Liveness with $(0, 1, \infty)$-Counter Abstraction*

Category A (Research) Paper

Amir Pnueli
Weizmann Institute of Science
amir@wisdom.weizmann.ac.il

Jessie Xu
New York University
jessie@cs.nyu.edu

Lenore Zuck
New York University
zuck@cs.nyu.edu

Abstract

We introduce the $(0, 1, \infty)$-*counter abstraction* method by which a parameterized system of unbounded size is abstracted into a finite-state system. Assuming that each process in the parameterized system is finite-state, the abstract variables are limited counters which count, for each local state s of a process, the number of processes which currently are in local state s. The counters are saturated at 2, which means that $\kappa(s) = 2$ whenever 2 *or more* processes are at state s. The emphasis of the paper is on the derivation of an adequate and sound set of fairness requirements (both weak and strong) which enable proofs of liveness properties of the abstract system, from which we can safely conclude a corresponding liveness property of the original parameterized system. We illustrate the method on few parameterized systems, including Szymanski's Algorithm for mutual exclusion.

The method is then extended to deal with parameterized systems whose processes may have infinitely many local states, such as the Bakery Algorithm. The extension is based on a choice of few "interesting" and "relevant" state assertions and then $(0, 1, \infty)$-counting the number of processes satisfying each of these assertions.

keywords: Parameterized systems, Finitary Abstraction, Verifying Liveness Properties

1 Introduction

The problem of *uniform verification of parameterized systems* is one of the most challenging problems in verification today. Given a parameterized system $S(N) : P[1] \| \cdots \| P[N]$ and a property p, uniform verification attempts to verify $S(N) \models p$ for every $N > 1$. One of the promising approaches to the verification of infinite-state systems (and a parameterized system is essentially such a system, due to the unbounded value of the parameter N), is the method of *finitary abstraction* by which we abstract an infinite-state system into a finite-state one, which can then be model checked. A general theory of finitary abstraction which jointly abstracts a system together with the property to be proven is presented in [KP00c, KP00a]. This method can be applied to the verification of arbitrary LTL properties, including liveness properties.

In this paper, we present a special finitary abstraction method for parameterized systems, to which we refer as *counter abstraction*. Assume that the processes $P[i]$ in the parameterized system have similar programs, where the program has locations $0, \ldots, L$. Assume first that processes do not have local variables (except for the program counter). In this case, the abstract state space is defined by having abstract variables $\kappa_0, \ldots, \kappa_L$. Abstract variable κ_j is intended to count the number of concrete processes which currently execute at location j. To guarantee that the abstract system is finite-state, we bound the range of κ_j to be $[0..2]$. A value $\kappa_j = n < 2$ describes the situation that precisely n processes currently execute at location j. A value $\kappa_j = 2$ describes the situation that 2 *or more* processes currently execute at location j. Certainly, such an abstraction is adequate for specifying properties of mutual exclusion which requires that the number of currently critical processes never exceed 1. The case that processes

*This research was supported in part by the Minerva Center for Verification of Reactive Systems, a gift from Intel, the European Community IST project "Advance", and ONR grant N00014-99-1-0131.

do possess additional (finite-state) local variables can also be treated by counter abstraction, only in this case we allocate a counter κ_s to each *local state* s of a process. The counter κ_s counts (up to 2) the number of processes which currently have the local state s.

The idea of this simple abstraction is certainly not new and usually serves as one of the first examples of abstraction and abstract interpretation [CC77]. The main contribution of this paper is to present a systematic method for enriching the counter abstraction with adequate set of fairness conditions which can be automatically derived and illustrate its application to verify effectively and efficiently various liveness properties of parameterized systems.

Even though the theory of linear abstraction, as presented in [KP00c], is aimed at verifying liveness (and general LTL) properties, it admits that the general recipe of abstraction is often too weak to provide a working finitary abstraction. In order to obtain a complete method, [KP00c] suggests to augment the system under consideration by an auxiliary monitoring module and abstract the composed system. While this may be theoretically satisfactory, no method is provided for designing the augmenting monitor.

For parameterized systems with counter abstraction, this paper provides a systematic construction of the necessary additional fairness requirements which are adequate for establishing liveness properties of non-trivial parameterized systems.

The method can be applied with no extensions to parameterized systems in which every process has only finitely many local states and the global variables range over finite domains. However, it can be extended to deal with systems in which individual processes may have infinitely many local states. The necessary extension is that we identify some state assertions and then allocate (bounded) counters which count the number of processes whose local states satisfy these assertions. This extension is illustrated in Section 5 where we verify the correctness of the Bakery Algorithm.

Related Work. The work which obtains results closest to ours is [BLS01] and previous articles describing the PAX system. They also address the problem of verification of parameterized systems, with emphasis on liveness properties and manage to verify individual accessibility of Szymanski's protocol. One of the differences between the two approaches is that [BLS01] is based on the method of *predicate abstraction* [BBM95] by which the abstract space is determined by the assertions appearing in the transition system and in the temporal property to be verified. As a result, the structure of the abstraction varies from case and the computation of the added fairness requirements depends on a marking algorithm which has to be re-applied for each case separately, searching for appropriate well-founded ranking. In comparison, we propose a uniform abstraction approach based on the notion of counter abstraction, which allows us to provides a standard recipe for the additional compassion (strong fairness) requirements, as well as the addition of justice (weak fairness) requirements, which [BLS01] claims cannot be automatically lifted from the concrete to the abstract systems (unless it is for a distinguished process which is not abstracted). Thus, due to our focus on a single uniform abstraction, we are able to derive a richer set of abstract fairness conditions, including weak as well as strong fairness. Like [BLS01], we also use an implementation of WS1S in order to compute the abstracted system. Another relative advantage of our method is its extended ability to deal with parameterized systems whose individual processes are not necessarily finite-state.

The problem of uniform verification of parameterized systems is, in general, undecidable [AK86]. There are two possible remedies to this situation: either we should look for restricted families of parameterized systems for which the problem becomes decidable, or devise methods which are sound but, necessarily incomplete, and hope that the system of interest will yield to one of these methods.

Among the representatives of the first approach we can count the work of German and Sistla [GS92] which assumes a parameterized system where processes communicate synchronously, and shows how to verify single-index properties. Similarly, Emerson and Namjoshi provided a decision procedure for proving a restricted set of properties on ring algorithms [EN95], and proved a PSPACE complete algorithm for verification of synchronously communicating processes [EN96]. Many of these methods fail when we move to asynchronous systems where processes communicate by shared variables. Perhaps the most advanced of this approach is the paper [EK00] which considers a general parameterized system allowing several different classes of processes. However, this work provides separate algorithms for the cases that the guards are either all disjunctive or all conjunctive. A protocol such as the cache example we consider in [PRZ01] which contains some disjunctive and some conjunctive guards, cannot be handled by the methods of [EK00].

The sound but incomplete methods include methods based on explicit induction ([EN95]) network invariants, which can be viewed as implicit induction ([KM95], [WL89], [HLR92], [LHR97]), methods that can be viewed as abstraction and approximation of network invariants ([BCG86], [SG89], [CGJ95], [KP00b]), and other methods that can be viewed as based on abstraction ([ID96]). The papers in [CR99a, CR99b, CR00] use structural induction based on the notion of a network invariant but significantly enhance its range of applicability by using a generalization of the data-independence approach which provides a powerful abstraction capability, allowing it to handle network with parameterized topologies. Most of these methods require the user to provide auxiliary constructs, such as a network invariant or an abstraction mapping. Other attempts to verify parameterized protocols such as Burn's protocol [JL98] and Szymanski's algorithm [GZ98, MAB+94] relied on abstraction functions or lemmas provided by the user. The work in [LS97] deals with the verification of safety properties of parameterized networks by abstracting the behavior of the system. PVS ([SOR93]) is used to discharge the generated VCs.

