in one data operation. We restrict the use of local variables in general. In particular, local variables introduced as process parameters or variables to store channel inputs cannot be modified by event associated programs. They can, however, be modified indirectly. The following illustrates alternative ways of achieving the same effect.

$$\begin{array}{lll} P(x) & = add\{x = x + 1\} \rightarrow P(x); & -\times \\ P(x) & = add \rightarrow P(x + 1); & -\checkmark \\ \textbf{var}\ x;\ P() & = add\{x = x + 1\} \rightarrow P(); & -\checkmark \end{array}$$

Because x cannot be modified, it becomes constant-like and therefore can be simply replaced by its value. This restriction allows us to perform efficient system verification. The reason is that, in this setting, it is sufficient to store only the valuation of the global variables and the process expression (with process parameters replaced with their values) when we explore the system states. Compared to software model checking, we can safely omit the *program stack* (which, combined with recursions, is very complicated to maintain) from the global state.

Figure 3 presents a process $Lift$ which concisely models the behavior of one lift. Notice that the process has multiple parameters, namely i which is an identifier of the lift, $level$ which denotes the residing floor and dir which denotes the current traveling direction (1 for traveling upwards and -1 for downwards). The condition at line 7 is used to check whether there is a request for the current floor, with the correct traveling direction if it is external. If yes, then the door is opened, the requests for the floor are cleared (using the code presented in Figure 1), and then the door is closed. Otherwise, the lift checks whether to continue traveling on the same direction (using the code presented in Figure 2). If the result is 1, then the lift moves to the next floor. Otherwise, the lift changes its direction and then repeats from the start. In this example, we have events which are associated with programs and simple events like $moving.i.dir$. The rest of the system model is presented in Section 4.

3. Semantics

In the section, we present operational semantics of CSP# models, which translates a model into a *labeled transition system* (LTS). The sets of behaviors can be extracted from the operational semantics, thanks to congruence theorems. We then define properties which are subject to model checking over the language of the LTS. We remark that the complication due to conflicts between global variables and CSP operational semantics (e.g., calculation of process alphabets) is discussed.

3.1. Operational Semantics

A system configuration is composed of two components (V, P) where V is a function mapping a variable name (or a channel name) to its value (or a sequence of items in the buffer), which we refer to as a valuation function in the following, and P is a process expression. A system transition is of the form $(V, P) \xrightarrow{e} (V', P')$ where e is an event.

The operational semantics is presented as firing rules associated with each process construct. The rules naturally extend the operational semantics for CSP [1] and Timed CSP [17]. Let Σ be the set of visible event names. For simplicity, we assume a function $upd(V, prog)$ which, given a sequential program $prog$ and V, returns the modified valuation function V' according to the semantics of the program. We write $V \models b$ (or $V \not\models b$) to denote that condition b evaluates to true (or false) given V. We write $eva(V, exp)$ to denote the value of the expression evaluated with variable valuations in V. To abuse notations, we write $app(V, ch!exp)$ to denote the function V' in which the respective channel buffer is appended with $eva(V, exp)$. We write $pop(V, ch?x)$ to denote the function V' in which the top element (written as $top(V, ch)$) in the respective channel buffer is removed.

Figure 4 illustrates part of the firing rules. The rest can be found in the Appendix. Rule $prefix$ captures how event associated with sequential programs are handled, i.e., the occurrence of the event and program is simultaneous and appears, to the system, to be atomic. Notice that, this is

```
7.  Lift(i, level, dir) = if ((dir > 0 && extUpReq[level] == 1) || (dir < 0 && extDownReq[level] == 1) ||
8.              intRequests[i][level] == dir){
9.              opendoor.i{doorOpen[i] = level; *code shown in Figure 1*} →
10.             closedoor.i{doorOpen[i] = −1} → Lift(i, level, dir)
11.         } else {
12.             checkIfToMove.i.level{*code shown in Figure 2*} →
13.             if (result[i] == 1){moving.i.dir →
14.                 if (level + dir == 0 || level + dir == NoOfFloors − 1){Lift(i, level + dir, −1 ∗ dir)}
15.                 else {Lift(i, level + dir, dir)}
16.             } else {
17.                 if ((level == 0 && dir == 1) || (level == NoOfFloors − 1 && dir == −1)){
18.                     Lift(i, level, dir)
19.                 }
20.                 else {changedir.i.level → Lift(i, level, −1 ∗ dir)}
21.             }
22.         };
```

