User-Driven Abstraction for Model Checking

Glenn Bruns
Bell Labs, Lucent Technologies

Abstract

Model checking has found a role in the engineering of reactive systems. However, model checkers are still strongly limited by the size of the system description they can check. Here we present a technique in which a system is simplified prior to model checking by the application of abstraction rules. The rules can greatly reduce the state space of a system description and help in understanding why a system satisfies a property. We illustrate the use of the technique on examples, including Dekker's mutual exclusion algorithm.

1 Motivation

Model checkers have become an important tool by which engineers can check properties of their systems. Model checkers have become successful partly because they require no user interaction. One must only describe the system of interest and the property to be checked. On the other hand, they provide little insight. With a model checker one can learn that a system satisfies a property, but not why it satisfies a property. Furthermore, the application of model checkers is limited strongly by the number of states in the system and in the complexity of the property to be checked.

Here we explore a technique by which an engineer can simplify a system description by removing features of the description believed to be irrelevant to the property to be checked. Simplification is performed by applying *abstraction rules*. For example, one rule allows any shared variable to be removed. These rules are sound in the sense that if the simplified description satisfies a property, then the original one does too. However, the simplified description may have a much smaller state space and therefore be able to be model checked, even if the original description could not.

This approach is also helpful in confirming one's understanding of *how* a system works. If the simplified description satisfies a property, then we know not only that the complete one does too, but also that our intuition is correct. Put another way, with our abstraction rules one can obtain a simplified description that does not satisfy a property even though the property *is* satisfied by the original description.

Our technique is sketched in the following section of the paper. Then we present abstraction rules and show how they can be applied to Dekker's mutual exclusion algorithm. We also discuss the use of our technique to other examples. We conclude with discussion and related work.

2 Abstraction with Preorders

Here we sketch our approach to simplifying the description of a system relative to a property to be checked. First of all, we describe systems as expressions in a process algebra. Such *processes* can model both normal terminating programs and reactive systems, which engage in ongoing interaction with their environment. We let E, E', F, … stand for processes.

We describe properties of systems as temporal logic formulas. In temporal logic we can express generic system properties like deadlock freedom as well as application-specific properties. We let ϕ and ψ range over temporal logic formulas and write $E \models \phi$ to mean that process E satisfies formula ϕ, which captures the idea that the system modelled by E has the property expressed by ϕ.

We want to simplify processes with respect to temporal logic formulas. We take a simplified process as one that stands in a certain relation to the original process. The kind of process relations we are interested in are *preorders*, which are reflexive and transitive, but not necessarily antisymmetric. Suppose $\leq$ is a process preorder. Then we say process E' is an *abstract* version of process E if

$$E \leq E'.$$

In this paper we explore preorders in which the abstract process E' can do whatever E can do and possibly more. So abstracting a system can be understood as a kind of "loosening" of its behavior.

We want to show that if an abstract process has a property, then so does the original one. To do so we find a class of formulas such that every formula ϕ in the class satisfies this logical condition:

$$(E \leq E' \text{ and } E' \models \phi) \Rightarrow E \models \phi.$$

The condition reads "if E' is an abstract version of E, and E' satisfies property ϕ, then so does E."

To make abstraction convenient, we do not want to have to prove that $E \leq E'$ whenever a simplification to E is made. If E describes a complex system this may be hard to do. Instead, we use a set of predefined *rules* for abstraction, each of which has the form $F \leq F'$. To keep the set of rules small, each rule must be able to be applied anywhere within a process. For example, suppose E has the form $E_1 \mid E_2$, where E_1 and E_2 are parallel components. Our rules should satisfy this algebraic condition:

$$E_1 \leq E_1' \Rightarrow (E_1 \mid E_2 \leq E_1' \mid E_2)$$

This condition reads "if E_1' is an abstract version of E_1, then $E_1' \mid E_2$ is an abstract version of $E_1 \mid E_2$." The preorders we use for abstraction satisfy this conditions not just for the parallel composition operator, but for all operators of the process notation.

