The United Nations
University

UNU/IIST

**International Institute for
Software Technology**

Limited Resource Models and Specifications for Programming Languages

Jin Naiyong and He Jifeng

UNU/IIST Report No. 277

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2003

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Limited Resource Models and Specifications for Programming Languages

Jin Naiyong and He Jifeng

Abstract

Traditional system specifications always abstract from resource. However, it is unavoidable to append resource constraints in later system implementation.This paper defines a hierarchy of semantic models for programm's execution with limited resources. Each model in the hierarchy is a refinement to the one defined formerly. We also investigate homomorphism and algebraic laws in these models. Based on that, we develop a set of program transformations to deal with the situation that resource are in shortage.

He Jifeng is a Senior Research Fellow of UNU/IIST. He is also a professor of Computer Science Department at East China Normal University and Shanghai Jiao Tong University. His research interests include the mathematical theory of programming and refinement methods, the techniques for design of the mixed software/hardware systems. His email is jifeng@iist.unu.edu

Jin Naiyong is a Fellow of UNU/IIST. He is also a PhD. student at The East China Normal University , Shanghai China. His research interest includes programming theory, formal methods application, formal language specification and compiler prototype. His email address is nyjin@iist.unu.edu

Contents

1 Introduction **1**

2 Programming Language **3**

3 A Primitive Resource Semantic Model **5**
3.1 Alphabet for Designs with Resources 5
3.2 The Primitive Resource Semantic Model 7

4 Algebraic Laws for the Primitive Resource Semantic Model **8**

5 Reusable Resource Models **10**

6 Resource Models with Garbage Collection **12**

7 Resource-Oriented Program Transformation **15**

8 Multiplexors for Reusable Resource **18**

9 Conclusion **20**

References **21**

1 Introduction

Electronic equipments are used extensively in modern applications. Failures of these systems can be costly or even fatal, especially for real-time systems. System correctness concerns both functional and temporal behavior. For a program to be implemented and executed, it needs sufficient resources. However, system specification abstract from these resources and assume they are always available. It is to simplify the analysis and design of a system. While it is unavoidable to address the resource constraints of a design in the implementation stage, for instance, the temporal behavior depends not only on its communication with other processes, but also on resources features provided by the environment.

Shade and Narayana [8] [9] developed a compositional, denotational semantic model which considers limited processing resource. Their semantic model takes into account the number of processors, relative speeds of processors and shared memory access. Gerber and Lee [2] [3] proposed a language, the Calculus of Communicating Shared Resources, reasoning about resource requirement in real-time systems. Their model is operational, with priorities, and in which the transition becomes enabled only if they do not cause a resource conflict. Goswami and Joseph's model [5] [4] also gave a compositional denotational semantic model which differs from that of Shade and Narayana by confining to local memory. One common point of these models are that the existence of resources and corresponding schedulers are independent of running processes, resources may be occupied by certain process, but will not be eliminated by execution of processes.

Gavin Lowe [7] built a hierarchy of scheduling-oriented models based on TAM, in which resource allocation is consigned to a fixed (compiler-time) scheduler.He also modelled the fact of a failure of an agent, either because of insufficiently available resources or failures in meeting a deadline. But he didn't mention how to represent a failure in his semantic model and how to perform a behavior-preserving transformation towards a program into one with less resource required. That is one of the topics addressed in this paper.

We view another aspect of the limited resource problem. The execution of a system consumes resources, available resources may get less and less. Probably, at a moment, the environment cannot run a system any more. If that happens, the system must be suspended, or even worse, aborted. It is quite common for us to face such kind of dilemma in our daily life. To prevent those bad things, people shall adjust their ways in using resources. For instance, one may try to recycle resources, or delegate resource hunger tasks to subtasks, and often share resources with others. This paper is going to formalize these treatment towards resources.

In order to analysis the limited resource problem systematically, we must answer four questions. 1) How to include the reusable resource information to semantic models? 2) How to process reusable resource? 3) How to convert a program to one requiring less resources? 4) How to provide reuse mechanism for hardware compiler design ?

The first problem has two challenges. First, newly added information shall not impact the

functional specification of the original program. So resource information shall be independent on other program variables. Second, we shall depict that the failures in satisfying the resource requirement of a program is the worst thing. That leads us to develop a primitive model in which checking and modifications to environment resources are in parallel with the execution of a program, and resource constraint violation will results in chaos, the worst in the predicate lattice. As any thing parallel with a chaos turns to be a chaos, so the general behavior of a system is degraded to chaos if environment cannot satisfy required resource of a program. That captures the common character of resource models. The method we adopt for this primitive model provides a good framework for further investigation.

The second question comes from synthesis tools for integrated circuit design. The electric current runs along wires, registers and transistors in parallel. If a resource is reusable, it is prohibited to use it concurrently. As stated in [11], resource allocation, scheduling and mapping are main functions to behavior synthesis. We enhance our resource model by introducing the concept of reuse. Yet the reusability also brings us non-determinism at our first try. That drives us to introduce garbage collection scheme to make explicit distinction between allocatable and recycled resources of the environment. Further more, we model the timing for turning resources in recycle bin to be allocatable again. After removing all non-determinism, we have an program for resource reuse.

Though with fast advances of technology it is easy to get giga-counted electronic equipment, the execution of a program may still run out of resource. In case that,we shall transform the program so that it is executable under limited resource environment. Resource models offer consolidated foundation for correct program transformations. That is our solution to the third question.

We meet with the fourth question when we construct hardware compiler for reusable resources. When like-named circuits resulting from software translation are put together, we must ensure that at any time only one electronic current drives the circuits and the result flows to right destination. That is the job of multiplexors. We propose a specification on how to build correct multiplexors for reusable resources.

We organize this paper as below. In Section 1, we introduce our programming language and necessary notations which will be used. After that, in Section 2, we develop a primitive semantic model in which no resource can be reused. In Section 3, we study the partial order for comparing two programs with resources, that results in a set of algebraic laws and inequations. In Section 4, we extend the model with information regarding reusable resource, and in Section 5 we discuss how to embed garbage collection scheme and make refinement to reusable resource models. In Section 6, we propose a set of program transformation rules to get less-resource-requiring program. The Section 7 is about specifications for multiplexors. In the end, we give a brief summary and discuss some further research directions.