Among the automatic incomplete approaches, we should mention the methods relying on "regular model-checking" [KMM+97, ABJN99, JN00, PS00], where a class of systems which include our bounded-data systems as a special case is analyzed representing linear configurations of processes as a word in a regular language. Unfortunately, many of the systems analyzed by this method cause the analysis procedure to diverge and special *acceleration* procedures have to be applied which, again, requires user ingenuity and intervention.

The works in [ES96, ES97, CEFJ96, GS97] study symmetry reduction in order to deal with state explosion. The work in [ID96] detects symmetries by inspection of the system description. Closer in spirit to our work is the work of McMillan on compositional model-checking (e.g. [McM98]), which combines automatic abstraction with finite-instantiation due to symmetry.

2 Parameterized Systems

The systems we consider here consist of a parallel composition of N symmetric processes, all executing the same program, that may refer to the id of the executing process. In addition, the process programs may access a set of global shared variables and each other's local variables. Accesses to the local variables of other processes can be done under existential or universal quantification. A simple example of such a system is MUX-SEM in Fig. 1. There, each of the processes is executing exactly the same code (that does not refer to the id of the executing process), and there are only references to the single shared variable y.

$$
\begin{array}{lll}
\textbf{in} & N & : \ \textbf{natural where } N > 1 \\
\textbf{local} & y & : \ \textbf{boolean where } y = 1
\end{array}
$$

$$
\overset{N}{\underset{i=1}{\Big|\Big|}} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0: & \textbf{Non-Critical} \\ 1: & \textbf{request } y \\ 2: & \textbf{Critical; release } y \end{array} \right] \end{array} \right]
$$

Figure 1: Program MUX-SEM

The semaphore instructions "**request** y" and "**release** y" appearing in the program stand, respectively, for $\langle \textbf{when } y = 1 \textbf{ do } y := 0 \rangle$ and $y := 1$. As seen in Fig. 1, we use the simple programming language SPL [MP95, MAB+94] for presenting programs.

2.1 The Computational Model

For a computational model we use *fair discrete systems*, which is a slight variation on the *fair transition systems* (FTS) model of [MP95]. An FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of typed *system variables*, containing data and control variables. The set of *states* (interpretation) over V is denoted by Σ.
- Θ : The *initial condition* – an *assertion* (first-order state formula) characterizing the initial states.
- ρ : A *transition relation* – an assertion $\rho(V, V')$, relating the values V of the variables in state $s \in \Sigma$ to the values V' in a $\mathcal{D}$-successor state $s' \in \Sigma$.
- $\mathcal{J} : \{J_1, \ldots, J_k\}$: A set of *justice* (*weak fairness*) *requirements*. The justice requirement $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-state (states satisfying J).
- $\mathcal{C} : \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$: A set of *compassion* (*strong fairness*) *requirements*. The compassion requirement $\langle p, q\rangle \in \mathcal{C}$ is a pair of assertions, intended to guarantee that every computation containing infinitely many p-states also contains infinitely many q-states.

We require that every state $s \in \Sigma$ has at least one $\mathcal{D}$-successor. This is often ensured by including in ρ the *idling* disjunct $V = V'$ (also called the *stuttering* step). In such cases, every state s is its own $\mathcal{D}$-successor.

There are several different ways to view a parameterized system $S(N)$. One way considers $S(N)$ as a representation of an infinite family of systems, one for each value assumed by N. We prefer to view $S(N)$ as a single FDS, where the parameter N is considered an input variable which retains its value throughout the computation. The single FDS MUX-SEM corresponding to program MUX-SEM is presented in Fig. 2.

$$V : \left\{ \begin{array}{ll} N : & \textbf{natural} \\ y : & \textbf{boolean} \\ \pi : & \textbf{array } [1..N] \textbf{ of } \{0,1,2\} \end{array} \right. \qquad \Theta : \quad y \ \wedge\ N > 1 \ \wedge\ \forall i : [1..N] : \pi[i] = 0$$

$$\rho : \ \exists i : [1..N] \left(\begin{array}{ll} & \pi'[i] = \pi[i] \ \wedge\ y' = y \\ \vee & \pi[i] = 0 \ \wedge\ \pi'[i] = 1 \ \wedge\ y' = y \\ \vee & \pi[i] = 1 \ \wedge\ y = 1 \ \wedge\ \pi'[i] = 2 \ \wedge\ y' = 0 \\ \vee & \pi[i] = 2 \ \wedge\ \pi'[i] = 0 \ \wedge\ y' = 1 \end{array} \right) \ \wedge\ \forall j \neq i : \pi'[j] = \pi[j] \ \wedge\ N = N'$$

$$\mathcal{J} : \ \Big\{ \pi[i] \neq 2 \quad | \quad i \in [1..N] \Big\} \qquad \mathcal{C} : \ \Big\{ \langle \pi[i] = 1 \ \wedge\ y \ ,\ \pi[i] = 2 \rangle \quad | \quad i \in [1..N] \Big\}$$

Figure 2: MUX-SEM–The FDS corresponding to program MUX-SEM.

A *computation* of an FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the following requirements:

Initiality: s_0 is initial, i.e., $s_0 \models \Theta$.

Consecution: For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

Justice: For each $J \in \mathcal{J}$, σ contains infinitely many J-positions.

Compassion: For each $\langle p, q\rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, then it also contain infinitely many q-positions.

For an FDS $\mathcal{D}$, we denote by $\mathcal{C}omp(\mathcal{D})$ the set of all computations of $\mathcal{D}$.

2.2 Verification of Parameterized Systems

When verifying properties of a parameterized system Sys, one wants to show that the property of interest is valid for all values of N. We focus on properties that are expressible by a formula of the type

$\varphi : \quad \forall i_1, \ldots, i_k : f(i_1, \ldots, i_k)$

where $i_1, \ldots, i_k$ are indices of processes and f is a temporal formula over the free variables $i_1, \ldots, i_k$. Thus, to show that φ is valid over a parameterized system Sys, it is necessary to show that every computation $\sigma \in \mathcal{C}omp(Sys)$ satisfies:

$$\sigma \ \models\ N \geq k \rightarrow \forall i_1, \ldots, i_k : [1..N] : f(i_1, \ldots, i_k)$$

For example, the safety property of MUX-SEM, requiring that no two processes are ever at the critical section at the same time, is given by $\forall i \neq j : \Box \neg(\pi[i] = 2 \;\wedge\; \pi[j] = 2)$. Thus, to prove safety one has to show that for every $N \geq 2$, the property

$$\forall i \neq j : \Box \neg(\pi[i] = 2 \;\wedge\; \pi[j] = 2)$$

holds for every computation of $\mathcal{C}omp$(MUX-SEM).

As stated in the introduction, automatic verification of parameterized systems is notoriously difficult. Many methods, including [PRZ01, APR+01], have been proposed for proving safety properties for a large class of such systems. Our goal here is to automatically prove liveness properties of such systems. As a matter of fact, our methodology allows us to also automatically prove safety properties, at times more efficiently than allowed by other methods.

Our approach is based on the method of *finitary abstraction* introduced in [KP00b]. The idea behind finitary abstraction (as well as other abstraction methods, all inspired by [CC77]) is to reduce the problem of verifying that a given *concrete* system satisfies its (concrete) specifications, into a problem of verifying that some (carefully crafted, finite-state) *abstract* system satisfies its (finite-state) abstract specifications.