3 Processes and Properties

We use CCS [14] processes to describe systems. Processes perform actions, which are either *names* $(a, b, \ldots)$, *co-names* $(\overline{a}, \overline{b}, \ldots)$, or the action τ. Names and co-names satisfy $\overline{\overline{\mathsf{a}}} = \mathsf{a}$. The set of all actions is denoted Act. Process expressions have the following syntax, where α ranges over actions, L ranges over sets of non-τ actions, A ranges over process constants, and f ranges over relabelling functions (functions from Act to Act satisfying $f(\tau) = \tau$ and $f(\overline{\mathsf{a}}) = \overline{f(a)}$):

$$E ::= A \,\big|\, 0 \,\big|\, \alpha.E \,\big|\, E_1 + E_2 \,\big|\, E_1 \mid E_2 \,\big|\, E \backslash L \,\big|\, E[f]$$

Additionally, families of process definitions are allowed, having the form $\{A_i \overset{\text{def}}{=} E_i \mid i \in I\}$. The meaning of CCS processes is given as a labelled transition system in which processes are states and transitions are labelled with actions. If there is a transition from process E to process F labelled with action α, we write $E \xrightarrow{a} F$ and say "E performs α and becomes F". We will only briefly recall the meaning of the CCS operators here. Process 0 is the deadlocked process. Operator . is action prefixing. Operator + is choice. Operator | is concurrent composition, with synchronization between complementary actions resulting in a τ action. Operator $\backslash L$ is restriction to labels in L and their complements, written $\overline{L}$. Operator $[f]$ is relabelling by f.

We will additionally use a *hiding* operator $\backslash\!\!\backslash L$. We define it directly for simplicity, but it could be defined as a derived CCS operator.

$$\frac{E \xrightarrow{\alpha} E'}{E \backslash\!\!\backslash L \xrightarrow{\alpha} E' \backslash\!\!\backslash L}\ \alpha \notin L \cup \overline{L} \qquad \frac{E \xrightarrow{\alpha} E'}{E \backslash\!\!\backslash L \xrightarrow{\tau} E' \backslash\!\!\backslash L}\ \alpha \in L \cup \overline{L}$$

We use a slightly extended modal mu-calculus [12, 16] to express properties of processes. We present the logic in its positive normal form. Formulas have the following syntax, where L ranges over sets of actions and Z ranges over variables:

$$\phi ::= Z \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; \phi_1 \vee \phi_2 \;\Big|\; [L]\phi \;\Big|\; \langle L\rangle\phi \;\Big|\; \nu Z.\phi \;\Big|\; \mu Z.\phi$$

The interesting operators are the modal and fixed-point operators. Informally, E satisfies $[L]\phi$ (resp. $\langle L\rangle\phi$) if all (resp. some) E' that can be reached from E through an a-transition ($a \in L$) satisfy ϕ. The fixed-point operators bind free occurrences of Z in ϕ. Informally, E satisfies $\nu Z.\phi$ (resp. $\mu Z.\phi$) if E belongs to the greatest (resp. least) solution of the recursive modal equation $Z = \phi$. We write $E \models \phi$ if E satisfies the closed formula ϕ. We use `tt` as an abbreviation for the (true) formula $\nu Z.Z$, and $[-L]\phi$ as an abbreviation for $[Act - L]\phi$.

We write the set of all closed modal mu-calculus formulas as μL.

4 Weak Simulation

Intuitively, one process simulates another if it can match any action the other can do, and can continue to match in this way indefinitely. Here we define a particular kind of simulation relation on processes called the *weak simulation* relation and show that it has the logical and algebraic properties we need.

The term *weak* indicates that the relation is based on what we can observe of a process. Recall that in CCS the τ action represents internal activity of a system that cannot be observed. So to formalize what it means to be observable we define a transition relation on processes in which τ transitions do not occur. However, transitions can be labelled by the new symbol ε, which represents the occurrence of zero or more τ actions.

$$\begin{aligned} E \overset{\varepsilon}{\Rightarrow} F &\overset{\text{def}}{=} E(\xrightarrow{\tau})^* F \\ E \overset{a}{\Rightarrow} F &\overset{\text{def}}{=} E \overset{\varepsilon}{\Rightarrow} \circ \xrightarrow{a} \circ \overset{\varepsilon}{\Rightarrow} F \qquad (a \neq \tau) \end{aligned}$$

We write Act_{obs} for the set $Act \setminus \{\tau\} \cup \{\varepsilon\}$ of observable actions.

Definition 1 *A binary relation $\mathcal{R}$ on processes is a* weak simulation *if (E, F) in $\mathcal{R}$ implies, for all α in Act_{obs}:*

Whenever $E \overset{\alpha}{\Rightarrow} E'$, then $F \overset{\alpha}{\Rightarrow} F'$ for some F' such that $(E', F') \in \mathcal{R}$.

E is weakly simulated by F, *written $E \leq F$, if (E, F) belongs to some weak simulation $\mathcal{R}$.*

If a process E is weakly simulated by another process F, then any sequence of actions that E can perform can be matched by F. However, the converse does not hold. For example, $B \overset{\text{def}}{=} a.b.0 + a.c.0$ can match every sequence of actions that $A \overset{\text{def}}{=} a.(b.0 + c.0)$ can perform, but B does not weakly simulate A.