2 Programming Language

Our language follows the convention of [6], and we confine ourself to "designs" defined in [6]. The syntax of a program can be in forms below

Definition 2.1

$$\begin{aligned} P ::= & \mathit{skip} \mid \ \bot \mid \mathit{wait}(n) \\ & \mid x := \mathit{Exp} \mid P;\ P \mid P \lhd B \rhd P \mid B * P \\ & \mid P \parallel Q \mid \textbf{var}\ x \mid \textbf{end}\ x \mid P \sqcap Q \end{aligned}$$

These terms stand for the following programs:
skip is a program which terminates immediately with all variables unchanged,
$\bot$ is the live-lock process, its behavior is totally unpredictable,
$\mathit{wait}(n)$ denotes a program suspended for n circles, and then terminates,
var x is the declaration of a new variable x,
end x is the declaration to end the use of the variable x,
$x := \mathit{Exp}$ is the assignment, where the expression *Exp* is supposed to have same data type as the variable x,
$P;\ Q$ is the sequential composition of processes P and Q,
$P \lhd B \rhd Q$ is the conditional, if B is evaluated to be true, it will behave like P,
otherwise it will behave like Q,
$B * P$ is a process which evaluates B first, if B is true it executes $(P;\ B * P)$, otherwise it terminates successfully and nothing changed,
$P \| Q$ is the parallel composition of P and Q which do not share variables
$P \sqcap Q$ is the non-deterministic choice between P and Q.

$[\![P]\!]$ stands for the semantic of a program P. In convention of [6], $[\![P]\!]$ is represented as $A_P \vdash C_P$, here A_P is the assumption, C_P is the commitment, and use αP for alphabet used in $[\![P]\!]$. By this, We have

1. $[\![\bot]\!] =_{df} \textbf{false} \vdash \textbf{false}$
2. $[\![\mathit{skip}]\!] =_{df} \textbf{true} \vdash x' = x \wedge y' = y \wedge ... \wedge z' = z$
3. $[\![\textbf{var}\ x]\!] =_{df} \exists x \cdot [\![\mathit{skip}]\!]$
4. $[\![\textbf{end}\ x]\!] =_{df} \exists x' \cdot [\![\mathit{skip}]\!]$
5. $[\![x := \mathit{Exp}]\!] =_{df} \textbf{true} \vdash x' = \mathit{Exp} \wedge y' = y \wedge ... \wedge z' = z$
6. $[\![P_1;\ P_2]\!] =_{df} \exists v_0 \cdot [\![P_1]\!](v_0/v') \wedge [\![P_2]\!](v_0/v)$
7. $[\![P_1 \lhd B \rhd P_2]\!] =_{df} (B \wedge [\![P_1]\!]) \vee (\neg B \wedge [\![P_2]\!])$

8. $[\![P_1 \sqcap P_2]\!] =_{df} (A_{P1} \wedge A_{P2}) \vdash (C_{P1} \vee C_{P2})$
9. $[\![B * P]\!] =_{df} \mu X \cdot ([\![P]\!];\ X) \lhd B \rhd [\![skip]\!]$
10. $[\![P_1 \| P_2]\!] =_{df} (A_{P1} \wedge A_{P2}) \vdash (C_{P1} \wedge C_{P2})$

Suppose $\alpha P = \alpha Q$, we define an refinement relation $\sqsubseteq$ between programs P and Q that holds if Q is as good as or better than P, i.e. $P \sqsubseteq Q =_{df} [[\![P]\!] \Leftarrow [\![Q]\!]]$.

Lemma 2.2

$$[A_P \vdash C_P \ \Leftarrow\ A_Q \vdash C_Q] \ \textbf{iff}\ [A_P \Rightarrow A_Q] \wedge [(A_P \wedge C_Q) \Rightarrow C_P]$$

We say two programs are equal if they are refinement to each other. That is $(P = Q) \equiv (P \sqsubseteq Q \wedge Q \sqsubseteq P)$.

So we have

Lemma 2.3

$$[\![P;\ Q]\!] = (A_P \wedge \neg(C_P;\ \neg A_Q)) \vdash (C_P;\ C_Q)$$

Usually, we write **var** $x := e$ in stead of **var** x; $x := e$. Following laws will be used in later discussion,

L1. (**end** x; **var** $x := e$) = $x := e$

L2. $((x = v)_\perp;\ x := v) \sqsubseteq$ *skip*
where $b_\perp =_{df}$ *skip* $\lhd b \rhd \perp$

L3. if $x \notin \alpha R$,
end x; R = R; **end** x **var** x; R = R; **var** x

L4. if $x \notin \alpha Q$
var x; $P \| Q$; **end** x = (**var** x; P; **end** x)$\| Q$

L5. (**end** v; **var** v) $\sqsubseteq$ *skip* = (**var** v; **end** v)

L6. ($x := e$; **end** x) = **end** x

L7. $(P;\ Q) \| (R;\ S) = (P \| R);\ (Q \| S)$ if $\alpha P \cap \alpha Q = \phi$

We adopt basic data types, integer and boolean for our language.

Definition 2.4

$$
\begin{aligned}
\mathbf{Exp} &::= \mathit{IntExp} \mid \mathit{BoolExp} \\
\mathit{IntExp} &::= \mathit{IntNum} \mid \mathit{IntVar} \mid \mathit{IntExp}\ \mathit{int.operator}\ \mathit{IntExp} \\
\mathit{BoolExp} &::= 1 \mid 0 \mid \mathit{BoolVar} \mid \neg \mathit{BoolExp} \\
&\mid \mathit{BoolExp}\ \mathit{bool.operator}\ \mathit{BoolExp} \\
&\mid \mathit{IntExp}\ \mathit{rel.operator}\ \mathit{IntExp} \\
\mathit{int.operator} &::= + \mid - \mid \times \mid \mathsf{div} \\
\mathit{bool.operator} &::= \vee \mid \wedge \mid \oplus \\
\mathit{rel.operator} &::= < \mid \leq \mid > \mid \geq \mid \neq \mid =
\end{aligned}
$$

3 A Primitive Resource Semantic Model

3.1 Alphabet for Designs with Resources

We introduce a new data type **RES**, whose element is a set of pairs $(\mathit{Op}, \mathit{Number})$, where the first component **Op** is the type of resource items, like *adder*, *comparator*, etc. And *Number* records the amount of resource items of type **Op** in the set. If R is a variable of type **RES**, function $\mathit{Res}(R, t) : \mathbf{RES} \times \mathbf{Op} \to \mathbf{N}$ gives the number of resource item t which R has. The comparison of resource is defined as

$$R_1 \subseteq R_2 \ =_{df}\ \forall\, t \cdot \ \mathit{Res}(R_1, t) \ \leq\ \mathit{Res}(R_2, t)$$

The operations $\cup$, $\cap$, $\setminus$ and $\uplus$ on **RES** are defined as

Definition 3.1

$$
\begin{aligned}
\mathit{Res}(R_1 \cup R_2, t) &=_{df}\ \mathit{Res}(R_1, t) \ +\ \mathit{Res}(R_2, t) \\
\mathit{Res}(R_1 \cap R_2, t) &=_{df}\ \min\{\ \mathit{Res}(R_1, t)\ ,\ \mathit{Res}(R_2, t)\ \} \\
\mathit{Res}(R_1 - R_2, t) &=_{df}\ \max\{\ \mathit{Res}(R_1, t)\ -\ \mathit{Res}(R_2, t)\ , 0\} \\
\mathit{Res}(R_1 \uplus R_2, t) &=_{df}\ \max\{\ \mathit{Res}(R_1, t)\ ,\ \mathit{Res}(R_2, t)\ \}
\end{aligned}
$$

We have all definitions and theorems related to the primitive model with a subscription $r0$. Function $\mathit{Req}_{r0}(\mathit{Exp})$: **Exp** $\to$ **RES** maps an expression to the collection of resources its evaluation requires.