Figure 3. CSP# model of the lift

the only way global variables are modified. Rule out and in captures semantics of channel output/input. We remark that there are two rules associated with *if* b $\{P\}$ *else* $\{Q\}$, whereas only one is associated with $[b]P$. Therefore, if b is false given $[b]P$, then the process will block until b becomes true.

The semantics of parallel composition $P \parallel Q$ depends on the alphabets of P and Q. Parallel composition is one of the most complicated operators in CSP. In CSP_M, users are asked to specify the synchronizing events[2]. This is sometimes troublesome. In CSP#, we use only one form of parallel composition (i.e., $P \parallel Q$). We remark this is not a restriction. In CSP#, users are allowed to manually specify the alphabet of a process[3] (e.g., **#alphabet** P X where X is a set of event names). In addition, we offer a sophisticated procedure to mechanically calculate the *default* alphabet of a process (and the synchronizing events), i.e., the set of simple events which constitute the process expression. We remark that in general, this is not always possible, e.g., if the event name is constituted with global variables or process parameters which are changing through recursive calls. In such cases, PAT will make a complaint, when a parallel composition is subject to simulation or model checking. Notice that an event which is associated with a data operation is excluded from the alphabet in CSP#. For instance, assume x is a global variable,

$$P() = a\{x = x + 1\} \rightarrow Stop;$$
$$Q() = a\{x = x + 2\} \rightarrow Stop;$$

Given the above, event a is not synchronized in the parallel composition of $P()$ and $Q()$. The intuition is that data operations are local actions, instead of communications. This prevents synchronizing events associated with different data operations but with the same name (e.g., a in the above example) and syntactically avoids potential data race.

We call a process expression and a function V a model. The transition system of the model is a LTS $L_P^V = (S, init, \rightarrow)$ where S is the set of reachable system configurations, $init$ is the initial configuration (V, P) and $\rightarrow$ is a labeled transition relationship conforming to the operational semantics presented in Figure 4 and the Appendix. A finite execution of P with V is a finite sequence of alternating states/events $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ where $s_0 = init$ and $s_i \stackrel{e_i}{\rightarrow} s_{i+1}$ for all $0 \leq i \leq n$. An infinite execution is an infinite sequence $\langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots\rangle$ where $s_0 = init$ and $s_i \stackrel{e_i}{\rightarrow} s_{i+1}$ for all $i \geq 0$.

A model is deadlock-free if and only if there does not exist a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that s_{n+1} is a deadlock state, i.e., no firing rules are applicable given s_{n+1}. Given a proposition p, a state satisfying the predicate is reachable (or equivalently p is reachable) if and only if there exists a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that $s_{n+1} = (V_{n+1}, P_{n+1})$ and $V_{n+1} \vDash p$.

3.2. Traces, Failures and Divergences

Verification of CSP models has been traditionally based on refinement checking. CSP refinement is expressive enough to cover a large class of properties [16]. In

2 in the form of $P[|\ a\ |]Q$, $P[a \parallel a']Q$ or $P[c <-> c']Q$.