The weakly simulates relation is preserved by all CCS operators, including recursive definition.

Theorem 1 $\leq$ *is preserved by all CCS operators.*

(Proofs for the theorems and abstraction rules in the paper can be found in [4].)

To describe the properties that are preserved by the weak simulation relation, we define a modal operator $[\![\,]\!]$ that is based on the weak transition relation. A process E satisfies formula $[\![a]\!]\phi$ if all E' that can be reached from E through a weak a-transition satisfy ϕ.

We write $\mu\mathbb{L}\Box$ for the set of closed modal mu-calculus formulas containing no modal operators except $[\![\,]\!]$. An example safety property that can be expressed in $\mu\mathbb{L}\Box$ is that no a action can ever occur. An example liveness property that can be expressed in $\mu\mathbb{L}\Box$ is that a must eventually occur if system operation never terminates.

If a process F satisfies a formula of $\mu\mathbb{L}\Box$, and F weakly simulates E, then E also satisfies the formula.

Theorem 2 *Let ϕ be a formula of $\mu\mathbb{L}\Box$. Then*

$$(E \leq E' \textit{ and } E' \models \phi) \Rightarrow E \models \phi.$$

5 Abstraction Rules

We now present abstraction rules for $\leq$. Each rule has the form $E \leq E'$, where E' is understood as the abstract version of E.

Proposition 1 (Restriction rules)

$$E\backslash L \leq E\backslash\!\backslash K\backslash L \quad \textit{if } K \subseteq L \cup \overline{L} \tag{1}$$

$$E\backslash L \leq E[f]\backslash L \quad \textit{if } f(\alpha) = \alpha \textit{ for } \alpha \notin L \cup \overline{L} \tag{2}$$

These rules are especially important because they allow actions that are used for process synchronization to be hidden or renamed. Doing so loosens synchronization between components, which intuitively increases the number of possible system states. However, the rule can allow the structure of components to be made more regular, which in terms reduces the system state space.

Proposition 2 (Hiding rules)

$$(\alpha.E)\backslash\!\backslash L \leq \tau.(E\backslash\!\backslash L) \quad \textit{if } \alpha \in L \tag{3}$$

$$(E + F)\backslash\!\backslash L \leq E\backslash\!\backslash L + F\backslash\!\backslash L \tag{4}$$

$$(E \mid F)\backslash\!\backslash L \leq E\backslash\!\backslash L \mid F\backslash\!\backslash L \quad \textit{if } L = \overline{L} \tag{5}$$

Proposition 3 (Relabelling rules)

$$(\alpha.E)[f] \leq f(\alpha).(E[f]) \tag{6}$$

$$(E + F)[f] \leq E[f] + F[f] \tag{7}$$

$$(E \mid F)[f] \leq E[f] \mid F[f] \tag{8}$$

Proposition 4 (Family rules)

$$\mathcal{F} \leq \mathrm{merge}(A_j, A_k, \mathcal{F}) \tag{9}$$

$$\mathcal{F}\{A \stackrel{\mathrm{def}}{=} A + E\} \leq \mathcal{F}\{A \stackrel{\mathrm{def}}{=} E\} \tag{10}$$

Rule 9 allows two constants in a family of process definitions to be "merged". To merge two constants one redefines the first of the constants by adding the definition of the second as a choice, and then renames the second constant to the first. For example, merge$(A, B, \{A \stackrel{\text{def}}{=} a.B, B \stackrel{\text{def}}{=} b.A\})$ yields $\{A \stackrel{\text{def}}{=} a.A + b.A\}$. In terms of process behavior, this operation can be understood as the combining to two process states. The rule is used when the difference between two process states is not important relative to the property to be proved.

Proposition 5 (Basic rules)

$$E \mid 0 \;\leq\; E \tag{11}$$
$$\tau.E \;\leq\; E \tag{12}$$
$$A \;\leq\; E \qquad \textit{if } A \stackrel{\text{def}}{=} E \tag{13}$$
$$E \;\leq\; A \qquad \textit{if } A \stackrel{\text{def}}{=} E \tag{14}$$

The abstraction rules above are typically applied in certain combinations. The following rules, which can be derived from the preceding ones, allow abstraction to proceed in coarser steps.