Definition 3.2

$Req_{r0}(Exp) \ =_{df} \ \{(op, n) \mid$ 'op' appears as an operator n times in expression $Exp\}$

Theorem 3.3

1. $Req_{r0}(C) \ = \ \phi$, here C is a numeric constant of type Bool and Int
2. $Req_{r0}(IntExp_1 \ + \ IntExp_2) \ = \ \{(adder, 1)\} \ \cup Req_{r0}(IntExp_1) \ \cup Req_{r0}(IntExp_2)$ this also applies to other binary operators, like $-$, $\times$, div, $<$, $\vee$, $\wedge$, etc.
3. $Req_{r0}(\sim BoolExp) \ = \ \{(not, 1)\} \ \cup Req_{r0}(BoolExp)$

And the function $Req_{r0}(P) \ :$ **Program** $\rightarrow$ **RES** maps the program P to its resources.

Definition 3.4

$Req_{r0}(P) \ =_{df} \ \{(op, n) \mid$ 'op' appears as an resource n times in program $P\}$

Theorem 3.5

1. $Req_{r0}(Skip) \ = \ \phi$
2. $Req_{r0}(Wait(n)) \ = \ \phi$
3. $Req_{r0}(x := Exp) \ = Req_{r0}(Exp)$
4. $Req_{r0}(P; \ \ Q) \ = Req_{r0}(P) \ \cup Req_{r0}(Q)$
5. $Req_{r0}(P \lhd B \rhd Q) \ = Req_{r0}(P) \ \cup Req_{r0}(Q) \ \cup Req_{r0}(B)$
6. $Req_{r0}(B * P) \ = Req_{r0}(P) \ \cup Req_{r0}(B)$
7. $Req_{r0}(P \sqcap Q) \ = Req_{r0}(P) \ \cup Req_{r0}(Q)$
8. $Req_{r0}(P \parallel Q) \ = Req_{r0}(P) \ \cup Req_{r0}(Q)$
9. $Req_{r0}(\bot) \ = \ \mathbf{Res}$
10. $Req_{r0}(\mathbf{var} \ x) \ = \ \phi$

11. $Req_{r0}(\textbf{end}\ x) = \phi$

Since $\perp$ is totally nondeterministic, it is impossible to estimate how much resource it requires, so we could only assume that it uses unlimited resources **Res**.

We have variable *Rsys* to stand for resources provided by the environment.

3.2 The Primitive Resource Semantic Model

For discussing the behavior of the program under the resource constraint, we use $[\![P]\!]_{r0}$ to denote the meaning of P in the primitive model.

Definition 3.6

$$[\![P]\!]_{r0} =_{df} [\![P]\!] \| (Req_{r0}(P) \subseteq Rsys \vdash R'sys \supseteq Rsys - Req_{r0}(P))$$

This definition states that to run a program, the environment shall provide enough resource to meet its requirement, otherwise the program will turn to be chaotic. And on termination of the program, used resources shall be no less than what required.

If we relax resource constraints by assuming there are unlimited resources, then the program is always a feasible one. Then we needn't concern the resource problem.

Theorem 3.7

$$[\![P]\!] = \textbf{var}\ Rsys := \textbf{Res};\ [\![P]\!]_{r0};\ \textbf{end}\ Rsys$$

Proof :

$$\begin{aligned}
RHS &= \{By\ definition\ of\ sequential\} \\
&\quad \textbf{var}\ Rsys;\ [\![P]\!]_{r0}[\textbf{Res}/Rsys];\ \textbf{end}\ Rsys \\
&= \{By\ definition\ of\ [\![P]\!]_{r0}\} \\
&\quad \textbf{var}\ Rsys;\ [\![P]\!] \| Req_{r0}(P) \subseteq \textbf{Res} \vdash R'sys \supseteq \textbf{Res} - Req_{r0}(P)\ ;\ \textbf{end}\ Rsys \\
&= \{By\ (Req_{r0}(P) \subseteq \textbf{Res}) = \textbf{true}\ and\ properties\ of\ \textbf{Res}\} \\
&\quad \textbf{var}\ Rsys;\ [\![P]\!] \| \textbf{true} \vdash R'sys \supseteq \textbf{Res} - Req_{r0}(P)\ ;\ \textbf{end}\ Rsys \\
&= \{By\ L4.\} \\
&\quad [\![P]\!] \| (\textbf{var}\ Rsys;\ \textbf{true} \vdash R'sys \supseteq \textbf{Res} - Req_{r0}(P)\ ;\ \textbf{end}\ Rsys) \\
&= \{By\ L6.\ L5.\} \\
&\quad [\![P]\!]
\end{aligned}$$

□

To explore further properties of programs under the resource model, we must ensure its homomorphism regarding general program constructs.

Theorem 3.8

1. $[\![P;\ Q]\!]_{r0} = [\![P]\!]_{r0};\ [\![Q]\!]_{r0}$
2. $[\![P \lhd B \rhd Q]\!]_{r0} = [\![P]\!]_{r0} \lhd [\![B]\!]_{r0} \rhd [\![Q]\!]_{r0}$
3. $[\![P \sqcap Q]\!]_{r0} = [\![P]\!]_{r0} \sqcap [\![Q]\!]_{r0}$
4. $[\![P \| Q]\!]_{r0} = [\![P]\!]_{r0} \| [\![Q]\!]_{r0}$