3 By default, the alphabet of a process includes ✓.

$$\frac{}{(V, e\{prog\} \rightarrow P) \xrightarrow{e} (upd(V, prog), P)}\ [\ prefix\ ] \qquad \frac{c \text{ is not full in } V}{(V, c!exp \rightarrow P) \xrightarrow{c!eva(V,exp)} (app(V, c!exp), P)}\ [\ out\ ]$$

$$\frac{c \text{ is not empty in } V}{(V, c?x \rightarrow P) \xrightarrow{c?top(c)} (pop(V, c?x), P)}\ [\ in\ ] \qquad \frac{V \vDash b, (V, P) \xrightarrow{e} (V', P')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e} (V', P')}\ [\ cond1\ ]$$

$$\frac{V \nvDash b, (V, Q) \xrightarrow{e} (V', Q')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e} (V', Q')}\ [\ cond2\ ] \qquad \frac{V \vDash b, (V, P) \xrightarrow{e} (V', P')}{(V, [b]P) \xrightarrow{e} (V', P')}\ [\ guard\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P'), x \in \alpha P, x \notin \alpha Q}{(V, P \parallel Q) \xrightarrow{x} (V', P' \parallel Q)}\ [\ par1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q'), x \in \alpha Q, x \notin \alpha P}{(V, P \parallel Q) \xrightarrow{x} (V', P \parallel Q')}\ [\ par2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V, P'), (V, Q) \xrightarrow{x} (V, Q'), x \in \alpha P \cap \alpha Q}{(V, P \parallel Q) \xrightarrow{x} (V, P' \parallel Q')}\ [\ par3\ ] \qquad \frac{P \mathrel{\widehat{=}} Q, (V, Q) \xrightarrow{*} (V', Q')}{(V, P) \xrightarrow{*} (V', Q')}\ [\ def\ ]$$

Figure 4. operational semantics where $e \in \Sigma$; $e_\tau \in \Sigma \cup \{\tau\}$; $x \in \Sigma \cup \{\checkmark\}$ and $* \in \Sigma \cup \{\tau, \checkmark\}$

the following, we extend the trace, stable failures and failures/divergences semantics of CSP to CSP# models. Unlike those in CSP, the definitions are defined on the labeled transition system L_P^V. We then briefly discuss how the respective refinement relationship can be checked mechanically.

A finite sequence of events $\langle x_0, x_1, \cdots, x_m\rangle$ is a trace of P with V if and only if there exists a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ of L_P^V such that $\langle e_0, e_1, \cdots, e_n\rangle \upharpoonright \{\tau\} = \langle x_0, x_1, \cdots, x_m\rangle$ where $tr \upharpoonright X$ removes the events in X from the sequence tr. The set of all traces is written as $traces(P, V)$.

Given a trace tr, we write $(P, V)/tr$ to denote the set of system configurations that can be reached from (P, V) via trace tr. Because of nondeterminism, multiple configurations can be reached via the same trace. Given process P and valuation function V, the refusals are the sets of event sets which may be *refused*.

$$refusals(P, V) = \{X \mid \forall\, e : X \\ \nexists (P', V')\ (P, V) \xrightarrow{e} (P', V')\}$$

The failures of the model (P, V) is defined as follows,

$$failures(P, V) = \{(tr, X) \mid tr \in traces(P, V) \wedge \\ X \in refusals((P, V)/tr)\}$$

If (tr, X) is a failure of the model, this means that the model can engage in the sequence of events recorded by tr, and then refuse to perform any event in X. A divergence of a model is defined as a trace after (a prefix of) which the model may perform infinite number of internal invisible actions. Equivalently, a trace tr is a divergence of the model (P, V) if and only if there exists an infinite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \tau, s_{i+1}, \cdots \tau, s_{i+n}, \cdots\rangle$ of L_P^V such that $\langle e_0, e_1, \cdots, e_i\rangle \upharpoonright \{\tau\}$ is tr or a prefix of tr. The set of divergence of a model is written as $divergences(P, V)$.

Given two models (P_1, V_1) and (P_2, V_2), model (P_1, V_1) refines (P_2, V_2) in the trace semantics if and only if $traces(P_1, V_1) \subseteq traces(P_2, V_2)$; model (P_1, V_1) refines (P_2, V_2) in the stable failures semantics if and only if $failures(P_1, V_1) \subseteq failures(P_2, V_2)$; model (P_1, V_1) refines (P_2, V_2) in the failures/divergences semantics if and only if $failures(P_1, V_1) \subseteq failures(P_2, V_2)$ and $divergences(P_1, V_1) \subseteq divergences(P_2, V_2)$.