Proposition 6 (Derived rules) *Let K and L be sets of actions such that $K = \overline{K} \subseteq L \cup \overline{L}$, and let f be a relabelling function satisfying $f(\alpha) = \alpha$ $(\alpha \notin L \cup \overline{L})$. Then*

$$(E_1 \mid \cdots \mid E_n)\backslash L \;\leq\; (E_1 \backslash\!\!\backslash K \mid \cdots \mid E_n \backslash\!\!\backslash K)\backslash L \tag{15}$$
$$(E_1 \mid \cdots \mid E_n)\backslash L \;\leq\; (E_1[f] \mid \cdots \mid E_n[f])\backslash L \tag{16}$$

6 Example

We illustrate our technique by abstracting Dekker's mutual exclusion algorithm [15] relative to a safety property and a liveness property. In each case, we show that if the abstract algorithm satisfies the property, then so does the concrete algorithm.

The purpose of a mutual exclusion algorithm is to control access by a set of processes to a shared resource so that at most one process has access at any time. A process is said to be in its *critical section* when it has access to the resource. The key safety property of a mutual exclusion algorithm is that at most one process is in its critical section at any time. The key liveness property is that a process wishing to enter its critical section will eventually be able to do so.

We now present a two-process version of Dekker's mutual exclusion algorithm. The top level of the algorithm, written in a concurrent while language, is as follows.

begin
    var b_1,b_2,k;
    b_1 := **false**; b_2 := **false**;
    k := 1;
    P_1 **par** P_2
end;

Processes P_1 and P_2 coordinate via shared variables b_1, b_2, and k. Process P_1 is defined as follows; P_2 is obtained from P_1 by interchanging 1 with 2 everywhere. The process does its useful work in the non-critical section.

while true do
begin
    ⟨ non-critical section ⟩;
    b_1 := **true**
    while b_2 **do**
        if $k = 2$ **then begin**
            b_1 := **false**;
            while $k = 2$ **do skip**;
            b_1 := **true**
        end;
    ⟨ critical section ⟩;
    $k := 2$;
    b_1 := **false**
end;

Informally, the b variables are "request" variables and the k variable is a "turn" variable. Variable b_i is *true* if P_i is requesting entry to its critical section; variable k is i if it is P_i's turn to enter its critical section. Only P_i writes on variable b_i, but both processes read b_i.

Translating the algorithm to CCS (from [17]) gives the following family of definitions. Actions $\mathtt{req_i}$, $\mathtt{enter_i}$, and $\mathtt{exit_i}$ have been added to indicate requests to enter, entrance to, and exit from the critical section by process i. A shared variable name (e.g., b_1) in a restriction operator stands for the read and write actions of the variable (e.g.,$\{\mathtt{b_1rt}, \overline{\mathtt{b_1wt}}, \mathtt{b_1rf}, \overline{\mathtt{b_1wf}}\}$). The definitions of *B2f* and *P2* are omitted, since they are symmetrical to *B1f* and *P1*.

$$
\begin{aligned}
K1 &\stackrel{\text{def}}{=} \overline{\mathtt{kr1}}.K1 + \mathtt{kw1}.K1 + \mathtt{kw2}.K2 \\
K2 &\stackrel{\text{def}}{=} \overline{\mathtt{kr2}}.K2 + \mathtt{kw1}.K1 + \mathtt{kw2}.K2 \\
\\
B1f &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}.B1f + \mathtt{b_1wf}.B1f + \mathtt{b_1wt}.B1t \\
B1t &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rt}}.B1t + \mathtt{b_1wf}.B1f + \mathtt{b_1wt}.B1t \\
\\
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathtt{req_1}.P11 \\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}.P13 + \mathtt{b_2rt}.(\mathtt{kr1}.P11 + \mathtt{kr2}.\overline{\mathtt{b_1wf}}.P12) \\
P12 &\stackrel{\text{def}}{=} \mathtt{kr1}.\overline{\mathtt{b_1wt}}.P11 + \mathtt{kr2}.\tau.P12 \\
P13 &\stackrel{\text{def}}{=} \mathtt{enter_1}.\mathtt{exit_1}.\overline{\mathtt{kw2}}.\overline{\mathtt{b_1wf}}.P1 \\
\\
Dekker &\stackrel{\text{def}}{=} (P1 \mid P2 \mid B1f \mid B2f \mid K1)\backslash\{\mathtt{b_1}, \mathtt{b_2}, \mathtt{k}\}
\end{aligned}
$$