Given an FDS $\mathcal{D}$ and a property ψ that we want to show true for every computation of $\mathcal{D}$, *verification by finitary abstraction* defines a finitary abstraction mapping α to abstract the concrete FDS $\mathcal{D}$ into a finite-state abstract FDS $\mathcal{D}^\alpha$ and the property ψ into a finitary abstract property ψ^α, and then verifies that $\mathcal{D}^\alpha \models \psi^\alpha$. From this $\mathcal{D} \models \psi$ is concluded. We sketch here the basic elements of the method, and refer the reader to [KP00b] for additional details. In the next section we show a family of cases where the recipe results in a too trivial system to be of use, and extend the method to encompass these new data abstractions.

Assume a set of *abstract variables* V_A and a set of expressions $\mathcal{E}^\alpha$, such that the $V_A = \mathcal{E}^\alpha(V)$. Let $p(V)$ be a (concrete) assertion. Define

$$\alpha^+(p)\colon\; \exists V\Big(V_A = \mathcal{E}^\alpha(V) \wedge p(V)\Big) \quad \text{and} \quad \alpha^-(p)\colon\; \exists V(V_A = \mathcal{E}^\alpha(V)) \wedge \forall V\Big(V_A = \mathcal{E}^\alpha(V) \;\rightarrow\; p(V)\Big)$$

Given an abstract state S, $\alpha^+(p)$ holds over S if p holds over *some* concrete state s that is α-abstracted into S. The abstraction $\alpha^-(p)$ holds over S if S is an abstraction of some concrete state and p holds over *all* concrete states s that are α-abstracted into S. The abstractions α^- and α^+ can be generalized to temporal formulae (see [KP00b]). Denote by ψ^α the temporal formula obtained by $\alpha^-(\psi)$.

Given an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, and an abstraction scheme α as described. The abstract system that corresponds to $\mathcal{D}$ under a is the system $\mathcal{D}^\alpha = \langle V^\alpha, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha\rangle$ where:

$$\begin{array}{lcllcl} \Theta^\alpha & = & \alpha^+(\Theta) & \qquad \rho^\alpha & = & \alpha^{++}(\rho) \\ \mathcal{J}^\alpha & = & \{\alpha^+(J) \mid J \in \mathcal{J}\} & \qquad \mathcal{C}^\alpha & = & \{\langle \alpha^-(p), \alpha^+(q)\rangle \mid \langle p, q\rangle \in \mathcal{C}\} \end{array}$$

where

$$\alpha^{++}(\rho) = \exists V, V' : \Big(V_A = \mathcal{E}^\alpha(V) \;\wedge\; V'_A = \mathcal{E}^\alpha(V') \;\wedge\; \rho(V, V')\Big).$$

It is proven in [KP00b] that this abstraction method is sound. That is, if $\mathcal{D}^\alpha \models \psi^\alpha$ then $\mathcal{D} \models \psi$. Since formula ψ is an arbitrary temporal formula, this provides a sound abstraction method for verifying liveness as well as safety properties.

3 Counter Abstraction

We present *counter abstraction*, a novel data abstraction method, that is a special case of the finitary abstraction approach. Attempts to verify liveness properties of counter abstracted system, however, reveal that the standard recipe for abstracting fairness requirements is inadequate. We show how to extend the recipe to derive stronger fairness requirements for abstracted systems in general and counter abstracted systems in particular. These stronger fairness requirements support automatic verification of liveness properties.

3.1 Counter Abstracting States and Transitions

Consider a parameterized system Sys described by an SPL program as described in Section 2. Assume that the control variable of processes ($\pi[i]$) ranges over $0, \ldots, L$, and that the shared variables are $y_1, \ldots, y_b$. For simplicity, we assume first that the processes do not have any local variables. Thus, the concrete state variables are given by

$$N, \pi[1..N], y_1, \ldots, y_b$$

We define a *counter abstraction* α, where the abstract variables are given by

$$\kappa_0, \ldots, \kappa_L : \{0, 1, 2\};\ Y_1, \ldots, Y_b$$

and the abstraction mapping $\mathcal{E}^a$ is given by

$$\begin{array}{rcll} \kappa_\ell & = & \left\{ \begin{array}{ll} 0 & \forall i : [1..N] : \pi[i] \neq \ell \\ 1 & \exists i_1 : [1..N] : \pi[i_1] = \ell \ \wedge\ \forall i_2 \neq i_1 : \pi[i_2] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} & \text{for } \ell = 0, \ldots, L \\ Y_j & = & y_j & \text{for } j = 1, \ldots, b \end{array}$$

That is, κ_ℓ is 0 when there are no processes at location ℓ, it is 1 if there is exactly one process at location ℓ, and it is 2 if there are two or more processes at location ℓ.[1]

Since the systems we are dealing with are symmetric, all the processes have initially the same control value. Without loss of generality, assume it's location 0. Thus we have

$$\Theta^\alpha : \quad \kappa_0 = 2 \wedge \kappa_1 = \kappa_3 = \cdots = \kappa_L = 0 \wedge Y_1 = y_1^0 \wedge \cdots \wedge Y_b = y_b^0,$$

where $y_1^0, \ldots y_b^0$ denote the initial values of the concrete shared variables $y_1, \ldots, y_b$.

Next, we obtain ρ^α according to the "recipe" of the previous section. This can be done using a system supporting the logic WS1S, such as TLV[P] [KMM+97] or MONA [HJJ+95], which is rather costly. Alternately, we can follow the procedure sketched below. The details are easy to flesh out and omitted here for brevity. For every location ℓ in the SPL program that describes a process, assume the instruction at location ℓ is of the form:

$$\ell : \textbf{if } c \textbf{ then } [\ \vec{y} := f_1(\vec{y}); \textbf{ goto } \ell_1\ ] \textbf{ else } [\ \vec{y} := f_2(\vec{y}); \textbf{ goto } \ell_2\ ]$$

Suppose c^α is defined and that $\ell \neq \ell_1, \ell_2$. Then we include in ρ^a the following disjunction:

$$\begin{array}{ll} & \big(\kappa_\ell > 0 \ \wedge\ \ \ c^a \ \wedge\ \kappa'_\ell = \kappa_\ell \ominus 1 \ \wedge\ \kappa'_{\ell_1} = \kappa_{\ell_1} \oplus 1 \ \wedge\ \forall j \neq \ell, \ell_1 : \kappa_j = \kappa'_j \ \wedge\ \vec{Y}' = f_1(\vec{Y})\big)\big) \\ \vee & \big(\kappa_\ell > 0 \ \wedge\ \neg c^a \ \wedge\ \kappa'_\ell = \kappa_\ell \ominus 1 \ \wedge\ \kappa'_{\ell_2} = \kappa_{\ell_2} \oplus 1 \ \wedge\ \forall j \neq \ell, \ell_2 : \kappa_j = \kappa'_j \ \wedge\ \vec{Y}' = f_2(\vec{Y})\big)\big) \end{array}$$

where $x' = x \oplus 1$ is an abbreviation for:

$$(x = 0 \wedge x' = 1) \vee (x = 1 \wedge x' = 2) \vee (x = 2 \wedge x' = 2)$$

and $x' = x \ominus 1$ is an abbreviation for:

$$(x = 1 \wedge x' = 0) \vee (x = 2 \wedge x' = 1) \vee (x = 2 \wedge x' = 2)$$

Example 1 Fig. 3 presents graphically the FDS obtained by counter abstracting MUX-SEM, where each state displays the values assigned to the abstract variables $\kappa_0, \kappa_1, \kappa_2; Y$. To make the figure more compact, we removed states and transitions that obtain only for small values of N (namely, for $N < 5$.) For example, all states where every κ_i is less than two (implying $N < 4$) is removed; so is the transition that leads from $\langle 2, 1, 1; 0\rangle$ into $\langle 1, 2, 1; 0\rangle$ (since it implies that $N = 4$.) The initial state is identified by an entry edge.

We note that there are self loop on each of the states, which is to be expected considering we assume there is always an idle transition, and when there are self loops in the concrete systems they also appear in the abstract system.