It can be shown that our definition is consistent with the CSP definition with respect to the CSP subset of CSP#. Moreover, the refinement checking algorithm implemented in the FDR model checker [14], which is based on the operational semantics of CSP, can be naturally extended to check refinement relationship between CSP# models. We refer the readers to [18] for details on the checking algorithm (combined with partial order reduction), which has been implemented in PAT.

3.3. State/Event LTL Semantics

Model checking based on temporal logic formulae has been proved effective as well as intuitive, which has gathered much attention, evidenced by the rich set of theories

and tools developed for CTL/LTL based verification [3, 9]. Because states/variables are made explicit in CSP#, state-based temporal logic is a natural candidate for property specification and verification. Furthermore, because we are dealing with an event-based formalism, it would be meaningful if the properties may concern both states and events. For instance, in the lift example, we may be interested in specifying and verifying the following property, where $\Box$ and $\Diamond$ are modal operators which read 'always' and 'eventually' respectively.

$$\Box(extUpReq[0] > 0 \Rightarrow \Diamond extUpReq[0] = 0) \wedge \Box\Diamond moving.0.1$$

The property states that a request at the ground floor must eventually be served (i.e., cannot be ignored forever) and the event $moving.0.1$ must always eventually occur (i.e., 0-th lift must always eventually move upwards).

In this section, we follow the work presented in [2] to define an interpretation of state/event linear temporal logic (LTL) based on CSP# semantics, which allows us to apply automata-based model checking [9] of temporal logic formulae constituted with both event and state propositions. Let Pr be a set of propositions (formulated using predicates on global variables in CSP#). An extended LTL formula is defined as follows.

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \psi \mid X\phi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

where p ranges over Pr and a ranges over Σ. Let $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots\rangle$ be an infinite execution. Let π^i be the suffix of π starting from P_i.

$$\begin{array}{ll}
\pi^i \vDash p & \Leftrightarrow s_i \vDash p \\
\pi^i \vDash a & \Leftrightarrow x_{i-1} = a \\
\pi^i \vDash \neg\phi & \Leftrightarrow \neg(\pi^i \vDash \phi) \\
\pi^i \vDash \phi \wedge \psi & \Leftrightarrow \pi^i \vDash \phi \wedge \pi^i \vDash \psi \\
\pi^i \vDash X \wedge \phi & \Leftrightarrow \pi^{i+1} \vDash \phi \\
\pi^i \vDash \Box\phi & \Leftrightarrow \forall j \geq i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \Diamond\phi & \Leftrightarrow \exists j \geq i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \phi U \psi & \Leftrightarrow \exists j \geq i \bullet \pi^j \vDash \psi \wedge \\
 & \qquad \forall k \mid i \leq k \leq j-1 \bullet \pi^j \vDash \phi
\end{array}$$

The simplicity of writing formulae concerning events as in the above example is not purely a matter of aesthetics. It may yield gains in time and space (refer to examples in [2]). A model satisfies ϕ if and only if every infinite execution of L_P^V satisfies ϕ. We refer the readers to [19] for details on temporal logic model checking (with or without fairness constraints) of CSP# models.

4. Case Studies

In this section, we complete the case study of the multi-lift model. Our modeling is related to the previous lift system models presented in [12]. In addition, we demonstrate how to write critical system properties as assertions.

Figure 5 shows the rest of the model. In particular, line 23 defines the rest of the variables (which are used in Figure 2). Lines 24 to 34 model users' behavior in the lift system. At line 24, the behavior of three users is defined as the interleaving of each user, where $||| \; x : \{i..j\} @ P(x) = P(i) \; ||| \cdots ||| \; P(j)$. Behavior of a user is specified as process $aUser$ at line 25. Each user may initially be at any floor. This is captured using indexed external choice. The user pushes a button (for traveling upwards or downwards, specified as $ExternalPush(pos)$) and then waits for the lift to come (specified as $Waiting(pos)$).

