An automated verification approach for scheduling in complex manufacturing machines

Kuijpers, T.J.H.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

An Automated Verification Approach for Scheduling in Complex Manufacturing Machines

by
T.J.H. Kuijpers

Supervisors: Prof.dr.ir. J.F. Groote
dr. A. Serebrenik

Eindhoven, January 2006

Preface

This document is my Master's Thesis, written as a part of graduation requirements to finish my study "Technische Informatica" (Technical Computer Science) at the Eindhoven University of Technology (TU/e). I chose the master-specialisation Embedded Systems. This document, and the accompanying CD, is the result of my research for ASML, manufacturer of advanced lithographic machines which are used for the production of semiconductors. The research has been done under supervision of dr. Alexander Serebrenik at the Laboratory for Quality Software (LaQuSo), an activity of the TU/e. This laboratory focuses on finding errors in software products.

I would like to thank dr. Alexander Serebrenik for being my supervisor and his reviews of this report, prof. dr. ir. Jan Friso Groote for suggesting the assignment and his support, ir. Barend van den Nieuwelaar and ir. Martin Driessen for their willingness to explain the working of the machine and the preceding documents and ir. Arjan Mooij for his PVS related support. Furthermore I would like to thank family and friends, my parents in particular, for their support and confidence.

— Thijs Kuijpers, December 2005 —

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Problem description

This report is a sequel to [2] and [10]. Some information about complex manufacturing machines is given below, followed by the goal of the project. This section is derived from [10].

The purpose of a manufacturing machine is to make products, which require physical manufacturing processes to be carried out. In complex manufacturing machines, different types of products are concurrently being processed. Each type of product requires different manufacturing processes. A complex manufacturing system consists of multiple parallel mechatronic components/systems. Mechatronics is a combination of mechanical engineering, electronic engineering and software engineering. These mechatronic components are referred to as resources. The manufacturing steps which are performed by these resources are referred to as tasks. Resources can perform only one task at the same time. Supervisory Machine Control (SMC) is responsible for deciding when to do which tasks on which resources [11].

In a complex manufacturing machine, different choices have to be made: which tasks to do, which resources to assign to them, and which sequence of tasks to do at each resource. The freedom of choice is restricted, e.g. by material logistics. The freedom of choice that is applicable for some manufacturing request and for some machine with its logistic restrictions can be defined in a scheduling model or system definition.

A formal system definition is presented in [10] and also a state machine can be found there. Dynamic scheduling of the production in complex industrial manufacturing machines can lead to problems such as, for instance, deadlocks bringing the production to a standstill. An approach to ensure that deadlock is avoided in these highly flexible systems is to investigate all potential schedules. But verification of industrially sized systems by state-space traversal is practically impossible because the number of states is far too large. To make it possible to investigate the state space, three situations in which state-space reduction techniques can be applied have been identified, formally defined and implemented in a dedicated checker. These three cases are:

1. Interleaving of independent parallel tasks
2. Resource symmetry
3. Non-logistic tasks

Results show that the first and third reduction technique can reduce state-spaces with an exponential factor, and the second technique has a linear effect. The application on a number of wafer scanners shows that this approach makes it possible to verify industrially sized systems. Correctness of these techniques is proved by hand in [2], but incorrectness of one or more proofs is expected since mistakes are made easily in this way. Some of the given proofs might be incorrect, definitions in [2] and [10] might be wrong or incomplete, and constraints on the system might be missing.

The assignment was to use a theorem prover (PVS) to check the correctness of the proofs in [2]. If a proof turned out to be incorrect, we tried to correct the proof, change definitions, or add constraints when needed. More about theorem proving and PVS can be found in the next chapter. Only interesting parts of proofs are described in this document. If you want to study all proofs in detail, this document could be used as a guide. Redoing the proofs - stored on the CD - in PVS and using both [2] and [10] is highly encouraged.

1.2 Overview of the report

This document is divided in three parts. Part I is the introduction, chapter 1 contains the problem description and chapter 2 is an introduction to theorem proving and PVS. Part II contains the system representation. In chapter 3 you will find the mathematical definition, the three reduction techniques can be found in chapter 4, and the translation from mathematical notations to PVS is explained in chapter 5. Part III consists of only one chapter; in this chapter, proving the reduction techniques are discussed. Finally, conclusions and further research can be found in chapters 7 and 8.

Chapter 2

Theorem proving

2.1 Overview of theorem proving

For this section we used some information from [9].
Theorem proving is an area of study to prove that some statement (conjecture) is a logical consequent of a set of statements (axioms and hypotheses) by using a computer. Conjectures, axioms and hypotheses are written in a language. This might be a classical 1^{st} order logic but this might also be a higher order logic or even a non-classical logic. All information has to be specified in a precise and formal way. With this information, theorem provers use a number of inference rules to derive simpler (sub)goals which can be discharged easily. The proof output, produced by theorem provers, shows that the conjecture is a logical consequence of the given axioms and hypotheses. But it can also describe a solution to a problem by enumerating the steps which have to be taken. Theorem proving can be very hard and finding a proof is usually undecidable. Proof checking, in fact, is decidable. Theorem proving requires considerable technical expertise and understanding of the specification. This can be seen as an advantage, but also as a disadvantage. Other advantages are: theorem provers are not limited by the size of the state space and deductions are correct. But there are also some disadvantages. If you fail to complete a proof, the tool will not tell you whether the property is indeed unprovable under the given circumstances.

Theorem provers are computer programs capable of solving problems ranging from trivial to immensely difficult. Because of this capability, theorem provers are often used in an interactive way. According to [12], three different interaction styles are distinguished. First, we have proof checkers. These are systems which check the correctness of the proof text written by the user. Second, there are interactive theorem provers or proof assistants. These systems maintain the proof state for the user. The proof state is modified by the user by providing rules to the system. Finally, automated theorem provers automatically prove conjectures, without being guided by the user. Both interactive and automated theorem provers contain a proof checker.

Interactive theorem provers can have powerful automation too. PVS is such an interactive theorem prover. Two reasons to prefer PVS over any other interactive theorem prover is the specification langauge, which is easy to understand, and the powerful rules. An overview of PVS is given in the next section. We used PVS 3.2 (November 2004) for the assignment, which is the latest version so far.

2.2 PVS in a nutshell

2.2.1 Introduction

PVS[1], short for Prototype Verification System, is an interactive theorem prover. This means that a user has to give hints to the system to prove a certain theorem or conjecture. PVS provides an integrated environment

[1] PVS is developed and maintained by SRI International. See *http://pvs.csl.sri.com/* for detailed information and downloads.

for the development and analysis of formal specifications. A PVS specification looks like math and consists of any number of ASCII files, each containing definitions and/or axioms. Before a (part of a) theory can be proven, the theory has to be parsed and typechecked. The parser checks the syntax of a theory and builds an internal representation. The typechecker analyses theories for semantic consistency and adds semantic information to the previously built internal representation by the parser. Theorem proving may be required to establish type-consistency of a PVS specification. The theorems that need to be proven in such a case are called type-correctness conditions (TCCs). We discuss the specification language in short in section 2.2.2.

PVS provides a powerful interactive proof checker with the ability to store and replay proofs. The prover maintains a proof tree. Each node of the tree is a proof goal that follows from its offspring nodes by means of a proof step. Each proof goal is a sequent consisting of a two sequences of formulas: antecedents (above the horizontal line; this is a conjunction of formulas) and consequents (below the horizontal line; this is a disjunction of formulas). It is the goal of the user to construct a tree, by submitting proof commands to PVS, where all the leaves are true. The most frequently used commands to achieve this are: grind, skosimp*, inst, flatten and split. The prover and some of its commands are discussed in section 2.2.3.

PVS supports also a wide range of activities involved in creating, analysing, modifying, managing and documenting theories and proofs. Some of these features are the possibility to generate documents, import/export theories, reuse files and theories, support parallel activity (e.g. editing and typechecking) and take care of lay-out issues by using the prettyprinter. More information about these additional features can be found in [8].

2.2.2 PVS specification language

The information in this section is derived from [6]. More about the PVS language can be found there.

The specification language of PVS is built on higher-order logic i.e. logic of quantification over propositional functions, whose range consists of properties. Functions are first class objects. Furthermore, recursive definitions and subtypes are supported. A rich set of built-in types and type-constructors and lots of useful definitions and lemmas are included in the prelude.

Theory. A PVS specification consists of a number of theories, which behave like modules in programming languages. Each theory contains mathematical and logical objects: types, axioms, functions, theorems, etcetera. Theories can be parametric and it is possible to place constraints, called assumptions, on these parameters. Within a theory, types can be defined starting from base types (booleans, naturals, reals) using the function, record and tuple type constructors. New interpreted base types may be introduced. The terms of the language can be constructed by function application, lambda abstraction, and record and tuple construction. Theories support modularity and reusability.

Declarations. Declarations are used to introduce, among others, types, variables, constants and functions. Each declaration has an identifier, an optional list of bindings and a body. The body determines the kind of the declaration. A declaration belongs to a unique theory. Overloading of declaration identifiers is allowed.

We will only give an overview of the various declarations we use in the rest of this document:

- Type declarations are used to introduce new type names to the context, which may be used in type expressions. Type names are introduced using the keyword `TYPE`, or for nonempty types `TYPE+`. We distinguish uninterpreted and interpreted type declarations, enumeration type declarations and subtype declarations. We will discuss the first three here; subtypes are discussed in the next paragraph.

- uninterpreted types are types with the minimum of assumptions on the type. The only assumptions is that the type is disjoint from all other types, e.g. `T1` and `T2` in `T1, T2 : TYPE` are disjoint. Further constraints may be put on these types by means of axioms (see 'Formula declarations' further on in this paragraph).
- interpreted types provide names for type expressions, e.g. if we want to have some function with integer domain and boolean range, we define `someFunction : TYPE = [int -> bool]`.
- enumeration types are types where all possible values are enumerated, e.g. for the days of the week we could define `dow : TYPE = sun, mon, tue, wed, thu, fri, sat.`

As mentioned before, types may be empty or nonempty. If a type is declared as nonempty and if the system cannot verify that the type is indeed nonempty, then an *existence TCC* is generated. Discharging this TCC ensures that an element of the type exists. Note that uninterpreted types introduced with the keyword `TYPE` may be empty and declaring a constant of that type will therefore lead to a TCC which is unprovable without further axioms.

- Variable declarations introduce new variables and associate a type with them. They provide a name and associated type so that binding expressions can be succinct. Variable declarations can also appear in binding expressions, such as `FORALL` and `LAMBDA`. Such local declarations overrule any earlier declarations. Predeclared variables are introduced using the keyword `VAR`, e.g. `n : VAR nat.` Local variables are not introduced by a keyword; variable `b` is a local variable in `... FORALL (b:bool) ...`. Predeclared and local variables can be mixed.

- Constant declarations introduce new constants, specifying their type and optionally providing a value. As with types, there are both uninterpreted and interpreted constants. Uninterpreted types make no assumptions but, as mentioned above, their types may not be empty. Examples of constants are:

```
c1 : int
c2 : int = 1
c3 : f(nat: n) = n + 1
```

Only the `c1` is an uninterpreted constant, `c2` and `c3` are interpreted.

- Recursive definitions are treated as constant declarations, except that the defining expression is required, a *measure* must be provided, and optionally, a well-founded order relation could be given. A *measure* is a function used to prove well-foundedness of the recursion and thus ensures termination. Recursive definitions are restricted in PVS. First, mutual recursion is not supported and the function must be total, meaning that the function should be defined for every value of its domain. Mutual recursion is a form of recursion where two functions are defined in terms of each other. An example of a recursive definition is given below.

```
factorial (x: nat): RECURSIVE nat =
  IF x = 0 THEN 1 ELSE x * factorial(x - 1) ENDIF
  MEASURE x
```

- Inductive definitions give some rules for generating elements of a set. An object is in the set only if it has been generated according to the rules; hence the generated set is the smallest set closed under the rules. Inductive definitions are predicates and therefore they have range type `boolean`. For example, the set of even numbers is generated by:

```
even(n: nat): INDUCTIVE boolean =
  n = 0 OR (n > 1 AND even(n - 2))
```

Some conjectures can be proven just by expanding the definition enough times, e.g. `even(1000)`. For the more general cases, say `even(n) IMPLIES NOT even(n+1)`. PVS creates two induction schemas, namely *weak_induction* and *induction*. For induction one needs to know that all previous steps hold, for weak induction this is just the previous one. The schemas can be viewed using the command "M-x prettyprint-expanded".

- Formula declarations introduce axioms, assumptions, theorems and obligations. The expression in the body of the function is a boolean expression. Keywords used to introduce axioms, assumptions and obligations are respectively `AXIOM` or `POSTULATE`, `ASSUMPTION` and `OBLIGATION`. Theorems can be introduced by various keywords. This makes specifications more readable for the user, but there is no difference between the various keywords. In this document we only use `CONJECTURE` and `LEMMA`. Assumptions are only allowed in assuming clauses and obligations are generated by the system for TCCs and cannot be specified by the user. Axioms need not be proved; they are assumed to hold at any time. Too many axioms are not desirable, since they can lead to inconsistencies.

Types and expressions. PVS specifications are strongly typed, meaning that every expression has an associated type. Type names, introduced in the previous paragraph, are the simplest type expressions. More complex type expressions are built from these, using type constructors for subtypes, function types, tuple types and record types. These are discussed below. Abstract datatypes (ADTs) are also supported, but we won't discuss them here. Furthermore, the PVS language offers expression constructs, including logical and arithmetic operators, quantifiers, lambda abstractions, function application, tuples, a polymorphic `IF-THEN-ELSE`, and function and record overrides. We will only discuss tuples (tuple and projection expressions), records (record expressions and record accessors) and `COND`-expressions (extension to the `IF-THEN-ELSE` construct. For more details about types and expressions we refer to [6].

- If any collection of a given type forms a new type, it is called a subtype. The type from which the elements are taken is called the supertype. The selection of elements is done by a subtype predicate on the supertype. A subtype looks like:

  ```
  t : TYPE = {x:s | p(x)}
  ```

  where `x` represents a set and `p` is a predicate over `x`.

- Function types can be represented in three equivalent ways. In this document we only use the form $[\mathtt{t}_1, ..., \mathtt{t}_n \rightarrow \mathtt{t}]$ where each $\mathtt{t}_i$ is a type expression. An element of this type is a function whose domain is the sequence of types $\mathtt{t}_1, ..., \mathtt{t}_n$ and whose range is `t`. In the prelude, `pred[t]` and `setof[t]` are provided as shorthand for $[\mathtt{t} \rightarrow \mathtt{bool}]$. Both forms are equivalent; the different names are introduced to distinguish different intentions.

- Tuple types have the form $[\mathtt{t}_1, ..., \mathtt{t}_n]$ where each $\mathtt{t}_i$ is a type expression. A tuple expression of the type $[\mathtt{t}_1, ..., \mathtt{t}_n]$ has the form $(\mathtt{e}_1, ..., \mathtt{e}_n)$ where each $\mathtt{e}_i$ is of type $\mathtt{t}_i$. For example, `(1, TRUE, (LAMBDA (x:int): x + 1))` is an expression of type $[\mathtt{int}, \mathtt{bool}, [\mathtt{int} \rightarrow \mathtt{int}]]$; the order is important. The 0-ary tuple type is not allowed. Accessing tuple components is done by the built-in *proj* function; *proj_1* represents the first component, *proj_2* the second and so on. Equivalent to the *proj* function is *'1*, *'2* and so on. Like reserved words, projection expressions are case insensitive and may not be redeclared.

- Record types are of the form $[\#\mathtt{a}_1 : \mathtt{t}_1, ..., \mathtt{a}_n : \mathtt{t}_n\#]$. The $\mathtt{a}_i$ are called *accessors* or fields and the $\mathtt{t}_i$ are types. Record types are similar to tuple types, except that the order is unimportant and accessors are used instead of projections. Record expressions of the type $[\#\mathtt{a}_1 : \mathtt{t}_1, ..., \mathtt{a}_n : \mathtt{t}_n\#]$ have the form $(\#\mathtt{a}_1 : \mathtt{e}_1, ..., \mathtt{a}_n : \mathtt{e}_n\#)$ where each $\mathtt{e}_i$ is of type $\mathtt{t}_i$. Partial record expressions are not allowed; all fields must be given. Use override expressions if partial records are desired. The components of an expression of a record are accessed using the corresponding field name. There are two forms of

access. For example, if we have a record `r` of type [#`x, y : real`#], we can access the `x`-component either by `r'x` or `x(r)`. We will use the first option in this document.

- The `COND` expression is an `IF-THEN-ELSE` statement with multiple conditions. Its form is shown below at the left-hand side. The right-hand side shows the equivalent `IF-THEN-ELSE` notation:

```
COND              |
  be_1 -> e_1,    | IF be_1 THEN e_1
  be_2 -> e_2,    | ELSEIF be_2 THEN e_2
    ...           |    ...
  be_n -> e_n,    | ELSEIF be_n THEN e_n
  ELSE -> e_z     | ELSE e_z
ENDCOND           |
```

  The `be_i` are boolean expressions and the `e_i` are expressions. The `ELSE` clause is optional. The boolean expressions must be pairwise disjoint and their disjunction, including the optional `ELSE` clause, must be a tautology.

2.2.3 PVS Prover

The information in this section is derived from [7]. More about the PVS prover can be found there.

The PVS proof checker provides a collection of powerful proof commands (*rules*) to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. These proof commands can be combined to form *proof strategies*. The proof checker is interactive, but supports a batch mode as well. Furthermore, the proof checker permits proof steps to be undone and PVS allows (partial) proofs to be edited and rerun. This section gives an overview of the presentation of proofs in PVS and describes the rules and strategies used in this document.

Sequents. The prover maintains a *proof tree* for the goal that has to be proved. The root of the proof tree represents the original goal. Each node of the tree is a *proof goal* that follows from its offspring nodes by means of a *proof step*. Proof steps can be used to introduce lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on. They affect the proof tree and are saved when the proof is saved. Any number of subgoals may arise from a goal by these proof steps. Each proof goal is a *sequent* consisting of a sequence of formulas, *antecedents* and *consequents*. All leaves of the tree have to be proved to complete the original goal.

In PVS, a sequent is displayed as:

```
{-1}   A1
       .
[-n]   An
  |-------
{1}    B1
       .
{m}    Bm
```

and must be read as $\mathtt{A}_1 \wedge \ldots \wedge \mathtt{A}_n \Rightarrow \mathtt{B}_1 \vee \ldots \vee \mathtt{B}_m$. Formulas are named by numbers; negative numbers denote antecedents, positive numbers belong to consequents. Names of functions which are unchanged in a subgoal from the parent goal are denoted in square brackets `[]`, new functions and changed ones are numbered in braces {}. At any time in a PVS proof, attention is focused on some sequent that is a leaf node in the current proof tree. This sequent is displayed by PVS while awaiting the user's command. Once a sequent is recognised as *true*, that branch of the proof tree is terminated. A sequent is *true* if any antecedent is the same as any consequent, if any antecedent is false, or if any consequent is true. Other sequents can also be recognised as *true*, but more powerful inferences are needed.

Besides proof steps there are *interactive commands* and *proof commands*. Interactive commands leave sequents unchanged or steps backwards in the partial proof of the current subgoal. They are not saved when the proof is saved. Examples are shifting the focus `(postpone)` to a sibling of the current sequent (if any) or returning to some ancestor node representing an earlier point in the proof `(undo)`.

Rules and strategies. A PVS proof command can either invoke a rule or a strategy. A proof is an atomic operation that generates zero or more subgoals from the given goal. A strategy need not be atomic; an application of a strategy expands into a number of atomic steps. Rules are either primitive or defined and the defined ones are defined as strategies but applied as atomic proof steps. For example, `prop` is the atomic propositional simplification rule and `prop$` is the corresponding strategy, which shows the atomic steps of `prop`. Strategies should be used when the expanded proof is of interest and otherwise, rules should be used. In this document we only use rules; we will give an overview of the used ones below. For full details about all rules and strategies we refer to [7].
The following rules are used in this document or in any of the proofs on the CD. A detailed description of each of these rules (and many more) can be found in [7]:

- Structural rules
  - hide
    Hide the selected formulas.
  - hide-all-but
    Hide all but the selected formulas.
  - reveal
    Reveal the selected formulas.
- Propositional rules
  - case
    Case analysis on functions; generates two or more subgoals.
  - flatten
    Transform each indicated formula into a list of formulas that contains no disjuncts. A sequent formula is disjunct if it is either an antecedent of the form $\neg$`A` or `A` $\wedge$ `B`, or a consequent formula of the form $\neg$`A`, `A` $\supset$ `B` or `A` $\vee$ `B`.
  - split
    Transform each indicated formula into a list of formulas that contains no conjuncts. A sequent formula is conjunct if it is either an antecedent of the form `A` $\vee$ `B`, `A` $\supset$ `B` or `IF`$(\mathtt{A},\mathtt{B},\mathtt{C})$, or a consequent formula of the form `A` $\wedge$ `B`, `A` $\Longleftrightarrow$ `B` or `IF`$(\mathtt{A},\mathtt{B},\mathtt{C})$.
  - propax
    Propositional simplification, using `flatten` and `split`.
- Quantifier rules
  - inst
    Instantiate universal quantified variables in the antecedent or existential quantified variables in the consequent.
  - skolem!
    Replace existential quantification in the antecedent or universal quantification in the consequent with automatically chosen constants. These constants are of the form $<$ `var` $>!<$ `num` $>$, for example `x!1`.
  - skosimp*
    Repeatedly using `skolem!` and `flatten`
- Equality rules

  - name
    Introducing names for terms.
  - replace*
    Iteratively rewriting using equalities in the antecedent.

- Using definitions and lemmas
  - expand
    Expanding and simplifying selected definitions.
  - lemma
    Introduce an instance of the selected lemma as a new formula in the antecedent.

- Using extensionality
  - apply-extensionality
    Use extensionality to prove equality. The extensionality axiom says that two sets are equal iff they have precisely the same members.
  - decompose-equality
    Decompose an antecedent or consequent equality of the form `t1` $=$ `t2` to component equalities.

- Simplification with decision procedures and rewriting
  - ground
    Propositional simplification followed by applying the following decision procedures: `record`, `simplify`, `beta` and `do-rewrite`. Since these commands use other rules too, we refer to [7] for more details.
  - smash
    Extension of `ground` with IF-lifting; a special rewrite rule lifting embedded IF connectives.
  - grind
    Most powerful command in PVS; install given theories and rewrite rules, followed by repeated simplification.

- Making type constraints explicit
  - typepred
    Making implicit type constraints explicit.

Part II

System representation

Chapter 3

System definition

In this chapter we give definitions which represent the mathematical model of the system. Section 3.1 gives the definitions representing the system and system invariants are enumerated. In section 3.2 the state space is defined. This chapter is taken from [10].

In the definitions the following symbols are used:

$\mathbb{N}$ is used to denote the natural numbers;
$\mathcal{P}(s)$ denotes the powerset of a given set s.
$\mathbb{B}$ is used to denote the boolean values.

3.1 Static and dynamic system

3.1.1 Main definitions

We use the manufacturing model from appendix B. This model consists of a static part and a dynamic part. The static part defines the machine-specific restrictions imposed by the hardware of the machine, which is resource related. It describes which capabilities the resources can provide, how many material instances can reside on them, and which logistic transports are possible.

Definition 3.1.1 (Static system definition) *A static system definition is a 5-tuple* $\Sigma = (R, C, A, R_m, M_f)$*, where*

- R *is a given set of available resources;*
- C *is a given set of capabilities;*
- $A : R \to C$ *is a function that gives the capability which a resource can provide;*
- $R_m : R \to \mathbb{N}$ *is a function that gives the maximum number of material instances that can reside on a resource, called the material capacity;*
- $M_f : R \to \mathcal{P}(R)$ *is a function that gives the resources to which material can be transported from a certain resource.*

A simple example can be found in appendix B. The dynamic system definition describes the tasks to do for a certain manufacturing request. They can recursively be structured in clusters and groups. The set of all tasks, clusters and groups are the nodes. A cluster indicates that if the cluster node is chosen all children of the cluster must be chosen, whereas a group has an attribute defining the allowed numbers of children to be chosen. Such an allowed number can be less than the number of children, which makes it possible to bypass tasks. Furthermore, as defined later, precedences of nodes cannot cross group boundaries, but they can pass over cluster boundaries.

For each task the set of involved capabilities is defined, and at which capability which material instances reside at the beginning and at the end of the task. This, together with the precedence relation between nodes outlines the room for choices with respect to task order.