Proof :1

$$
\begin{array}{ll}
 & [\![P]\!]_{r0};\ [\![Q]\!]_{r0} \\
= & \{By\ L7.\ and\ Lemma\ 2.3\} \\
 & ([\![P]\!];\ [\![Q]\!]) \| ((Req_{r0}(P)\ \subseteq Rsys) \wedge \neg((R'sys \supseteq Rsys - Req_{r0}(P)\ );\ \neg(Req_{r0}(Q)\ \subseteq Rsys)) \\
 & \quad \vdash (R'sys \supseteq Rsys - Req_{r0}(P)\ );\ (R'sys \supseteq Rsys - Req_{r0}(Q)\ )) \\
= & \{By\ definition\ of\ sequential\} \\
 & ([\![P]\!];\ [\![Q]\!]) \| ((Req_{r0}(P)\ \subseteq Rsys) \wedge \neg((R'sys \supseteq Rsys - Req_{r0}(P)\ );\ (Req_{r0}(Q)\ \not\subseteq Rsys)) \\
 & \quad \vdash (R'sys \supseteq Rsys - Req_{r0}(P)\ - Req_{r0}(Q)\ )) \\
= & ([\![P]\!];\ [\![Q]\!]) \| ((Req_{r0}(P)\ \subseteq Rsys) \wedge (Rsys - Req_{r0}(P)\ \supseteq Req_{r0}(Q)\ )) \\
 & \quad \vdash (R'sys \supseteq Rsys - Req_{r0}(P)\ - Req_{r0}(Q)\ )) \\
= & ([\![P]\!];\ [\![Q]\!]) \| ((Req_{r0}(P)\ \cup Req_{r0}(Q)\ \subseteq Rsys) \vdash (R'sys \supseteq Rsys - Req_{r0}(P)\ - Req_{r0}(Q)\ )) \\
= & \{By\ theorem\ 2.5\} \\
 & [\![P;\ Q]\!]_{r0}
\end{array}
$$

All others can be proved in the same way. □

4 Algebraic Laws for the Primitive Resource Semantic Model

To distinguish the general non-resource refinement partial order $\sqsubseteq$, we use $\sqsubseteq_{r0}$ for those with resource.

Theorem 4.1

$$P \sqsubseteq_{r0} Q \equiv [[\![P]\!] \Leftarrow [\![Q]\!]] \wedge (Req_{r0}(Q) \subseteq Req_{r0}(P))$$

Proof:

$$\begin{aligned}
& P \sqsubseteq_{r0} Q \\
\equiv\ & [A_P \wedge Req_{r0}(P) \subseteq Rsys \vdash C_P \wedge R'sys \supseteq Rsys - Req_{r0}(P) \\
& \Leftarrow A_Q \wedge Req_{r0}(Q) \subseteq Rsys \vdash C_Q \wedge R'sys \supseteq Rsys - Req_{r0}(Q)\ ] \\
\equiv\ & \{By\ Lemma\ 2.2\} \\
& [(A_P \wedge Req_{r0}(P) \subseteq Rsys) \Rightarrow (A_Q \wedge Req_{r0}(Q) \subseteq Rsys) \\
& \wedge ((A_P \wedge Req_{r0}(P) \subseteq Rsys) \wedge (C_Q \wedge R'sys \supseteq Rsys - Req_{r0}(Q))) \\
& \quad \Rightarrow (C_P \wedge R'sys \supseteq Rsys - Req_{r0}(P))] \\
\equiv\ & [A_P \Rightarrow A_Q] \wedge [Req_{r0}(P) \subseteq Rsys \Rightarrow Req_{r0}(Q) \subseteq Rsys] \wedge [A_P \wedge C_Q \Rightarrow C_P] \\
\equiv\ & [[\![P]\!] \Leftarrow [\![Q]\!]] \wedge (Req_{r0}(Q) \subseteq Req_{r0}(P))
\end{aligned}$$

□

This theorem states that if Q is a refinement of P, besides Q implies P, Q shall also use less resource than P does, as the feasibility of a program requiring more-resource can always implies that of a program requiring less resource.

With the new resource information, it is necessary for us to review algebraic laws of programs. The partial order brings us many inequations of designs which are equations under non-resource model. The refinement is due to the removing of duplicated programs, and resulting in less resource required.

Theorem 4.2 *(Conditional)*

1. $P \lhd B \rhd P \sqsubseteq_{r0} P$
2. $P \lhd B \wedge C \rhd (Q \lhd C \rhd R) \sqsubseteq_{r0} (P \lhd B \rhd Q) \lhd C \rhd R$
3. $(P \lhd B \rhd Q) \lhd C \rhd (P \lhd B \rhd Q) \sqsubseteq_{r0} P \lhd B \rhd (Q \lhd C \rhd R)$
4. $P \lhd true \rhd Q \sqsubseteq_{r0} P$ and $Q \lhd false \rhd P \sqsubseteq_{r0} P$
5. $P \lhd B \rhd (Q \lhd B \rhd R) \sqsubseteq_{r0} P \lhd B \rhd R$

Proof: of 1. we only need to show that $Req_{r0}(P) \subseteq Req_{r0}(P \lhd B \rhd P)$

$$\begin{aligned}
Req_{r0}(P \lhd B \rhd P) &= Req_{r0}(B) \cup Req_{r0}(P) \cup Req_{r0}(P) \\
&\supseteq Req_{r0}(P)
\end{aligned}$$

□

Theorem 4.3 *(Sequential Composition)*

1. $(P;\ R) \lhd B \rhd (Q;\ R) \sqsubseteq_{r0} (P \lhd B \rhd Q);\ R$

Theorem 4.4 *(Internal Choice)*

1. $P \sqcap P \sqsubseteq_{r0} P$
2. $(P \sqcap Q) \sqcap (P \sqcap R) \sqsubseteq_{r0} P \sqcap (Q \sqcap R)$
3. $(P \lhd B \rhd Q) \sqcap (P \lhd B \rhd R) \sqsubseteq_{r0} P \lhd B \rhd (Q \sqcap R)$
4. $(P;\ R) \sqcap (Q;\ R) \sqsubseteq_{r0} (P \sqcap Q);\ R$
5. $(P;\ Q) \sqcap (P;\ R) \sqsubseteq_{r0} P;\ (Q \sqcap R)$
6. $(P \sqcap Q) \lhd B \rhd (P \sqcap R) \sqsubseteq_{r0} P \sqcap (Q \lhd B \rhd R)$

Theorem 4.5 *(Parallel)*

1. $(P \| R) \lhd B \rhd (Q \| R) \sqsubseteq_{r0} (P \lhd B \rhd Q) \| R$
2. $(P \| R) \sqcap (Q \| R) \sqsubseteq_{r0} (P \sqcap Q) \| R$

5 Reusable Resource Models

The primitive resource semantic model presents us many essential properties of a program with resource constraint. But it is still not satisfactory. The most shortcoming is that it is an uneconomical model, resource consumption in that way will make us run out of budget easily. One solution is to introduce the concept of reusability.

Before extending our theory, we shall first modify the resource calculation formula to reflect characters of reusable resource. For that, we introduce another three functions $Cons(P)$: **Program** $\rightarrow$ **RES**, $Reus(P)$: **Program** $\rightarrow$ **RES** and $Req_{r1}(P)$: **Program** $\rightarrow$ **RES**. $Cons(P)$ gives us those resources which are consumed by the execution of P, $Reus(P)$ gives the rest of those resource which are reusable by P's successors. $Req_{r1}(P)$ returns all resources required by P, for its different components from $Req_{r0}(P)$, we give it a new subscription.