A **case** statement, which is a syntactic sugar for multiple if-then-else statements, is used in process $ExternalPush(pos)$. We remark that the conditions in the case statement are evaluated in the order until one which evaluates to be true is found. Otherwise, the $default$ branch is taken. In the example, if the user is at the ground floor or the top one, only one direction to travel can be requested. Otherwise, the user may choose either to go upwards or downwards. Lines 27 to 30 capture how the external requests are updated.

The user then waits until a lift opened its door at the user's floor (captured by condition $doorOpen[i] == pos$) and then enters the lift. We remark that this model allows users to enter the lift with the wrong traveling direction (which may happen in real world). After making an internal request, the user may exit when the door is opened again at his/her destination floor. The lift system is modeled as the interleaving of users and multiple lifts at line 35. Initially, the lifts are residing at the ground floor, ready to travel upwards. In this example, we demonstrate how variable updates and compositional operators may be used together seamlessly to capture system behavior.

Once we have a model, we may use PAT to simulate its behaviors. Alternatively, we may write assertions about critical system properties and invoke the PAT model checkers to examine the model in order to find counterexamples. In particular, line 36 asserts that the lift system is deadlock-free. Line 37 and 38 define propositions which are used to constitute the temporal logic formula defined at line 39, which has been discussed in Section 3.3.

5. Conclusions

In this work, we propose a mixing of high-level specification languages with low-level procedural codes for the purpose of efficient system analysis, in particular, model checking. A multi-lift system is used to illustrate the language. We remark that CSP# has been applied to model and verify a variety of systems, ranging from recently proposed distributed algorithms, concurrent programming algorithms to real-world systems like the pacemaker system. Previously unknown bugs have been discovered. Further-

23. **var** $index$; **var** $result[NoOfLift]$;
24. $Users() =||| \ x : \{0..2\}@aUser()$;
25. $aUser() = [] \ pos : \{0..NoOfFloor-1\}@(ExternalPush(pos); \ Waiting(pos))$;
26. $ExternalPush(pos) =$ **case** $\{$
27. $\quad pos == 0 : \ pushup.pos\{extUpReq[pos] = 1\} \rightarrow Skip$
28. $\quad pos == NoOfFloor - 1 \ : \ pushdown.pos\{extDownReq[pos] = 1\} \rightarrow Skip$
29. $\quad$ **default** $: \ pushup.pos\{extUpReq[pos] = 1\} \rightarrow Skip \ []$
30. $\quad\quad\quad pushdown.pos\{extDownReq[pos] = 1\} \rightarrow Skip$
31. $\quad \};$
32. $Waiting(pos) = [] \ i : \{0..NoOfLift-1\}@([doorOpen[i] == pos]enter.i \rightarrow$
33. $\quad []x : \{0..NoOfFloor-1\}@(push.x\{intRequests[i][x] = 1\} \rightarrow$
34. $\quad [doorOpen[i] == x]exit.i.x \rightarrow User()))$;
35. $LiftSystem() = Users() \ ||| \ (||| \ x : \{0..NoOfLift-1\}@Lift(x, 0, 1))$;
36. #**assert** $LiftSystem()$ **deadlockfree**;
37. #**define** $pr1 \ extUpReq[0] > 0$;
38. #**define** $pr2 \ extUpReq[0] == 0$;
39. #**assert** $LiftSystem() \ |= \ \Box(pr1 \Rightarrow \Diamond pr2) \ \&\& \ \Box\Diamond moving.0$

Figure 5. CSP# model of the lifts system

more, we formally define the semantic models for CSP#, which facilitates PAT to perform sound and complete system verification. One future research direction is to continue our previous works on verifying real-time systems [6, 5] by introducing clock variables in CSP# and extend PAT to handle simple real variables.

References

[1] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An Operational Semantics for CSP. Technical report, 1986.

[2] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *IFM'04: Proc. of the 4th Inter. Conf. on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 128–147, 2004.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV'02: Proc. of the 14th Inter. Conf. on Computer Aided Veri.*, pages 359–364, 2002.