Establishing the safety property is now trivial, since the counter abstraction of the safety property $\varphi : \ \forall i \neq j : \Box\, \neg(\pi[i] = 2 \ \wedge\ \pi[j] = 2)$ is $\varphi^\alpha \ = \ \Box(\kappa_2 \leq 1)$, which trivially holds for MUX-SEM$^\alpha$.

[1]The upper bound of 2 can be substituted with other positive integer. In particular 1 can be used for most locations.

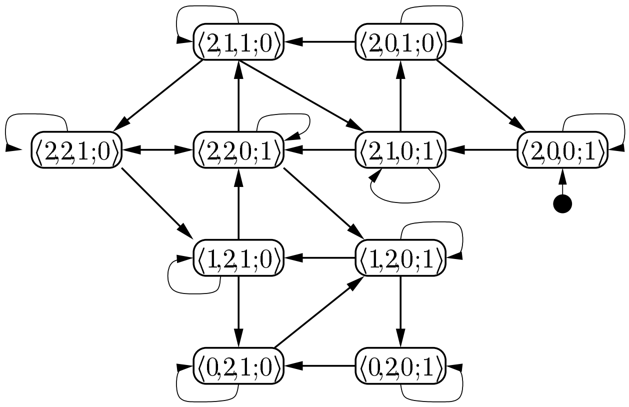


Figure 3: Counter Abstraction of MUX-SEM ($N \geq 5$)

3.2 Deriving Abstract Justice Requirements

Consider a single justice requirement of MUX-SEM $J: \ (\pi[i] \neq 2)$. The recipe in the previous section defines $J^\alpha = \alpha^+(J)$, which should encompass all abstract states S that have at least one α-source satisfying $\pi[i] \neq 2$. It is not difficult to see that this yields the abstract assertion $J^\alpha = \kappa_0 + \kappa_1 > 0$. Since, as can be seen from Fig. 3, all reachable states satisfy this assertion, J^α does not introduce any meaningful constraint on computations, and the counter abstraction of a system with the requirement J will be equivalent to the system without this requirement. It follows that any liveness property (such as accessibility) that is not valid for system MUX-SEM without the justice requirement J cannot be proven by an abstraction which only abstracts J into $\alpha^+(J)$.

We now develop a method by which stronger abstract justice requirements can be derived. Let φ be an abstract assertion, i.e., an assertion over the abstract state variables V_A. We say that φ *suppresses the concrete justice requirement* J if, for every two concrete states s_1 and s_2 such that s_2 is a ρ-successor of s_1, and both $\alpha(s_1)$ and $\alpha(s_2)$ satisfy φ, $s_1 \models \neg J$ implies $s_2 \models \neg J$.

For example, the abstract assertion $\varphi: \kappa_2 = 1$ suppresses the concrete justice requirement $J: \pi[i] \neq 2$. Indeed, assume two states s_1 and s_2 such that s_2 is a successor of s_1 and both are counter-abstracted into abstract states satisfying $\kappa_2 = 1$. This implies that both s_1 and s_2 have precisely one process executing at location 2. If s_1 satisfies $\neg J = (\pi[i] = 2)$ then the single process executing at location 2 within s_1 must be $P[i]$. Since s_2 is a successor of s_1 and also has a single process executing at location 2, it must also be the same process $P[i]$, because it is impossible for $P[i]$ to exit location 2 and another process to enter the same location all within a single transition.

The abstract assertion φ is defined to be *justice suppressing* if, for every concrete state s such that $\alpha(s) \models \varphi$, there exists a concrete justice requirement J such that $s \models \neg J$ and φ suppresses J.

For example, the assertion $\kappa_2 = 1$ is justice suppressing, because every concrete state s whose counter-abstraction satisfies $\kappa_2 = 1$ must have a single process, say $P[i]$, executing at location 2. In that case, s violates the justice requirement $J: \pi[i] \neq 2$ which is suppressed by φ.

Theorem 1 *Let $\mathcal{D}$ be a concrete system and α be an abstraction that we apply to $\mathcal{D}$. Assume that $\varphi_1, \ldots, \varphi_k$ is a list of justice suppressing abstract assertions. Let $\mathcal{D}^\alpha$ be the abstract system obtained by following the abstraction recipe described in Section 2 and then adding $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ to the set of abstract justice requirements. If $\mathcal{D}^\alpha \models \psi^\alpha$ then $\mathcal{D} \models \psi$.*

Thus, we can safely add $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ to the set of justice requirement, while preserving the soundness of the method.

The proof of the theorem is based on the observation that every abstraction of a concrete computation must contain infinitely many $\neg\varphi$-states for every justice suppressing assertion φ. Therefore, the abstract computations removed from the abstract system by the additional justice requirements can never correspond to abstractions of concrete computations, and it is safe to remove them.

Theorem 1 is very general (not even restricted to counter abstraction) but does not provide us with guidelines for the choice of the justice suppressing assertions. For the case of counter-abstraction, we can provide some practical guidelines, as follows:

G1. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell)$, then the assertion $\kappa_\ell = 1$ is justice suppressing.

G2. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge\ c)$, where c is a condition on the shared variables (that are kept intact by the counter-abstraction), then the assertion $\kappa_\ell = 1 \wedge c$ is justice suppressing.

G3. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell)$ and the only possible move from location ℓ is to location $\ell + 1$, then the two assertions $\kappa_\ell > 0 \ \wedge\ \kappa_{\ell+1} = 0$ and $\kappa_\ell > 0 \ \wedge\ \kappa_{\ell+1} = 1$ are justice suppressing.

G4. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge\ c)$, where c is a condition on the shared variables, and the only possible move from location ℓ is to location $\ell + 1$, then the assertions $\kappa_\ell > 0 \ \wedge\ \kappa_{\ell+1} = 0 \ \wedge\ c$ and $\kappa_\ell > 0 \ \wedge\ \kappa_{\ell+1} = 1 \ \wedge\ c$ are justice suppressing.

Example 2 We state justice properties of MUX-SEM$^\alpha$ according to the above guidelines. Since for MUX-SEM we have the justice $\neg(\pi[i] = 2)$, and since every move from location 2 leads to location 0, then the assertions $\kappa_2 = 1$, $\kappa_2 > 0 \ \wedge\ \kappa_0 = 0$, and $\kappa_2 > 0 \ \wedge\ \kappa_0 = 1$ are all justice suppressing and their negation can be added to $\mathcal{J}^\alpha$. Similarly, the concrete compassion requirement $\langle \pi[i] = 1 \ \wedge\ y, \pi[i] = 2 \rangle$ implies that the concrete assertion $\neg(\pi[i] = 1 \ \wedge\ y)$ is a justice requirement for system MUX-SEM. We can therefore add $\neg(\kappa_1 = 1 \ \wedge\ y)$ to the abstract justice requirement. Since every move from location 1 leads to location 2, we can also add $\neg(\kappa_1 > 0 \ \wedge\ \kappa_2 = 0 \ \wedge\ y)$ to the abstract justice requirements.

Note: All justice requirements of MUX-SEM$^\alpha$ have been generated automatically by a general procedure implementing all the rules specified in the guidelines above.

The fact that abstract assertion φ is justice suppressing can be formalized by the following formula:

$$\forall V : \varphi(\mathcal{E}^\alpha(V)) \ \rightarrow\ \exists J \in \mathcal{J} : \neg J(V) \wedge \forall V_1, V_2 : \rho(V_1, V_2) \wedge \varphi(\mathcal{E}^\alpha(V_1)) \wedge \varphi(\mathcal{E}^\alpha(V_2)) \wedge \neg J(V_1) \ \rightarrow\ \neg J(V_2)$$

3.3 Proving Liveness

The liveness property one usually associates with parameterized systems is *individual accessibility* of the form $\forall i : \Box(\pi[i] = \ell_1 \rightarrow \Diamond(\pi[i] = \ell_2))$. Unfortunately, counter-abstraction does not allow us to observe the behavior of an individual process. Therefore, the property of individual accessibility cannot be expressed (and, consequently, verified) in a counter-abstracted system. In Section 3.4 we show how to extend counter-abstraction to handle individual accessibility properties.