Definition 3.1.2 (Dynamic system definition) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition. Then a dynamic system definition is a 12-tuple*
$\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$, where

- *T is a given set of tasks;*
- *G is a given set of groups;*
- *L is a given set of clusters;*

  Now we define N, the set of nodes, as: $N = T \cup G \cup L$

- *$L_n : L \to \mathcal{P}(N)$ gives the nodes contained in a certain cluster;*
- *$G_n : G \to \mathcal{P}(N)$ gives the nodes contained in a certain group;*
- *$G_a : G \to \mathcal{P}(\mathbb{N})$ gives the allowed numbers of node alternatives to be selected from a group;*
- *$I : T \to \mathcal{P}(C)$ is a function that gives the capabilities required to perform a certain task;*
- *$P : N \to \mathcal{P}(N)$ is a function that gives the predecessors of a certain node;*
- *M is the set of material instances;*
- *$C_b, C_e : T \times C \to \mathcal{P}(M)$ are functions that give the material instances residing on a resource with a certain capability that is involved in a certain task at the beginning and end of that task, respectively;*
- *$\hat{m}_s : R \to \mathcal{P}(M)$ is a function that gives the material instances initially residing on a certain resource.*

Again, a simple example can be found in appendix B.

3.1.2 Additional definitions

To be able to specify the system invariants in the next subsection, we need some additional definitions:

- The ancestor function $anc : N \to \mathcal{P}(N)$ gives the nodes in which a certain node is contained. The set $anc(n)$ is the smallest set satisfying the following conditions:
  - if $n \in G_n(n')$ or $n \in L_n(n')$, then $n' \in anc(n)$;
  - if $n'' \in anc(n')$ and $n' \in anc(n)$ then $n'' \in anc(n)$;
- The function $allsucc : N \to \mathcal{P}(N)$ gives *all* successors of a node. It is the smallest set satisfying the following condition:
  - for all nodes $n' \in N$, $n'' \in \{n\} \cup anc(n)$, if $n'' \in P(n')$ then $n' \in allsucc(n)$, and $allsucc(n') \subseteq allsucc(n)$;
- Function mat gives the materials involved with a node:

$$mat(n) = \{m{\in}M \mid \exists t{\in}T, c{\in}C.\ n{\in}\{t\} \cup anc(t)\ \wedge\ m{\in}C_b(t,c) \cup C_e(t,c)\};$$

3.1.3 System invariants

We only consider static and dynamic system definitions that satisfy the following properties:

1. The nodes in the system have a hierarchical structure. For all nodes $n \in N$ it must hold that $n \notin anc(n)$.
2. No group has only 0 as allowed number. I.e. $G_a(g) \neq \{0\}$ for all groups $g \in G$;
3. The capabilities in $C_b(t)$ and $C_e(t)$ exist also in $I(t)$. I.e. for all tasks $t \in T$ and capabilities $c \in C$ if $C_b(t,c) \cup C_e(t,c) \neq \emptyset$ then $c \in I(t)$;
4. P contains no cycles. For each node $n \in N$ it must hold that $n \notin allsucc(n)$.
5. There is no precedence relation between node alternatives in a group. For all groups $g \in G$ and nodes $n \in N$ it holds that if $n \in G_n(g)$ then $P(n) = \emptyset$;
6. Precedence relations do not cross group boundaries. For all groups $g \in G$ and nodes $n, n' \in N$ it is the case that if $g \in anc(n)$ then $n' \in P(n) \Rightarrow g \in anc(n')$ and $n \in P(n') \Rightarrow g \in anc(n')$;
7. All tasks in a group concern the same material:

$$\forall t, t' \in T, g \in G \cap anc(t) \cap anc(t').\ mat(t) = mat(t').$$

8. The subsets of material instances involved in a task remain the same from the beginning to the end of a task. I.e. for all tasks $t \in T$:

$$\bigcup_{c \in I(t)} C_b(t,c) = \bigcup_{c \in I(t)} C_e(t,c).$$

   This constraint implies that only closed systems are considered where no material enters or leaves the system;
9. Initially, the material capacity of all resources is not exceeded. For any resource $r \in R$ it holds that $\#\hat{m}_s(r) \leq R_m(r)$. Here $\#S$ gives the number of elements in the set S.

3.2 State space

3.2.1 Main definitions

Given the two system definition parts, the physically feasible behaviour of the machine carrying out the schedule is defined in the form of a transition system, which is often also called a state-space or (behavioural) automaton, in Def. 3.2.5. The states represent which tasks have been executed and which material instances reside at which resources. The transitions indicate all allowed possibilities to go from one such state to another.
Before giving the main definitions two auxiliary concepts must be characterised. The predicate $successful(n, tp)$ expresses whether node n is successfully executed given the successful tasks in tp. A task is successful if it occurs in tp. A cluster is successful if all its subnodes are successful. A group is successful iff an allowed number of nodes in the group are successful. If this number is zero, all predecessors of the group need to be successful.

Definition 3.2.1 *Let $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Let $n \in N$ be a node and $tp \subseteq T$ a set of executed tasks. We inductively define that n is successful in tp, notation $successful(n, tp)$, iff*

- *if $n \in T$ is a task, then $n \in tp$;*
- *if $n \in L$ is a cluster, then for all $n' \in L_n(n)$ it must hold that $successful(n', tp)$;*
- *if $n \in G$ is a group, then*

$$\#\{n' \in G_n(n) \mid successful(n', tp)\} \in G_a(n) \wedge$$
$$(\#\{n' \in G_n(n) \mid successful(n', tp)\}) = 0 \Rightarrow \forall n'' \in P(n).\ successful(n'', tp).$$

For a set of nodes np and a set of executed tasks tp the predicate $successful(np, tp)$ holds if for all nodes $n{\in}np$ $successful(n, tp)$ is valid.

The second auxiliary predicate that we define is $bypassed$. In order to define it, we must introduce two additional predicates. For a node $n \in N$ the expression $successor(n)$ gives the successor nodes of n that can immediately be executed after n. The definition is somewhat involved, because it can be that a direct successor of n is a group in which 0 tasks can be executed. Then the successor of such a group is also a successor of n.

Definition 3.2.2 *Let $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. The predicate $successor(n)$ for a node $n \in N$ is defined as the smallest predicate satisfying:*

$$\begin{array}{rcl} successor(n) & = & \{n'{\in}N \mid (n \cup anc(n)) \cap P(n') \neq \emptyset\} \cup \\ & & \{successor(n') \mid n'{\in}successor(n) \cap G \ \wedge \ 0{\in}G_a(n')\}. \end{array}$$

Another additional definition is the notion of *initiated* nodes. These are nodes that have already been started.

Definition 3.2.3 *Let $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. For a set of executed tasks tp the set of nodes $initiated(tp)$ is defined by*

$$initiated(tp) = \{n{\in}N \mid \exists t{\in}T.\ t{\in}tp \ \wedge \ n{\in}anc(t)\}.$$

The second auxiliary predicate that we define is $bypassed(t, tp)$ for a task $t \in T$ and a set of tasks $tp \subseteq T$. Its definition is quite involved. The predicate $bypassed(t, tp)$ holds for task t and set of tasks tp, if t is not successful (i.e. $t \notin tp$) and

- either a subsequent task is already successful,
- or the number of initiated nodes in a group containing that task is equal to the maximum number of nodes that can be successful in that group.

Definition 3.2.4 *Let $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Let $t \in T$ be a task and $tp \subseteq T$ a set of executed tasks. We define that t is bypassed, notation $bypassed(t, tp)$ iff*

$$\begin{array}{l} t \notin tp \wedge \exists g{\in}G \cap anc(t). \\ \qquad successor(g) \cap tp \neq \emptyset \ \vee \\ \qquad ((anc(t) \cup \{t\}) \cap G_n(g)) \setminus initiated(tp) \neq \emptyset \wedge \\ \qquad\qquad \#(G_n(g) \cap initiated(tp)) = \max(G_a(g)). \end{array}$$

Here $\max(S)$ for a finite set of natural numbers S is the largest number in S. The set $bypassed(tp)$ is defined as $\{t{\in}T \mid bypassed(t, tp)\}$.

This provides sufficient basic material to define the state-space of a system.

Definition 3.2.5 (The state-space of the system) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition and $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Then the state-space is defined as the transition system $\Omega = (St, \hat{s}, Ac, \tau)$, where*

- *$St = (\mathcal{P}(T) \times (R \to \mathcal{P}(M)))$ are the states. The first component indicates the tasks that have been executed and the second component indicates which material instances can be found at which resource.*
- *$\hat{s} = (\emptyset, \hat{m}_s)$ is the initial state;*
- *$Ac = (T \times \mathcal{P}(R))$ are the labels of states indicating which task is executed at which resources;*

- $\tau = \{((tp, ms), (t, r), (tp', ms')) \in St \times Ac \times St$

$$
\begin{aligned}
&| \quad successful(P(t), tp) && (1)\\
&\wedge \quad t \in T \backslash (tp \cup bypassed(tp)) && (2)\\
&\wedge \quad \forall cap \in I(t).\ \exists res \in r.\ A(res) = cap && (3)\\
&\wedge \quad \forall res \in r.\ C_b(t, A(res)) \subseteq ms(res) && (4)\\
&\wedge \quad \forall res \in r, m \in M. && (5)\\
&\qquad m \in ms(res) \Rightarrow\ m \in ms'(res) \vee (\exists res' \in M_f(res).\ m \in ms'(res'))\\
&\wedge \quad \forall res \in R.\ \#ms'(res) \leq R_m(res) && (6)\\
&\wedge \quad tp' = tp \cup \{t\} && (7)\\
&\wedge \quad \forall res \in r.\ ms'(res) = (ms(res) \backslash C_b(t, A(res))) \cup C_e(t, A(res)) && (8)\\
&\wedge \quad \forall res \notin r.\ ms'(res) = ms(res)\} && (9)
\end{aligned}
$$

contains the transitions from state to state.

To make sure that the transitions conform the intuition of the dynamic system definition, several cases must be distinguished. Cases one through three are properties that must hold for the task that is performed in the transition and cases four through seven are properties that must hold for the resources assigned to the task. Seven through nine express the update of the system state. Below each line of the definition of a state-space is explained separately.

1. All predecessors of the task that is performed in the transition were successful before the transition.
2. The task to be performed did not happen and has not been bypassed.
3. For every capability involved in the task, there is a resource assigned to the task that can provide that capability.
4. The material to be used in the task is present at the resources that are assigned to the task.
5. Material that is at an involved resource before a transition is at that same resource after the transition, or it has been transported to a resource to which it could be transported.
6. After a transition, the material capacity of all resources is not exceeded.
7. The set tp' contains all tasks that have been performed before the transition and the task that has been performed during the transition.
8. After the transition, the material configuration of the resources involved in the task is the same as the material configuration at the beginning of the transition except for the changes made by the task.
9. After the transition, the material configuration of the resources not involved in the task is the same as the material configuration at the beginning of the transition.

3.2.2 Additional definitions

Finally, we need one more definition before we can define a set of τ-confluent transitions. Confluence and the set of τ-confluent transitions are defined in section 4.1.

- Function $succ(n,\ V)$ gives the successors of a node n, given a precedence relation V:

$$
\begin{aligned}
succ(n, V) = &\{n' \in N \mid n \in V(n')\} \cup\\
&\{n'' \in N \mid \neg \exists n' \in N.\ n \in V(n') \wedge\\
&\qquad \exists n''' \in N.\ n \in G_n(n''') \cup L_n(n''') \wedge n'' \in succ(n''', V)\}.
\end{aligned}
$$

If no nodes succeed n directly, a node one level higher in the node hierarchy is looked at. Note that due to the hierarchical node structure, $succ$ is well defined.

Chapter 4

Reducing the state space

In this chapter the three reduction techniques are described. This chapter is taken from [10].

4.1 Interleaving of independent parallel tasks

The first reduction technique has to do with interleaving of independent parallel tasks. In some situations, no matter what interleaving is chosen during traversal, the same state will be encountered in the end. A general state-space reduction technique that can be applied in this case is the prioritisation of actions [3], also called τ-prioritisation. In literature, τ is used to denote internal transitions, which applies to all our transitions, because we are interested in deadlocks, and not in the particular nature of individual transitions. It is allowed to apply prioritisation of actions, if the system has no infinite behaviour and is confluent. Our transition system has no infinite behaviour, as with each step, an additional task is executed. Confluence is defined below. It says that in every state, where a transition from a confluent set of transitions can be chosen, and another transition is possible, a common state can be reached. The definition of confluence is given in the form of a set of conditions on a confluent set of transitions (cf. [4]).

Definition 4.1.1 (Confluence) *Let $(St, \hat{s}, Ac, \tau)$ be a state-space. Let $\mathbf{C} \subseteq \tau$ be a set of transitions. The set $\mathbf{C}$ is called τ-confluent if for all $(s, a, s') \in \mathbf{C}$ and $(s, a', s'') \in \tau$ it holds that:*

$$(\exists s''' \in St.\ (s', a', s''') \in \tau \wedge (s'', a, s''') \in \mathbf{C}) \vee \quad \text{(I)}$$

$$\exists a'' \in Ac.\ (s'', a'', s') \in \mathbf{C} \vee \quad \text{(II)}$$

$$\exists a''' \in Ac.\ (s', a''', s'') \in \tau \vee \quad \text{(III)}$$

$$s' = s'' \quad \text{(IV)}$$

A graphical representation of these four cases is given in Fig. 4.1.

The prioritised state-space is defined as follows. It says that Ω' can be constructed out of Ω by removing outgoing transitions from a state, as long as at least one transition in $\mathbf{C}$ remains. This generally reduces the size of a transition system substantially, especially, because many states become unreachable. We have the important theorem that says that if we apply τ-prioritisation with a τ-confluent set, then the state-space of the system has *exactly* the same deadlocks.

Definition 4.1.2 (τ-prioritisation) *Let $\Omega = (St, \hat{s}, Ac, \tau)$ and $\Omega' = (St, \hat{s}, Ac, \tau')$ be state-spaces. Let $\mathbf{C} \subseteq \tau$ be a set of transitions. We say that Ω' is a τ-prioritised reduction of Ω iff*

- $\tau' \subseteq \tau$;
- $\forall s, s' \in St, a \in Ac.\ (s, a, s') \in \tau \ \Rightarrow\ (s, a, s') \in \tau' \vee \exists s'' \in St, a' \in Ac.\ (s, a', s'') \in \tau \cap \mathbf{C}$.

A set of τ-confluent transitions $\mathbf{C}$ in the state-space of a system is defined as follows. A more detailed explanation is given after the definition.

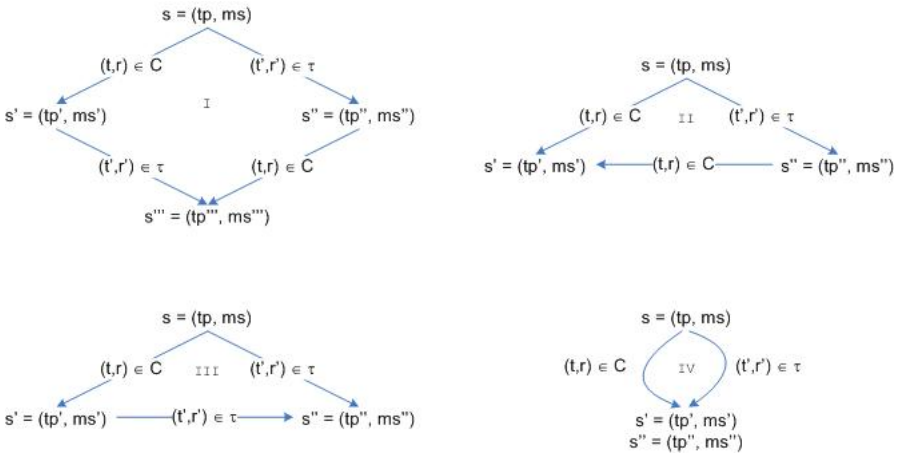


Figure 4.1: Confluence

Definition 4.1.3 (Confluent transition set C) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition and $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Let $(St, \hat{s}, Ac, \tau)$ be the state-space of the system. We define $\mathbf{C} \subseteq \tau$ to be the set consisting of the transitions $((tp, ms), (t, r), (tp', ms'))$ for which for all transitions $((tp, ms), (t', r'), (tp'', ms''))$ with $t \neq t'$ the following conditions hold.*

1. $\forall t'' \in T.\ mat(t'') \cap mat(t) \neq \emptyset \Rightarrow (t'' \cup anc(t'')) \cap allsucc(t) \neq \emptyset$;
2. $\forall g \in G, n \in (g \cup anc(g)).\ (anc(t) \cup \{t\}) \cap succ(n, P) = \emptyset$;
3. $G \cap anc(t) = \emptyset$;
4. $\forall res \in r.\ res \in unsafe \Rightarrow C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset$.

The auxiliary functions used in this definition are defined in sections 3.1.2 and 3.2.2, except for $unsafe$, which is given below. The set of unsafe resources contains those resources for which it is possible to put material onto it from more than one other resource.

$$unsafe = \{res \in R \mid \sum_{res' \in R, res \in M_f(res')} R_m(res') > 1\}.$$

Condition one says that parallel tasks involving the same material are not necessarily confluent. Furthermore, the fact that in the end the same state will be encountered implies that no deadlock may be involved in the confluent set. If none of the transitions in the confluent set loads to an unsafe resource, i.e. a resource that can receive material from multiple other locations, confluence is not harmed. This is condition 4. Finally, the logistic effect of a group depends on which tasks are chosen in that group. Therefore, groups form an unreliable basis for confluence and are excluded as predecessor or relative of tasks in the confluent set. This yields conditions 2 and 3, respectively. Note that these conditions on set $\mathbf{C}$ are at the safe side. The set can be extended, but τ-confluence of an extended set has not been proved in [2].

4.2 Resource symmetry

The second reduction technique concerns equivalent resources in the system. In such a case multiple equivalent schedules can be generated, in which the only difference is that equivalent resources are swapped. The state-space can be reduced in such cases using strong bisimulation [5], provided that it does not relate finished and deadlocked states. In such a case strong bisimulation reduction maintains deadlocks in our production machines, and so, only the reduced system needs to be investigated.

Definition 4.2.1 (Strong bisimilarity) *Let $(St, \hat{s}, Ac, \tau)$ be a state-space. For two states $s, t \in St$ to be strongly bisimilar, there must be a strong bisimulation relation R relating s and t, i.e. sRt. A relation R is a strong bisimulation relation iff it is symmetric, and for all states s, t such that sRt it holds that:*

$$(s, a, s') \in \tau \Rightarrow \exists t' \in St.\ (t, a, t') \in \tau \wedge s' R t'.$$

Two resources are symmetric in our transition system if they are of the same capability, have the same material capacity, and material can flow from and to the same resources. Note again that this definition can be extended, as e.g. also groups of resources can be symmetric, but this is not taken into account here.

Definition 4.2.2 (Symmetric resources) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition. We say that two resources $r \in R$ and $r' \in R$ are symmetric, notation $symm(r, r')$, iff*

$$\begin{gathered} A(r) = A(r') \wedge \\ R_m(r) = R_m(r') \wedge \\ M_f(r) = M_f(r') \wedge \\ \forall res \in R.\ r \in M_f(res) \Rightarrow r' \in M_f(res) \end{gathered}$$

Consider the state-space $(St, \hat{s}, Ac, \tau)$ of the system. If there are two states $(tp, ms) \in St$ and $(tp, ms') \in St$ such that $\forall res \in R.\ ms(res) = ms'(res) \vee \exists res' \in R.\ symm(res, res') \wedge ms(res) = ms'(res')$ then it holds that (tp, ms) and (tp, ms') are strongly bisimilar. Furthermore, it is easy to see that with this definition no deadlocked and finished states are related. So, this definition can then be employed in the following way. It is only necessary to investigate one state from the whole set of bisimilar states. So, in order to explore the full state-space, it is only necessary to investigate only the first state encountered from the whole bisimilar set of states.

4.3 Non-logistic tasks

The third state-space reduction deals with mutually exclusive non-logistic tasks. A non-logistic task has no logistic effect as the materials stay at the same resources. Tasks having no logistic effect can be executed in any arbitrary sequence without influencing deadlock behaviour. Using a similar confluence argument as in Section 4.1 the state-space can be reduced. We use the following confluent transition set.

Definition 4.3.1 (Confluent transition set D) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition and $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Let $(St, \hat{s}, Ac, \tau)$ be the state-space of the system. We define $\mathbf{D} \subseteq \tau$ to be the set consisting of the transitions $((tp, ms), (t, r), (tp', ms'))$ for which the following condition hold.*

$$anc(t) \cap NL \neq \emptyset \wedge \forall g \in G \cap anc(t).\ g \in NL.$$

Here, the definition of the non-logistic groups and clusters, notation

$$NL = \{n \in (G \cup L) \mid \forall t \in T, res \in R.\ n \in anc(t) \Rightarrow C_b(t, A(res)) = C_e(t, A(res))\}.$$

This definition only considers groups or clusters containing only non-logistic tasks.

Using the fact that $\mathbf{D}$ is a confluent set of transitions, it is only necessary to investigate only one edge labelled with a non-logistic task in each state having such an edge. All other edges in this state can be ignored, while exactly finding all deadlocks in the system.

Chapter 5

From system definition to PVS theory

In this chapter we look at the PVS theory, in which all definitions described in chapter 3 are represented. Further, we explain choices and lack of clarity and mention mistakes in [10]. To check the definitions and to get familiar with them, we translate the example called *SimpleMachine* in appendix B and prove correctness of some results of these definitions with PVS.

5.1 Static and dynamic system

5.1.1 Main definitions

First, we take a look at the structure of the system. The static and dynamic system are given as tuples. There are two ways to model this in PVS: tuples and records. Recall that accessing tuple components is done by the built-in *proj* function; *proj_1* represents the first component, *proj_2* the second and so on. While using records, we can refer to components by their names. We want to keep the model simple and easy to understand. Hence, for the sake of readability, we use records for modelling. Additional differences between tuples and records, discussed in [6], are of no importance for us.

Again to enhance readability, we choose to leave some components out of the records. We choose to leave functions inside the record and move given set declarations (like resources and tasks) outside the record. We come back to this later.

Static system. Leaving set declarations out of the record means that the static system will be a record with three components, namely the function that gives the capabilities which a resource can provide `A`, the material capacity `Rm`, and the function that gives the resources to which material can be transported from a certain resource `Mf`. The set of available resources `R` and the set of capabilities `C` will be defined outside the record. In practice, all used sets are non-empty and finite. Unfortunately, the information that sets are finite is used in, among others, system invariants 8 and 9. This means that we have to model this, resulting in a number of type-correctness conditions (TCCs) as we will see later in part III. It seems that two types are suitable for modelling a finite set: *is_finite* and *finite_set*. We will need the cardinality function from the prelude; this function needs a variable of type *finite_set*, so we use the latter one. Another reason why we prefer *finite_set* over *is_finite* is because in our case *is_finite* generates an extra existence TCC. *Finite_set* is defined as follows:

```
finite_set : TYPE = (is_finite) CONTAINING emptyset[T], T : TYPE
```

For *finite_set* we do not get this existence TCC since *finite_set* has a witness, namely `emptyset[T]`. Thus, *finite_set* is non-empty and this had to be proved by using `TYPE+` in the definition of `R` and `C`.

The code below represents the static system and consists of three parts; the set and function definitions, the record and an instance of the record:

```
R  : TYPE+ = finite_set
C  : TYPE+ = finite_set
A  : TYPE+ = [R -> C]
Rm : TYPE+ = [R -> nat]
Mf : TYPE+ = [R -> setof[R]]

staticSystem: TYPE+ = [#
  a  : A,
  rm : Rm
  mf : Mf
#]

ss : staticSystem
```

As you can see, both sets are non-empty, finite and do not show up in the record. Functions are non-empty and an instance of each of them is present in the record. We have one static system, called `ss`, and we can access the components by `ss`a, `ss`rm and `ss`mf.

Dynamic system. We can do the same for the dynamic system. Again, we leave the set definitions out of the record. So, the dynamic system will be a record containing eight components, namely the function that gives the nodes contained in a certain cluster `Ln`, the function that gives the nodes contained in a certain group `Gn`, the function that gives the allowed numbers of node alternatives to be selected from a group `Ga`, the function that gives the capabilities required to perform a certain task `I`, the function that gives the predecessors of a certain node `P`, the function that gives the material instances residing on a capability, and involved in a task, at the beginning of that task `Cb`, the function that gives the material instances residing on a capability, and involved in a task, at the end of that task `Ce`, and the function that gives the material instances initially residing on a certain resource `Ms`. Furthermore, we have four sets which are defined outside the record: one non-empty, finite set (material instances `M`) and three, possibly empty, finite sets (the set of tasks `T`, clusters `L`, and groups `G`). In practice, none of these sets will be empty.