6.1 Safety

The property of mutual exclusion is that at most one process can be in its critical section at any time. The property can be expressed as formula *ME* of $\mu\mathbb{L}\Box$ requiring that `enter` and `exit` actions alternate:

$$
\begin{aligned}
Cycle(L_1, L_2) &\stackrel{\text{def}}{=} \nu X_1.[\![L_2]\!]\,\mathtt{ff} \wedge [\![-L_1, L_2]\!]X_1 \ \wedge\ [\![L_1]\!] \\
&\qquad \nu X_2.[\![L_1]\!]\,\mathtt{ff} \wedge [\![-L_1, L_2]\!]X_2 \ \wedge\ [\![L_2]\!]X_1 \\
ME &\stackrel{\text{def}}{=} Cycle(\{\mathtt{enter}_1, \mathtt{enter}_2\}, \{\mathtt{exit}_1, \mathtt{exit}_2\})
\end{aligned}
$$

Dekker's algorithm satisfies the property of mutual exclusion because a process sets its request variable to *true* before attempting to enter its critical section, and waits for the request variable of the other process to be *false* before actually entering it. Dekker's algorithm can be abstracted greatly with respect to mutual exclusion because much of the algorithm's design deals with the problem of ensuring liveness.

We now apply abstraction rules to remove other details from the algorithm. To make the presentation concise, we will redefine the constants of the algorithm at each step, rather than use new ones, and will present only the constant definitions that were affected by the abstraction step. As a preliminary step we remove the indices of the $\mathtt{enter}_i$ and $\mathtt{exit}_i$ actions by relabelling, and hide the $\mathtt{req}_i$ actions, yielding process $Dekker_1$ below.

$$
\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11\\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \mathtt{b_2rt}\,.(\mathtt{kr1}\,.P11 + \mathtt{kr2}\,.\overline{\mathtt{b_1wf}}\,.P12)\\
P12 &\stackrel{\text{def}}{=} \mathtt{kr1}\,.\overline{\mathtt{b_1wt}}\,.P11 + \mathtt{kr2}\,.\tau.P12\\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\mathtt{exit}\,.\overline{\mathtt{kw2}}\,.\overline{\mathtt{b_1wf}}\,.P1
\end{aligned}
$$

By the correspondence rule of [3] we have

$$Dekker \models ME \Leftrightarrow Dekker_1 \models Cycle(\{\mathtt{enter}\},\{\mathtt{exit}\}).$$

Formula $Cycle(\{\mathtt{enter}\},\{\mathtt{exit}\})$ is also a formula of $\mu\mathbb{L}\Box$. In all further abstraction steps we use only the abstraction rules of Section 5, which all preserve the property $Cycle(\{\mathtt{enter}\},\{\mathtt{exit}\})$.

Our informal explanation of why Dekker's algorithm satisfies mutual exclusion mentions only the request variables, not the turn variable k. We therefore hide k-related actions $\{\mathtt{kr1},\mathtt{kw1},\mathtt{kr2},\mathtt{kw2}\}$ and their complements using derived rule 15, and move the hiding inward using the hiding rules. We refer to *Dekker* of the resulting family as $Dekker_2$.

$$
\begin{aligned}
K1 &\stackrel{\text{def}}{=} \tau.K1 + \tau.K1 + \tau.K2\\
K2 &\stackrel{\text{def}}{=} \tau.K2 + \tau.K1 + \tau.K2
\end{aligned}
$$

$$
\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11\\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \mathtt{b_2rt}\,.(\tau.P11 + \tau.\overline{\mathtt{b_1wf}}\,.P12)\\
P12 &\stackrel{\text{def}}{=} \tau.\overline{\mathtt{b_1wt}}\,.P11 + \tau.\tau.P12\\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\mathtt{exit}\,.\tau.\overline{\mathtt{b_1wf}}\,.P1
\end{aligned}
$$

The actions related to variables b_1 and b_2 cannot all be hidden, as variables b_1 and b_2 play a part in ensuring mutual exclusion. However, only some of the b-related actions are involved. We can hide the actions that represent when a b variable is read with value *true*. The effect is to allow P_1 to proceed as if b_2 is true whether b_2 is true or not. Thus P_1 can elect not to enter its critical section even if b_2 is *false*. Similarly P_2 can wait instead of entering its critical section.