There are, however, liveness properties that *are* expressible and verifiable by counter abstraction. Such are *live-lock freedom* (sometime called *communal accessibility*) properties of the form $\psi : \ \Box(\exists i : \pi[i] = \ell_1 \rightarrow \Diamond(\exists i : \pi[i] = \ell_2))$, stating that if *some* process is at location ℓ_1, then eventually *some* process (not necessarily the same) will enter ℓ_2. The counter-abstraction of such a property is $\psi^\alpha : \ \Box(\kappa_{\ell_1} > 0 \rightarrow \Diamond(\kappa_{\ell_2} > 0))$. Model checking that ψ^α holds over Sys^α can be accomplished by standard model checking techniques of response properties. E.g., the procedure in [LP85] suggests extracting from the state-transition graph of the subgraph of *pending states* and showing that it contains no infinite fair path. A pending state for a property $p \rightarrow \Diamond\, q$ is any state which is reachable from a p-state by a q-free path.

Example 3 Consider the system MUX-SEM$^\alpha$ of Fig. 3 and the abstract live-lock freedom property $\chi : \Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_2 > 0))$. In Fig. 4, we present the subgraph of pending states for the property χ over the system MUX-SEM$^\alpha$.

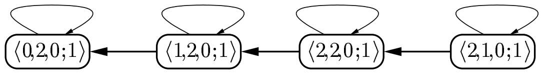


Figure 4: Reachability Graph for χ

The way we show that this graph contains no infinite fair path is to decompose the graph into maximal strongly connected components and show that each of them is *unjust*. A strongly connected subgraph (SCC) S is unjust if there exists a justice requirement which is violated by all states within the subgraph. In the case of the graph in Fig. 4 there are four maximal SCC's. Each of these subgraphs is unjust towards the abstract justice requirement $\neg(\kappa_1 > 0 \wedge y)$ derived in Example 2.

We conclude that the abstract property $\square(\kappa_1 > 0 \rightarrow \diamondsuit(\kappa_2 > 0))$ is valid over system MUX-SEM$^\alpha$ and, therefore, the property $\square(\exists i : \pi[i] = 1 \rightarrow \diamondsuit(\exists i : \pi[i] = 2))$ is valid over MUX-SEM.

3.4 Proving Individual Accessibility

As indicated before, individual accessibility cannot be directly verified by standard counter abstraction which cannot observe individual processes. To prove individual accessibility for the generic process $P[t]$, we abstract the system by counter abstracting all the processes except for $P[t]$, whom we leave intact. We then prove that the abstracted system satisfies the liveness property (the abstraction of which leaves it unchanged, since it refers to $\pi[t]$ that is kept intact by the abstraction), from which we derive that the concrete system satisfies it as well.

The new abstraction, which is "counter abstraction save one", is denoted by γ. As before, we assume for simplicity that the processes possess no local variables except for their program counter. The abstract variables for γ are given by

$$\kappa_0, \ldots, \kappa_L : [0..2],\ \Pi : [0..L];\ Y_1, \ldots, Y_b$$

and the abstraction mapping $\mathcal{E}^\gamma$ is given by

$$\begin{array}{rcll}
\kappa_\ell & = & \left\{ \begin{array}{ll} 0 & \forall r : [1..N] - \{t\} : \pi[r] \neq \ell \\ 1 & \exists r : [1..N] - \{t\} : \pi[r] = \ell \ \wedge\ \forall j \notin \{r,t\} : \pi[t] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} & \text{for } \ell = 0, \ldots, L \\
\Pi & = & \pi[t] & \\
Y_k & = & y_k & \text{for } k = 1, \ldots, b
\end{array}$$

We obtain ρ^γ in the obvious way. For $\mathcal{J}^\gamma$, we include all the justice requirements obtained by the recipe of [KP00b] and the guidelines of Subsection 3.2, with all requirements in $\mathcal{J}$ that relate to $P[t]$. For $\mathcal{C}^\gamma$ we take all the requirements in $\mathcal{C}$ that relate only to $P[t]$.

To prove $\varphi : \square(\Pi = 1 \rightarrow \diamondsuit(\Pi = 2))$, consider the subgraph of $\mathcal{D}^\gamma$ that consists of all the abstract states that are reachable from a $(\Pi = 1)$-state by a $(\Pi = 2)$-free paths, and show, as before, that this subgraph contains no infinite fair path.

Example 4 Consider the system MUX-SEM$^\gamma$ and the liveness property $\varphi^\gamma :\ \square((\Pi = 1) \rightarrow \diamondsuit(\Pi = 2))$. The subgraph of pending states is presented in Fig. 5. Each state in this graph is labeled by a tuple which specifies the values assigned by the state variables $\langle \kappa_0, \kappa_1, \kappa_2; \Pi; Y \rangle$.
Unlike the previous case, this system has the compassion requirement $\langle \Pi = 1 \ \wedge\ Y, \Pi = 2 \rangle$ associated with $P[t]$. Since the subgraph contains no state satisfying $\Pi = 2$, no fair path can pass infinitely often through any state satisfying $\Pi = 1 \ \wedge\ Y$. Therefore, as a first step, we remove from the graph all states satisfying $\Pi = 1 \ \wedge\ Y$. This leads to the graph presented in Fig. 6.
This graph consists of five maximal SCC's, each of which is unjust towards the abstract justice requirement $\neg(\kappa_2 = 1)$ derived according to rule 1 of the guidelines of Subsection 3.2.

We conclude that the abstract property $\square((\Pi = 1) \rightarrow \diamondsuit(\Pi = 2))$ is valid over MUX-SEM$^\gamma$ and, therefore, $\square((\pi[t] = 1) \rightarrow \diamondsuit(\pi[t] = 2))$ is valid over MUX-SEM.

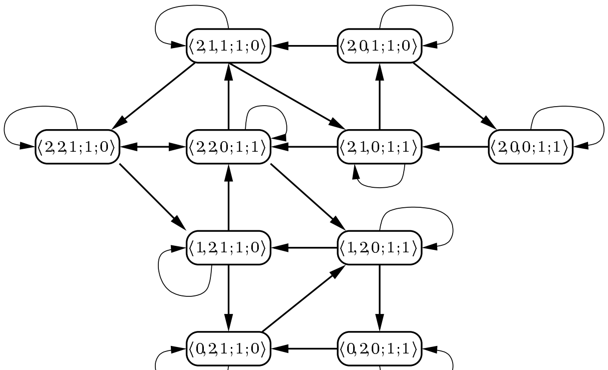


Figure 5: The states of MUX-SEM$^\gamma$ which are pending with respect to φ

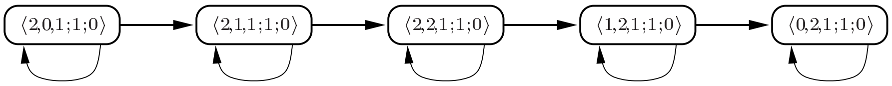


Figure 6: Pending states of MUX-SEM$^\gamma$ after removal of all $(\Pi = 1 \ \wedge \ Y)$-states