Before we define the sets and functions mentioned above, we have to look at two given constraints. Firstly, the set of nodes `N` is defined as `N = T ∪ L ∪ G` and tasks, clusters and groups are disjunct. It is not possible to construct a new type `N` from the existing three sets. We could do it the other way round, so we define a non-empty, finite set `N` and make three subtypes out of this one. With some creativity, we have a number of options to achieve this. For example, we could pretend that we have to do with natural numbers instead of nodes. This has no impact on the correctness of the model; with the remainder function `rem` we get three disjoint subsets:

```
N : TYPE+ = nat

T : TYPE = {n:N | rem(3)(n) = 0}
L : TYPE = {n:N | rem(3)(n) = 1}
G : TYPE = {n:N | rem(3)(n) = 2}
```

Natural numbers are infinite and we want `N` to be a finite set. We could use a range to solve this. However, at this point it is not clear whether this can lead to difficulties later on in the model or during proofs. Another possibility is a definition with a predicate `p` for the remainder function and an enumeration type. We could define:

```
N : TYPE+ = finite_set

n : VAR N

enum : TYPE = {0,1,2}
```

```
typeP : TYPE+ = {p: [N -> enum] | (EXISTS (n:N): p(n) = 0) AND
                                  (EXISTS (n:N): p(n) = 1) AND
                                  (EXISTS (n:N): p(n) = 2)}
                CONTAINING (LAMBDA (n:N): IF (n=0)
                                            THEN 0
                                            ELSE IF (n=1)
                                                   THEN 1
                                                   ELSE 2
                                                 ENDIF
                                          ENDIF)

v_typeP : VAR typeP
```

Let's say that `v_typeP(n)=0` means that `n` represents a task, for clusters `v_typeP(n)=1` and for groups `v_typeP(n)=2`. Using `T`, `L` and `G` instead of `0`, `1` and `2` in the enumeration would probably have been more obvious. Here, we use natural numbers in the `CONTAINING` clause. Nodes have no connection at all with natural numbers and still we want to stick to the original model as much as possible. Although both solutions are correct, we choose another one without using natural numbers and more readable than the previous two. The code below is a part of the dynamic system:

```
N :  TYPE+ = finite_set

x : VAR N

T(x) : bool
G(x) : bool
L(x) : bool
```

Here, we use that `setof[N]` is nothing more than a function `(N -> bool)`. In the code above, `T`, `G` and `L` are subsets of `N`. Modelling the subsets in this way is readable and easy to understand, but has one drawback. Since we do not use natural numbers anymore, we have to specify the disjunction explicitly. We do this by defining four axioms:

```
disj0 : AXIOM T(x) OR L(x) OR G(x)
disj1 : AXIOM T(x) => NOT G(x) AND NOT L(x)
disj2 : AXIOM G(x) => NOT T(x) AND NOT L(x)
disj3 : AXIOM L(x) => NOT T(x) AND NOT G(x)
```

Axiom `disj0` ensures that there are no other categories than tasks, clusters and groups. A node has to be part of at least one of these three categories. The other three axioms ensure that a node is part of at most one of the three categories. Introducing too many axioms is very risky; they can be inconsistent. In our case, the model is small and it is obvious that these axioms are consistent. We think that it is more important to ensure a good readability without needing some tricks like the natural numbers in the previous attempts.

Secondly, no group has only 0 as allowed number, i.e. $\forall(g \in G) : Ga(g) \neq \{0\}$ by system invariant 2. Here, no axiom is needed. We define a new type, namely a set containing all possible sets of natural numbers, except the singleton $\{0\}$. We call this set of sets `setofNat`:

```
setofNat : TYPE+ = {x:setof[nat] |
                      NOT (subset?(x, singleton[nat](0)) AND
                           subset?(singleton[nat](0),x))}
```

Now, the function `Ga` is of type `[N -> setofNat]`. The code below represents the final part of the dynamic system:

```
Ln : TYPE+ = [N -> setof[N]]
Gn : TYPE+ = [N -> setof[N]]
Ga : TYPE+ = [N -> setofNat]
I  : TYPE+ = [N -> setof[C]]
P  : TYPE+ = [N -> setof[N]]
M  : TYPE+ = finite_set
Cb : TYPE+ = [N, C -> setof[M]]
Ce : TYPE+ = [N, C -> setof[M]]
Ms : TYPE+ = [R -> setof[M]]

dynamicSystem: TYPE+ = [#
  ln : Ln,
  gn : Gn,
  ga : Ga,
  i  : I,
  p  : P,
  cb : Cb,
  ce : Ce,
  ms : Ms
#]

ds : dynamicSystem
```

Note that all occurrences of `T`, `L` and `G` in the function parameters have been replaced by `N`. If not all three flavours are allowed, then we have to define some axioms or we have to add this restriction when needed. Since we do not want to look for any restrictions in the static and dynamic system, in the definitions of the additional functions and in axioms, we choose the latter one. Now we have a clear view of the restrictions. And it is easier to define restrictions in the function itself, because they show up immediately in the proofs. Axioms have to be called separately and it is easy to forget their existence.

Now it is easier to explain why we left the set declarations out of the records. The model must not be larger than necessary. If we want the set declarations inside the record, then we have to add the following lines to the record:

```
t: setof[N],
g: setof[N],
l: setof[N],
```

Finally, we have to make sure that these three sets are disjunct. This is done by making an instance of the record. `T`, `L` and `G` are disjunct; we assign `T`, `L` and `G` to `t`, `l` and `g` respectively to the record. But unfortunately, we are not allowed to give some components a value, just all of them or no one at all. We can complete this by declaring a constant of each field's type and assign this constant to the field as follows:

```
cLn: Ln
cMs: Ms
...

ds: dynamicSystem = (#
  t  := T,
  l  := L,
  g  := G,
  ln := cLn,
  ...
  ms := cMs
 #)
```

Another possibility to model this is to instantiate a temporary dynamic system and use this temporary dynamic system to create a new one where only those fields which need a value are modified. This looks as follows:

```
dsTemp: dynamicSystem

ds: dynamicSystem = dsTemp WITH [`t := T, `l := L, `g := G]
```

Modelling in the first way is confusing since the additional value is very small and we get a lot of extra code. The second possibility is confusing because a non-existing dynamic system is needed; we only have one such a system. Furthermore, function parameters have to be clear otherwise we have to use destructors to pass the desired components of the records to the functions. And, when needed, we explicitly would have to make sure that the records contain the full set and not just a subset of any declared set. Of course, both problems can be solved and specified in PVS, but it would be hard to understand. By splitting it up, we still have a distinction between a static and a dynamic system, and we achieve a good readability.

5.1.2 Additional definitions

Ancestor. Say, we have a binary relation R, where xRy means "x is immediately contained in y (in one step)". This means that y is either a cluster or a group. If y is a cluster, then x is a member of `Ln(y)`. Else, x is a member of `Gn(y)`. The ancestor function is the transitive closure of R, which means that x is contained in y in one or more steps. This transitive closure can be inductively defined:

```
ancRel(ds:dynamicSystem)(n1:N, n:N): INDUCTIVE boolean =
  (G(n1) AND  member(n,ds`gn(n1)))  OR            % base
  (L(n1) AND  member(n,ds`ln(n1)))  OR
  (EXISTS (n2:N): ancRel(ds)(n1,n2) AND           % compound
                  ancRel(ds)(n2,n))

anc(ds:dynamicSystem)(n:N): setof[N] =
  LAMBDA(n1:N): ancRel(ds)(n1,n)
```

The binary relation is of type `(N -> N -> bool)`; we omit the `dynamicSystem` parameter. Relation *ancRel* returns `true` if `n1` is an ancestor of `n`. If `n1` is a group or a cluster and `n` is contained in this group or cluster, then we are done and the result is `true` (base). As mentioned above, we have to add explicitly that function `Gn` accepts groups only and function `Ln` accepts clusters only. This is done by adding `G(n1)` resp. `L(n1)`. If we are not done in one step, then we have to search for an existing node in-between. So, we have to search a node `n2`, for which `n1` is an ancestor of `n2` and `n2` is an ancestor of `n` (compound).

With *ancRel*, we can define the *anc*-function. This function has type `(N -> [N])`. All `n1` for which *ancRel* returns `true` are an element of the final result of *anc(n)*.

Allsucc. The function *allsucc* can be defined in the same way. It is the transitive closure of a relation S, where xSy means that y is a direct successor of x.

```
allsuccRel(ds:dynamicSystem)(n1:N, n:N): INDUCTIVE boolean =
  member(n,ds`p(n1)) OR
  (EXISTS (n2:N): member(n2,anc(ds)(n)) AND
                  member(n2,ds`p(n1))) OR         % base
  (EXISTS (n2:N): allsuccRel(ds)(n1,n2) AND
                  allsuccRel(ds)(n2,n))           % compound

allsucc(ds: dynamicSystem)(n:N): setof[N] =
  LAMBDA(n1:N): allsuccRel(ds)(n1,n)
```

If the predecessor of a node `n1`, called `n`, is contained in a group or cluster, then this group or cluster is a predecessor of `n1` and not node `n` itself. Therefore, we have to distinct two base cases: one for `n` itself and one for the ancestors of `n`. The inductive step is similar to the ancestor function.

Mat. The *mat*-function is a straight-forward translation; we only have to make sure that `t` has type `T` and not `N`, so we add this immediately after the existential quantifier.

```
mat(ds:dynamicSystem)(n:N): setof[M] =
  {m:M | EXISTS (t:N),(c:C):
    T(t) AND
    (n=t OR member(n,anc(ds)(t))) AND
    member(m,union(ds`cb(t,c),ds`ce(t,c)))}
```

5.1.3 System invariants

Most of the system invariants, mentioned in section 3.1.3 can easily be translated. Below you see the definitions of them in PVS. We give a more detailed explanation when needed.

1. The nodes in the system have a hierarchical structure. For all nodes $n \in N$ it must hold that $n \notin anc(n)$.

   ```
   hierarchical: AXIOM
     FORALL (n:N): NOT member(n,anc(ds)(n))
   ```

2. No group has only 0 as allowed number. I.e. $G_a(g) \neq \{0\}$ for all groups $g \in G$.

   This is a part of the definition of the dynamic system. See section 5.1.1.

3. The capabilities in $C_b(t)$ and $C_e(t)$ exist also in $I(t)$, i.e. for all tasks $t \in T$ and capabilities $c \in C$ if $C_b(t,c) \cup C_e(t,c) \neq \emptyset$ then $c \in I(t)$;

   ```
   cap: AXIOM
     FORALL (t:N), (c:C):
       T(t) AND union(ds`cb(t,c), ds`ce(t,c)) /= emptyset IMPLIES
         member(c,ds`i(t))
   ```

4. P contains no cycles. For each node $n \in N$ it must hold that $n \notin allsucc(n)$.

   ```
   noCycles: AXIOM
     FORALL (n:N): NOT member(n,allsucc(ds)(n))
   ```

5. There is no precedence relation between node alternatives in a group. For all groups $g \in G$ and nodes $n \in N$ it holds that if $n \in G_n(g)$ then $P(n) = \emptyset$;

   ```
   noPrecedence: AXIOM
     FORALL (g:N), (n:N): G(g) AND member(n,ds`gn(g)) IMPLIES
       ds`p(n) = emptyset
   ```

6. Precedence relations do not cross group boundaries. For all groups $g \in G$ and nodes $n, n' \in N$ it is the case that if $g \in anc(n)$ then $n' \in P(n) \Rightarrow g \in anc(n')$ and $n \in P(n') \Rightarrow g \in anc(n')$;

   ```
   crossGroup: AXIOM
     FORALL (g:N), (n,n1:N):
       G(g) AND member(g,anc(ds)(n)) IMPLIES
         ((member(n1,ds`p(n)) IMPLIES
           member(g,anc(ds)(n1))) AND (member(n,ds`p(n1)) IMPLIES
             member(g,anc(ds)(n1))))
   ```

7. All tasks in a group concern the same material:

$$\forall t, t' \in T, g \in G \cap anc(t) \cap anc(t').\ mat(t) = mat(t').$$

```
sameMaterial: AXIOM
  FORALL (t,t1: N), (g:N):
    T(t) AND T(t1) AND G(g) AND
      member(g,intersection(anc(ds)(t),anc(ds)(t1))) IMPLIES
        mat(ds)(t) = mat(ds)(t1)
```

8. The subsets of material instances involved in a task remain the same from the beginning to the end of a task, i.e. for all tasks $t \in T$:

$$\bigcup_{c \in I(t)} C_b(t,c) = \bigcup_{c \in I(t)} C_e(t,c).$$

This constraint implies that only closed systems are considered where no material enters or leaves the system.

Unfortunately, the $\bigcup$-operator is not defined in the PVS prelude or in any other available library so we define it ourselves. We call this operator *setUnion*. The base case is obvious, if we have an empty set then the result is $\emptyset$. If we have a non-empty set, then we have to apply a function *f* on every element of the set. The function *f* returns a set and the final result is computed by taking the union of all of these resulting sets. We define this recursively. We choose an element from a set `S`, apply *f* on this element and do the same with the rest of the set `S`. Finally, we take the union of all these results. When defining a function recursively, we have to specify a *measure*, i.e. we have to show that *setUnion* terminates. Here, *card(S)* does the job since we remove one element from the set; the length of the set reduces and will be zero after a number of iterations. Our definition of *setUnion*:

```
setUnion(S:setof[C],t:N,f:[N,C -> setof[M]]): RECURSIVE setof[M] =
    IF (empty?(S)) THEN emptyset
    ELSE union(f(t,choose(S)),setUnion(rest(S),t,f))
    ENDIF
    MEASURE
      (LAMBDA (S:setof[C]),(t:N),(f:[N,C -> setof[M]]): card(S))
```

Now, defining the system invariant is easy:

```
closedSystem: AXIOM
  FORALL (t:N), (c:C):
    T(t) AND member(c,ds`i(t)) AND
      setUnion(ds`i(t),t,ds`cb) = setUnion(ds`i(t),t,ds`ce)
```

9. Initially, the material capacity of all resources is not exceeded. For any resource $r \in R$ it holds that $\#\hat{m}_s(r) \leq R_m(r)$. Here $\#S$ gives the number of elements in the set S.

```
matCap: AXIOM
  FORALL (r:R): card(ds`ms(r)) <= ss`rm(r)
```

5.2 State space

5.2.1 Main definitions

Successful. *Successful* is a predicate and therefore we put a question mark in the function name in PVS. The definition is straight-forward and consists of three parts; one base case $n \in T$ and two inductively

defined cases $n \in L$ and $n \in G$. For $n \in G$, we changed $(\#\{n' \in G_n(n) \mid \mathit{successful}(n', tp)\}) = 0$ into $(\{n' \in G_n(n) \mid \mathit{successful}(n', tp)\}) = \emptyset$. This means exactly the same, but the empty set is easier to use than the cardinality function. So, in PVS we get the following definition:

```
successful?(ds:dynamicSystem)(n:N, tp:setof[N]): INDUCTIVE boolean =
  (T(n) AND  member(n,tp)) OR                                  % base
  (L(n) AND (FORALL (n1:N): member(n1,ds`ln(n)) IMPLIES        % compound
    successful?(ds)(n1,tp))) OR
  (G(n) AND                                                    % compound
    member(card({n1:N | member(n1,ds`gn(n)) AND
      successful?(ds)(n1,tp)}),ds`ga(n)) AND
    ({n1:N | member(n1,ds`gn(n)) AND
      successful?(ds)(n1,tp)} = emptyset IMPLIES
    (FORALL (n2:N): member(n2,ds`p(n)) AND successful?(ds)(n2,tp))))
```

If we have a set of nodes instead of one single node then the predicate *successful* should hold for all nodes $\in$ np. We use overloading:

```
successful?(ds:dynamicSystem)(np:setof[N], tp:setof[N]): boolean =
  FORALL (n:N): member(n,np) IMPLIES successful?(ds)(n,tp)
```

Successor. In chapter 3.2.1, the function $\mathit{successor}(n)$ is defined as follows (definition 3.2.2):

$$\begin{array}{rcl} \mathit{successor}(n) & = & \{n' \in N \mid (n \cup \mathit{anc}(n)) \cap P(n') \neq \emptyset\} \cup \\ & & \{\mathit{successor}(n') \mid n' \in \mathit{successor}(n) \cap G \ \wedge \ 0 \in G_a(n')\}. \end{array}$$

Here we have a problem with the types. The function *successor* should have type `(N -> [N])`. The first part of the definition is correct indeed but the second part is of type `(N -> [[N]]`. We solve this problem by changing the definition above in:

$$\begin{array}{rcl} \mathit{successor}(n) & = & \{n' \in N \mid (n \cup \mathit{anc}(n)) \cap P(n') \neq \emptyset\} \cup \\ & & \bigcup_{n' \in N} \{\mathit{successor}(n') \mid n' \in \mathit{successor}(n) \cap G \ \wedge \ 0 \in G_a(n')\}. \end{array}$$

Before we model this in PVS, we want to have both parts of the definition in the same form. After rewriting we obtain the following definition, which replaces Definition 3.2.2:

Definition 5.2.1 *Let $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. The predicate $\mathit{successor}(n)$ for a node $n \in N$ is defined as the smallest predicate satisfying:*

$$\begin{array}{rcl} \mathit{successor}(n) & = & \{n'' \in N \mid (n \cup \mathit{anc}(n)) \cap P(n'') \neq \emptyset\} \cup \\ & & \{n'' \in N \mid \exists n' \in N : n'' \in \mathit{successor}(n') \ \wedge \ n' \in \mathit{successor}(n) \cap G \ \wedge \ 0 \in G_a(n')\}. \end{array}$$

Now, *successor* can be modelled inductively just like *anc* and *allsucc*:

```
successorRel(ds:dynamicSystem)(n:N, n2:N): INDUCTIVE boolean =
  intersection(union(n,anc(ds)(n)),ds`p(n2)) /= emptyset OR
  (EXISTS (n1:N): successorRel(ds)(n1,n2) AND
                  successorRel(ds)(n,n1) AND
                  G(n1) AND
                  member(0,ds`ga(n1)))

successor(ds:dynamicSystem)(n:N): setof[N] =
  LAMBDA(n1:N): successorRel(ds)(n,n1)
```

Initiated. The function *initiated* can be written down almost literally in PVS:

```
initiated(ds:dynamicSystem)(tp:setof[N]): setof[N] =
  {n:N | EXISTS (t:N): T(t) AND member(t,tp) AND member(n,anc(ds)(t))}
```

Bypassed. Defining the function max is not difficult too. Max is a part of the function $bypassed$ (defined below) and is only used with Ga. So, max takes a set `S` of type `setofNat`. It returns the largest number in that set; the result has type `nat`. The result `r` of max is an element of `S` and no other element in `S` is larger than `r`. We use the function the from the prelude to extract the only element out of a singleton. The definition looks as follows:

```
max(S:setofNat): nat =
  the({n:nat | member(n,S) AND
    FORALL (m:nat): member(m,S) IMPLIES n >= m})
```

With these definitions, both $bypassed(t, tp)$ and $bypassed(tp)$ are very straight-forward:

```
bypassed(ds:dynamicSystem)(t:N,tp:setof[N]): boolean =
  T(t) AND NOT member(t,tp) AND
    EXISTS (g:N): G(g) AND member(g,anc(ds)(t)) AND
                  (intersection(successor(ds)(g),tp) /= emptyset OR
                  (difference
                     (intersection
                           (union(anc(ds)(t),singleton(t)),
                        ds`gn(g)),
                     initiated(ds)(tp)) /= emptyset AND
                   card(intersection(ds`gn(g),initiated(ds)(tp))) =
                     max(ds`ga(g))))

bypassed(ds:dynamicSystem)(tp:setof[N]): setof[N] =
  {t:N | bypassed(ds)(t,tp)}
```

State space. Now we have the functions needed to define the state space of the system. The state space consists of a number of states, actions and transitions. First, we define these three components:

```
State: TYPE+ = [# tp: setof[N], ms: Ms #]
Action: TYPE+ = [# t: N, r: setof[R] #]
Transition: TYPE+ = [# s: State, ac: Action, s1: State #]
```

A state contains the information which tasks have been successfully processed and which material instances can be found at which resource. The first one is a set of nodes, actually a set of tasks, and the second one is a function of type Ms, see chapter 5.1.1. An action indicates which task is executed at which resource. And, finally, a transition represents the connection of a beginning and an end state by a single action. Furthermore, we define two axioms:

```
tp_ax: AXIOM
  FORALL (t: N,tp:setof[N]): member(t,tp) IMPLIES T(t)

ac_ax: AXIOM
  FORALL (ac:Action): T(ac`t)
```

The first one ensures that the set of processed tasks consists of tasks only. As mentioned above, no groups or clusters are allowed here. And the second one does the same for the task in the $action$-record. Then we have to model the set of transitions, called τ. This set contains transitions for which nine conditions hold. Translating the conditions does not lead to any problems:

```
Tau: TYPE+ =
  {tr:Transition  |
    successful?(ds)(ds`p(tr`ac`t),tr`s`tp) AND                    % (1)
    member(tr`ac`t, difference(T, union(tr`s`tp,                   % (2)
```

```
      bypassed(ds)(tr`s`tp))))  AND
    (FORALL (cap:C): member(cap,ds`i(tr`ac`t)) IMPLIES            % (3)
      EXISTS (res:R): member(res,tr`ac`r) AND ss`a(res) = cap) AND
    (FORALL (res:R): member(res,tr`ac`r) IMPLIES                  % (4)
      subset?(ds`cb(tr`ac`t,ss`a(res)),tr`s`ms(res))) AND
    (FORALL (res:R),(m:M):                                        % (5)
      member(res,tr`ac`r) AND member(m,tr`s`ms(res)) IMPLIES
        member(m,tr`s1`ms(res)) OR
      EXISTS (res1:R): member(res1,ss`mf(res)) AND
                       member(m,tr`s1`ms(res1))) AND
    (FORALL (res:R): card(tr`s1`ms(res)) <= ss`rm(res)) AND       % (6)
    tr`s1`tp = union(tr`s`tp,singleton(tr`ac`t)) AND              % (7)
    (FORALL (res:R): member(res,tr`ac`r) IMPLIES                  % (8)
      tr`s1`ms(res) = union(difference(tr`s`ms(res),
         ds`cb(tr`ac`t,ss`a(res))),ds`ce(tr`ac`t,ss`a(res)))) AND
    (FORALL (res:R): (NOT member(res,tr`ac`r)) IMPLIES            % (9)
      tr`s1`ms(res) = tr`s`ms(res)) }
```

Just like we did for the static and dynamic system, we make a record for the state space. We have four components: a set of states, a single initial state, a set of actions and a set of transitions for which the nine conditions mentioned above hold.

```
statespace: TYPE+ = [#
  state: setof[State],
  state0: State,            % initial state
  action: setof[Action],
  tau: setof[Tau]
#]
```

Finally, we make an instance of the state space:

```
stsp_s: setof[State]
stsp_s0: State = (# `tp := emptyset, `ms :=  ds`ms #)
stsp_a: setof[Action]
stsp_t: setof[Tau] = fullset[Tau]

stsp: statespace = (#
  state := stsp_s,
  state0 := stsp_s0,
  action := stsp_a,
  tau := stsp_t
#)
```

For the initial state, $tp = \emptyset$ and $ms = \hat{m}_s$ and for τ in the state space we take the full set of transitions for which the nine conditions hold. The example in section 5.3 shows that state `(C0,p0)` is not possible due to the capacity limit of `C0`. Also, action `(t0p2, E0)` is not valid. So, both sets are no full sets. Since we have to add some information to the initial state and the set of transitions, we cannot just say `stsp: statespace`.

5.2.2 Additional definitions

Before we can start with the proofs of the first reduction technique, two more definitions are needed: the functions $succ$ and $unsafe$.