Applying derived rule 15 with actions $\{\mathtt{b_1rt},\mathtt{b_2rt}\}$ and their complements, and moving hiding inward with the hiding rules gives the following.

$$
\begin{aligned}
B1f &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.B1f + \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t\\
B1t &\stackrel{\text{def}}{=} \tau.B1t + \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t
\end{aligned}
$$

$$
\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11\\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \tau.(\tau.P11 + \tau.\overline{\mathtt{b_1wf}}\,.P12)\\
P12 &\stackrel{\text{def}}{=} \tau.\overline{\mathtt{b_1wt}}\,.P11 + \tau.\tau.P12\\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\mathtt{exit}\,.\tau.\overline{\mathtt{b_1wf}}\,.P1
\end{aligned}
$$

We now remove all τ actions by repeatedly applying basic rule 12, and then apply family rule 10 to remove all unguarded occurrences of constants.

$$\begin{aligned}
\mathit{B1f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}.\mathit{B1f} + \mathtt{b_1wf}.\mathit{B1f} + \mathtt{b_1wt}.\mathit{B1t} \\
\mathit{B1t} &\stackrel{\text{def}}{=} \mathtt{b_1wf}.\mathit{B1f} + \mathtt{b_1wt}.\mathit{B1t}
\end{aligned}$$

$$\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathit{P11} \\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}.\mathit{P13} + \overline{\mathtt{b_1wf}}.\mathit{P12} \\
\mathit{P12} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathit{P11} \\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}.\mathtt{exit}.\overline{\mathtt{b_1wf}}.\mathit{P1}
\end{aligned}$$

Next we want to merge the states that are are outside the critical section but have not yet requested entry to the critical section. To prepare for this we apply basic rule 14 to introduce constants *P14* and *P24*.

$$\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathit{P11} \\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}.\mathit{P13} + \overline{\mathtt{b_1wf}}.\mathit{P12} \\
\mathit{P12} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathit{P11} \\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}.\mathtt{exit}.\mathit{P14} \\
\mathit{P14} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wf}}.\mathit{P1}
\end{aligned}$$

Family rule 9 is now applied to merge constants *P1*, *P12*, and *P14*. Similarly, constants *P2*, *P22*, and *P24* of *P2* are merged. Constants *K1* and *K2* are then removed using basic rule 11 to obtain the following family. We refer to *Dekker* of this family as Dekker_3.

$$\begin{aligned}
\mathit{B1f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}.\mathit{B1f} + \mathtt{b_1wf}.\mathit{B1f} + \mathtt{b_1wt}.\mathit{B1t} \\
\mathit{B1t} &\stackrel{\text{def}}{=} \mathtt{b_1wf}.\mathit{B1f} + \mathtt{b_1wt}.\mathit{B1t}
\end{aligned}$$

$$\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathit{P11} + \overline{\mathtt{b_1wf}}.\mathit{P1} \\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}.\mathit{P13} + \overline{\mathtt{b_1wf}}.\mathit{P1} \\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}.\mathtt{exit}.\mathit{P1}
\end{aligned}$$

$$\mathit{Dekker} \stackrel{\text{def}}{=} (\mathit{P1} \mid \mathit{P2} \mid \mathit{B1f} \mid \mathit{B2f})\backslash\{\mathtt{b_1}, \mathtt{b_2}\}$$

The abstract algorithm contains little more than the protocol for handling the request variables to ensure mutual exclusion. While process *Dekker* has 153 states, Dekker_3 has only 16 states. With a tool like the Concurrency Workbench [7] it is easy to show that Dekker_3 satisfies $\mathit{Cycle}(\{\mathtt{enter}\},\{\mathtt{exit}\})$. Since Dekker_3 was obtained by abstraction rules that preserve $\mathit{Cycle}(\{\mathtt{enter}\},\{\mathtt{exit}\})$ we know Dekker_1 also satisfies $\mathit{Cycle}(\{\mathtt{enter}\},\{\mathtt{exit}\})$. Then, since

$$\mathit{Dekker}_1 \models \mathit{Cycle}(\{\mathtt{enter}\},\{\mathtt{exit}\}) \Rightarrow \mathit{Dekker} \models \mathit{ME}$$

we know $\mathit{Dekker} \models \mathit{ME}$.

The soundness of our abstraction of Dekker's algorithm does not depend on the assumptions we made about the shared variables in it. For example, our decision to hide actions of variable k was based on the assumption that k plays no part in ensuring mutual exclusion. Regardless of the truth of this assumption, the algorithm will satisfy the formula expressing mutual exclusion if the abstract version of it does. Since mutual exclusion does hold in the abstract algorithm, we know not only that it also holds in the concrete algorithm, but also that our assumption about variable k's role in mutual exclusion is correct.

6.2 Liveness

An important liveness property for mutual exclusion algorithms is that if one process requests entry to its critical section, then it will not wait forever while the other processes continue to enter their critical sections. Here we use our abstraction rules to show that Dekker's algorithm satisfies this property, with process 2 as the requesting process.