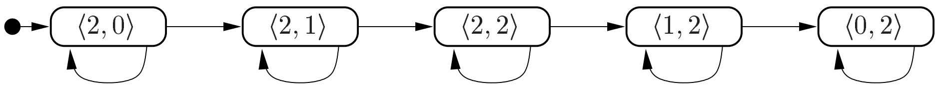
4 Compassion

In Subsection 3.2 we showed how to derive additional justice requirements for a counter-abstracted system. We now turn to abstract compassion requirements, that do not always correspond to concrete compassion requirements. Here we derive abstract compassion requirements which reflect well-founded properties of some of the concrete data domains. Consider program TERMINATE presented in Fig. 7.

$$\begin{array}{l} \textbf{in} \quad N : \textbf{natural where } N > 1 \\ \displaystyle\mathop{\Big\|}_{i=1}^{N} P[i] :: \quad \left[\begin{array}{ll} 0: & \textbf{skip} \\ 1: & \end{array} \right] \end{array}$$

Figure 7: Program TERMINATE

The liveness property we would like to establish for this program is given by the formula $\varphi : \diamondsuit(\forall i : \pi[i] = 1)$, stating that eventually, all processes reach location 1. The counter abstraction for this program (assuming $N > 2$) is given by the following graph, representing system TERMINATE$^\alpha$:



In this graph, each state s is labeled by the values state s assigns to the abstract variables κ_0 and κ_1. The abstracted property φ^α is given by $\diamondsuit(\kappa_0 = 0)$. However, this property is not valid over TERMINATE$^\alpha$, even when we take into account all the justice requirements derived according to Subsection 3.2. These justice requirements force the computation to eventually exit states $\langle 2, 0\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$, but they do not prevent the computation from staying forever in state $\langle 2, 2\rangle$.

To obtain a fairness requirement which will force the computation to eventually exit state $\langle 2, 2\rangle$ we augment the system with two additional abstract variables and a corresponding compassion requirement that governs their behavior.

Let $Sys : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an FDS representing a concrete parameterized system, where the locations of each process are $[0..L]$. We define an *augmented system* $\mathcal{D}^* : \langle V^*, \Theta^*, \rho^*, \mathcal{J}^*, \mathcal{C}^* \rangle$ as follows:

$$
\begin{array}{lcl}
V^* & : & V \ \cup \ \{\mathit{from}, \mathit{to} : [-1..L]\} \\
\Theta^* & : & \Theta \ \wedge \ (\mathit{from} = -1) \ \wedge \ (\mathit{to} = -1) \\
\rho^* & : & \rho \ \wedge \left(\begin{array}{ll} & \exists i : [1..N], \ell_1 \neq \ell_2 : [0..L] : \pi[i] = \ell_1 \ \wedge \ \pi'[i] = \ell_2 \ \wedge \ (\mathit{from}' = \ell_1) \ \wedge \ (\mathit{to}' = \ell_2) \\ \vee & \forall i : [1..N] : \pi'[i] = \pi[i] \ \wedge \ (\mathit{from} = -1) \ \wedge \ (\mathit{to} = -1) \end{array} \right) \\
\mathcal{J}^* & : & \mathcal{J} \\
\mathcal{C}^* & : & \mathcal{C} \ \cup \ \{\langle \mathit{from} = \ell, \ \mathit{to} = \ell \rangle \ \mid \ \ell \in [0..L]\}
\end{array}
$$

Thus, system Sys is augmented with two auxiliary variables, *from* and *to*. Whenever a transition causes some process to move from location ℓ_1 to location $\ell_2 \neq \ell_1$, the same transition sets *from* to ℓ_1 and *to* to ℓ_2. Transitions that cause no process to change its location set both *from* and *to* to -1. For every $\ell \in [0..L]$, we add to Sys^* the compassion requirement $\langle \mathit{from} = \ell, \ \mathit{to} = \ell \rangle$. This compassion requirement represents the obvious fact that, since the overall number of processes is bounded (by N), processes cannot leave location ℓ infinitely many times without processes entering location ℓ infinitely many times.

Comparing the observable behavior of Sys and Sys^*, we can prove the following:

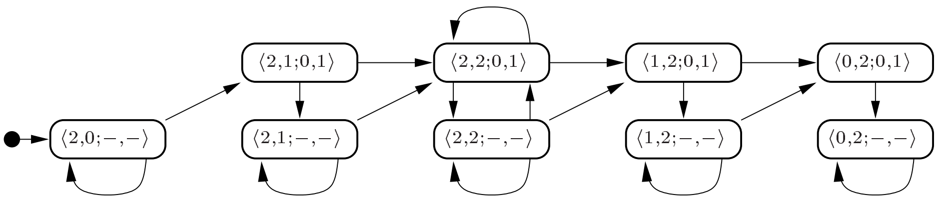
Claim 2 *Let σ be an infinite sequence of V-states. Then σ is a computation of Sys iff it is a V-projection of a computation of system Sys^*.*

Thus, the augmentation of Sys does not change its observable behavior.

Consequently, instead of counter-abstracting the system Sys, we can counter-abstract Sys^*. We denote by $Sys^\dagger = (Sys^*)^\alpha$ the counter abstraction of the augmented system Sys^*. In the presence of the auxiliary variable *from*, we can derive an even sharper justice requirements. We can replace the guidelines of Subsection 3.2 by the following single rule:

G5. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge \ c)$, where c is a condition on the shared variables and process indices other than i, then we may add to $Sys^\dagger$ the justice requirement $\mathit{from} = \ell \ \vee \ \neg(\kappa_\ell > 0 \wedge c^\alpha)$, where c^α is the counter abstraction of c.

Reconsider program TERMINATE. Applying the augmented abstraction, we obtain the following abstraction TERMINATE†, where each state s is labeled by the values s assigns to the variables $\langle \kappa_0, \kappa_1, \mathit{from}, \mathit{to} \rangle$ ($-$ stands for -1):



This system is augmented by the additional justice requirement $(\mathit{from} = 0) \ \vee \ (\kappa_0 = 0)$ (generated according to G5) and the compassion requirement $\langle \mathit{from} = 0, \ \mathit{to} = 0 \rangle$.

Analyzing the SCSs in the above graph, we find out that all of them, except for $\{\langle 0, 2; -, - \rangle\}$ are unfair. Subgraphs $\{\langle 2, 0; -, - \rangle\}$, $\{\langle 2, 1; -, - \rangle\}$, and $\{\langle 1, 2; -, - \rangle\}$ violate the justice requirement $(\mathit{from} = 0) \vee (\kappa_0 = 0)$. Subgraph $\{\langle 2, 2; 0, 1 \rangle, \langle 2, 2; -, - \rangle\}$ violates the compassion requirement $\langle \mathit{from} = 0, \ \mathit{to} = 0 \rangle$. The standard remedy for this is to remove the state satisfying $\mathit{from} = 0$, namely $\langle 2, 2; 0, 1 \rangle$ this leaves us with the SCS $\{\langle 2, 2; -, - \rangle\}$ which violates the justice requirement $(\mathit{from} = 0) \ \vee \ (\kappa_0 = 0)$. The remaining SCS's ($\{\langle 2, 1; 0, 1 \rangle\}$, $\{\langle 1, 2; 0, 1 \rangle\}$, and $\{\langle 0, 2; 0, 1 \rangle\}$) are singleton subgraphs which are not connected to themselves so the fair run cannot visit them more than once.

It follows that every fair run traversing the subgraph of TERMINATE† must eventually reach state $\langle 0, 2; -, - \rangle$ and remain there forever. It follows that TERMINATE† satisfies $\diamondsuit(\kappa_0 = 0)$ and, therefore, that the concrete system TERMINATE satisfies $\forall i : \diamondsuit(\pi[i] \neq 0)$.

5 Examples

In this section we describe some of our test cases. The codes for these are in *http://cs.nyu.edu/~zuck/counter*. Some of the FDSs are in appendices.