Succ. The function $succ$ is defined as follows in section 3.2.2:

$$\begin{aligned} succ(n,V) = &\{n' \in N \mid n \in V(n')\} \cup \\ &\{n'' \in N \mid \neg \exists n' \in N.\ n \in V(n') \wedge \\ &\qquad \exists n''' \in N.\ n \in G_n(n''') \cup L_n(n''') \wedge n'' \in succ(n''', V)\}. \end{aligned}$$

To convert this to PVS, we write this in a slightly different way:

$$\begin{aligned} succ(n,V) = &\{n'' \in N \mid n \in V(n'')\} \cup \\ &\{n'' \in N \mid \neg \exists n' \in N.\ n \in V(n') \wedge \\ &\qquad \exists n' \in N.\ n \in G_n(n') \cup L_n(n') \wedge n'' \in succ(n', V)\}. \end{aligned}$$

Now it is easy to write down the definition in PVS. We have seen this in section 3.2.1 for the *successor*-function. The definition looks as follows:

```
succRel(ds:dynamicSystem)(V:P)(n:N, n2:N): INDUCTIVE boolean =
  member(n,V(n2)) OR
  (NOT EXISTS (n1:N): member(n,V(n1))) AND
  EXISTS (n1:N): (member(n,union(ds`gn(n1),ds`ln(n1))) AND
                  succRel(ds)(V)(n1,n2))

succ(ds:dynamicSystem)(n:N, V:P): setof[N] =
  LAMBDA(n1:N): succRel(ds)(V)(n,n1)
```

Unsafe. For $unsafe$ we need some extra definitions. Firstly, we define the $\sum$-operator, just like we did for the $\bigcup$-operator in section 5.1.3. We call the function *setSum*:

```
setSum(S:setof[R], f:[R -> nat]): RECURSIVE nat =
  IF (empty?(S)) THEN 0
  ELSE f(choose(S)) + setSum(rest(S),f)
  ENDIF MEASURE (LAMBDA (S:setof[R]),(f:[R -> nat]): card(S))
```

Then, we define the range we want to use with the $\sum$-operator. This is not difficult; both `res` and `res'` must be an element of `R` and `res` must be an element of `Mf(res')`. We call the range `RS`, resource subset, and is defined as:

```
RS(ss:staticSystem)(res:R): setof[R] =
  LAMBDA (res1:R): member(res,ss`mf(res1))
```

Using the definitions above, we write down the function *unsafe*:

```
unsafe(ss:staticSystem): setof[R] =
  {res:R | setSum(RS(ss)(res),ss`rm) > 1}
```

5.3 Example

To check the correctness of the definitions in the previous sections, we use the *Simple machine* from appendix B. We computed a number of results of some functions by hand and we wrote this as conjectures. These conjectures are mentioned in appendix B.3 and we try to prove them with PVS. We discuss some parts of these conjectures in this section. Of course, we can only prove that the definitions return the correct results for the values mentioned in appendix B, we cannot prove correctness for all possible values. But if we choose some interesting cases, i.e. cases involving groups or clusters, we assume the definitions are correct indeed.

As our first example we take the conjecture `anc(t4p2) = {c0p2, g0p2}`. We can use PVS only to check the correctness of a conjecture, not to compute a result. So we can prove that `anc(t4p2) =`

{c0p2, g0p2} holds, but we cannot use PVS to compute the outcome of anc(t4p2). The *ancestor* function is inductively defined. For each inductively defined function, we get two functions for free: one of them deals with induction and the other one deals with weak induction. These functions become visible by the command "M-x prettyprint-expanded" and can be used to prove a conjecture. To illustrate the use of one of these given functions, we try to prove the mentioned conjecture with PVS:

```
  |-------
[1]   subset?(anc(ds)(t4p2), add(c0p2, singleton(g0p2))) AND
       subset?(add(c0p2, singleton(g0p2)), anc(ds)(t4p2))
```

This equality could be proved by splitting it up in two subgoals: anc(t4p2) $\subseteq$ {c0p2, g0p2} and {c0p2, g0p2} $\subseteq$ anc(t4p2). The latter subgoal is not a problem; we do not need any of the given induction functions. Expanding definitions and instantiating them with appropriate values is enough to prove this subgoal. The first one needs some more attention. After expanding all definitions we get the following subgoal:

```
  |-------
{1}   FORALL (x: N): ancRel(ds)(x, t4p2) IMPLIES c0p2 = x OR x = g0p2
```

We use weak induction. The generated function is shown below:

```
ancRel_weak_induction: AXIOM
  FORALL (ds: dynamicSystem, P: [[N, N] -> boolean]):
    (FORALL (n1, n: N):
       (G(n1) AND member(n, ds`gn(n1))) OR
        (L(n1) AND member(n, ds`ln(n1))) OR
         (EXISTS (n2: N): P(n1, n2) AND P(n2, n))
        IMPLIES P(n1, n))
     IMPLIES (FORALL (n1, n: N): ancRel(ds)(n1, n) IMPLIES P(n1, n));
```

Now we have to match our subgoal with FORALL (n1,n:N): ancRel(ds)(n1,n) IMPLIES P(n1,n). So we unify n with t4p2. Then n1 equals either c0p2 or g0p2. We could instantiate the predicate in *ancRel_weak_induction* with (n=t4p2 => n1=c0p2 OR n1=g0p2) but this will not be enough because we need information about c0p2 and g0p2. So we use the command (use) to introduce and instantiate *ancRel_weak_induction*:

```
(use ancRel_weak_induction
  "ds" "ds"
  "P" "(n=t4p2 => n1=c0p2 OR n1=g0p2) AND
       (n=c0p2 => n1=g0p2) AND
       (n=g0p2 => FALSE)")
```

Now we get the following sequent; the proof can be finished by typing (grind):

```
{-1}  (FORALL (n1_1, n_1: N):
         ((G(n1_1) AND member(n_1, ds`gn(n1_1))) OR
           (L(n1_1) AND member(n_1, ds`ln(n1_1))) OR
            (EXISTS (n2: N):
                    (n2 = t4p2 => n1_1 = c0p2 OR n1_1 = g0p2)
                AND (n2 = c0p2 => n1_1 = g0p2) AND NOT n2 = g0p2
                AND (n_1 = t4p2 => n2 = c0p2 OR n2 = g0p2)
                AND (n_1 = c0p2 => n2 = g0p2) AND NOT n_1 = g0p2))
          IMPLIES
          (n_1 = t4p2 => n1_1 = c0p2 OR n1_1 = g0p2) AND
           (n_1 = c0p2 => n1_1 = g0p2) AND NOT n_1 = g0p2)
       IMPLIES
```

```
        (FORALL (n1_1, n_1: N):
           ancRel(ds)(n1_1, n_1) IMPLIES
            (n_1 = t4p2 => n1_1 = c0p2 OR n1_1 = g0p2) AND
             (n_1 = c0p2 => n1_1 = g0p2) AND NOT n_1 = g0p2)
  |-------
[1]   ancRel(ds)(x, t4p2) IMPLIES c0p2 = x OR x = g0p2
```

The test cases for *allsucc*, *mat*, *successful*, *successor* and *initiated* can all be proved in the same way. These functions are either inductively defined or *anc* is a part of them or both. Already proved conjectures can be used to prove unproved conjectures. Furthermore, test cases for *succ* are no problem. Expanding and instantiating the definition of *succ* and *succRel* is enough to prove the conjectures.

Now, let us take a look at another conjecture mentioned in appendix B.3, namely `setUnion(I(t0p1), t0p1,Ce) = ∅`. In PVS this looks as follows:

```
  |-------
[1]   setUnion(ds`i(t0p1), t0p1, ds`ce) = emptyset
```

We start our proof with `(apply-extensionality)` and `(expand "emptyset").` `I(t0p1)` contains two elements, namely `P` and `C`. This means we can choose an element twice and then the base case of *setUnion* holds. So we apply `(expand "setUnion")` three times followed by `(smash)` to split the if-statements. Now we have four subgoals. The first two of them say that we do not have an empty set by removing zero or one element from `I(t0p1)` and the third one says that there is an x $\in$ M for which x $\neq$ p1. These cases can easily be proved by `(grind)`. Finally, we have to prove that we get an empty set when we remove two elements from `I(t0p1)`. This looks obvious but we have to guide PVS step by step through it. The *choose* function from the prelude is defined as:

```
% A choice function for nonempty sets
choose(p: (nonempty?)): (p) = epsilon(p)
```

After expanding and flattening a number of functions we get:

```
{-1}  x!2 = P OR x!2 = C
  |-------
{1}   epsilon({y: C |
                  NOT epsilon({x: C | x = P OR x = C}) = y AND
                   (y = P OR y = C)})
       = x!2
{2}   epsilon({x: C | x = P OR x = C}) = x!2
```

About epsilons, the prelude says the following: "Epsilons provides a *choice* function that does not have a non-emptiness requirement. Given a predicate `p` over the type `t`, epsilon produces an element of satisfying that predicate if one exists, and otherwise produces an arbitrary element of that type." We use case splitting to distinguish three cases: (1) `epsilon(x: C | x = P OR x = C) = C`, (2) `epsilon(x: C | x = P OR x = C) = P` and (3) the negation of (1) and (2). The latter case is obvious since there is no other choice than (1) or (2); `(grind)` completes this subgoal. For case (1) we get:

```
  |-------
{1}   epsilon({y: C | NOT C = y AND (y = P OR y = C)}) = P
[2]   C = P
```

To prove this we need the only available axiom for epsilon. This axiom is defined as follows:

```
epsilon_ax: AXIOM (EXISTS x: p(x)) => p(epsilon(p))
```

This axiom leads to:

```
{-1}  FORALL (p: pred[C]): (EXISTS (x: C): p(x)) => p(epsilon(p))
  |-------
[1]   epsilon({y: C | NOT C = y AND (y = P OR y = C)}) = P
[2]   C = P
```

After instantiating rule -1 with `LAMBDA (y:C): C /= y AND (y = P OR y = C)`, we can complete the subgoal without any difficulties. Case (2) can be proved in a symmetric way.

Unsafe is a unique set of resources. In our example *Simple Machine*, this set of resources contains just one element: singleton {`R0`}.

```
  |-------
[1]   subset?(unsafe(ss), singleton(R0)) AND
       subset?(singleton(R0), unsafe(ss))
```

Proving this was not as easy as expected it would be. We need a lot of case splitting and we also use some extra lemmas to make the proof more transparant. It is impossible to explain the proof clearly here and it is also quite useless. These proofs are - in our opinion - all understandable and easy to follow so we refer to the CD for the complete proof.

Just like we did with *setUnion*, we use axiom *epsilon_ax* to prove some conjectures with *max*. We take a look at conjecture `max({2,4,3}) = 4`:

```
  |-------
[1]   max(restrict[real,nat,boolean](add(2,add(4,singleton(3))))) = 4
```

After expanding as much as possible and introducing the axiom, we have to prove:

```
{-1}  FORALL (p: pred[nat]): (EXISTS (x: nat): p(x) => p(epsilon(p))
  |-------
[1]   epsilon({n: nat |
                 (2 = n OR 4 = n OR n = 3) AND
                  (FORALL (m: nat):
                    2 = m OR 4 = m OR m = 3 IMPLIES n >= m)})
       = 4
```

Instantiating rule -1 with `LAMBDA (n:nat):  (2 = n OR 4 = n OR n = 3) AND (FORALL (m:nat):  2 = m OR 4 = m OR m = 3 IMPLIES n >= m))` does the job.

All conjectures in appendix B.3 are proved. So, now it seems that the definitions mentioned so far are translated well. The model and the proofs of the *Simple Machine* are included on the CD. You can see the used strategies by typing `"M-x show-proof"` or redo the proofs by typing `"M-x xpr"`. Note that we did not try to optimise the proofs; many of them could be done in a faster or more elegant way. But that was not the goal we wanted to achieve; correctness of the results is more important.

Part III

Proving correctness of reduction techniques

Chapter 6

Interleaving of independent parallel tasks

6.1 Confluence

Before we can start with the first reduction technique, we need to translate a few more definitions. We have to prove that set C is confluent, so we need to translate the definitions of confluence (Definition 4.1.1) and set C (Definition 4.1.3).

Translating the definition of confluence results in the following PVS code:

```
confluent(c:setof[setC_el]): boolean =
  FORALL (tr1:setC_el,tr2:Tau): (member(tr1,c) AND tr1`s = tr2`s) =>
    ((EXISTS (tr3:Tau,tr4:setC_el): member(tr4,c) AND
    (EXISTS (s3:State): tr3`s = tr1`s1 AND tr3`ac = tr2`ac  AND  % I
                        tr3`s1 = s3 AND tr4`s = tr2`s1 AND
                        tr4`ac = tr1`ac AND tr4`s1 = s3) OR
    (tr4`s = tr2`s1 AND tr4`ac = tr1`ac AND tr4`s1 = tr1`s1) OR  % II
    (tr3`s = tr1`s1 AND tr3`ac = tr2`ac AND tr3`s1 = tr2`s1)) OR % III
    tr1`s1 = tr2`s1)                                             % IV
```

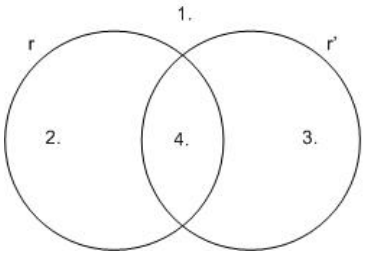
The implementation of Set C in PVS is given below:

```
setC_el: TYPE = {tr: Tau |
  (FORALL (t2:N):                                                % 1
    (T(t2) AND intersection(mat(ds)(t2), mat(ds)(tr`ac`t))
      /= emptyset) IMPLIES
    intersection(union(t2,anc(ds)(t2)),allsucc(ds)(tr`ac`t))
      /= emptyset) AND
  (FORALL (g,n:N): (G(g) AND member(n,union(g,anc(ds)(g)))) IMPLIES % 2
    intersection(union(anc(ds)(tr`ac`t),singleton(tr`ac`t)),
      succ(ds)(n,ds`p)) = emptyset) AND
  (intersection(G,anc(ds)(tr`ac`t)) = emptyset) AND               % 3
  (FORALL (res:R): member(res,unsafe(ss)) IMPLIES                 % 4
    difference(ds`ce(tr`ac`t,ss`a(res)),ds`cb(tr`ac`t,ss`a(res)))
      = emptyset)}

setC: setof[setC_el] = fullset[setC_el]
```

As you can see in Fig. 4.1, state s''' is related to states s' and s''. The set of executed tasks in state s''' equals either the executed tasks in state s' plus task t or the executed tasks in state s'' plus task t'. We

model this as a function with two parameters. Note that it is up to the user to take appropriate values. For the material instances ms‴ we take the values mentioned in [2]. These values are depicted in Fig. 6.1. Since every state consists of only one set of executed tasks and one set of material instances, we can define state s‴ as well. Here, it does not matter which of the two options we choose for the set of executed tasks; both options give the same result.



1. $\forall res \notin (r \cup r')$: $ms'''(res) = ms(res)$
2. $\forall res \in (r \setminus r')$: $ms'''(res) = ms'(res)$
3. $\forall res \in (r' \setminus r)$: $ms'''(res) = ms''(res)$
4. $\forall res \in (r \cap r')$: $ms'''(res) = (((ms(res) \setminus Cb(t, A(res))) \cup Ce(t, A(res))) \setminus Cb(t', A(res))) \cup Ce(t', A(res)))$

Figure 6.1: Values for ms‴

```
tp3(tp:setof[N],t:N): setof[N] = union(tp,singleton(t))

ms3(ss:staticSystem,ds:dynamicSystem)(tr1:setC_el, tr2:Tau): Ms =
  LAMBDA (res:R):
    COND
      NOT member(res,union(tr1`ac`r,tr2`ac`r)) -> tr1`s`ms(res),
      member(res,difference(tr1`ac`r,tr2`ac`r)) -> tr1`s1`ms(res),
      member(res,difference(tr2`ac`r,tr1`ac`r)) -> tr2`s1`ms(res),
      member(res,intersection(tr1`ac`r,tr2`ac`r)) ->
        union(difference(union(difference(tr1`s`ms(res),
        ds`cb(tr1`ac`t,ss`a(res))),ds`ce(tr1`ac`t,ss`a(res))),
        ds`cb(tr2`ac`t,ss`a(res))),ds`ce(tr2`ac`t,ss`a(res)))
    ENDCOND

s3(ss:staticSystem, ds:dynamicSystem)(tr1:setC_el, tr2:Tau): State =
  (# `tp := tp3(tr1`s1`tp,tr2`ac`t), `ms := ms3(ss,ds)(tr1,tr2) #)
```

The begin state of transition tr3 is the end state of transition tr1 and transition tr3 executes task t′. Symmetrically, the begin state of transition tr4 is the end state of transition tr2 and transition tr4 executes task t. Furthermore, state s‴ is the end state for both transition tr3 and transition tr4. These relations are modelled below:

```
tr3(tr1:setC_el, tr2:Tau, s3:State): Transition =
  (# `s := tr1`s1, `ac := tr2`ac, `s1 := s3 #)

tr4(tr1:setC_el, tr2:Tau, s3:State): Transition =
  (# `s := tr2`s1, `ac := tr1`ac, `s1 := s3 #)
```

Now we have all the information needed to prove that set C is confluent; i.e. we have to prove the conjecture `setCIsConfluent: CONJECTURE confluent(setC)`. For the start of the proof we have only one option, namely replacing `confluent(setC)` by its definition. Then we remove the universal quantifier in that definition. We obtain:

```
[-1]  member(tr1!1, setC)
[-2]  tr1!1`s = tr2!1`s
  |-------
```

```
[1]   EXISTS (tr3: Tau, tr4: setC_el):
        (member(tr4, setC) AND
          (EXISTS (s3: State):
                  tr3`s = tr1!1`s1 AND tr3`ac = tr2!1`ac AND
                  tr3`s1 = s3 AND tr4`s = tr2!1`s1 AND
                  tr4`ac = tr1!1`ac AND tr4`s1 = s3))
         OR
           (tr4`s = tr2!1`s1 AND tr4`ac = tr1!1`ac AND
           tr4`s1 = tr1!1`s1)
         OR
           (tr3`s = tr1!1`s1 AND tr3`ac = tr2!1`ac AND
            tr3`s1 = tr2!1`s1)
[2]   tr1!1`s1 = tr2!1`s1
```

We continue by instantiating the existential quantifier. Both transitions `tr3` and `tr4` are defined above. We have only one option for the parameters; there is one existing transition in set `C`, namely `tr1` and one existing transition in τ, namely `tr2`, so we instantiate the first variable in consequent `1` with `tr3(tr1!1, tr2!1,s3(ss,ds)(tr1!1,tr2!1))` and the second variable with `tr4(tr1!1,tr2!1, s3(ss,ds)(tr1!1,tr2!1))`. We obtain three subgoals; these are the goal we wanted to prove after instantiating and two type-correctness conditions. In consequent `1`, we have to instantiate the first parameter with a transition in τ and the second parameter with a transition in set C. This leads to two TCCs; in the first TCC we have to prove that `tr4` is in set `C` and in the second TCC we have to prove that `tr3` is in τ.

The first case does not lead to any difficulties. For the remaining TCCs we define two additional conjectures:

```
tr3InTau: CONJECTURE
  FORALL (tr1:setC_el, tr2:Tau):
    tr1`s = tr2`s AND tr1`s1 /= tr2`s1 =>
      member[Transition](tr3(tr1,tr2,s3(ss,ds)(tr1,tr2)),stsp`tau)

tr4InSetC: CONJECTURE
  FORALL (tr1:setC_el, tr2:Tau):
    tr1`s = tr2`s AND tr1`s1 /= tr2`s1 =>
      member[Transition](tr4(tr1,tr2,s3(ss,ds)(tr1,tr2)),setC)
```

These conjectures assume that transition `tr3` is an element of τ and `tr4` is an element of set `C`, provided that transitions `tr1` and `tr2` have the same starting state and case four of *confluent* does not hold. We complete the subgoals by instantiating these conjectures with transitions `tr1` and `tr2` followed by expanding some definitions.

Now we have to prove these additional conjectures. For `tr3InTau` we have to prove the nine conditions mentioned in definition 3.2.5. We have to prove:

1. $\mathit{successful}(P(t'), tp')$
2. $t' \in T \setminus (tp' \cup \mathit{bypassed}(tp'))$
3. $\forall \mathit{cap} \in I(t').\exists res \in r'.A(res) = \mathit{cap}$
4. $\forall res \in r'.Cb(t', A(res)) \subseteq ms'(res)$
5. $\forall res \in r', m \in M.m \in ms'(res) \Rightarrow m \in ms'''(res) \vee (\exists res' \in Mf(res).m \in ms'''(res'))$
6. $\forall res \in R.\#(ms'''(res)) \leq Rm(res)$

7. $tp''' = tp' \cup \{t'\}$
8. $\forall res \in r'.ms'''(res) = (ms'(res) \setminus Cb(t', A(res))) \cup Ce(t', A(res))$
9. $\forall res \notin r'.ms'''(res) = ms'(res)$

Since set C is a subset of τ, for transitions in set C we have to prove the nine proof obligations for transitions in τ again, but with some other parameters, and four proof obligations which must hold for transitions in set C, see definition 4.1.3. So, for the transition in set C we get the following proof obligations:

1. $successful(P(t), tp'')$
2. $t \in T \setminus (tp'' \cup bypassed(tp''))$
3. $\forall cap \in I(t). \exists res \in r. A(res) = cap$
4. $\forall res \in r. Cb(t, A(res)) \subseteq ms''(res)$
5. $\forall res \in r, m \in M. m \in ms''(res) \Rightarrow m \in ms'''(res) \vee (\exists res' \in Mf(res). m \in ms'''(res'))$
6. $\forall res \in R. \#(ms'''(res)) \leq Rm(res)$
7. $tp''' = tp'' \cup \{t\}$
8. $\forall res \in r. ms'''(res) = (ms''(res) \setminus Cb(t, A(res))) \cup Ce(t, A(res))$
9. $\forall res \notin r. ms'''(res) = ms''(res)$
10. $\forall t'' \in T.\ mat(t'') \cap mat(t) \neq \emptyset \Rightarrow (t'' \cup anc(t'')) \cap allsucc(t) \neq \emptyset$
11. $\forall g \in G, n \in (g \cup anc(g)).\ (anc(t) \cup \{t\}) \cap succ(n, P) = \emptyset$
12. $G \cap anc(t) = \emptyset$
13. $\forall res \in r.\ res \in unsafe \Rightarrow C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset$

We will refer to these proof obligations by tau-1 through tau-9 and setC-1 through setC-13. For all proof obligations, we tried to stick to the proofs described in [2] as much as possible. This is the easiest and fastest way to prove them and we can compare the PVS results with the ones in [2]. Unfortunately, due to lack of time, not all proof obligations are finished.

6.2 Proving proof obligations

In this section we give an overview of successful proofs, corrected proofs and failed proofs. Successful proofs are proof obligations which are correctly proved in [2]; no changes had to be made there. We will discuss some of these proofs in detail in section 6.2.1. Corrected proofs are proof obligations in which one or more steps in the proof in [2] were not true but we were able to correct these steps. These changes are discussed in section 6.2.2. Failed proofs are proof obligations which contain mistakes in [2] but these mistakes are not corrected so far. These proof obligations are discussed in section 6.2.3.

Since transitions `tr1` and `tr2` have the same starting state and case four of *confluent* does not hold for the proof obligations, the PVS sequent has the form:

```
[-1]   tr1!1`s = tr2!1`s
[-n]   << list of antecedents >>
  |-------
[m-1]  << list of consequents >>
[ m ]  tr1!1`s1 = tr2!1`s1
```

We split equivalences `A` $\equiv$ `B` in two subgoals, namely `A` $\Rightarrow$ `B` and `A` $\Leftarrow$ `B`. Discussing all subgoals in detail would be very time consuming, so we discuss only those which are not self-explanatory. For more details we refer to the CD.

6.2.1 Successful proofs

The following proof obligations are correctly proved in [2]: tau-3 and setC-3, tau-7 and setC-7, tau-8, tau-9 and setC-9, setC-10, setC-11, setC-12 and setC-13. We will discuss tau-3, tau-9, setC-12 and setC-13. PVS proofs of all of these proof obligations can be found on the CD.