Definition 5.1

1. $Req_{r1}(P) = \{(op, n) \mid n$ instances of resource op being used in $P\}$
2. $Cons(P) = \{(op, n) \mid (op, n) \in Req_{r1}(P) \ \wedge \ op$ is consumed by the execution of $P\}$
3. $Reus(P) = \{(op, n) \mid (op, n) \in Req_{r1}(P) \ \wedge \ op$ is reusable after termination of $P\}$

From this definition, we conclude that one resource item can't be both consumptive and reusable that is

Theorem 5.2

1. $Req_{r1}(P) = Cons(P) \cup Reus(P)$
2. $Cons(P) \cap Reus(P) = \phi$

The semantic model will be changed after introducing the distinction between consumptive and reusable resource.

Definition 5.3

$$[\![P]\!]_{r1} =_{df} [\![P]\!] \| Req_{r1}(P) \subseteq Rsys \vdash \left(\begin{array}{ll} & R'sys \supseteq Rsys - Req_{r1}(P) \\ \wedge & R'sys \subseteq Rsys - Cons(P) \end{array} \right)$$

This definition gives a range for available resources on termination of program P, that is actually used resources shall be not only less than what required, but also no less than $Cons(P)$. The existence of the range is due to the absence of specification on how to process reusable resources, which may not be reused though they are claimed to be reusable.

By induction on program constructs, we have the fact $Req_{r1}(P) \subseteq Req_{r0}(P)$ which leads us to

Theorem 5.4

$$[\![P]\!]_{r0} \Leftarrow [\![P]\!]_{r1}$$

Which means the new model is a refinement of the primitive resource model.

To investigate properties of the reusable resource model, we must have a new partial order,

Theorem 5.5

$$P \sqsubseteq_{r1} Q \equiv [[[P]] \Leftarrow [[Q]]] \wedge (Req_{r1}(P) \supseteq Req_{r1}(Q)) \wedge (Cons(P) \subseteq Cons(Q))$$

This theorem may seems contra-intuitive, it discourages us to have reusable resource. Yet it is right, as the absence of explicit specification on reuses brings non-determinism to resource model, more reusable resources contributes more non-determinism. Hence it demands for further refinement in later section.

6 Resource Models with Garbage Collection

The concept of reusability has two aspects, resource release and resource allocation. Resource release is to return resources to environment or to think it as the environment collects reusable resources. Yet it doesn't means recycled resource can be allocated again immediately. There are different options on this issue. For example, it is common to have garbage collection schema in operating system and JVM for memory administration. We put the reuse control schema to the environment behavior. In light of this, we introduce two new environment variables of resource data type, $Gsys$ which bears resources in a recycle bin, and $Asys$ which is the allocatable resource. The union of $Asys$ and $Gsys$ is the available resource.

As stated before, the reusable model above introduces non-determinism. They are the non-determinism on the *de facto* reuse and the non-determinism on when to reuse. The reason why some reusability not be exploited or not be exploited immediately is that we may pay cost for reuse, these cost may appear in several forms, for instance, extra resources or reduction in temporal behavior. We solve these problems one by one.

First we assume all reusable resources are worth reusing. That is the extra resource cost is tiny, so all reusable resource can be reused. We set that cost to $Cons(P)$. Thus we have a new model

Definition 6.1

$$\begin{array}{lll} [[P]]_{r2} =_{df} & [[P]] & \\ & \| Req_{r1}(P) \subseteq Asys \cup Gsys \vdash & (A'sys \supseteq Asys - Req_{r1}(P) \\ & & \wedge G'sys \supseteq Reus(P) \\ & & \wedge A'sys \cup G'sys = Asys \cup Gsys - Cons(P)) \end{array}$$

This definition says that if resource constraint is not violated, on termination of program P, the actually consumed resource is just $Cons(P)$.

Theorem 6.2

$$P \sqsubseteq_{r2} Q \equiv [[\![P]\!] \Leftarrow [\![Q]\!]] \wedge (Req_{r1}(P) \supseteq Req_{r1}(Q)\ ) \wedge (\ Reus(P) \subseteq Reus(Q)\ )$$

The new model $[\![P]\!]_{r2}$ is more deterministic. But because of different alphabet with $[\![P]\!]_{r1}$, we cannot compare them directly. The gap can be bridged by a simulation pair (U, D).

Definition 6.3

$$S =_{df} \mathbf{var} Asys, Gsys;\ Asys := Rsys;\ Gsys := \phi;\ \mathbf{end} Rsys$$
$$T =_{df} \mathbf{var} Rsys;\ Rsys := Asys \cup Gsys;\ \mathbf{end} Asys, Gsys$$

The simulation functions are

Definition 6.4

$$U(P) =_{df} S;\ P \qquad D(P) =_{df} P;\ T$$

The next theorem shows us that the new model is more dedicated than the previous one.

Theorem 6.5

$$[\![P]\!]_{r1} \sqsubseteq_{r1} D(U([\![P]\!]_{r2})) \qquad U(D([\![P]\!]_{r1})) \sqsubseteq_{r2} [\![P]\!]_{r2}$$

Proof:

$$
\begin{array}{ll}
 & D(U([\![P]\!]_{r2})) \\
= & \{By\ L3.\} \\
 & \textbf{var}\ Asys := Rsys, Gsys := \phi;\ [\![P]\!]_{r2}; \\
 & \textbf{end}\ Rsys;\ \textbf{var}\ Rsys := Asys \cup Gsys; \\
 & \textbf{end}\ Asys, Gsys \\
= & \{By\ L1.\} \\
 & \textbf{var}\ Asys := Rsys, Gsys := \phi;\ [\![P]\!]_{r2}; \\
 & Rsys := Asys \cup Gsys;\ \textbf{end}\ Asys, Gsys \\
= & \{By\ sequential\ and\ parallel\ composition\} \\
 & \textbf{var}\ Asys, Gsys; \\
 & [\![P]\!] \| Req_{r1}(P)\ \subseteq Rsys \vdash R'sys = Rsys - Cons(P) \\
 & \| \textbf{true}\ \vdash \left(\begin{array}{l} A'sys \supseteq Rsys - Req_{r1}(P) \\ \wedge\ G'sys \supseteq\ Reus(P) \\ \wedge\ A'sys \cup G'sys = Rsys - Cons(P) \end{array} \right) \\
 & \textbf{end}\ Asys, Gsys \\
= & \{By\ Definiton\ of\ parallel\ ,\ L3.\ and\ L7.\} \\
 & [\![P]\!] \| Req_{r1}(P)\ \subseteq Rsys \vdash R'sys = Rsys - Cons(P) \\
 & \textbf{var}\ Asys, Gsys;\ \textbf{true}\ \vdash (...);\ \textbf{end}\ Asys, Gsys \\
\sqsupseteq_{r1} & \{By\ L6.,\ L5.\ and\ Cons(P)\ \subseteq Req_{r1}(P)\ \} \\
 & [\![P]\!]_{r1}
\end{array}
$$