5.1 Szymanski's Mutual Exclusion Algorithm

Consider the mutual exclusion algorithm due to B. Szymanski [Szy88] presented in Fig. 8. The original algorithm used two local boolean variables: s and w in each process. However, since the values of these variables is determined by the location of the process owning them, it is possible to represent the algorithm using pure locations, which is the version presented in Fig. 8.

$$
\begin{array}{l}
\quad\quad\quad\quad\quad\quad \textbf{in} \quad N: \quad \textbf{integer where } N > 1 \\
\mathop{\Big\|}_{i=1}^{N} P[i] :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{ll}
0: & \textbf{NonCritical} \\
1: & \textbf{await } \forall j : \pi[j] \in \{0..2, 4\} \\
2: & \textbf{skip} \\
3: & \textbf{If } \exists j : \pi[j] \in \{1, 2\} \\
 & \quad\quad \textbf{then} \quad \textbf{goto } \ell_4 \\
 & \quad\quad \textbf{else} \quad \textbf{goto } \ell_5 \\
4: & \textbf{await } \exists j : \pi[j] \in \{5..7\} \\
5: & \textbf{await } \forall j : \pi[j] \notin \{3, 4\} \\
6: & \textbf{await } \forall j : j < i : \pi[j] \in \{0..2\} \\
7: & \textbf{Critical}
\end{array} \right]
\end{array} \right]
\end{array}
$$

Figure 8: Parameterized mutual exclusion Algorithm SZYMANSKI

Strictly speaking, the algorithm is not symmetric since the instruction in ℓ_6 depends on the id of the executing process. To prove mutual exclusion with a counter abstraction, it was necessary to add an additional abstract variable *lmin* to the abstraction. This variable maintains the location of the process with the minimal index among those in 3..7; *lmin* is 0 when there are no processes in locations 3..7. Once the abstraction mapping of this variable is defined, the abstract transition relation can be automatically computed. We present the resulting abstract FDS $\mathsf{szyman}^\dagger$, following the recipe of Section 4 in Appendix A. Using this abstraction we were able to verify the property of mutual exclusion of the original system as presented in Fig. 8.

To facilitate the proof of the live-lock freedom property $\chi : (\exists i : \pi[i] = 1) \rightarrow \diamondsuit(\exists i : \pi[i] = 7)$, $\mathsf{szyman}^\dagger$ contains the abstract justice requirements $\mathit{from} = \ell \vee \neg En(\tau)$ for each abstract transition τ departing from location $\ell \neq 0$, where $En(\tau)$ is the enabling condition for τ. Consider, for example, the transition corresponding to a process moving from location 6 to location 7. The enabling condition for the corresponding abstract transition is $\kappa_6 > 0 \;\wedge\; \mathit{lmin} = 6$. Consequently, we include in the abstract FDS the justice requirement $\mathit{from} = 6 \;\vee\; \neg(\kappa_6 > 0 \;\wedge\; \mathit{lmin} = 6)$. In addition, we included in the abstracted system the compassion requirements $\langle \mathit{from} = \ell, \mathit{to} = \ell \rangle$, for each $\ell = 0, \ldots, 7$. Using the abstraction $\mathsf{szyman}^\dagger$, we were able to verify the property χ ensuring freedom from live-lock. Note that the auxiliary variable *lmin* is essential only for the proof of mutual exclusion. Live-lock freedom can also be established without this auxiliary variable.

To prove individual accessibility, we applied γ abstraction to szyman^*. Some bookkeeping was needed to maintain the value of *lmin*, that was used, as before, to store the location of the non-t process whose index is minimal among those in locations 3..7 (including t.)

5.2 The Bakery Algorithm

Consider a variant of Lamport's Bakery Algorithm for mutual exclusion presented in Fig. 9. An interesting feature of the algorithm is the infinite data domain–processes choose "tickets" whose values are unboundedly large. To counter abstract bakery, we used a variable $lmin$, with a role similar to the one

$$
\begin{array}{l}
\qquad\qquad\qquad \textbf{in} \quad N: \ \textbf{naturl where } N > 1 \\
\qquad\qquad\qquad \textbf{local} \quad y: \ \textbf{array } [1..N] \textbf{ of natural init 0} \\
\mathop{\Big\|}\limits_{i=1}^{N} P[i] :: \left[\begin{array}{l}
\textbf{local } my\textbf{: natural} \\
\textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
0: & \textbf{NonCritical} \\
1: & my := (\max_{j=1}^{N} y[j]) + 1 \\
2: & y[i] := my \\
3: & \textbf{await } \forall j \neq i : \pi[j] \neq 2 \wedge (y[j] = 0 \vee (y[i], i) \prec_{lex} (y[j], j)) \\
4: & \textbf{Critical; } y[i] := 0
\end{array}\right]
\end{array}\right]
\end{array}
$$

Figure 9: The Bakery Algorithm

used in *Szymanski*. Here $lmin$ is the location of the process j whose $(y[j], j)$ is (lexicographically) minimal among those in locations 2..4. The system bakery† is described in Appendix B. The mutual exclusion property of the protocol, $\square(\kappa_4 < 2)$, as well as the live-lock freedom property, $\square(\kappa_1 > 0 \rightarrow \diamondsuit(\kappa_4 > 0))$ were easily established in TLV (the Weizmann Institute programmable model checker [PS96]). To establish the individual accessibility property, $\pi[t] = 1 \rightarrow \diamondsuit(\pi[t] = 4)$, we applied γ abstraction to bakery*. As in the case of *Szymanski*, some bookkeeping was needed to maintain the value of $lmin$, that was used, as before, to store the location of the non-t process whose index is minimal among those in locations 2..4 (including t.)

5.3 Run Time Results

In Fig. 10 we give the user run-time (in seconds) results of our various experiments. All verifications were carried in TLV.

Algorithm	Mutual Exclusion and Live-lock Freedom	Mutual Exclusion and Individual Accessibility (γ abstraction)
MUX-SEM	0.02	0.03
szyman	0.69	95.87
bakery	0.12	1.06

Figure 10: Run Time Results

6 Conclusions

We have presented the abstraction method of *counter abstraction*. The main novelty and contribution is the derivation of additional fairness conditions (both weak and strong) which enable us to perform automatic verification of interesting liveness properties of non-trivial parameterized systems.

References

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *CAV'99, LNCS 1633*, pages 134–145, 1999.

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV'01, LNCS 2102*, pages 221–234, 2001.

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *Lect. Notes in Comp. Sci.*, pages 589–623. Springer-Verlag, 1995.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[BLS01] K. Baukus, Y. Lakhnesche, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2), 8 1996.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR'95)*, pages 395–407, 1995.

[CR99a] S.J. Creese and A.W. Roscoe. Formal verification of arbitrary network topologies. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, 1999. CSREA Press.

[CR99b] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and fdr. In *Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99)*, Beijing, 1999. Kluwer Academic Publishers.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, Las Vegas, June 2000. CSREA Press.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In *CAV'96, LNCS 1102*, 1996.

[ES96] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2), 8 1996.

[ES97] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions. *ACM Trans. Prog. Lang. Sys.*, 19(4), 1997.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[GS97] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *CAV'97, LNCS 1254*, 1997.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In B. Steffen, editor, *TACAS'98, LNCS 1384*, pages 424–438, 1998.

[HJJ+95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS'95, LNCS 1019*, 1995. Also available through http://www.brics.dk/~klarlund/Mona/main.html.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In *CAV'96, LNCS 1102*, 1996.

[JL98] E. Jensen and N.A. Lynch. A proof of burn's n-process mutual exclusion algorithm using abstraction. In *TACAS'98, LNCS 1384*, pages 409–423, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *TACAS'00, LNCS 1785*, 2000.

[KM95] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV'97, LNCS 1254*, pages 424–435, 1997.

[KP00a] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(1):328–342, 2000.