Proof obligation: tau-3

We have to proof that $\forall cap \in I(t').\exists res \in r'.A(res) = cap$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (cap: C):
        member(cap, ds`i(tr2!1`ac`t)) IMPLIES
         (EXISTS (res: R): member(res, tr2!1`ac`r) AND ss`a(res) = cap)
[2]   tr1!1`s1 = tr2!1`s1
```

We start our proof by hiding antecedent `-1` and consequent `2` and we remove the universal quantifier followed by simplifying the result. This leads to:

```
{-1}  member(cap!1, ds`i(tr2!1`ac`t))
  |-------
{1}   EXISTS (res: R): member(res, tr2!1`ac`r) AND ss`a(res) = cap!1
```

Since transition `tr2` is a valid transition in τ, we know that the nine conditions for τ hold. We make these conditions visible, using the command `(typepred tr2!1)`. Condition 3 is exactly the same as the one we have to prove. After hiding all irrelevant conditions we obtain:

```
[-1]  FORALL (cap: C):
        member(cap, ds`i(tr2!1`ac`t)) IMPLIES
         (EXISTS (res: R): member(res, tr2!1`ac`r) AND ss`a(res) = cap)
[-2]  member(cap!1, ds`i(tr2!1`ac`t))
  |-------
[1]   EXISTS (res: R): member(res, tr2!1`ac`r) AND ss`a(res) = cap!1
```

Finally, we could again remove the universal quantifier and simplify the result. But this is not necessary; PVS completes the proof for us by using `(grind)`. Proving setC-3 is done symmetrically.

Proof obligation: tau-9

This is the last property of τ. We have to prove $\forall res \notin r'.ms'''(res) = ms'(res)$. In PVS this looks as follows:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R):
        (NOT member(res, tr2!1`ac`r)) IMPLIES
         s3(ss, ds)(tr1!1, tr2!1)`ms(res) = tr1!1`s1`ms(res)
[2]   tr1!1`s1 = tr2!1`s1
```

Again, we start to remove the universal quantifier and then we redo the proof in [2]. So we expand the definition of `s3` followed by expanding `ms3`. Then we need property 9 of transition `tr1`, saying $\forall res \notin r.ms'(res) = ms(res)$. We get four cases:

```
[-1]  FORALL (res: R):
        (NOT member(res, tr1!1`ac`r)) IMPLIES
         tr1!1`s1`ms(res) = tr1!1`s`ms(res)
[-2]  tr1!1`s = tr2!1`s
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(res!1, tr2!1`ac`r)
[3]   COND NOT member(res!1, union(tr1!1`ac`r, tr2!1`ac`r)) ->   % 1
             tr1!1`s`ms(res!1),
           member(res!1, difference(tr1!1`ac`r, tr2!1`ac`r)) ->  % 2
             tr1!1`s1`ms(res!1),
           member(res!1, difference(tr2!1`ac`r, tr1!1`ac`r)) ->  % 3
             tr2!1`s1`ms(res!1),
           ELSE ->                                               % 4
             union(difference(union(difference
                                      (tr1!1`s`ms(res!1),
                                       ds`cb(tr1!1`ac`t, ss`a(res!1))),
                                      ds`ce(tr1!1`ac`t, ss`a(res!1))),
                                ds`cb(tr2!1`ac`t, ss`a(res!1))),
                    ds`ce(tr2!1`ac`t, ss`a(res!1)))
      ENDCOND
       = tr1!1`s1`ms(res)
```

In Fig. 6.1 you see the four cases. Cases 3 and 4 are not possible since consequent 2 says that $res \notin r'$. So we only have to prove cases 1 and 2. For case 1 we have to prove $ms(res) = ms'(res)$. This is what antecedent -1, which is property nine of τ, says. For case 2 we have to prove $ms'(res) = ms'(res)$, which is trivial.

Proof obligation: setC-12

We have to prove $G \cap anc(t) = \emptyset$. In PVS this looks as follows:

```
  |-------
[1]   intersection(G,
         anc(ds)(tr4(tr1!1, tr2!1, s3(ss, ds)(tr1!1, tr2!1))`ac`t)
```

This subgoal is a property of set `C`. So by expanding the definition of transition `tr4`, PVS completes this subgoal immediately, without reporting it to us. Therefore one subgoal is 'missing' after expanding this definition.

Proof obligation: setC-13

We have to prove $\forall res \in r.\ res \in unsafe \Rightarrow C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R):
        member(res, unsafe(ss)) IMPLIES
         difference(ds`ce(tr1!1`ac`t, ss`a(res)),
                    ds`cb(tr1!1`ac`t, ss`a(res)))
          = emptyset
[2]   tr1!1`s1 = tr2!1`s1
```

This is exactly the same as property 11 of transition `tr1`. To proof this goal, we remove the universal quantifier and extract this property from `tr1`. We obtain:

```
[-1]  FORALL (g, n: N):
        (G(g) AND member(n,union(singleton[N](g),anc(ds)(g)))) IMPLIES
         intersection(
           union(
             anc(ds)(tr1!1`ac`t), singleton(tr1!1`ac`t)),
           succ(ds)(n, ds`p))
          = emptyset
[-3]  G(g!1)
[-4]  member(n!1, union(singleton[N](g!1), anc(ds)(g!1)))
  |-------
[1]   intersection(union(anc(ds)(tr1!1`ac`t), singleton(tr1!1`ac`t)),
                   succ(ds)(n!1, ds`p))
       = emptyset
```

Now we instantiate antecedent -1 with the only possible values `g!1` and `n!1` and PVS will do the rest by using `(grind)`.

6.2.2 Corrected proof

Proof obligation: setC-2

We have to prove $t \in T \setminus (tp'' \cup bypassed(tp''))$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   member(tr1!1`ac`t,
         difference(T, union(tr2!1`s1`tp, bypassed(ds)(tr2!1`s1`tp))))
[2]   tr1!1`s1 = tr2!1`s1
```

The proof of this proof obligations is in [2] done in six steps. We modelled these steps as lemmas; applying these lemmas after each other completes the goal above. Five steps did not lead to any difficulties; we will therefore refer to the CD for these proofs. The only one which causes difficulties is the third step. Here we have to prove the following lemma; the reason why this should hold is given in braces (according to [2]):

$$\begin{aligned} & t' \notin (tp \cup \{t\} \cup t_b(tp \cup \{t\}))\backslash(tp \cup t_b(tp)) \\ \equiv\ & \{t \neq t', \text{and } tp \cap t_b(tp \cup \{t\}) = \emptyset\} \\ & t' \notin (t_b(tp \cup \{t\}))\backslash(t_b(tp)) \end{aligned}$$

This is modelled in PVS as follows:

```
le_setC13: LEMMA
  FORALL (tr1:setC_el, tr2:Tau):
    tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
      NOT member(tr1`ac`t,
            difference(union(tr1`s`tp, union(singleton(tr2`ac`t),
        bypassed(ds)(union(tr1`s`tp,singleton(tr2`ac`t))))),
        union(tr1`s`tp, bypassed(ds)(tr1`s`tp)))) =
      NOT member(tr1`ac`t, difference(bypassed(ds)(union(tr1`s`tp,
        singleton(tr2`ac`t))), bypassed(ds)(tr1`s`tp)))
```

As you can see, in [2] $\mathtt{t} \neq \mathtt{t}'$ is used to prove this lemma. Since there is no constraint which says that $\mathtt{t} = \mathtt{t}'$ is not allowed, this step deserves a closer look. Let us try to prove the lemma. We distinguish case $\mathtt{t} \neq \mathtt{t}'$ and case $\mathtt{t} = \mathtt{t}'$. The first one can easily be proved, so we move on to case $\mathtt{t} = \mathtt{t}'$. This case is not proved in [2]. Here, we distinguish three more cases: $\mathtt{r} = \mathtt{r}'$, $\mathtt{ms}' = \mathtt{ms}''$ and $\mathtt{r} \neq \mathtt{r}' \wedge \mathtt{ms}' \neq \mathtt{ms}''$. Case $\mathtt{t} = \mathtt{t}' \wedge \mathtt{r} = \mathtt{r}'$ rewrites to $tp' = tp'' \wedge ms' = ms''$ because we now have just one transition: $(\mathtt{t}, \mathtt{r})$. Case

$\mathtt{t} = \mathtt{t}' \wedge \mathtt{ms}' = \mathtt{ms}''$ also leads to $tp' = tp'' \wedge ms' = ms''$ because $ms' = ms''$ is a part of our assumption and $tp' = tp''$ holds by property 7 of τ. This completes both cases since $tp' = tp'' \wedge ms' = ms''$ is equivalent to $s' = s''$, which is case IV of definition 4.1.1.

Finally we have to prove the lemma for case $\mathtt{r} \neq \mathtt{r}' \wedge \mathtt{ms}' \neq \mathtt{ms}''$. After a number of steps we get two symmetrical subgoals. We will discuss one of these:

```
{-1}  tr1!1`s1`ms(x!1) = tr1!1`s`ms(x!1)
[-2]  tr2!1`s1`ms(x!1) =
       union(difference(tr2!1`s`ms(x!1), ds`cb(tr2!1`ac`t, ss`a(x!1))),
             ds`ce(tr2!1`ac`t, ss`a(x!1)))
[-3]  member(x!1, tr2!1`ac`r)
[-4]  tr1!1`s1`tp = tr2!1`s1`tp
[-5]  tr1!1`ac`t = tr2!1`ac`t
[-6]  tr1!1`s = tr2!1`s
  |-------
[1]   member(x!1, tr1!1`ac`r)
[2]   tr1!1`s1`ms(x!1) = tr2!1`s1`ms(x!1)
```

This means we have to prove $ms(x!1) = ms''(x!1)$ since $ms'(x!1) = ms(x!1)$ holds by antecedent -1. Equality $ms(x!1) = ms''(x!1)$ is a consequence of antecedent -2 iff $ms(x!1) = (ms(x!1) \setminus Cb(t', A(x!1))) \cup Ce(t', A(x!1))$. Because property 4 of τ says $\forall res \in r'.Cb(t', A(res)) \subseteq ms(res)$, there exist only one option namely $Cb(t', A(x!1)) = Ce(t', A(x!1))$. If this does not hold, consequent 2 is false so that consequent 2, $x!1 \in r$, must hold. To achieve this, we need a new constraint on set C: $\forall((tp, ms), (t', r'), (tp'', ms'')) \in Tau, res \in R : Cb(t, A(res)) \neq Ce(t, A(res)) \Rightarrow res \in r$. For the symmetrical subgoal we need $\forall((tp, ms), (t', r'), (tp'', ms'')) \in Tau, res \in R : Cb(t, A(res)) \neq Ce(t, A(res)) \Rightarrow res \in r'$; redo the proof on the CD for more details. We combine both constraints and we obtain the fifth constraint on set C: $\forall((tp, ms), (t', r'), (tp'', ms'')) \in Tau, res \in R : Cb(t, A(res)) \neq Ce(t, A(res)) \Rightarrow res \in (r \cap r')$. The new definition of set C is given below and replaces definition 4.1.3. This change does not affect previously proved proof obligations.

Definition 6.2.1 (Confluent transition set C) *Let $\Sigma = (R, C, A, R_m, M_f)$ be a static system definition and $\Delta = (T, G, L, L_n, G_n, G_a, I, P, M, C_b, C_e, \hat{m}_s)$ be a dynamic system definition. Let $(St, \hat{s}, Ac, \tau)$ be the state-space of the system. We define $\mathbf{C} \subseteq \tau$ to be the set consisting of the transitions $((tp, ms), (t, r), (tp', ms'))$ for which the following conditions hold:*

1. $\forall t'' \in T.\ mat(t'') \cap mat(t) \neq \emptyset \Rightarrow (t'' \cup anc(t'')) \cap allsucc(t) \neq \emptyset$;
2. $\forall res \in r.\ res \in unsafe \Rightarrow C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset$;
3. $\forall g \in G, n \in (g \cup anc(g)).\ (anc(t) \cup \{t\}) \cap succ(n, P) = \emptyset$;
4. $G \cap anc(t) = \emptyset$.
5. $\forall((tp, ms), (t', r'), (tp'', ms'')) \in Tau, res \in R : Cb(t, A(res)) \neq Ce(t, A(res)) \Rightarrow res \in (r \cap r')$

The implementation of the new set C in PVS is given below:

```
setC_el: TYPE = {tr: Tau |
  (FORALL (t2:N):                                                  % 1
    (T(t2) AND intersection(mat(ds)(t2), mat(ds)(tr`ac`t))
      /= emptyset) IMPLIES
    intersection(union(t2,anc(ds)(t2)),allsucc(ds)(tr`ac`t))
      /= emptyset) AND
  (FORALL (g,n:N): (G(g) AND member(n,union(g,anc(ds)(g)))) IMPLIES % 2
    intersection(union(anc(ds)(tr`ac`t),singleton(tr`ac`t)),
      succ(ds)(n,ds`p)) = emptyset) AND
```

```
  (intersection(G,anc(ds)(tr`ac`t)) = emptyset) AND                  % 3
  (FORALL (res:R): member(res,unsafe(ss)) IMPLIES                    % 4
    difference(ds`ce(tr`ac`t,ss`a(res)),ds`cb(tr`ac`t,ss`a(res)))
      = emptyset) AND
  (FORALL (tr1: Tau, res: R):                                        % 5
    ds`cb(tr1`ac`t, ss`a(res)) /= ds`ce(tr1`ac`t, ss`a(res)) IMPLIES
      member(res, tr`ac`r) AND member(res, tr1`ac`r))}

setC: setof[setC_el] = fullset[setC_el]
```

Now we use the extra property in our proof. We get the following sequent:

```
{-1}  ds`cb(tr2!1`ac`t, ss`a(x!1)) /= ds`ce(tr2!1`ac`t, ss`a(x!1)) =>
       member(x!1, tr1!1`ac`r) AND member(x!1, tr2!1`ac`r)
[-2]  tr1!1`s1`ms(x!1) = tr1!1`s`ms(x!1)
[-3]  tr2!1`s1`ms(x!1) =
       union(difference(tr2!1`s`ms(x!1), ds`cb(tr2!1`ac`t, ss`a(x!1))),
             ds`ce(tr2!1`ac`t, ss`a(x!1)))
[-4]  member(x!1, tr2!1`ac`r)
[-5]  tr1!1`s1`tp = tr2!1`s1`tp
[-6]  tr1!1`ac`t = tr2!1`ac`t
[-7]  tr1!1`s = tr2!1`s
  |-------
[1]   member(x!1, tr1!1`ac`r)
[2]   tr1!1`s1`ms(x!1) = tr2!1`s1`ms(x!1)
```

We split this goal into two subgoals. For the first one we have to prove

```
[-1]  member(x!1, tr1!1`ac`r)
[-2]  member(x!1, tr2!1`ac`r)
  |-------
[1]   member(x!1, tr1!1`ac`r)
```

which is trivial. The second subgoal looks as follows:

```
{-1}  ds`cb(tr2!1`ac`t, ss`a(x!1)) = ds`ce(tr2!1`ac`t, ss`a(x!1))
[-2]  tr1!1`s1`ms(x!1) = tr1!1`s`ms(x!1)
[-3]  tr2!1`s1`ms(x!1) =
       union(difference(tr2!1`s`ms(x!1), ds`cb(tr2!1`ac`t, ss`a(x!1))),
             ds`ce(tr2!1`ac`t, ss`a(x!1)))
[-4]  member(x!1, tr2!1`ac`r)
[-5]  tr1!1`s1`tp = tr2!1`s1`tp
[-6]  tr1!1`ac`t = tr2!1`ac`t
[-7]  tr1!1`s = tr2!1`s
  |-------
[1]   member(x!1, tr1!1`ac`r)
[2]   tr1!1`s1`ms(x!1) = tr2!1`s1`ms(x!1)
```

Rewriting this sequent, using the equalities in the antecedent and hiding the formulas we are not interested in, results in:

```
[-1]  tr1!1`s1`ms(x!1) = tr1!1`s`ms(x!1)
[-2]  tr2!1`s1`ms(x!1) = tr2!1`s`ms(x!1)
[-3]  tr1!1`s = tr2!1`s
  |-------
[1]   tr1!1`s1`ms(x!1) = tr2!1`s1`ms(x!1)
```

which is also true.

Finally we have to prove our additional property for set C. The proof obligation, setC-14, looks as follows:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (tr1: Tau, res: R):
        ds`cb(tr1`ac`t, ss`a(res)) /= ds`ce(tr1`ac`t, ss`a(res)) =>
         member(res, tr1!1`ac`r) AND member(res, tr1`ac`r)
[2]   tr1!1`s1 = tr2!1`s1
```

This goal can be proved in a similar way as setC-10 through setC-12. The proof can be found on the CD. So, the additional property of set C holds and therefore we are allowed to use it to prove other proof obligations, as we did for setC-2.

Proof obligation tau-2 is done in a symmetrical way, provided that $\forall g \in G : succnil(g) \cap (tp \cup \{t\}) = succnil(g) \cap tp$ and $\forall g \in G : G_n(g) \cap (n_i(tp \cup \{t\}) \cup n_s(tp \cup \{t\})) = G_n(g) \cap (n_i(tp) \cup n_s(tp))$ hold; see section 6.3.

6.2.3 Failed proofs

Proof obligation: tau-4

We have to prove $\forall res \in r'.Cb(t', A(res)) \subseteq ms'(res)$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R):
        member(res, tr2!1`ac`r) IMPLIES
         subset?(ds`cb(tr2!1`ac`t, ss`a(res)), tr1!1`s1`ms(res))
[2]   tr1!1`s1 = tr2!1`s1
```

We follow the seven steps in [2] to prove this. For the sake of clarity, we prove these seven steps separately and use these parts to complete the proof. Five of these steps cause no difficulties; we will only discuss the other two steps.

The first one of these steps is the third step of the proof. It looks as follows (see [2]):

$$
\begin{aligned}
& (\forall res \in r' \cap r : C_b(t', A(res)) \subseteq \\
& \quad ms(res) \backslash C_b(t, A(res)) \cup C_e(t, A(res))) \\
\equiv \quad & \{calculus\} \\
& (\forall res \in r' \cap r : C_b(t', A(res)) \cap (C_b(t, A(res)) \backslash C_e(t, A(res))) = \emptyset)
\end{aligned}
$$

This step holds, but the given hint is not sufficient. We need property 4 of τ, $\forall res{\in}r.\ C_b(t, A(res)) \subseteq ms(res)$. After retrieving this property, PVS completes the proof easily using `(grind)`.

The second one leads to more problems; this is the sixth step in [2]. The mathematical and PVS representations are shown below, with Lemma 6.2.1:

Lemma 6.2.1 *When there exists tasks t' and t, such that t' is in $allsucc(t)$ and t and t' are both enabled in a state (tp, ms) then there must be an ancestor of t which is a group.*

$$\begin{array}{ll} & \neg((t' \cup anc(t')) \cap allsucc(t) \neq \emptyset) \\ \Leftarrow & \{\text{Lemma 6.2.1}\} \\ & anc(t) \cap G = \emptyset \end{array}$$

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s IMPLIES
         intersection(G, anc(ds)(tr1`ac`t)) = emptyset =>
          intersection(union(singleton[N](tr2`ac`t),anc(ds)(tr2`ac`t)),
                       allsucc(ds)(tr1`ac`t))
           = emptyset
```

We remove the universal quantifier and we introduce the lemma mentioned in the hint:

```
{-1}  FORALL (ac, ac1: Action):
        member(ac1`t, allsucc(ds)(ac`t)) AND
         (EXISTS (st: State):
            member(ac, enabled(st)) AND member(ac1, enabled(st)))
         IMPLIES intersection(G, anc(ds)(ac`t)) /= emptyset
[-2]  tr1!1`s = tr2!1`s
[-3]  intersection(G, anc(ds)(tr1!1`ac`t)) = emptyset
  |-------
[1]   intersection(union(singleton[N](tr2!1`ac`t),anc(ds)(tr2!1`ac`t)),
                   allsucc(ds)(tr1!1`ac`t))
       = emptyset
```

Antecedent `-1` can only be instantiated in one way: we take `tr1!1`ac` for `ac` and `tr2!1`ac` for `ac1`. Another option is not possible because of the existence of antecedent `-3`. Splitting the result leads to three subgoals; for two of these subgoals we refer to the CD, both subgoals are proved. Now we take a look at the only remaining subgoal:

```
[-1]  tr1!1`s = tr2!1`s
[-2]  intersection(G, anc(ds)(tr1!1`ac`t)) = emptyset
  |-------
{1}   member(tr2!1`ac`t, allsucc(ds)(tr1!1`ac`t))
[2]   intersection(union(singleton[N](tr2!1`ac`t),anc(ds)(tr2!1`ac`t)),
                   allsucc(ds)(tr1!1`ac`t))
       = emptyset
```

Let us translate this into math:

$$(G \cap anc(t)) = \emptyset \Rightarrow t' \in allsucc(t) \vee (t' \cup anc(t')) \cap allsucc(t) = \emptyset$$

We prove that this subgoal does not hold by giving a counter example. Take a look at the picture below: Task `t` is not contained in a group, so either $t' \in allsucc(t)$ or $(t' \cup anc(t')) \cap allsucc(t) = \emptyset$ must hold. The only successors of task `t` are group `g` and task $\mathtt{t}'''$ and thus $t' \notin allsucc(t)$. The ancestor of both task $\mathtt{t}'$ and task $\mathtt{t}''$ is group `g`. For $(t' \cup anc(t')) \cap allsucc(t) = \emptyset$ we get $(t' \cup g) \cap \{g, t'''\} = \emptyset$ which is obviously false.

This means that the proof of the sixth step of proof obligation tau-4 does not hold. But this does not mean that this step does not hold at all. Maybe there exists another way to prove this step or maybe there exists a complete other way to prove tau-4. The same holds for setC-4.

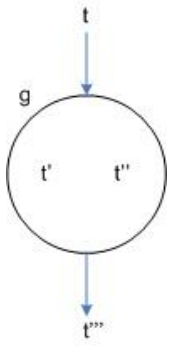


Figure 6.2: Counterexample tau-4.6

Proof obligation: tau-5

We have to prove $\forall res \in r', m \in M.m \in ms'(res) \Rightarrow m \in ms'''(res) \vee (\exists res' \in Mf(res).m \in ms'''(res'))$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R), (m: M):
        member(res, tr2!1`ac`r) AND member(m, tr1!1`s1`ms(res)) IMPLIES
         member(m, s3(ss, ds)(tr1!1, tr2!1)`ms(res)) OR
          (EXISTS (res1: R):
             member(res1, ss`mf(res)) AND
              member(m, s3(ss, ds)(tr1!1, tr2!1)`ms(res1)))
[2]   tr1!1`s1 = tr2!1`s1
```

In [2], the proof is done in eight steps. Some of these steps caused a lot of trouble; let us take a look at the second step:

$$
\begin{aligned}
& (\forall res \in r' \backslash r, m \in \mathcal{M} : m \in ms(res) \Rightarrow m \in ms'''(res) \\
& \quad \vee (\exists res' \in M_f(res) : m \in ms'''(res'))) \\
& \wedge (\forall res \in r' \cap r, m \in \mathcal{M} : m \in ms'(res) \Rightarrow m \in ms'''(res) \\
& \quad \vee (\exists res' \in M_f(res) : m \in ms'''(res'))) \\
\equiv \quad & \{(\forall res \in r' \backslash r : ms'''(res) = ms''(res))\} \\
& (\forall res \in r' \backslash r, m \in \mathcal{M} : m \in ms(res) \Rightarrow m \in ms''(res) \\
& \quad \vee (\exists res' \in M_f(res) : m \in ms''(res'))) \\
& \wedge (\forall res \in r' \cap r, m \in \mathcal{M} : m \in ms'(res) \Rightarrow m \in ms'''(res) \\
& \quad \vee (\exists res' \in M_f(res) : m \in ms'''(res')))
\end{aligned}
$$

Initially the proof looks very convincing, but we were not able to prove this with the help of PVS. Taking a closer look shows why. The second conjunct is no problem; on both sides of the equality we have the same formula, so this holds. The first conjunct however is not provable by applying the hint. Rewriting $\exists res' \in M_f(res) : m \in ms'''(res')$ into $\exists res' \in M_f(res) : m \in ms''(res')$ is only allowed for those res' which are contained in `r'` and not in `r`. So the given proof is incomplete; this could possibly be solved by using case distinction and add the missing cases. The fourth step contains a similar problem, but now with the intersection of `r` and `r'` instead of the difference.