□

Secondly, we'd like to address the timing of reuse. As our resource models are un-timed, we can't compare programs' temporal behavior. We postpone emptying recycle bin until it is unavoidable. That is

Definition 6.6

$$
\begin{array}{lll}
[\![P]\!]_{r3} & =_{df} & [\![P]\!] \| \\
 & & Req_{r1}(P)\ \subseteq Asys \cup Gsys \vdash \\
 & & (\quad (Req_{r1}(P)\ \subseteq Asys) \\
 & & \qquad \wedge (A'sys = Asys - Req_{r1}(P)\ ) \wedge (G'sys = Gsys \cup\ Reus(P)\ ) \\
 & & \vee\quad (Req_{r1}(P)\ \not\subseteq Asys) \\
 & & \qquad \wedge (A'sys = Asys \cup Gsys - Req_{r1}(P)\ ) \wedge (G'sys =\ Reus(P)\ ))
\end{array}
$$

This definition expresses the idea that only when there is no sufficient allocatable resources, but with resources in the recycle bin the resource constraint can still be satisfied, then will we clean up the recycle bin. On termination of the program, all reusable resource will be put to recycle bin.

By arithmetic calculation, we have

Theorem 6.7

$[\![P]\!]_{r2} \sqsubseteq_{r2} [\![P]\!]_{r3}$

With all inequations in the commitment turns to be equations, it is direct to write a program for it.

Definition 6.8

$$\begin{aligned} [\![Emp]\!]_{rg} &=_{df} (Req_{r1}(P) \subseteq Asys \cup Gsys) \vdash \\ &\quad (A'sys = Asys \cup Gsys) \wedge (G'sys = \phi) \\ [\![P]\!]_{rg} &=_{df} [\![P]\!] \| \mathbf{true} \vdash \left(\begin{array}{l} A'sys = Asys - Req_{r1}(P) \\ \wedge G'sys = Gsys \cup \ Reus(P) \end{array} \right) \end{aligned}$$

Theorem 6.9

$[\![P]\!]_{r3} = ([\![Emp]\!]_{rg} \lhd Req_{r1}(P) \not\subseteq Asys \rhd skip);\ [\![P]\!]_{rg}$

Theorem 6.10 $[\![\ ]\!]_{r3}$*is homomorphic to general program constructs*

1. $[\![P;\ Q]\!]_{r3} = [\![P]\!]_{r3};\ [\![Q]\!]_{r3}$
2. if $Reus(b) = \phi$
   $[\![P \lhd b \rhd Q]\!]_{r3} = [\![P]\!]_{r3} \lhd [\![b]\!]_{r3} \rhd [\![Q]\!]_{r3}$
3. $[\![P \sqcap Q]\!]_{r3} = [\![P]\!]_{r3} \sqcap [\![Q]\!]_{r3}$
4. $[\![P \| Q]\!]_{r3} = [\![P]\!]_{r3} \| [\![Q]\!]_{r3}$

7 Resource-Oriented Program Transformation

Reusable resource semantic models provide us good ways for resource utilization. However, we may often be unable to afford the least required resource of a program. That the condition

$Req_{rs}(P) \not\subseteq Asys \cup Gsys$ is definitely to be violated, though certain schema for reuse are implemented. In our daily life, if that takes place, we must change the way of doing things, like splitting a resource hunger task into sub-tasks so that the resource constraint can be satisfied in sub-tasks. The motivation is to gain the feasibility of a task by sacrificing deadline. The tradeoff between resource and time is always an interesting topic in computer science, in this paper, we discuss the feasibility problem, that is to convert a program into a behavior equivalent one with less required resources.

Program $I(Q, x, y, u, v)$ leaves the execution of a program Q under environment with initial allocatable resource x, recycle bin y, and ensure on its termination they are u and v accordingly.

$$\begin{aligned} I(Q, x, y, u, v) =_{df} & \ \textbf{var}\ Asys := x, Gsys := y; \\ & [\![Q]\!]_{r3}; \\ & (Asys = u \wedge Gsys = v)_{\perp}; \\ & \textbf{end}\ Asys, Gsys \end{aligned}$$

We define a predicate $\mathcal{PC}(P, Q, x, y, u, v)$, where P is the original program, Q is the program after transformation.

Definition 7.1 $\mathcal{PC}(P, Q, x, y, u, v) =_{df} P \Leftarrow I(Q, x, y, u, v)$

So a transformation is correct only if, under an environment with initial resource constraints, the execution of Q is better than or equal to the original program P. If Q still violates the resource constrain, as we know it will behave chaotically, then the transformation fails and it is necessary to modify resource settings.

Algebraic laws and refinement theorems give us a consolidated foundation for program transformation. For reusable resources, we have

Theorem 7.2

1. $\mathcal{PC}(skip,\ skip, A, G, A, G)$
2. $\mathcal{PC}(wait(n),\ wait(n), A, G, A, G)$
3. $\mathcal{PC}((P_1;\ Q_1),\ (P_2;\ Q_2),\ A, G, A', G')$
   $if\ \exists A_0, G_0 \quad \mathcal{PC}(P_1,\ P_2,\ A, G, A_0, G_0)$
   $\wedge \quad \mathcal{PC}(Q_1,\ Q_2,\ A_0, G_0, A', G')$
4. $\mathcal{PC}(x := Exp,\ x := Exp, A, G, A - Cons(Exp), G \cup Reus(Exp))$
   if $Req_{r1}(Exp) \subseteq A$

5. $\mathcal{PC}(x := Exp,\ P,\ A, G, A', G')$
   if $\quad Req_{r1}(Exp)\ \not\subseteq A$
   $\wedge\ \ Req_{r1}(Exp)\ \subseteq A \cup G$
   $\wedge\ \ \mathcal{PC}(x := Exp,\ P,\ A \cup G, \phi, A', G')$

6. $\mathcal{PC}(x := Exp,\ P,\ A, G, A', G')$
   if $\quad Req_{r1}(Exp)\ \not\subseteq A \cup G$
   $\wedge\ \ Exp = Exp_2(Exp_1)$
   $\wedge\ \ \mathcal{PC}((x := Exp_1;\ \ x := Exp_2(x)), P,\ A, G, A', G')$

7. $\mathcal{PC}(P_1 \lhd B \rhd Q_1,\ P_2 \lhd B \rhd Q_2, A, G, A', G')$
   if $\quad Reus(B)\ =\ \phi$
   $\wedge\ \ \exists\, A_1, G_1 \cdot$
   $\mathcal{PC}(P_1,\ P_2,\ A - Cons(B)\ , G, A_1, G_1)$
   $\wedge\ \ \mathcal{PC}(Q_1,\ Q_2,\ A_1, G_1, A', G')$