Dekker's algorithm satisfies this property only under the fairness assumption that the requesting process continues to execute after it requests entry to its critical section. We handle the fairness assumption by incorporating it into the formula we use to express the liveness property. However, we must also revise process 2 by adding "probe" action p2, which continues to occur while process P_2 is waiting to enter its critical section.

$$\begin{array}{rcl}
P2 & \stackrel{\text{def}}{=} & \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.\,P21 \\
P21 & \stackrel{\text{def}}{=} & \mathtt{b_1rf}\,.\,P23 + \mathtt{b_1rt}\,.\,\mathtt{p2}\,.(\mathtt{kr2}\,.\,P21 + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.\,P22) \\
P22 & \stackrel{\text{def}}{=} & \mathtt{kr2}\,.\,\overline{\mathtt{b_2wt}}\,.\,P21 + \mathtt{kr1}\,.\,\mathtt{p2}\,.\tau.P22 \\
P23 & \stackrel{\text{def}}{=} & \mathtt{enter_2}\,.\,\mathtt{exit_2}\,.\,\overline{\mathtt{kw1}}\,.\,\overline{\mathtt{b_2wf}}\,.\,P2
\end{array}$$

The following $\mu\mathbb{L}\Box$ formula expresses that process 2 will not wait forever to enter its critical section while process 1 continues to enter its critical section, provided that process 2 continues to execute while waiting.

$$\begin{array}{l}
Live \stackrel{\text{def}}{=} \nu X.[\![-\ \mathtt{req_2}]\!]X \wedge \\
\qquad\qquad [\![\mathtt{req_2}]\!]\mu Y.\nu X_1.[\![-\ \mathtt{enter}_1, \mathtt{enter}_2, \mathtt{p2}]\!]X_1 \ \wedge\ [\![\mathtt{enter}_2]\!]X \ \wedge\ [\![\mathtt{enter}_1]\!] \\
\qquad\qquad\qquad\qquad \nu X_2.[\![-\ \mathtt{enter}_1, \mathtt{enter}_2, \mathtt{p2}]\!]X_2 \ \wedge\ [\![\mathtt{enter}_2]\!]X \ \wedge\ [\![\mathtt{p2}]\!]Y
\end{array}$$

Paraphrasing the formula gives: "after $\mathtt{req_2}$ occurs, no path containing infinitely many $\mathtt{enter}_1$ and p2 actions but no $\mathtt{enter}_2$ actions, can occur". Informally, Dekker's algorithm satisfies this property because Process 1 will only enter its critical section if process 2's request variable is false, and will set the turn variable to 2 upon leaving its critical section. If Process 2 has requested entry but not yet entered its critical section, and the turn variable is set to 2, then it will eventually set its request variable to *true* and keep it *true* until after it exits its critical section.

To abstract Dekker's algorithm relative to this formula, we first hide both exit actions and action $\mathtt{req}_1$. This step is justified by the correspondence rule of [3]. Then we hide all actions corresponding to reads of b variables of value *true*, to all reads and writes of b_1 variables, and to reads of k of value 2. Finally all τ actions are removed and some constants are merged. The abstract form of the algorithm is as follows.

$$\begin{array}{rcl}
B2f & \stackrel{\text{def}}{=} & \overline{\mathtt{b_2rf}}\,.\,B2f + \mathtt{b_2wf}\,.\,B2f + \mathtt{b_2wt}\,.\,B2t \\
B2t & \stackrel{\text{def}}{=} & \mathtt{b_2wf}\,.\,B2f + \mathtt{b_2wt}\,.\,B2t
\end{array}$$

$$\begin{array}{rcl}
K1 & \stackrel{\text{def}}{=} & \overline{\mathtt{kr1}}\,.\,K1 + \mathtt{kw1}\,.\,K1 + \mathtt{kw2}\,.\,K2 \\
K2 & \stackrel{\text{def}}{=} & \mathtt{kw1}\,.\,K1 + \mathtt{kw2}\,.\,K2
\end{array}$$

$$
\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.\,\mathit{P13} + \mathtt{kr1}\,.\,\mathit{P1} \\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}_1\,.\,\overline{\mathtt{kw2}}\,.\,\mathit{P1}
\end{aligned}
\qquad
\begin{aligned}
\mathit{P2} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\,\mathtt{req}_2\,.\,\mathit{P21} \\
\mathit{P21} &\stackrel{\text{def}}{=} \mathit{P23} + \mathtt{p}_2\,.\,\mathit{P21} + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.\,\mathit{P22} \\
\mathit{P22} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\,\mathit{P21} + \mathtt{kr1}\,.\,\mathtt{p}_2\,.\,\mathit{P22} \\
\mathit{P23} &\stackrel{\text{def}}{=} \mathtt{enter}_2\,.\,\overline{\mathtt{kw1}}\,.\,\overline{\mathtt{b_2wf}}\,.\,\mathit{P2}
\end{aligned}
$$

$$
\mathit{Dekker} \stackrel{\text{def}}{=} (\mathit{P1} \mid \mathit{P2} \mid \mathit{B2f} \mid \mathit{K1})\backslash\{\mathtt{b_2}, \mathtt{k}\}
$$