Apart from this incompleteness, the seventh step of the proof is not clear. This step looks as follows in [2]:

$$
\begin{array}{ll}
 & (\forall res \in r' \cap r, m \in (C_b(t, A(res)) \cup C_e(t, A(res))) : \\
 & \quad m \in ms'(res) \Rightarrow m \in ms'(res) \vee (\exists res' \in M_f(res) : m \in ms'(res'))) \\
 & \wedge (\forall res \in r' \cap r, m \in (C_b(t', A(res)) \cup C_e(t', A(res))) : \\
 & \quad m \in ms'(res) \Rightarrow m \in ms''(res) \vee (\exists res' \in M_f(res) : m \in ms''(res'))) \\
\equiv & \{(\forall res \in R, m \notin (C_b(t, A(res)) \cup C_e(t, A(res))) : ms'(res) = ms(res))\} \\
 & (\forall res \in r' \cap r, m \in (C_b(t', A(res)) \cup C_e(t', A(res))) : \\
 & \quad m \in ms(res) \Rightarrow m \in ms''(res) \vee (\exists res' \in M_f(res) : m \in ms''(res')))
\end{array}
$$

We split this step in two smaller ones and omit the implication the other way around. We get

$$
\begin{array}{ll}
 & (\forall res \in r' \cap r, m \in (C_b(t, A(res)) \cup C_e(t, A(res))) : \\
 & \quad m \in ms'(res) \Rightarrow m \in ms'(res) \vee (\exists res' \in M_f(res) : m \in ms'(res'))) \\
\Leftarrow & \{calculus\} \\
 & (\forall res \in r' \cap r, m \in (C_b(t', A(res)) \cup C_e(t', A(res))) : \\
 & \quad m \in ms(res) \Rightarrow m \in ms''(res) \vee (\exists res' \in M_f(res) : m \in ms''(res')))
\end{array}
$$

and

$$
\begin{array}{ll}
 & (\forall res \in r' \cap r, m \in (C_b(t', A(res)) \cup C_e(t', A(res))) : \\
 & \quad m \in ms'(res) \Rightarrow m \in ms''(res) \vee (\exists res' \in M_f(res) : m \in ms''(res'))) \\
\Leftarrow & \{(\forall res \in R, m \notin (C_b(t, A(res)) \cup C_e(t, A(res))) : ms'(res) = ms(res))\} \\
 & (\forall res \in r' \cap r, m \in (C_b(t', A(res)) \cup C_e(t', A(res))) : \\
 & \quad m \in ms(res) \Rightarrow m \in ms''(res) \vee (\exists res' \in M_f(res) : m \in ms''(res')))
\end{array}
$$

The first one does not lead to any problems. For the second one, we need the hint given in braces. Even while using the hint we are not able to solve this goal because the hint says something about $C_b(t, A(res)) \cup C_e(t, A(res))$ and antecedent and consequent are about $C_b(t', A(res)) \cup C_e(t', A(res))$. We can complete the proof with the following lemma:

$$
\begin{array}{l}
\forall (tr1 : setC_el, tr2 : Tau, res : R, m : M) : \\
\quad m \in C_b(t', a(res)) \cup C_e(t', a(res)) \Rightarrow m \notin C_b(t, a(res)) \cup C_e(t, ss`a(res)
\end{array}
$$

Finally we have to show that both the additional lemma and the given hint hold. For the lemma we have to prove:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau, res: R, m: M):
        tr1`s = tr2`s AND
         member(m,
                union(ds`cb(tr2`ac`t, ss`a(res)),
                      ds`ce(tr2`ac`t, ss`a(res))))
         IMPLIES
         NOT member(m,
                    union(ds`cb(tr1`ac`t, ss`a(res)),
                          ds`ce(tr1`ac`t, ss`a(res))))
```

After a number of steps, we have one subgoal left:

```
{-1}  anc(ds)(tr1!1`ac`t)(x!1)
{-2}  allsucc(ds)(tr1!1`ac`t)(x!1)
[-3]  tr1!1`s = tr2!1`s
[-4]  member(m!1,
             union(ds`cb(tr2!1`ac`t, ss`a(res!1)),
                   ds`ce(tr2!1`ac`t, ss`a(res!1))))
```

```
[-5]  member(m!1,
             union(ds`cb(tr1!1`ac`t, ss`a(res!1)),
                   ds`ce(tr1!1`ac`t, ss`a(res!1))))
  |-------
[1]   allsucc(ds)(tr1!1`ac`t)(tr1!1`ac`t)
```

This is true iff `tr1!1`ac`t` = `x!1`. Since we don't know anything from `x!1`, we cannot conclude that this is indeed true.

For the hint we have a similar problem. We have to prove:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau, res: R, m: M):
        tr1`s = tr2`s AND
         member(res, intersection(tr1`ac`r, tr2`ac`r)) AND
          NOT member(m,
                     union(ds`cb(tr1`ac`t, ss`a(res)),
                           ds`ce(tr1`ac`t, ss`a(res))))
         IMPLIES tr1`s1`ms(res) = tr1`s`ms(res)
```

This sequent should hold by property 1 of set `C`, which says $\forall t'' \in T.\ mat(t'') \cap mat(t) \neq \emptyset \Rightarrow (t'' \cup anc(t'')) \cap allsucc(t) \neq \emptyset$. Making this information explicit results in:

```
[-1]  FORALL (res: R):
        member(res, tr1!1`ac`r) IMPLIES
         tr1!1`s1`ms(res) =
          union(difference(tr1!1`s`ms(res),
                           ds`cb(tr1!1`ac`t, ss`a(res))),
                ds`ce(tr1!1`ac`t, ss`a(res)))
[-2]  tr1!1`s = tr2!1`s
[-3]  member(res!1, intersection(tr1!1`ac`r, tr2!1`ac`r))
  |-------
[1]   member(m!1,
             union(ds`cb(tr1!1`ac`t, ss`a(res!1)),
                   ds`ce(tr1!1`ac`t, ss`a(res!1))))
[2]   tr1!1`s1`ms(res!1) = tr1!1`s`ms(res!1)
```

After using a number of rules we get the following sequent. This is the only remaining subgoal which has to be proved to complete the proof of the hint.

```
[-1]  tr1!1`s1`ms(res!1)(m!1) =
       union(difference(tr1!1`s`ms(res!1),
                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
             ds`ce(tr1!1`ac`t, ss`a(res!1)))
            (m!1)
[-2]  tr1!1`s = tr2!1`s
[-3]  member(res!1, intersection(tr1!1`ac`r, tr2!1`ac`r))
  |-------
{1}   ds`cb(tr1!1`ac`t, ss`a(res!1))(m!1)
{2}   ds`ce(tr1!1`ac`t, ss`a(res!1))(m!1)
[3]   tr1!1`s1`ms(res!1) = tr1!1`s`ms(res!1)
```

We decompose equality `-3`, which results in $\forall(m : M) : m \in ms'(res) = m \in ms(res)$.

```
[-1]  tr1!1`s1`ms(res!1)(m!1) =
       union(difference(tr1!1`s`ms(res!1),
                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
             ds`ce(tr1!1`ac`t, ss`a(res!1)))
            (m!1)
[-2]  tr1!1`s = tr2!1`s
[-3]  member(res!1, intersection(tr1!1`ac`r, tr2!1`ac`r))
  |-------
{1}   tr1!1`s1`ms(res!1)(x!1) = tr1!1`s`ms(res!1)(x!1)
[2]   ds`cb(tr1!1`ac`t, ss`a(res!1))(m!1)
[3]   ds`ce(tr1!1`ac`t, ss`a(res!1))(m!1)
```

Now we see the same problem as we saw for the additional lemma. The sequent holds for $\texttt{x!1} = \texttt{m!1}$, but not necessarily for $\texttt{x!1} \neq \texttt{m!1}$. Because this is not the only problematic step in the proof of tau-5, we decided to take a look at an alternative proof which is not proved in [2]. This alternative proof was given on the fly by N.J.M. van den Nieuwelaar and M.M.H. Driessen during a discussion about the problems described above.

$$\begin{array}{ll} & \forall res \in r', m \in M.m \in ms'(res) \Rightarrow m \in ms'''(res) \vee (\exists res' \in Mf(res).m \in ms'''(res')) \\ \Leftarrow & \{(1)\ and\ (2)\} \\ & true \end{array}$$

(1)

$$\begin{array}{ll} & \forall res \in R, m \in M.m \in ms'''(res) \Rightarrow m \in ms''(res) \vee m \in ms'(res) \\ \Leftarrow & \{property\ 8\ and\ 9\ of\ \tau\} \\ & \forall res \in R, m \in M.m \in mat(t') \vee (m \in mat(t) \wedge m \notin (t \cup t')) \\ \Leftarrow & \{property\ 1\ of\ set\ \texttt{C}\} \\ & true \end{array}$$

(2)

$$\begin{array}{ll} & \forall res \in r', m \in M.m \in ms'(res) \wedge m \notin ms(res) \Rightarrow (\exists res' \in Mf(res).m \in ms'(res')) \\ \Leftarrow & \{property\ 5\ of\ \tau\} \\ & true \end{array}$$

In the last step of (1) we have to prove:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (m: M):
            (member(m, mat(ds)(tr2`ac`t)) OR
              member(m, mat(ds)(tr1`ac`t)) AND
               NOT member(m,
                          intersection(mat(ds)(tr2`ac`t),
                                       mat(ds)(tr1`ac`t))))
               /= emptyset)
```

This is a consequence of the first property of set C which says $\forall t'' \in T.\ mat(t'') \cap mat(t) \neq \emptyset \Rightarrow (t'' \cup anc(t'')) \cap allsucc(t) \neq \emptyset$. Removing the universal quantifiers results in:

```
[-1]  tr1!1`s = tr2!1`s
{-2}  (T(tr2!1`ac`t) AND
        intersection(mat(ds)(tr2!1`ac`t),
                     mat(ds)(tr1!1`ac`t)) /= emptyset)
       IMPLIES
        intersection(union(singleton[N](tr2!1`ac`t),
                           anc(ds)(tr2!1`ac`t)),
                     allsucc(ds)(tr1!1`ac`t))
        /= emptyset
  |-------
[1]   member(m!1, mat(ds)(tr2!1`ac`t))
[2]   member(m!1, mat(ds)(tr1!1`ac`t)) AND
       NOT member(m!1,
              intersection(mat(ds)(tr2!1`ac`t), mat(ds)(tr1!1`ac`t)))
[3]   tr1!1`s1 = tr2!1`s1
```

Splitting the antecedent leads to three subgoals; $\mathtt{t}'$ is an element of the set of tasks, so this does not lead to any problems. The other two subgoals do. For both

```
{-1}  intersection(union(singleton[N](tr2!1`ac`t),
                         anc(ds)(tr2!1`ac`t)),
                   allsucc(ds)(tr1!1`ac`t))
       /= emptyset
  |-------
[1]   member(m!1, mat(ds)(tr2!1`ac`t))
[2]   member(m!1, mat(ds)(tr1!1`ac`t)) AND
       NOT member(m!1,
             intersection(mat(ds)(tr2!1`ac`t), mat(ds)(tr1!1`ac`t)))
```

and

```
  |-------
{1}   intersection(mat(ds)(tr2!1`ac`t),
                   mat(ds)(tr1!1`ac`t)) /= emptyset
[2]   member(m!1, mat(ds)(tr2!1`ac`t))
[3]   member(m!1, mat(ds)(tr1!1`ac`t)) AND
       NOT member(m!1,
              intersection(mat(ds)(tr2!1`ac`t), mat(ds)(tr1!1`ac`t
```

we need more information to complete the proofs. So we change the last step of proof of (1) in:

$$\begin{array}{ll} & \forall res \in R, m \in M.(m \in mat(t') \vee (m \in mat(t) \wedge m \notin (t \cup t'))) \vee \\ & \quad (m \notin mat(t) \wedge m \notin mat(t')) \vee (\{t'\} \cup anc(t')) \cap allsucc(t) \neq \emptyset \\ \Leftarrow & \{property\ 1\ of\ set\ \mathtt{C}\} \\ & true \end{array}$$

Now we complete the proof without any difficulties. This change has no impact on the other steps of the proof. In PVS, the second step of (1) looks as follows:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (m: M, res: R):
            member(res, tr2`ac`r) AND
             member(m, s3(ss, ds)(tr1, tr2)`ms(res))
```

```
             IMPLIES member(m, tr2`s1`ms(res)) OR
                     member(m, tr1`s1`ms(res)))
```

We use the first step and we expand the definition of `s3`. Now we only have to make properties 8 and 9 of τ explicit; we use the mentioned properties of both transition `tr1` and transition `tr2`, so we get the following sequent which can be completed by the command `(grind)`:

```
[-1]  member(res!1, tr1!1`ac`r) IMPLIES
       tr1!1`s1`ms(res!1) =
        union(difference(tr1!1`s`ms(res!1),
                         ds`cb(tr1!1`ac`t, ss`a(res!1))),
              ds`ce(tr1!1`ac`t, ss`a(res!1)))
{-2}  (NOT member(res!1, tr1!1`ac`r)) IMPLIES
       tr1!1`s1`ms(res!1) = tr1!1`s`ms(res!1)
[-3]  member(res!1, tr2!1`ac`r) IMPLIES
       tr2!1`s1`ms(res!1) =
        union(difference(tr2!1`s`ms(res!1),
                         ds`cb(tr2!1`ac`t, ss`a(res!1))),
              ds`ce(tr2!1`ac`t, ss`a(res!1)))
[-4]  (NOT member(res!1, tr2!1`ac`r)) IMPLIES
       tr2!1`s1`ms(res!1) = tr2!1`s`ms(res!1)
[-5]  tr1!1`s = tr2!1`s AND tr1!1`s1 /= tr2!1`s1 IMPLIES
       (FORALL (m: M):
          (member(m, mat(ds)(tr2!1`ac`t)) OR
            member(m, mat(ds)(tr1!1`ac`t)) AND
             NOT member(m,
                        intersection(mat(ds)(tr2!1`ac`t),
                                     mat(ds)(tr1!1`ac`t))))
           OR
           (NOT member(m, mat(ds)(tr2!1`ac`t)) AND
             NOT member(m, mat(ds)(tr1!1`ac`t)))
            OR
            intersection(union(singleton[N](tr2!1`ac`t),
                               anc(ds)(tr2!1`ac`t)),
                         allsucc(ds)(tr1!1`ac`t))
             /= emptyset)
[-6]  tr1!1`s = tr2!1`s
[-7]  member(res!1, tr2!1`ac`r)
[-8]  member(m!1, ms3(ss, ds)(tr1!1, tr2!1)(res!1))
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(m!1, tr2!1`s1`ms(res!1))
[3]   member(m!1, tr1!1`s1`ms(res!1))
```

Part (2) of the proof looks as follows:

```
  |-------
{1}   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (m: M, res: R):
            member(m, tr1`s1`ms(res)) AND
               NOT member(m, tr1`s`ms(res))
             IMPLIES
```

```
            (EXISTS (res1: R):
               member(res1, ss`mf(res)) AND
                  member(m, tr1`s1`ms(res1))))
```

We remove the universal quantifier and rewrite the implication to a disjunction of terms, resulting in
$\forall res \in r', m \in M.m \notin ms''(res) \vee m \in ms(res) \vee (\exists res' \in Mf(res).m \in ms(res'))$

```
{-1}  tr1!1`s = tr2!1`s
{-2}  member(m!1, tr1!1`s1`ms(res!1))
  |-------
{1}   tr1!1`s1 = tr2!1`s1
{2}   member(m!1, tr1!1`s`ms(res!1))
{3}   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND member(m!1, tr1!1`s1`ms(res1))
```

This should be a consequence of property 5 of τ, so we make these properties explicit both for transition `tr1` and transition `tr2`. We obtain:

```
[-1]  FORALL (res: R), (m: M):
        member(res, tr1!1`ac`r) AND member(m, tr1!1`s`ms(res)) IMPLIES
         member(m, tr1!1`s1`ms(res)) OR
           (EXISTS (res1: R):
              member(res1, ss`mf(res)) AND member(m, tr1!1`s1`ms(res1)))
[-2]  FORALL (res: R), (m: M):
        member(res, tr2!1`ac`r) AND member(m, tr2!1`s`ms(res)) IMPLIES
         member(m, tr2!1`s1`ms(res)) OR
           (EXISTS (res1: R):
              member(res1, ss`mf(res)) AND member(m, tr2!1`s1`ms(res1)))
[-3]  tr1!1`s = tr2!1`s
[-4]  member(res!1, tr2!1`ac`r)
[-5]  member(m!1, tr1!1`s1`ms(res!1))
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(m!1, tr1!1`s`ms(res!1))
[3]   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND member(m!1, tr1!1`s1`ms(res1))
```

Removing the universal quantifiers again, followed by splitting the implications into smaller subgoals results in 13 subgoals. Four of these are trivial, for the remaining nine subgoals we need either $\texttt{ms}(\texttt{res}) = \texttt{ms}'(\texttt{res})$ or $\texttt{ms}(\texttt{res}) = \texttt{ms}''(\texttt{res})$. Both equalities need not be true, so we cannot prove (2) in this way.

Also the original goal causes some problems. We have to prove:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1
        IMPLIES
         (FORALL (res: R), (m: M):
            member(res, tr2`ac`r) AND member(m, tr1`s1`ms(res))
            IMPLIES
             member(m, s3(ss, ds)(tr1, tr2)`ms(res)) OR
               (EXISTS (res1: R):
                  member(res1, ss`mf(res)) AND
                   member(m, s3(ss, ds)(tr1, tr2)`ms(res1))))
```

We will need (1) and (2) to prove this goal, so we add these lemmas to the sequent. This results in:

```
{-1}  FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (m: M, res: R):
            member(res, tr2`ac`r) AND
             member(m, tr1`s1`ms(res)) AND
                NOT member(m, tr1`s`ms(res))
             IMPLIES
             (EXISTS (res1: R):
                member(res1, ss`mf(res)) AND
                   member(m, tr1`s1`ms(res1))))
[-2]  FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (m: M, res: R):
            member(res, tr2`ac`r) AND
             member(m, s3(ss, ds)(tr1, tr2)`ms(res))
             IMPLIES member(m, tr2`s1`ms(res)) OR
                     member(m, tr1`s1`ms(res)))
[-3]  tr1!1`s = tr2!1`s
[-4]  member(res!1, tr2!1`ac`r)
[-5]  member(m!1, tr1!1`s1`ms(res!1))
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res!1))
[3]   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND
         member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res1))
```

Now we instantiate antecedent -1, which represents lemma (2). We take values `tr1!1` and `tr2!1` for the outer universal quantifier, followed by an instantiation of the inner universal quantifier with values `m!1` and `res!1`. This are the only possible values. We obtain four subgoals; the first one is shown below:

```
{-1}  member(m!1, tr2!1`s1`ms(res!1))
[-2]  FORALL (m: M, res: R):
        member(res, tr2!1`ac`r) AND
         member(m, tr1!1`s1`ms(res)) AND NOT member(m, tr1!1`s`ms(res))
         IMPLIES
         (EXISTS (res1: R):
            member(res1, ss`mf(res)) AND member(m, tr1!1`s1`ms(res1)))
[-3]  tr1!1`s = tr2!1`s
[-4]  member(res!1, tr2!1`ac`r)
[-5]  member(m!1, tr1!1`s1`ms(res!1))
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res!1))
[3]   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND
         member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res1))
```

This goal can be proved by using property 8 of τ for both transition `tr1` and `tr2`. The proof can be found on the CD. The third subgoal is trivial; $\texttt{res} \in \texttt{r}' \Rightarrow \texttt{res} \in \texttt{r}'$ is indeed true. The second and fourth subgoal are equal:

```
[-1]  FORALL (m: M, res: R):
        member(res, tr2!1`ac`r) AND
```

```
          member(m, tr1!1`s1`ms(res)) AND NOT member(m, tr1!1`s`ms(res))
          IMPLIES
          (EXISTS (res1: R):
             member(res1, ss`mf(res)) AND member(m, tr1!1`s1`ms(res1)))
[-2]  tr1!1`s = tr2!1`s
[-3]  member(res!1, tr2!1`ac`r)
[-4]  member(m!1, tr1!1`s1`ms(res!1))
  |-------
{1}   member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res!1))
[2]   tr1!1`s1 = tr2!1`s1
[3]   member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res!1))
[4]   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND
         member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res1))
```

We instantiate antecedent -1 with `m!1` and `res!1` and split the obtained goal. This results in four subgoals. Two of these are trivial; we discuss one of the others:

```
{-1}  EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND member(m!1, tr1!1`s1`ms(res1))
[-2]  member(m!1, tr1!1`s1`ms(res!1))
[-3]  tr1!1`s = tr2!1`s
[-4]  member(res!1, tr2!1`ac`r)
[-5]  member(m!1, tr1!1`s1`ms(res!1))
  |-------
[1]   tr1!1`s1 = tr2!1`s1
[2]   member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res!1))
[3]   EXISTS (res1: R):
        member(res1, ss`mf(res!1)) AND
         member(m!1, s3(ss, ds)(tr1!1, tr2!1)`ms(res1))
```

To prove this subgoal we have to show $\texttt{ms}'''(\texttt{res}) = \texttt{ms}'(\texttt{res})$. Let us take a look at the case $\texttt{res} \notin \texttt{r}$. Case 3 in figure 6.1 says $\texttt{ms}'''(\texttt{res}) = \texttt{ms}''(\texttt{res})$ and property 9 of τ says $\texttt{ms}'(\texttt{res}) = \texttt{ms}(\texttt{res})$. Both properties cannot be used to prove this goal.

This means that also the alternative solution for tau-5 is not proved so far. Proving setC-5 leads to similar problems.

Proof obligation: setC-6

Proof obligations tau-6 and setC-6 are exactly the same. We have to prove $\forall res \in R.\#(ms'''(res)) \leq Rm(res)$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R):
         card(s3(ss, ds)(tr1!1, tr2!1)`ms(res)) <= ss`rm(res)
[2]   tr1!1`s1 = tr2!1`s1
```

The proof consists of nine steps; seven of these do not lead to any difficulties and can be found on the CD. The first step we will discuss is the second step of the proof:

$$
\begin{array}{ll}
 & (\forall res \notin (r \cup r') : |ms(res)| \leq R_m) \\
 & \quad \wedge (\forall res \in (r \cup r') : |ms'''(res)| \leq R_m) \\
\equiv & \{(\forall res \in R : |ms(res)| \leq R_m)\} \\
 & (\forall res \in (r \cup r') : |ms'''(res)| \leq R_m)
\end{array}
$$

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s IMPLIES
         (FORALL (res: R):
            member(res, union(tr1`ac`r, tr2`ac`r)) IMPLIES
             card(s3(ss, ds)(tr1, tr2)`ms(res)) <= ss`rm(res))
          =>
          ((FORALL (res: R):
              NOT member(res, union(tr1`ac`r, tr2`ac`r)) IMPLIES
               card(tr1`s`ms(res)) <= ss`rm(res))
            AND
            (FORALL (res: R):
               member(res, union(tr1`ac`r, tr2`ac`r)) IMPLIES
                card(s3(ss, ds)(tr1, tr2)`ms(res)) <= ss`rm(res)))
```

Again we start by removing the universal quantifier, then we split the goal in two subgoals. This leads to

```
[-1]  tr1!1`s = tr2!1`s
[-2]  FORALL (res: R):
        member(res, union(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         card(s3(ss, ds)(tr1!1, tr2!1)`ms(res)) <= ss`rm(res)
  |-------
{1}   FORALL (res: R):
        member(res, union(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         card(s3(ss, ds)(tr1!1, tr2!1)`ms(res)) <= ss`rm(res)
```

which is trivial, and

```
[-1]  tr1!1`s = tr2!1`s
[-2]  FORALL (res: R):
        member(res, union(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         card(s3(ss, ds)(tr1!1, tr2!1)`ms(res)) <= ss`rm(res)
  |-------
{1}   FORALL (res: R):
        NOT member(res, union(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         card(tr1!1`s`ms(res)) <= ss`rm(res)
```

Now we use system invariant 9, for any resource $r \in R$ it holds that $\#\hat{m}_s(r) \leq R_m(r)$. After removing the universal quantifier in the consequent, instantiating the antecedents with `res!1` and hiding useless functions, we obtain the following sequent:

```
{-1}  card(ds`ms(res!1)) <= ss`rm(res!1)
  |-------
[1]   card(tr1!1`s`ms(res!1)) <= ss`rm(res!1)
```

This holds iff $\hat{m}_s(res!1) = ms(res)$.

The second step we will discuss is step eight of the proof in [2]. This step looks as follows:

$$
\begin{aligned}
& (\forall res \in (r' \cap r) : |(((ms(res) \backslash C_b(t, A(res))) \cup C_e(t, A(res))) \\
& \quad \backslash C_b(t', A(res))) \cup C_e(t', A(res)))| \leq R_m(res)) \\
\equiv \quad & \{\text{Lemma 6.2.2}\} \\
& (\forall res \in (r' \cap r) : |ms'(res)| \leq R_m(res)) \\
& \wedge (\forall res \in (r' \cap r) : |ms''(res)| \leq R_m(res))
\end{aligned}
$$

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s IMPLIES
         ((FORALL (res: R):
             member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
              card(tr1`s1`ms(res)) <= ss`rm(res))
          AND
          (FORALL (res: R):
             member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
              card(tr2`s1`ms(res)) <= ss`rm(res)))
         =>
         (FORALL (res: R):
            member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
             card(union(difference(union
                                   (difference
                                    (tr1`s`ms(res),
                                     ds`cb(tr1`ac`t, ss`a(res))),
                                    ds`ce(tr1`ac`t, ss`a(res))),
                                   ds`cb(tr2`ac`t, ss`a(res))),
                        ds`ce(tr2`ac`t, ss`a(res))))
              <= ss`rm(res))
```

This sequent should be proved by using the following lemma:

Lemma 6.2.2 *When a task t only loads to safe resources (i.e. resources not from $unsafe$) then there cannot be another task t' that can load different material onto that resource at the same time.*