8. $\mathcal{PC}(P_1 \lhd B \rhd Q_1,\ P, A, G, A', G')$
   if $\quad Reus(B)\ \not\subseteq \phi$
   $\wedge\ \mathcal{PC}((\ \mathbf{var} x := B\ ;\ \ P_1 \lhd x \rhd Q_1;\quad \mathbf{end} x\ ), P, A, G, A', G')$

9. $\mathcal{PC}(B * P,\ B * Q,\ A, G, A', G')$
   if $\quad Reus(B)\ =\ \phi$
   $\wedge\ \mathcal{PC}(P,\ Q,\ A - Cons(B)\ , G, A', G')$

10. $\mathcal{PC}(B * P,\ Q, A, G, A', G')$
    if $\quad Reus(B)\ \neq \phi$
    $\wedge\ \mathcal{PC}((\ \mathbf{var} x := B\ ;\ \ (x) * (P;\ \ x := B);\ \ \mathbf{end} x\ ), Q, A, G, A', G')$

11. $\mathcal{PC}(P_1 \| Q_1,\ P_2 \| Q_2, A_1 \cup A_2, G_1 \cup G_2, A_1' \cup A_2', G_1' \cup G_2')$
    if $\quad \mathcal{PC}(P_1,\ P_2,\ A_1, G_1, A_1', G_1')$
    $\wedge\ \ \mathcal{PC}(Q_1,\ Q_2, A_2, G_2, A_2', G_2')$

Proof: Here we prove Clause 3

$$
\begin{aligned}
P_1;\ Q_1 <= & \ I(P_2, A, G, A_0, G, G_0);\ \ I(Q_2, A_0, G_0, A', G') \\
= & \ \mathbf{var} Asys := A, Gsys := G; \\
& [\![P]\!]_{r3};\ (Asys = A_0 \wedge Gsys = G_0)_\perp; \\
& \mathbf{end} Asys, Gsys; \\
& \mathbf{var} Asys := A_0, Gsys := G_0; \\
& [\![Q]\!]_{r3};\ (Asys = A' \wedge Gsys = G')_\perp; \\
& \mathbf{end} Asys, Gsys \\
<= & \ \{By\ L1.\ and\ L2.\} \\
& \mathbf{var} Asys := A, Gsys := G; \\
& [\![P]\!]_{r3};\ [\![Q]\!]_{r3}; \\
& (Asys = A' \wedge Gsys = G')_\perp; \\
& \mathbf{end}\ Asys, Gsys \\
= & \ \{By\ definition\ and\ theorem\ 6.10\} \\
& I((P_2;\ Q_2), A, G, A', G')
\end{aligned}
$$

8 Multiplexors for Reusable Resource

Prolog [10] [12] is strong in transforming theorems to programs. In [1], it is used for compiling occam programs to integrated circuits. Here we can apply the transformation clauses to our compiler design to allocate and map reusable resources at compile time likely. In actual implementation of the compiler, we must guarantee data consistency for reusable resource. So we have to pay attention to multiplexors, which are designed to avoid input data collision. The multiplexor is bound with resource one-to-one.

Also in [1], the target of compilation is in Hardware Normal Form(HWNF) $C(s, f)$, the impulse on s triggers the running of circuits C, and f's turning to 1 indicates the end of its execution, its behavior is $< s, C, f >$, which is defined as

$$
\begin{aligned}
< s, C, f > =_{df} & \ \mathrm{var} s, f, \underline{l}, \underline{w}; \\
& \ Init; \\
& \ \mathrm{while}\ \neg f\ \mathrm{do}\ Step\ ; \\
& \ (\neg w \wedge \neg l)_\perp; \\
& \ \mathrm{end} s, f, \underline{l}, \underline{w}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Init} &=_{df} \; s, f, \underline{l}, \underline{w}, \underline{c}, \underline{d} :\in (s' = 1) \wedge (f' = 0) \\
&\quad \wedge (\underline{l} = 0) \wedge (\underline{w'} = \underline{W'}) \wedge (\underline{d'} = \underline{D'})) \\
\mathit{Step} &=_{df} \; \mathit{busy}1; \\
&\quad (s, f, \underline{l}, \underline{w}, \underline{c}, \underline{d} :\in (s' = 0) \wedge (f' = F) \\
&\quad \wedge (\underline{l} = 0) \wedge (\underline{w'} = \underline{W'}) \\
&\quad \wedge (\underline{x'} = E \lhd B \rhd \underline{x}) \wedge (\underline{d'} = \underline{D'}))
\end{aligned}
$$

Here those with underlines are set of symbols, $\underline{w}$, $\underline{l}$, $\underline{x}$ and $\underline{d}$ are set of combinational circuits, delays, registers and output ports of C. The notation $v :\in R$ represents a generalized assignment which assigns v a value such that the post-condition R holds at its termination. The correctness of a hardware compiler is specified as $P \sqsubseteq < s, C, f >$. The principle for multiplexer is based on one important character of $C(s, f)$, that is to a sequence of processes $P_0, ..., P_n$, there are correspondingly disjoint $s_0, ..., s_n$, at anytime

$$\sum_i s_{i_t} \leq 1$$

We can use this information to pick up correct input data. The multiplexor can be specified as

$$u_t = x_{i_t} \; \mathit{if} \; s_{i_t} = 1$$

We add the multiplexor as a new term to the HWNF. That is in the form

$$u : \mathit{mux}((s_0 \wedge x_0) \vee \ldots \vee (s_n \wedge x_n))$$

Two multiplexors with a same name, $u : \mathit{mux}((s_0 \wedge x_0) \vee \ldots \vee (s_i \wedge x_i))$and $u : \mathit{mux}((s_j \wedge x_j) \vee \ldots \vee (s_n \wedge x_n))$ can be merged together as $u : \mathit{mux}((s_0 \wedge x_0) \vee \ldots \vee (s_i \wedge x_i) \vee (s_j \wedge x_j) \vee \ldots \vee (s_n \wedge x_n))$.