7 Other Examples

Dekker's algorithm was presented as our main example because it is a non-trivial algorithm but simple enough to work through in detail. We have used our abstraction rules on other examples, including other the mutual exclusion algorithms of Dijkstra [9], Knuth [11], and Peterson [15]. For example, using our abstraction operations we reduced a three process version of Dijkstra's algorithm from 10570 to 109 states. In abstracting two-process versions of the algorithms we found that all the algorithms reduce to virtually the same 16 state algorithm. Thus, abstraction shows that the same idea is used to achieve safety in all the algorithms. However, in abstracting the processes with respect to liveness, different algorithms are reached.

We have used our abstraction technique with preorders other than weak simulation. In [4] we use the *anonymous ready simulation* preorder to abstract *prioritized* CCS processes [6]. The sole abstraction rule here is the adding of priorities to a process. Using this rule we have been able to check a property of Ben-Ari's concurrent garbage collection algorithm [1], which we could not do at all without abstraction.

8 Discussion and Related Work

We have presented a general technique for abstracting a system description relative to a property to be checked, and illustrated it by developing abstraction rules for CCS processes. We would like to develop similar abstraction rules for notations that are more suitable for the description of complex systems. One possibility is *value-passing CCS* [14], which we used in our work in abstraction using priorities. Other possibilities include concurrent while languages, LOTOS [2], CRL [10], or even a programming language such as C. The main problem faced in using a higher-level notation is to establish the algebraic condition we need to apply the technique. One way to deal with this problem is to try to define the notation in terms of a low-level notation like CCS. Then if the algebraic condition we need holds of CCS, it also holds of the higher-level notation.

The possibilities for automation with our technique are not clear. In the example we presented the abstraction rules were selected and applied manually. It would not be difficult to provide tool support for the automatic application of rules. A more interesting question is whether it would be worthwhile to try to select abstraction rules automatically. A problem is that there may be many abstracted forms of an algorithm that have small state spaces but fail to satisfy the property of interest.

There is existing work on abstraction using preorders, but none that gives abstraction rules. Lynch [13] abstracts I/O automata using the simulation preorder, but requires that a simulation relation be invented and checked. The logical effect of abstraction with simulation is not given; it is only stated that simulation is a stronger relation than trace inclusion. Clarke *et al* [5] abstract finite-state, procedural programs by mappings on program inputs and outputs. No abstraction operations on control structure are given. The technique is justified by showing that a kind of homomorphic mapping on transition systems preserves CTL* formulas with only the universal path quantifier. This mapping is stronger than the simulation relation, and permits only operations that immediately reduce the state space. Cleaveland and Riely [8] abstract value-passing CCS processes by mappings on data domains. They show that an abstract value-passing process is greater in the specification preorder than the original process. The abstraction operations do not operate on process structure, and their logical effects are not given.

References

[1] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 3(6):333–344, 1984.

[2] T. Bolognesi and E. Brinksma. Introduction to the specification language LOTOS. In van Eijk, Vissars, and Diaz, editors, *The Formal Description Technique LOTOS*. Elsevier, 1989.

[3] Glenn Bruns. A practical technique for process abstraction. In *Proceedings of CONCUR '93, LNCS 715*, pages 37–49, 1993.

[4] Glenn Bruns. *Process Abstraction in the Verification of Temporal Properties*. PhD thesis, University of Edinburgh, 1997. Published as report ECS-LFCS-98-380 by the Department of Computer Science.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[6] Rance Cleaveland and Matthew Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2), 1990.

[7] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[8] Rance Cleaveland and James Riely. Testing-based abstractions for concurrent systems. In *Proceedings of CONCUR '94, LNCS 836*, pages 417–432, 1994.

[9] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[10] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, Centre for Mathematics and Computer Science, CWI, 1990.

[11] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5), 1966.

[12] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[13] Nancy A. Lynch. Multivalued possibilities mapping. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 519–543, 1989. LNCS 430.

[14] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[15] J.L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison Wesley, 1985.

[16] Colin Stirling. An introduction to modal and temporal logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, pages 2–20, 1989. LNCS 491.

[17] D. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1:273–292, 1989.