This lemma is proved in an informal way in [2]. From this proof, we use:

$$
\begin{array}{ll}
(& \forall res \in (r \cap r') \\
: & C_e(t, A(res)) = C_e(t', A(res)) \\
\vee & C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset \\
\vee & C_e(t', A(res)) \backslash C_b(t', A(res) = \emptyset \\
) &
\end{array}
$$

We get the following sequent:

```
{-1}  FORALL (tr1: setC_el, tr2: Tau, res: R):
        member(res, intersection(tr1`ac`r, tr2`ac`r)) IMPLIES
         ds`ce(tr1`ac`t, ss`a(res)) = ds`ce(tr2`ac`t, ss`a(res)) OR
          difference(ds`ce(tr1`ac`t, ss`a(res)),
                     ds`cb(tr1`ac`t, ss`a(res)))
           = emptyset
           OR
           difference(ds`ce(tr2`ac`t, ss`a(res)),
                      ds`cb(tr2`ac`t, ss`a(res)))
            = emptyset
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s IMPLIES
         ((FORALL (res: R):
             member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
              card(tr1`s1`ms(res)) <= ss`rm(res))
          AND
```

```
          (FORALL (res: R):
             member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
              card(tr2`s1`ms(res)) <= ss`rm(res)))
         =>
         (FORALL (res: R):
            member(res, intersection(tr2`ac`r, tr1`ac`r)) IMPLIES
             card(union(difference(union
                                  (difference
                                   (tr1`s`ms(res),
                                    ds`cb(tr1`ac`t, ss`a(res))),
                                   ds`ce(tr1`ac`t, ss`a(res))),
                                  ds`cb(tr2`ac`t, ss`a(res))),
                        ds`ce(tr2`ac`t, ss`a(res))))
              <= ss`rm(res))
```

We remove the universal quantifier in the consequent and instantiate the antecedent with `tr1!1` and `tr2!1` for the outer quantifier and we take `res!1` for the inner quantifier. We get the sequent below:

```
[-1]  member(res!1, intersection(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
       ds`ce(tr1!1`ac`t, ss`a(res!1)) =
          ds`ce(tr2!1`ac`t, ss`a(res!1)) OR
        difference(ds`ce(tr1!1`ac`t, ss`a(res!1)),
                   ds`cb(tr1!1`ac`t, ss`a(res!1)))
         = emptyset
         OR
         difference(ds`ce(tr2!1`ac`t, ss`a(res!1)),
                    ds`cb(tr2!1`ac`t, ss`a(res!1)))
          = emptyset
[-2]  tr1!1`s = tr2!1`s
[-3]  member(res!1, intersection(tr2!1`ac`r, tr1!1`ac`r)) IMPLIES
       card(tr1!1`s1`ms(res!1)) <= ss`rm(res!1)
{-4}  member(res!1, intersection(tr2!1`ac`r, tr1!1`ac`r)) IMPLIES
       card(tr2!1`s1`ms(res!1)) <= ss`rm(res!1)
[-5]  member(res!1, intersection(tr2!1`ac`r, tr1!1`ac`r))
  |-------
[1]   card(union(difference(
             union(difference(tr1!1`s`ms(res!1),
                              ds`cb(tr1!1`ac`t, ss`a(res!1))),
                   ds`ce(tr1!1`ac`t, ss`a(res!1))),
             ds`cb(tr2!1`ac`t, ss`a(res!1))),
                 ds`ce(tr2!1`ac`t, ss`a(res!1))))
       <= ss`rm(res!1)
```

Splitting this sequent leads to four subgoals; one for each disjunct in the used part of the lemma, and the trivial subgoal $\texttt{res} \in \texttt{r} \cup \texttt{r}' \Rightarrow \texttt{res} \in \texttt{r} \cup \texttt{r}'$. We were not able to prove the first three subgoals; let us take a look at one of these. The second subgoal uses $C_e(t, A(res)) \backslash C_b(t, A(res)) = \emptyset$; this is antecedent `-3` in the sequent given below:

```
{-1}  card(tr2!1`s1`ms(res!1)) <= ss`rm(res!1)
[-2]  card(tr1!1`s1`ms(res!1)) <= ss`rm(res!1)
[-3]  difference(ds`ce(tr2!1`ac`t, ss`a(res!1)),
                 ds`cb(tr2!1`ac`t, ss`a(res!1)))
       = emptyset
[-4]  tr1!1`s = tr2!1`s
[-5]  member(res!1, intersection(tr2!1`ac`r, tr1!1`ac`r))
```

```
  |-------
[1]   card(union(difference(
            union(difference(tr1!1`s`ms(res!1),
                             ds`cb(tr1!1`ac`t, ss`a(res!1))),
                  ds`ce(tr1!1`ac`t, ss`a(res!1))),
            ds`cb(tr2!1`ac`t, ss`a(res!1))),
                ds`ce(tr2!1`ac`t, ss`a(res!1))))
       <= ss`rm(res!1)
```

Properties 4 and 8 of both τ and set C contain useful information; property 4 says $\forall res \in r'.Cb(t', A(res)) \subseteq ms'(res)$ and $\forall res \in r.Cb(t, A(res)) \subseteq ms''(res)$, and property 8 says $\forall res \in r'.ms'''(res) = (ms'(res) \setminus Cb(t', A(res))) \cup Ce(t', A(res))$ and $\forall res \in r.ms'''(res) = (ms''(res) \setminus Cb(t, A(res))) \cup Ce(t, A(res))$. We add these properties to the sequent, instantiate all of these properties with `res!1` and split the goal in five subgoals. Four subgoals are trivial, the remaining subgoal is given below:

```
{-1}  tr1!1`s1`ms(res!1) =
       union(difference(tr2!1`s`ms(res!1),
                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
             ds`ce(tr1!1`ac`t, ss`a(res!1)))
{-2}  subset?(ds`cb(tr1!1`ac`t, ss`a(res!1)), tr2!1`s`ms(res!1))
[-3]  tr2!1`s1`ms(res!1) =
       union(difference(tr2!1`s`ms(res!1),
                        ds`cb(tr2!1`ac`t, ss`a(res!1))),
             ds`ce(tr2!1`ac`t, ss`a(res!1)))
[-4]  subset?(ds`cb(tr2!1`ac`t, ss`a(res!1)), tr2!1`s`ms(res!1))
{-5}  card(union(difference(tr2!1`s`ms(res!1),
                            ds`cb(tr2!1`ac`t, ss`a(res!1))),
                  ds`ce(tr2!1`ac`t, ss`a(res!1))))
       <= ss`rm(res!1)
{-6}  card(union(difference(tr2!1`s`ms(res!1),
                            ds`cb(tr1!1`ac`t, ss`a(res!1))),
                  ds`ce(tr1!1`ac`t, ss`a(res!1))))
       <= ss`rm(res!1)
[-7]  difference(ds`ce(tr2!1`ac`t, ss`a(res!1)),
                 ds`cb(tr2!1`ac`t, ss`a(res!1)))
       = emptyset
[-8]  tr1!1`s = tr2!1`s
[-9] member(res!1, intersection(tr2!1`ac`r, tr1!1`ac`r))
  |-------
{1}   card(union(difference(
            union(difference(tr2!1`s`ms(res!1),
                             ds`cb(tr1!1`ac`t, ss`a(res!1))),
                  ds`ce(tr1!1`ac`t, ss`a(res!1))),
            ds`cb(tr2!1`ac`t, ss`a(res!1))),
                ds`ce(tr2!1`ac`t, ss`a(res!1))))
       <= ss`rm(res
```

To improve the readability of the sequent above, we rename some parts of the formulas using the following abbreviations: A = tr2!1`s`ms(res!1), B = ds`cb(tr1!1`ac`t, ss`a(res!1)), C = ds`ce(tr1!1`ac`t, ss`a(res!1)), D = ds`cb(tr2!1`ac`t, ss`a(res!1)), E = ds`ce(tr2!1`ac`t, ss`a(res!1)) and N = ss`rm(res!1). This results in the goal below, given in mathematical notation:

$$
\begin{array}{ll}
 & ms'(res) = (A \setminus B) \cup C \\
\wedge & B \subseteq A \\
\wedge & ms''(res) = (A \setminus D) \cup E \\
\wedge & D \subseteq A \\
\wedge & |(A \setminus D) \cup E| \leq N \\
\wedge & |(A \setminus B) \cup C| \leq N \\
\wedge & E \setminus D = \emptyset \\
\wedge & res \in (r \cap r') \\
\Rightarrow & |(((A \setminus B) \cup C) \setminus D) \cup E| \leq N
\end{array}
$$

Now we use property 10 of set C which says that there do not exist two tasks using the same material or there exists a precedence relation between the two tasks using the same material. We only look at the first branch due to lack of time. If tasks do not share material, we get $(\mathtt{B} \cup \mathtt{C}) \cap (\mathtt{D} \cup \mathtt{E}) = \emptyset$. Furthermore, we rewrite $\mathtt{E} \setminus \mathtt{D} = \emptyset$ to $\mathtt{E} \subseteq \mathtt{D}$ and replace $(\mathtt{A} \setminus \mathtt{B}) \cup \mathtt{C})$ by Z. We get the following proof obligation:

$$
\begin{array}{ll}
 & B \subseteq A \\
\wedge & D \subseteq A \\
\wedge & E \subseteq D \\
\wedge & (B \cup C) \cap (D \cup E) = \emptyset \\
\wedge & |Z| \leq N \\
\Rightarrow & |(Z \setminus D) \cup E| \leq N
\end{array}
$$

Since $\mathtt{D} \cup \mathtt{E}$ has no relation with $\mathtt{B} \cup \mathtt{C}$, this inequality holds. But we were not able to prove this with PVS so far. We did not study the other mentioned branches in detail, so we are not sure if these are correct or not at this moment.

6.3 Unfinished proofs

Due to lack of time we were not able to prove all proof obligations in PVS. Therefore we cannot say anything about the correctness of the proofs, given in [2], of the following proof obligations. Both lemmas are proved in an informal way there.

- Proof obligation: tau-1
  $successful(P(t'), tp')$

- Proof obligation: setC-1
  $successful(P(t), tp'')$

- Lemma 6.2.1
  When there exists tasks t' and t, such that t' is in $allsucc(t)$ and t and t' are both enabled in a state (tp, ms) then there must be an ancestor of t which is a group.

- Lemma 6.2.2
  When a task t only loads to safe resources (i.e. resources not from $unsafe$) then there cannot be another task t' that can load different material onto that resource at the same time.

Finally, we started with the proof attempts of tau-2 and setC-8. The results are discussed below.

Proof obligation: tau-2

The proof for tau-2 is not complete in section 6.2.2. We had to prove $t' \in T \setminus (tp' \cup bypassed(tp'))$ and assumed that $\forall g \in G : succnil(g) \cap (tp \cup \{t\}) = succnil(g) \cap tp$ and $\forall g \in G : G_n(g) \cap (n_i(tp \cup \{t\}) \cup n_s(tp \cup \{t\})) = G_n(g) \cap (n_i(tp) \cup n_s(tp))$ hold. The first assumption consists of three steps in [2]:

$$
\begin{aligned}
& (\forall g \in G : succnil(g) \cap (tp \cup \{t\}) = succnil(g) \cap tp) \\
\equiv\ & \{A \cap (B \cup C) = (A \cap B) \cup (A \cap C)\} \\
& (\forall g \in G : ((succnil(g) \cap tp) \cup (succnill(g) \cap \{t\})) = succnil(g) \cap tp) \\
\equiv\ & \{calculus\} \\
& (\forall g \in G : succnil(g) \cap \{t\} = \emptyset) \\
\equiv\ & \{(\forall g \in G, n \in (g \cup anc(g)) : (anc(t) \cup \{t\}) \cap succ(n) = \emptyset)\} \\
& true
\end{aligned}
$$

We proved the first step without any difficulties in PVS. We also tried to prove the second step; this step does not hold. Let us simplify this step. We have to prove:

$$
\begin{aligned}
& A \cup B = A \\
\equiv\ & \{calculus\} \\
& B = \emptyset
\end{aligned}
$$

Now we replace the equivalence by $\Rightarrow$ and $\Leftarrow$; both should hold. The latter one does not lead to any problems, but the $\Rightarrow$-version does. For example, take `B` $=$ `A` as counterexample. Fortunately, we only need the $\Leftarrow$-version to prove the original goal. We did not have enough time to prove the last step, neither for the second assumption.

Proof obligation: setC-8

We have to prove $\forall res \in r.ms'''(res) = (ms''(res) \setminus Cb(t, A(res))) \cup Ce(t, A(res))$:

```
[-1]  tr1!1`s = tr2!1`s
  |-------
{1}   FORALL (res: R):
        member(res, tr1!1`ac`r) IMPLIES
         s3(ss, ds)(tr1!1, tr2!1)`ms(res) =
          union(difference(tr2!1`s1`ms(res),
                           ds`cb(tr1!1`ac`t, ss`a(res))),
                ds`ce(tr1!1`ac`t, ss`a(res)))
[2]   tr1!1`s1 = tr2!1`s1
```

The proof consists of five steps in [2]. Only the second step is interesting, the other ones do not lead to any problems. The second step looks as follows:

$$
\begin{aligned}
& (\forall res \in (r \cap r') : ms'''(res) = (((ms(res) \setminus C_b(t', A(res))) \cup C_e(t', A(res))) \\
& \quad \setminus C_b(t, A(res))) \cup C_e(t, A(res))) \\
& \wedge (\forall res \in (r \setminus r') : ms'''(res) = (ms''(res) \setminus C_b(t, A(res))) \cup C_e(t, A(res))) \\
\equiv\ & \{(\forall res \in (r' \cap r) : ms'''(res) = (((ms(res) \setminus C_b(t, A(res))) \cup C_e(t, A(res))) \\
& \quad \setminus C_b(t', A(res))) \cup C_e(t', A(res)))\} \\
& (\forall res \in (r \setminus r') : ms'''(res) = (ms''(res) \setminus C_b(t, A(res))) \cup C_e(t, A(res)))
\end{aligned}
$$

We replace the equivalence by $\Rightarrow$ and $\Leftarrow$; the first one is trivial, so we move on to the $\Leftarrow$-part. We get the following sequent:

```
  |-------
[1]   FORALL (tr1: setC_el, tr2: Tau):
        tr1`s = tr2`s AND tr1`s1 /= tr2`s1 IMPLIES
         (FORALL (res: R):
            member(res, difference(tr1`ac`r, tr2`ac`r)) IMPLIES
             s3(ss, ds)(tr1, tr2)`ms(res) =
              union(difference(tr2`s1`ms(res),
                               ds`cb(tr1`ac`t, ss`a(res))),
                    ds`ce(tr1`ac`t, ss`a(res))))
          =>
          ((FORALL (res: R):
              member(res, intersection(tr1`ac`r, tr2`ac`r)) IMPLIES
               s3(ss, ds)(tr1, tr2)`ms(res) =
                union(difference(union(difference
                                        (tr2`s`ms(res),
                                         ds`cb(tr2`ac`t, ss`a(res))),
                                        ds`ce(tr2`ac`t, ss`a(res))),
                                 ds`cb(tr1`ac`t, ss`a(res))),
                      ds`ce(tr1`ac`t, ss`a(res))))
            AND
            (FORALL (res: R):
               member(res, difference(tr1`ac`r, tr2`ac`r)) IMPLIES
                s3(ss, ds)(tr1, tr2)`ms(res) =
                 union(difference(tr2`s1`ms(res),
                                  ds`cb(tr1`ac`t, ss`a(res))),
                       ds`ce(tr1`ac`t, ss`a(res)))))
```

We remove the universal quantifier and split the obtained goal. We get two subgoals; one is of the form `A` $\Rightarrow$ `A` which is trivial, the other one looks as follows:

```
[-1]  tr1!1`s = tr2!1`s
[-2]  FORALL (res: R):
        member(res, difference(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         s3(ss, ds)(tr1!1, tr2!1)`ms(res) =
          union(difference(tr2!1`s1`ms(res),
                           ds`cb(tr1!1`ac`t, ss`a(res))),
                ds`ce(tr1!1`ac`t, ss`a(res)))
  |-------
{1}   FORALL (res: R):
        member(res, intersection(tr1!1`ac`r, tr2!1`ac`r)) IMPLIES
         s3(ss, ds)(tr1!1, tr2!1)`ms(res) =
          union(difference(
            union(difference(tr2!1`s`ms(res),
                             ds`cb(tr2!1`ac`t, ss`a(res))),
                  ds`ce(tr2!1`ac`t, ss`a(res))),
            ds`cb(tr1!1`ac`t, ss`a(res))),
                ds`ce(tr1!1`ac`t, ss`a(res)))
[2]   tr1!1`s1 = tr2!1`s1
```

The antecedent -2 is useless here; we need the given hint to prove this sequent. The hint is nothing more than the definition of $\mathtt{ms}'''$. So, if we hide antecedent -2, remove the universal quantifier in consequent `1`, expand, among others, the definition of `s3` followed by expanding the definition of `ms3` and decompose the equality in consequent `1`, we get:

```
[-1]  tr1!1`ac`r(res!1)
[-2]  tr2!1`ac`r(res!1)
```

```
[-3]  tr1!1`s = tr2!1`s
  |-------
{1}   IF NOT member(res!1, union(tr1!1`ac`r, tr2!1`ac`r))
        THEN tr1!1`s`ms(res!1)(x!1)
      ELSE IF member(res!1, difference(tr1!1`ac`r, tr2!1`ac`r))
             THEN tr1!1`s1`ms(res!1)(x!1)
           ELSE IF member(res!1, difference(tr2!1`ac`r, tr1!1`ac`r))
                  THEN tr2!1`s1`ms(res!1)(x!1)
                ELSE union(difference
                       (union(difference
                                (tr1!1`s`ms(res!1),
                                   ds`cb(tr1!1`ac`t, ss`a(res!1))),
                              ds`ce(tr1!1`ac`t, ss`a(res!1))),
                                 ds`cb(tr2!1`ac`t, ss`a(res!1))),
                            ds`ce(tr2!1`ac`t, ss`a(res!1)))
                           (x!1)
                ENDIF
           ENDIF
      ENDIF
       =
       union(difference(union(difference(tr2!1`s`ms(res!1),
                                    ds`cb(tr2!1`ac`t, ss`a(res!1))),
                              ds`ce(tr2!1`ac`t, ss`a(res!1))),
                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
             ds`ce(tr1!1`ac`t, ss`a(res!1)))
            (x!1)
[2]   tr1!1`s1 = tr2!1`s1
```

Because $\texttt{res} \in (\texttt{r} \cup \texttt{r}')$, case 4 of Fig. 6.1 holds, which is the ELSE-clause in consequent 1. We use the command (smash) to split the sequent above in eight subgoals. Six of these are trivial, the other two differ only in antecedent -2 and consequent 3: they are symmetrical. One of these subgoals is shown below:

```
{-1}  member(res!1, union(tr1!1`ac`r, tr2!1`ac`r))
{-2}  union(difference(union(difference(tr1!1`s`ms(res!1),
                                    ds`cb(tr1!1`ac`t, ss`a(res!1))),
                              ds`ce(tr1!1`ac`t, ss`a(res!1))),
                        ds`cb(tr2!1`ac`t, ss`a(res!1))),
             ds`ce(tr2!1`ac`t, ss`a(res!1)))
            (x!1)
[-3]  tr1!1`ac`r(res!1)
[-4]  tr2!1`ac`r(res!1)
[-5]  tr1!1`s = tr2!1`s
  |-------
{1}   member(res!1, difference(tr1!1`ac`r, tr2!1`ac`r))
{2}   member(res!1, difference(tr2!1`ac`r, tr1!1`ac`r))
{3}   union(difference(union(difference(tr2!1`s`ms(res!1),
                                    ds`cb(tr2!1`ac`t, ss`a(res!1))),
                              ds`ce(tr2!1`ac`t, ss`a(res!1))),
                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
             ds`ce(tr1!1`ac`t, ss`a(res!1)))
            (x!1)
[4]   tr1!1`s1 = tr2!1`s1
```

We see that $\texttt{t}$ and $\texttt{t}'$ in antecedent -2 are swapped in consequent 3. To prove this sequent, we need $(\texttt{C}_\texttt{b}(\texttt{t}, \texttt{A}(\texttt{res})) \cup \texttt{C}_\texttt{e}(\texttt{t}, \texttt{A}(\texttt{res}))) \cap (\texttt{C}_\texttt{b}(\texttt{t}', \texttt{A}(\texttt{res})) \cup \texttt{C}_\texttt{e}(\texttt{t}', \texttt{A}(\texttt{res}))) = \emptyset$. Property 10 of set C could be

useful, so we make this property explicit, instantiate it with the only possible value t' and split the result. We obtain three subgoals; for one of them we have to prove $t' \in T$ which holds. The second subgoal is given below: here, two tasks do not use the same material.

```
[-1]  member(res!1, union(tr1!1`ac`r, tr2!1`ac`r))
[-2]  union(difference(union(difference(tr1!1`s`ms(res!1),
                                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
                             ds`ce(tr1!1`ac`t, ss`a(res!1))),
                       ds`cb(tr2!1`ac`t, ss`a(res!1))),
            ds`ce(tr2!1`ac`t, ss`a(res!1)))
           (x!1)
[-3]  tr1!1`ac`r(res!1)
[-4]  tr2!1`ac`r(res!1)
[-5]  tr1!1`s = tr2!1`s
  |-------
{1}   intersection(mat(ds)(tr2!1`ac`t),
                   mat(ds)(tr1!1`ac`t)) /= emptyset
[2]   member(res!1, difference(tr1!1`ac`r, tr2!1`ac`r))
[3]   member(res!1, difference(tr2!1`ac`r, tr1!1`ac`r))
[4]   union(difference(union(difference(tr2!1`s`ms(res!1),
                                        ds`cb(tr2!1`ac`t, ss`a(res!1))),
                             ds`ce(tr2!1`ac`t, ss`a(res!1))),
                       ds`cb(tr1!1`ac`t, ss`a(res!1))),
            ds`ce(tr1!1`ac`t, ss`a(res!1)))
           (x!1)
[5]   tr1!1`s1 = tr2!1`s1
```

This sequent does not lead to any problems. Finally we have to prove the third subgoal: there exists a precedence relation between two tasks using the same material.

```
{-1}  intersection(union(singleton[N](tr2!1`ac`t),
                         anc(ds)(tr2!1`ac`t)),
                   allsucc(ds)(tr1!1`ac`t))
       /= emptyset
[-2]  member(res!1, union(tr1!1`ac`r, tr2!1`ac`r))
[-3]  union(difference(union(difference(tr1!1`s`ms(res!1),
                                        ds`cb(tr1!1`ac`t, ss`a(res!1))),
                             ds`ce(tr1!1`ac`t, ss`a(res!1))),
                       ds`cb(tr2!1`ac`t, ss`a(res!1))),
            ds`ce(tr2!1`ac`t, ss`a(res!1)))
           (x!1)
[-4]  tr1!1`ac`r(res!1)
[-5]  tr2!1`ac`r(res!1)
[-6]  tr1!1`s = tr2!1`s
  |-------
[1]   member(res!1, difference(tr1!1`ac`r, tr2!1`ac`r))
[2]   member(res!1, difference(tr2!1`ac`r, tr1!1`ac`r))
[3]   union(difference(union(difference(tr2!1`s`ms(res!1),
                                        ds`cb(tr2!1`ac`t, ss`a(res!1))),
                             ds`ce(tr2!1`ac`t, ss`a(res!1))),
                       ds`cb(tr1!1`ac`t, ss`a(res!1))),
            ds`ce(tr1!1`ac`t, ss`a(res!1)))
           (x!1)
[4]   tr1!1`s1 = tr2!1`s1
```

We move the negation of antecedent 1 to the consequents and decompose this equality, introducing variable x!2. Then we expand all definitions in this formula. We get the following goal:

```
{-1}  x!2 = tr2!1‘ac‘t OR anc(ds)(tr2!1‘ac‘t)(x!2)
{-2}  allsucc(ds)(tr1!1‘ac‘t)(x!2)
[-3]  member(res!1, union(tr1!1‘ac‘r, tr2!1‘ac‘r))
[-4]  union(difference(union(difference(tr1!1‘s‘ms(res!1),
                                       ds‘cb(tr1!1‘ac‘t, ss‘a(res!1))),
                             ds‘ce(tr1!1‘ac‘t, ss‘a(res!1))),
                      ds‘cb(tr2!1‘ac‘t, ss‘a(res!1))),
            ds‘ce(tr2!1‘ac‘t, ss‘a(res!1)))
           (x!1)
[-5]  tr1!1‘ac‘r(res!1)
[-6]  tr2!1‘ac‘r(res!1)
[-7]  tr1!1‘s = tr2!1‘s
  |-------
[1]   member(res!1, difference(tr1!1‘ac‘r, tr2!1‘ac‘r))
[2]   member(res!1, difference(tr2!1‘ac‘r, tr1!1‘ac‘r))
[3]   union(difference(union(difference(tr2!1‘s‘ms(res!1),
                                       ds‘cb(tr2!1‘ac‘t, ss‘a(res!1))),
                             ds‘ce(tr2!1‘ac‘t, ss‘a(res!1))),
                      ds‘cb(tr1!1‘ac‘t, ss‘a(res!1))),
            ds‘ce(tr1!1‘ac‘t, ss‘a(res!1)))
           (x!1)
[4]   tr1!1‘s1 = tr2!1‘s1
```

We were not able to prove this or to construct a counterexample, due to lack of time.