Let's define the less-than comparator as $v : \mathit{lessThan}(a, b)$ with meaning $v_t = a_t < b_t$. The behavior of the less-than comparator and the multiplexer is

$$
\begin{aligned}
& v : \mathit{lessThan}(a, b) \\
& \;\& \; a : \mathit{mux}((s_0 \wedge x_0) \vee \ldots \vee (s_n \wedge x_n)) \\
& \;\& \; b : \mathit{mux}((s_0 \wedge y_0) \vee \ldots \vee (s_n \wedge y_n)) \\
= \; & (v = a < b) \wedge (a = (s_0 \wedge x_0) \vee \ldots \vee (s_n \wedge x_n)) \\
& \wedge (b = (s_0 \wedge y_0) \vee \ldots \vee (s_n \wedge y_n)) \\
= \; & (v = a_i < b_i \; \mathit{if} \; s_i = 1)
\end{aligned}
$$

If the comparator v is allocated to carry the result of comparison, multiplexors a and b are bound to operands of v, and the comparison happens during s is 1, the compiler clause is

$$x < y \sqsubseteq \exists \, a, b \cdot \begin{pmatrix} & v : \mathit{lessThan}(a, b) \\ \& & a : \mathit{mux}(s \wedge x) \\ \& & b : \mathit{mux}(s \wedge y) \end{pmatrix}$$

The multiplexor problem to reusable procedures is similar, but with a little expansion to the HWNF . Suppose x is the value parameter, y is the return parameter, then the correctness of compiling a procedure with parameters can be specified as

$$P(x,y) \sqsubseteq < s, C(x,y), f >$$

$$\begin{aligned} < s, C(x,y), f > =_{df}\ & \text{var} s, f, \underline{l}, \underline{w}, u, v; \\ & \mathit{Init}_p \\ & \text{while } \neg f \text{ do } \mathit{Step}\ ; \\ & (\neg w \wedge \neg l \wedge y = v)_\perp; \\ & \text{end} s, f, \underline{l}, \underline{w}, u, v \end{aligned}$$

$$\begin{aligned} \mathit{Init}_p\ =_{df}\ & s, f, \underline{l}, \underline{w}, \underline{c}, \underline{d}, u :\in (s' = 1) \wedge (f' = 0) \\ & \wedge (\underline{l} = 0) \wedge (\underline{w'} = \underline{W'}) \wedge (\underline{d'} = \underline{D'}) \\ & \wedge (u' = x)) \end{aligned}$$

To a non-reusable procedure, if it is called n times, there will be n instance of $C_i(s_i, x_i, y_i, f_i)$. But if it is reusable, we shall merge them together. We use pair (s_i, f_i) to enable correct value parameters and assign correct result parameters to the right y_i. The merged one is in form $C(\underline{s_i}, \underline{x_i}, \underline{y_i}, \underline{f_i})$. Its semantic is

$$\begin{aligned} < \underline{s_i}, C(\underline{x_i}, \underline{y_i}), \underline{f_i} > =_{df}\ & \text{var} s, f, \underline{l}, \underline{w}, u, v, \underline{b_i}; \\ & \mathit{Init}_{pr} \\ & \text{while } \neg f \text{ do } \mathit{Step}\ ; \\ & (\neg w \wedge \neg l \wedge y_i = (b_i \wedge v) \wedge f_i = (b_i \wedge f))_\perp; \\ & \text{end} s, f, \underline{l}, \underline{w}, u, v, \underline{b_i} \end{aligned}$$

$$\begin{aligned} \mathit{Init}_{pr}\ =_{df}\ & s, f, \underline{l}, \underline{w}, \underline{c}, \underline{d}, u, b_i :\in (s' = \vee_i s_i) \wedge (f' = 0) \\ & \wedge (\underline{l} = 0) \wedge (\underline{w'} = \underline{W'}) \wedge (\underline{d'} = \underline{D'}) \\ & \wedge (u' = \vee_i (s_i \wedge x_i)) \wedge (b_i' = s_i) \end{aligned}$$

In this schema, variables $\underline{b_i}$ are all cost for reuse.

9 Conclusion

In this paper, following the method used in [6], we proposed four resource semantic models for programming languages. The first primitive model lights the way on how to include resource information to semantic models, and it contributes a set of inequations which tells us duplicated programs shall be eliminated during implementation. The second model distinguishes consumptive and reusable resources. Though it brings nondeterminism, it requires less resource, it is a refinement to the primitive model. To remove nondeterminism, we proposed another two models with assumptions that reusable resources are always reused and they are reused only when the

resource constraints is violated. These two models are in refinement hierarchy as well. After all inequations turn to be equations, the refining process stops and we get a program for resource allocation and reuse. The framework adopted in this paper for building models helps us reason programs with limited resources. In the end, we discuss transformation clauses for converting a program to one which is executable with less resources.

Yet there are still some unsolved issues need further exploration. One issue is that theorems must be complete in the sense that the model shall be homomorphic to all constructs. Our current model can fit resource allocation to finite programs. But it doesn't tackle resources for infinitive programs, like iteration. From the transformation laws, all resources inside a loop body are reused, no matter whether they are set as reusable and consumptive. That hampers us giving a reusable resource calculus. We can see the essential point is a piece of program can request resource allocation only once. To solve that we may take time into the model to record the trace for resource allocation. If in the record, allocation is already applied to a program, next time when we meet with the same program, we need only to provide some scheme for it to reuse allocated resource. That asks for a complete resource model, otherwise there are still chances for the hardware compiler to produce dangerous circuits.

Besides that, we hope the time information can give us more power in describing real-time system. A transformed program may slow the execution of a program. Yet time is a very important consumptive resource, a program's failure in meeting a deadline is also a violation to specifications. That enlightens us to expand the domain of resources by taking time as a resource, and establish refinement based on time resource. Further more, we carry program transformation in guide with timed refinement. That refinement is critical to optimal partition method for hardware software co-design.

References

[1] Jonathan Bowen and He Jifeng. An approach to the specification and verification of a hardware compilation scheme. *The Journal of Supercomputing*, 19((1)):23–29, 2001.

[2] R. Gerber and I.Lee. A calculus for communicating shared resources. Concur'90 LNCS 458. Springer Verlag, 1990.

[3] R. Gerber and I.Lee. Specification and analysis of resource-bound real-time systems. Theory in Practice, LNCS 600. Springer Verlag, 1992.

[4] A. Goswami and M. Joseph. A semantics for the specification of real-time processes. Technical Report 121, Department of Computer Science, University of Warwick, 1988.

[5] A. Goswami and M. Joseph. Semantics of real-time distributed programs. In *Concurrency'88*, 1988.

[6] C.A.R Hoare and Jifeng He. *Unifying Theories of Programming.* Series in Computer Science. Prentice-Hall Europe, 1st edition, 1998.

[7] Gavin Lowe. Scheduling-oriented models for real-time systems. *The Computer Journal*, 38(6), 1995.

[8] E. Shade and K.T. Narayana. Real-time semantics for concurrency with shared variables. Technical Report CS-88-28, The Pennsylvania State University, 1988.

[9] E. Shade and K.T. Narayana. *Concepts and Models for Real-Time Concurrency*. Foundation of Real-Time Computing, Formal Specification and Methods. Kluwer Academic Publishers, 1991.

[10] Sterling and Leon. *The Art of Prolog*. The MIT Press, Cambridge, Mass, 2nd edition, 1994.

[11] Thomas and Mooby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, fourth edition edition, 1998.

[12] Jan Wielemaker. *SWI-Prolog 5.0 Reference Manual*.