[KP00b] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.

[KP00c] Y. Kesten and A. Pnueli. Verification by finitary abstraction. *Information and Computation, a special issue on Compositionality*, 163:203–243, 2000.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.

[LS97] D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction. In *2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, 1997.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, 1994.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV'98, LNCS 1427*, pages 110–121, 1998.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS'01*, volume 2031, pages 82–97, 2001.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV'96, LNCS 1102*, pages 184–195, 1996.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV'00, LNCS 1855*, pages 328–343, 2000.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 151–165. Springer-Verlag, 1989.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, 1988.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1989.

A Description of $\mathsf{szyman}^\dagger$

The transition system $\mathsf{szyman}^\dagger : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is defined as follows:

$$
\begin{array}{rcl}
V & : & \{\kappa_0,\ldots,\kappa_7 : [0..2],\ \mathit{lmin} : [0,3..7],\ \mathit{from},\mathit{to} : [-1..7]\} \\
\Theta & : & \kappa_0 = 2 \ \wedge\ \forall \ell \in \{1..7\} : \kappa_\ell = 0 \ \wedge\ \mathit{lmin} = 0 \ \wedge\ \mathit{from} = -1 \ \wedge\ \mathit{to} = -1 \\
\rho & : & (\kappa_0 > 0 \ \wedge\ (\kappa_0, \kappa_1, \mathit{from}, \mathit{to}) := (\kappa_0 \ominus 1, \kappa_1 \oplus 1, 0, 1)) \\
& \vee & E_1 : (\kappa_1 > 0 \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 = 0) \ \wedge\ (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2) \\
& \vee & E_2 : (\kappa_2 > 0) \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 3) \\
& \vee & E_2 : (\kappa_2 > 0) \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 > 0) \ \wedge\ \mathit{lmin} \neq 3 \ \wedge\ (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 > 0) \ \wedge\ \mathit{lmin} = 3 \ \wedge\ (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 4) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 > 0) \ \wedge\ \mathit{lmin} = 3 \ \wedge\ \kappa_3 > 1) \ \wedge \\
& & \qquad (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 3) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 = 0) \ \wedge\ \mathit{lmin} \neq 3 \ \wedge\ (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 = 0) \ \wedge\ \mathit{lmin} = 3 \ \wedge\ (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5, 5) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge\ \kappa_1 + \kappa_2 = 0) \ \wedge\ \mathit{lmin} = 3 \ \wedge\ \kappa_3 > 1 \ \wedge \\
& & \qquad (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5, 3) \\
& \vee & E_4 : (\kappa_4 > 0 \ \wedge\ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge\ \mathit{lmin} \neq 4 \ \wedge\ (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5) \\
& \vee & E_4 : (\kappa_4 > 0 \ \wedge\ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge\ \mathit{lmin} = 4 \ \wedge\ (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5, 5) \\
& \vee & E_4 : (\kappa_4 > 0 \ \wedge\ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge\ \mathit{lmin} = 4 \ \wedge\ \kappa_3 > 2 \ \wedge \\
& & \qquad (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5, 4) \\
& \vee & E_5 : (\kappa_5 > 0 \ \wedge\ \kappa_3 + \kappa_4 = 0) \ \wedge\ \mathit{lmin} \neq 5 \ \wedge\ (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6) \\
& \vee & E_5 : (\kappa_5 > 0 \ \wedge\ \kappa_3 + \kappa_4 = 0) \ \wedge\ \mathit{lmin} = 5 \ \wedge\ (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6, 6) \\
& \vee & E_5 : (\kappa_5 > 0 \ \wedge\ \kappa_3 + \kappa_4 = 0) \ \wedge\ \mathit{lmin} = 5 \ \wedge\ \kappa_5 > 1 \ \wedge \\
& & \qquad (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}, \mathit{lmin}) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6, 6) \\
& \vee & E_6 : (\kappa_6 > 0) \ \wedge\ \mathit{lmin} = 7 \ \wedge\ (\kappa_6, \kappa_7, \mathit{from}, \mathit{to}) := (\kappa_6 \ominus 1, \kappa_7 \oplus 1, 6, 7) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 = 1 \ \wedge\ (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 0) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge\ \kappa_3 > 0 \ \wedge \\
& & \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 3) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge\ \kappa_4 > 0 \ \wedge \\
& & \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 4) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge\ \kappa_5 > 0 \ \wedge \\
& & \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 5) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge\ \kappa_6 > 0 \ \wedge \\
& & \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 6) \\
& \vee & E_7 : (\kappa_7 > 0) \ \wedge\ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge\ \kappa_7 > 1 \ \wedge \\
& & \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, \mathit{lmin}) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 7) \\
& \vee & \kappa_0 := \kappa_0 \\
\mathcal{J} & : & \Big\{\mathit{from} = \ell \ \vee\ \neg E_\ell \ \mid\ \ell \in \{1,2,4,5,6,7\}\Big\} \ \cup\ \{\mathit{from} = 3 \ \vee\ \neg E_{34}\} \ \cup\ \{\mathit{from} = 3 \ \vee\ \neg E_{35}\} \\
\mathcal{C} & : & \displaystyle\bigcup_{\ell=0}^{7} \langle \mathit{from} = \ell,\ \mathit{to} = \ell\rangle
\end{array}
$$

Each statement of the form $(v_1', \ldots, v_k') := (E_1, \ldots, E_k)$ is an abbreviation for

$$v_1' = E_1 \ \wedge \ldots \wedge\ v_k' = E_k \ \wedge \forall v \in V - \{v_1, \ldots, v_k\} : v' = v$$

B Description of bakery†

The transition system bakery† : $\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is defined as follows:

$$
\begin{array}{lcl}
V & : & \{\kappa_0, \ldots, \kappa_4 : [0..2],\ lmin : [0, 2..4],\ \mathit{from}, \mathit{to} : [-1..4]\} \\
\Theta & : & \kappa_0 = 2 \ \wedge\ \forall \ell \in \{1..4\} : \kappa_\ell = 0 \ \wedge\ lmin = 0 \ \wedge\ \mathit{from} = -1 \ \wedge\ \mathit{to} = -1 \\
\rho & : & \qquad (\kappa_0 > 0 \ \wedge\ (\kappa_0, \kappa_1, \mathit{from}, \mathit{to}) := (\kappa_0 \ominus 1, \kappa_1 \oplus 1, 0, 1)) \\
& & \vee \quad E_1 : (\kappa_1 > 0) \ \wedge\ lmin \neq 0 \ \wedge\ (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2) \\
& & \vee \quad E_1 : (\kappa_1 > 0) \ \wedge\ lmin = 0 \ \wedge\ (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}, lmin) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2, 2) \\
& & \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin \neq 2 \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3) \\
& & \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin = 2 \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, lmin) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 3) \\
& & \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin = 2 \ \wedge\ \kappa_2 > 1 \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, lmin) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 2) \\
& & \vee \quad E_3 : (\kappa_3 > 0 \ \wedge\ lmin = 3) \ \wedge\ (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 4) \\
& & \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_2 + \kappa_3 + \kappa_4 = 1 \ \wedge\ (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 0) \\
& & \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_2 > 0 \ \wedge\ (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 2) \\
& & \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_3 > 0 \ \wedge\ (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 3) \\
& & \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_4 > 1 \ \wedge\ (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 4) \\
& & \vee \quad \kappa_0 := \kappa_0 \\
\mathcal{J} & : & \left\{ \mathit{from} = \ell \ \vee\ \neg E_\ell \ \mid\ \ell \in [1..4] \right\} \\
\mathcal{C} & : & \displaystyle\bigcup_{\ell=0}^{4} \langle \mathit{from} = \ell,\ \mathit{to} = \ell \rangle
\end{array}
$$