[4] E. W. Dijkstra. Programming: From Craft to Scientific Discipline. In *International Computing Symposium 1977*, pages 23–30, 1977.

[5] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Y. Wang. Timed automata patterns. *IEEE Transsactions on Software Engineering*, 34(6):844–859, 2008.

[6] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM'06*.

[7] C. Fischer. CSP-OZ: a combination of object-Z and CSP. In *FMOODS'97*.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[9] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[10] S. Y. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engeering*, 24(1):24–45, 1998.

[11] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[12] B. P. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In *ZUM'98*.

[13] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[14] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[15] A. W. Roscoe. Compiling Shared Variable Programs into CSP. In *Proceedings of PROGRESS workshop 2001*, 2001.

[16] A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17(2):93–112, 2005.

[17] S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[18] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISOLA'08*.

[19] J. Sun, Y. Liu, J. S. Dong, and H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *ICFEM'08*, pages 318–337, 2008.

[20] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203, 2002.

Appendix: Operational Semantics where $e \in \Sigma$; $e_\tau \in \Sigma \cup \{\tau\}$; $x \in \Sigma \cup \{\checkmark\}$ and $* \in \Sigma \cup \{\tau, \checkmark\}$

$$\frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Stop)} [\ skip\ ]$$

$$\frac{(V,P) \xrightarrow{e} (V',P'), e \in X}{(V, P \setminus X) \xrightarrow{\tau} (V', P')} [\ hide1\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P'), x \notin X}{(V, P \setminus X) \xrightarrow{x} (V', P' \setminus X)} [\ hide2\ ]$$

$$\frac{(V,P) \xrightarrow{e_\tau} (V',P')}{(V, P;\ Q) \xrightarrow{e_\tau} (V', P';\ Q)} [\ seq1\ ]$$

$$\frac{(V,P) \xrightarrow{\checkmark} (V',P')}{(V, P;\ Q) \xrightarrow{\tau} (V', Q)} [\ seq2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V, P \,\square\, Q) \xrightarrow{x} (V', P')} [\ ch1\ ]$$

$$\frac{(V,Q) \xrightarrow{x} (V',Q')}{(V, P \,\square\, Q) \xrightarrow{x} (V', Q')} [\ ch2\ ]$$

$$\frac{(V,P) \xrightarrow{\tau} (V',P')}{(V, P \,\square\, Q) \xrightarrow{x} (V', P' \,\square\, Q)} [\ ch3\ ]$$

$$\frac{(V,Q) \xrightarrow{\tau} (V',Q')}{(V, P \,\square\, Q) \xrightarrow{\tau} (V', P \,\square\, Q')} [\ ch4\ ]$$

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, P)} [\ non1\ ]$$

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, Q)} [\ non2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V, P \,|||\, Q) \xrightarrow{x} (V', P' \,|||\, Q)} [\ int1\ ]$$

$$\frac{(V,Q) \xrightarrow{x} (V',Q')}{(V, P \,|||\, Q) \xrightarrow{x} (V', P \,|||\, Q')} [\ int2\ ]$$

$$\frac{(V,P) \xrightarrow{\checkmark} (V',P'), (V,Q) \xrightarrow{\checkmark} (V',Q')}{(V, P \,|||\, Q) \xrightarrow{\checkmark} (V', P' \,|||\, Q')} [\ int3\ ]$$

$$\frac{(V,P) \xrightarrow{*} (V',P')}{(V, P \triangle Q) \xrightarrow{*} (V', P' \triangle Q)} [\ inter1\ ]$$

$$\frac{(V,Q) \xrightarrow{e} (V',Q')}{(V, P \triangle Q) \xrightarrow{e} (V', Q')} [\ inter2\ ]$$

$$\frac{(V,Q) \xrightarrow{\tau} (V',Q')}{(V, P \triangle Q) \xrightarrow{\tau} (V', P \triangle Q')} [\ inter3\ ]$$