Chapter 7

Concluding remarks

The assignment was to check the correctness of the proofs in [2] by a theorem prover (PVS). We translated the mathematical model in [10] into a PVS model and we tried to prove the proof obligations in [2] with the help of PVS. If a proof seemed to be incorrect, we tried to correct a proof, to change definitions or to add constraints when needed.

Reduction techniques. Apart from the design decisions we had to make while translating the mathematical model into the PVS model, we had to change the definition of $successor$. The original version contained a problem with the types, this has been solved. Furthermore, we had to add one property to set `C`. The definition and the intuition behind it did not match. The changes we made are:

- Definition of `successor`

$$\begin{aligned} successor(n) \quad = \quad & \{n' \in N \mid (n \cup anc(n)) \cap P(n') \neq \emptyset\} \cup \\ & \{successor(n') \mid n' \in successor(n) \cap G \ \wedge \ 0 \in G_a(n')\}. \end{aligned}$$

  has been replaced by

$$\begin{aligned} successor(n) \quad = \quad & \{n' \in N \mid (n \cup anc(n)) \cap P(n') \neq \emptyset\} \cup \\ & \bigcup_{n' \in N} \{successor(n') \mid n' \in successor(n) \cap G \ \wedge \ 0 \in G_a(n')\}. \end{aligned}$$

- Properties of set `C`
  We added the following property to set `C`:
  $\forall((tp, ms), (t', r'), (tp'', ms'')) \in \tau, res \in R : Cb(t, A(res)) \neq Ce(t, A(res)) \Rightarrow res \in (r \cap r')$

All other definitions, functions or properties remained unchanged. The table contains the results for each proof attempt. A ✓ means that the proof obligation is proved in PVS and a ✗ means that we found one or more steps in the proof in [2] to be incorrect and we were not able to correct this proof. A list of proof obligations can be found in 6.1; here we use the identifiers to enumerate them. If the proof attempt is discussed in this document, a reference to the section where the proof attempt is explained is provided.

Proof obligation	Status	Remarks
tau-1	unknown	Unable to start with this proof due to lack of time
tau-2	unknown	Two lemmas not proved, see section 6.2.2
tau-3	✓	See section 6.2.1
tau-4	✗	See section 6.2.3
tau-5	✗	See section 6.2.3
tau-6	✗	Consult setC-6
tau-7	✓	For this proof we refer to the CD
tau-8	✓	For this proof we refer to the CD
tau-9	✓	See section 6.2.1
setC-1	unknown	Unable to start with this proof due to lack of time
setC-2	✓	Modified, see section 6.2.2
setC-3	✓	For this proof we refer to the CD, or check tau-3
setC-4	✗	Consult tau-4
setC-5	✗	Consult tau-5
setC-6	✗	See section 6.2.3
setC-7	✓	For this proof we refer to the CD
setC-8	unknown	See section 6.3
setC-9	✓	For this proof we refer to the CD, or check tau-9
setC-10	✓	For this proof we refer to the CD
setC-11	✓	For this proof we refer to the CD
setC-12	✓	See section 6.2.1
setC-13	✓	See section 6.2.1
setC-14	✓	For this proof we refer to the CD

The proof obligations we failed to prove may still hold by adding properties or there may exist a complete different way to prove them. The mentioned changes do not affect other proofs; if a proof obligation holds before we made the changes, this proof obligation still holds after the changes have been made.

PVS. Since this was the first time we used PVS, or a theorem prover in general, it might be useful to share some experiences. Using PVS is more time consuming than we had thought at a first glance. Learning the basics of PVS, without having used any other theorem prover before, will take some months. Knowing everything is almost impossible, since PVS is a very large and complex system. There are hardly any books available, the best way to start is skimming through the tutorial [1]. Spending too much time on this tutorial is inefficient because the information in it is not very detailed and the reader will encounter a lot of problems which are not discussed anywhere. Just making your own examples and use only that part of PVS you need is the best way to get started. The PVS language is easy to understand, it is almost plain English. Proof sequents are well organised and interaction is intuitive. The online manuals are good but not perfect: for both the language and the prover, it is not always clear when to choose method `X` instead of `Y` when `X` and `Y` have a lot in common.

Theorem provers definitely contribute to more robust systems. Some proofs in [2] seemed to be correct and errors would not be found without the use of a theorem prover. But, of course, one has to be sure the model is correct, or else proofs in PVS are worthless. Despite the strengths of theorem proving, using a theorem prover to verify the correctness of any system, either large and complex or small and simple, is lightyears away. Learning and using theorem provers require a great deal of effort and is therefore too expensive for companies. Also, theorem provers have to be improved; for example, using sets in PVS is not very amusing. Last but not least, experts are needed to guide a theorem prover and there exist various kinds of theorem provers for different fields.

Chapter 8

Further research

Because theorem proving is more difficult than it seems to be, we were not able to complete the proofs for all three reduction techniques. As you can see above, some proof obligation for the first reduction technique are not proved yet. These still have to be proved in some way or rejected by giving a counter example. Also, the used lemmas are only informally defined and proved; the definitions and the proofs have to be formalised and checked by a theorem prover. And the model as it is now, generates a number of type-correctness conditions (TCCs). Most of them result from the use of non-empty sets and the cardinality function. These TCCs have to be discharged too. We started to prove some of them, but because this would take too much time we decided to skip this part for the time being. We only checked roughly if there were problematic ones which could be a consequence of inconsistencies in the model, but they should all be provable.

Finally, for the other two reduction techniques, a model has to be made and proofs have to be given. These techniques are possibly easier and less time consuming.

Bibliography

[1] J. Crow, S. Owre, J.M. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995. Computer Science Laboratory, SRI International. Updated June 1995.

[2] M.M.H. Driessen. Verification of task resource scheduling. Technical report, Technische Universiteit Eindhoven, jun 2004. ASML Confidential.

[3] J.F. Groote and M.P.A Sellink. Confluence for process verification. *TCS: Theoretical Computer Science*, 170:47–81, 1996.

[4] J.F. Groote and J.C. van de Pol. State space reduction using partial tau-confluence. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, volume 1893, pages 383–393. Springer-Verlag, 2000.

[5] R.M. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[6] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[7] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[8] S. Owre, N. Shankar, J.M Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[9] G. Sutcliffe. Automated theorem proving: A review. *AI Magazine*, 23(1):121, 2002.

[10] N.J.M. van den Nieuwelaar, M.M.H. Driessen, and J.F. Groote. A dedicated verification approach for scheduling in complex manufacturing systems. Computer Science Report 04-27, Technische Universiteit Eindhoven, 2004.

[11] N.J.M. van den Nieuwelaar, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Design of Supervisory Machine Control*. Accepted for European Control Conference ECC03, 2003.

[12] F. Wiedijk. Comparing mathematical provers. In *MKM*, pages 188–202, 2003.

Appendix A

Math vs. PVS notation

Below you find a conversion table to transform mathematical notation into PVS notation and vice versa. We use case `I` of confluence, depicted in Fig. 4.1. This means that the following pairs of states are equal: the begin states of transition `tr1` and `tr2`, the end state of transition `tr1` and the begin state of transition `tr3`, the end state of transition `tr2` and the begin state of transition `tr4`, and the end states of transition `tr3` and `tr4`. That is why you find two different PVS representations for the one mathematical representation in some cases.

Math	$\Leftrightarrow$	**PVS**
s	$\Leftrightarrow$	$tr1`s$ **or** $tr2`s$
s'	$\Leftrightarrow$	$tr1`s1$
s''	$\Leftrightarrow$	$tr2`s1$
s'''	$\Leftrightarrow$	$tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1$ **or** $tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1$
ms	$\Leftrightarrow$	$tr1`s`ms$ **or** $tr2`s`ms$
ms'	$\Leftrightarrow$	$tr1`s1`ms$ **or** $tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`s`ms$
ms''	$\Leftrightarrow$	$tr2`s1`ms$ **or** $tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`s`ms$
ms'''	$\Leftrightarrow$	$tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1`ms$ **or**
		$tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1`ms$
tp	$\Leftrightarrow$	$tr1`s`tp$ **or** $tr2`s`tp$
tp'	$\Leftrightarrow$	$tr1`s1`tp$ **or** $tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`s`tp$
tp''	$\Leftrightarrow$	$tr2`s1`tp$ **or** $tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`s`tp$
tp'''	$\Leftrightarrow$	$tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1`tp$ **or**
		$tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`s1`tp$
t	$\Leftrightarrow$	$tr1`ac`t$ **or** $tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`ac`t$
t'	$\Leftrightarrow$	$tr2`ac`t$ **or** $tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`ac`t$
r	$\Leftrightarrow$	$tr1`ac`r$ **or** $tr4(tr1, tr2, s3(ss, ds)(tr1, tr2))`ac`r$
r'	$\Leftrightarrow$	$tr2`ac`r$ **or** $tr3(tr1, tr2, s3(ss, ds)(tr1, tr2))`ac`r$

Appendix B

Simple Machine

B.1 Definition

An example called *Simple Machine* is depicted in Fig. B.1. This example is taken from [10]. The machine consists of 2 equivalent main processing resources: P0 and P1. Furthermore, products can be transported from the environment (E0) to a processing resource using a robot (R0). At the processing units, a cleaning resource (C0) is available to clean products before processing. Note that although the cleaning unit is essential in the production process, products will never be put on the cleaning resource itself. The robot and the processing resources can each contain one product (denoted between brackets in Fig. B.1). The arrows in Fig. B.1 depict how products can be transferred from resource to resource. The static system definition of *Simple Machine* looks as follows:

- $R = \{\mathsf{E0}, \mathsf{R0}, \mathsf{P0}, \mathsf{P1}, \mathsf{C0}\}$;
- $C = \{\mathsf{E}, \mathsf{R}, \mathsf{P}, \mathsf{C}\}$;
- $A = \{(\mathsf{E0}, \mathsf{E}), (\mathsf{R0}, \mathsf{R}), (\mathsf{P0}, \mathsf{P}), (\mathsf{P1}, \mathsf{P}), (\mathsf{C0}, \mathsf{C})\}$;
- $R_m = \{(\mathsf{E0}, 100), (\mathsf{R0}, 1), (\mathsf{P0}, 1), (\mathsf{P1}, 1), (\mathsf{C0}, 0)\}$;
- $M_f = \{(\mathsf{E0}, \{\mathsf{R0}\}), (\mathsf{R0}, \{\mathsf{E0}, \mathsf{P0}, \mathsf{P1}\}), (\mathsf{P0}, \{\mathsf{R0}\}), (\mathsf{P1}, \{\mathsf{R0}\})\}$.

As an example, we consider three products (p0, p1 and p2) to be manufactured by *Simple Machine*. The work to be done for these products is depicted in Fig. B.2. By convention, node names end with a hyphen and the product id. Each product must be loaded, processed, and unloaded, respectively. Loading consists of an E2R task to transfer the product from the environment onto the robot, followed by an R2P task to transfer the product from the robot onto a processing unit, which is similar for unloading. The horizontal precedence arrows show that the products are loaded in the sequence p0, p1 and then p2. Products p0 and p1 are of the same type. They require a cleaning task (t0), and after that two processing tasks after another: t1 and t2, respectively. The cleaning of product p2, however, can also be done otherwise. Instead of the cleaning task, t0, cleaning can also be done by two successive other tasks, t3 and t4, that do not require the cleaning unit. Either one of the alternatives can be scheduled. In Fig. B.2 this choice with respect to

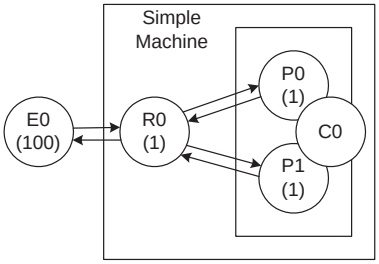


Figure B.1: Layout of a simple manufacturing machine

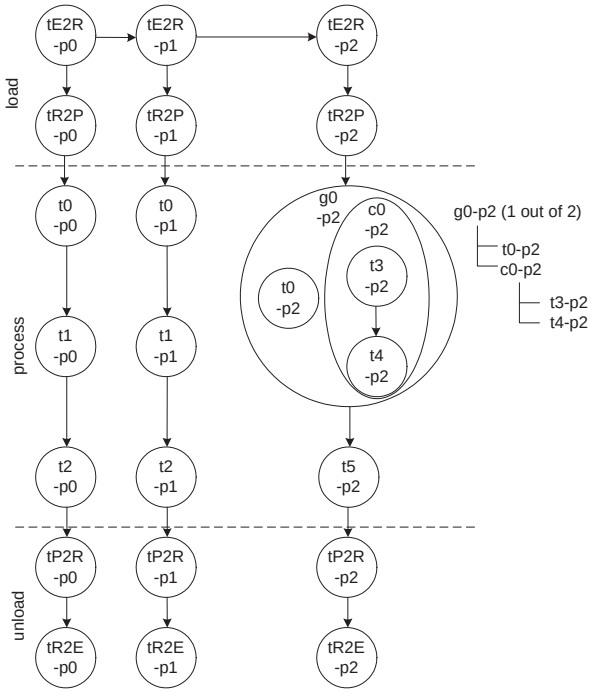


Figure B.2: Three products to be produced by the simple manufacturing machine

cleaning tasks is depicted as nested nodes in the graph, and additionally as a hierarchical node structure at the right. After cleaning, p2 requires one process task, t5. The dynamic system definition for this example looks as follows:

- T={tE2R-p0, tE2R-p1, tE2R-p2, tR2P-p0, tR2P-p1, tR2P-p2, t0-p0, t0-p1, t0-p2, t3-p2, t4-p2, t1-p0, t1-p1, t2-p0, t2-p1, t5-p2, tP2R-p0, tP2R-p1, tP2R-p2, tR2E-p0, tR2E-p1, tR2E-p2};
- G={g0-p2};
- L={c0-p2};
- L_n={(c0-p2, {t3-p2, t4-p2})};
- G_n={(g0-p2, {t0-p2, c0-p2})};
- G_a={(g0-p2, {1})};
- I={(tE2R-p0, {E, R}), (tE2R-p1, {E, R}), (tE2R-p2, {E, R}), (tR2P-p0, {R, P}), (tR2P-p1, {R, P}), (tR2P-p2, {R, P}), (t0-p0, {P, C}), (t0-p1, {P, C}), (t0-p2, {(P, C}), (t3-p2, {P}), (t4-p2, {P}), (t1-p0, {P}), (t1-p1, {P}), (t2-p0, {P}), (t2-p1, {P}), (t5-p2, {P}), (tP2R-p0, {P, R}), (tP2R-p1, {P, R}), (tP2R-p2, {P, R}), (tR2E-p0, {P, E}), (tR2E-p1, {P, E}), (tR2E-p2, {P, E})};
- P={(tR2P-p0, {tE2R-p0}), (t0-p0, {tR2P-p0}), (t1-p0, {t0-p0}), (t2-p0, {t1-p0}), (tP2R-p0, {t2-p0}), (tR2E-p0, {tP2R-p0}), (tE2R-p1, {tE2R-p0}), (tR2P-p1, {tE2R-p1}), (t0-p1, {tR2P-p1}), (t1-p1, {t0-p1}), (t2-p1, {t1-p1}), (tP2R-p1, {t2-p1}), (tR2E-p1, {tP2R-p1}), (tE2R-p2, {tE2R-p1}), (tR2P-p2, {tE2R-p2}), (g0-p2, {tR2P-p2}), (t4-p2, {t3-p2}), (t5-p2, {g0-p2}), (tP2R-p2, {t5-p2}), (tR2E-p2, {tP2R-p2})};
- M={p0, p1, p2};
- C_b={((tE2R-p0, E), {p0}), ((tE2R-p0, R), ∅), ((tE2R-p1, E), {p1}), ((tE2R-p1, R), ∅), ((tE2R-p2, E), {p2}), ((tE2R-p2, R), ∅), ((tR2P-p0, R), {p0}), ((tR2P-p0, P), ∅), ((tR2P-p1, R), {p1}), ((tR2P-p1, P), ∅), ((tR2P-p2, R), {p2}), ((tR2P-p2, P), ∅), ((t0-p0, P), {p0}), ((t0-p0, C), ∅), ((t0-p1, P), {p1}), ((t0-p1, C), ∅), ((t0-p2, P), {p2}), ((t0-p2, C), ∅), ((t3-p2, P), {p2}), ((t4-p2, P), {p2}), ((t1-p0, P), {p0}), ((t1-p1, P), {p1}),

$((\text{t2-p0}, \text{P}), \{\text{p0}\}), ((\text{t2-p1}, \text{P}), \{\text{p1}\}), ((\text{t5-p2}, \text{P}), \{\text{p2}\}), ((\text{tP2R-p0}, \text{P}), \{\text{p0}\}),$
$((\text{tP2R-p0}, \text{R}), \emptyset), ((\text{tP2R-p1}, \text{P}), \{\text{p1}\}), ((\text{tP2R-p1}, \text{R}), \emptyset), ((\text{tP2R-p2}, \text{P}), \{\text{p2}\}),$
$((\text{tP2R-p2}, \text{R}), \emptyset), ((\text{tR2E-p0}, \text{R}), \{\text{p0}\}), ((\text{tR2E-p0}, \text{E}), \emptyset), ((\text{tR2E-p1}, \text{R}), \{\text{p1}\}),$
$((\text{tR2E-p1}, \text{E}), \emptyset), ((\text{tR2E-p2}, \text{R}), \{\text{p2}\}), ((\text{tR2E-p2}, \text{E}), \emptyset)\}$;

- C_e=$\{((\text{tE2R-p0}, \text{E}), \emptyset), ((\text{tE2R-p0}, \text{R}), \{\text{p0}\}), ((\text{tE2R-p1}, \text{E}), \emptyset), ((\text{tE2R-p1}, \text{R}), \{\text{p1}\}),$
$((\text{tE2R-p2}, \text{E}), \emptyset), ((\text{tE2R-p2}, \text{R}), \{\text{p2}\}), ((\text{tR2P-p0}, \text{R}), \emptyset), ((\text{tR2P-p0}, \text{P}), \{\text{p0}\}),$
$((\text{tR2P-p1}, \text{R}), \emptyset), ((\text{tR2P-p1}, \text{P}), \{\text{p1}\}), ((\text{tR2P-p2}, \text{R}), \emptyset), ((\text{tR2P-p2}, \text{P}), \{\text{p2}\}),$
$((\text{t0-p0}, \text{P}), \{\text{p0}\}), ((\text{t0-p1}, \text{P}), \{\text{p1}\}), (\text{t0-p0}, \text{C}), \emptyset), ((\text{t0-p1}, \text{C}), \emptyset), ((\text{t0-p2}, \text{P}), \{\text{p2}\}),$
$((\text{t0-p2}, \text{C}), \emptyset), ((\text{t3-p2}, \text{P}), \{\text{p2}\}), ((\text{t4-p2}, \text{P}), \{\text{p2}\}), ((\text{t1-p0}, \text{P}), \{\text{p0}\}),$
$((\text{t1-p1}, \text{P}), \{\text{p1}\}), ((\text{t2-p0}, \text{P}), \{\text{p0}\}), ((\text{t2-p1}, \text{P}), \{\text{p1}\}), ((\text{t5-p2}, \text{P}), \{\text{p2}\}),$
$((\text{tP2R-p0}, \text{P}), \emptyset), ((\text{tP2R-p0}, \text{R}), \{\text{p0}\}), ((\text{tP2R-p1}, \text{P}), \emptyset), ((\text{tP2R-p1}, \text{R}), \{\text{p1}\}),$
$((\text{tP2R-p2}, \text{P}), \emptyset), ((\text{tP2R-p2}, \text{R}), \{\text{p2}\}), ((\text{tR2E-p0}, \text{R}), \emptyset), ((\text{tR2E-p0}, \text{E}), \{\text{p0}\}),$
$((\text{tR2E-p1}, \text{R}), \emptyset), ((\text{tR2E-p1}, \text{E}), \{\text{p1}\}), ((\text{tR2E-p2}, \text{R}), \emptyset), ((\text{tR2E-p2}, \text{E}), \{\text{p2}\})\}$;

- $\hat{m}_s$=$\{(\text{E0}, \{\text{p0}, \text{p1}, \text{p2}\}), (\text{R0}, \emptyset), (\text{P0}, \emptyset), (\text{P1}, \emptyset), (\text{C0}, \emptyset)\}$.

B.2 PVS Model

Below you see the translation of the *Simple Machine* into PVS:

```
R:  TYPE+ = {E0, R0, P0, P1, C0}
C:  TYPE+ = {E, R, P, C}
A:  TYPE+ = [R -> C]
Rm: TYPE+ = [R -> nat]
Mf: TYPE+ = [R -> setof[R]]

staticSystem: TYPE+ = [#
  a:A,
  rm:Rm,
  mf:Mf
#]

ss: staticSystem = (#
  a  := LAMBDA (r:R):
    COND
      r=E0         -> E,
      r=R0         -> R,
      r=P0 OR r=P1 -> P,
      r=C0         -> C
    ENDCOND,
  rm := LAMBDA (r:R):
    COND
      r=E0                  -> 100,
      r=R0 OR r=P0 OR r=P1 -> 1,
      r=C0                  -> 0
    ENDCOND,
  mf := LAMBDA (r:R):
    COND
      r=E0 OR r=P0 OR r=P1 -> singleton(R0),
      r=R0                  -> {x:R | x=E0 OR x=P0 OR x=P1},
      ELSE                  -> emptyset
    ENDCOND
#)
```

```
setofNat: TYPE+ = {x:setof[nat] |
                     NOT (subset?(x, singleton[nat](0)) AND
                     subset?(singleton[nat](0),x))}

N:  TYPE+ = {tE2Rp0, tE2Rp1, tE2Rp2, tR2Pp0, tR2Pp1, tR2Pp2,
             t0p0, t0p1, t0p2, t3p2, t4p2, t1p0, t1p1, t2p0,
             t2p1, t5p2, tP2Rp0, tP2Rp1, tP2Rp2, tR2Ep0,
             tR2Ep1, tR2Ep2, g0p2, c0p2}

x: VAR N

T(x): bool = x /= g0p2 AND x /= c0p2
G(x): bool = x = g0p2
L(x): bool = x = c0p2

Ln: TYPE+ = [N -> setof[N]]
Gn: TYPE+ = [N -> setof[N]]
Ga: TYPE+ = [N -> setofNat]
I:  TYPE+ = [N -> setof[C]]
P:  TYPE+ = [N -> setof[N]]
M:  TYPE+ = {p0, p1, p2}
Cb: TYPE+ = [N, C -> setof[M]]
Ce: TYPE+ = [N, C -> setof[M]]
Ms: TYPE+ = [R -> setof[M]]

dynamicSystem: TYPE+ = [#
  ln:Ln,
  gn:Gn,
  ga:Ga,
  i:I,
  p:P,
  cb:Cb,
  ce:Ce,
  ms:Ms
#]

ds: dynamicSystem = (#
  ln := LAMBDA (l:N):
    COND
      l=c0p2 -> {x | x = t3p2 OR x = t4p2},
      ELSE   -> emptyset
    ENDCOND,
  gn := LAMBDA (g:N):
    COND
      g=g0p2 -> {x | x = t0p2 OR x = c0p2},
      ELSE   -> emptyset
    ENDCOND,
  ga := LAMBDA (g:N):
    COND g=g0p2 -> singleton(1),
    ELSE        -> emptyset
  ENDCOND,
  i  := LAMBDA (t:N):
    COND
      t=tE2Rp0 OR t=tE2Rp1 OR t=tE2Rp2     -> {x:C | x = E OR x = R},
```

```
    t=tR2Pp0 OR t=tR2Pp1 OR t=tR2Pp2     -> {x:C | x = R OR x = P},
    t=t0p0 OR t=t0p1 OR t=t0p2           -> {x:C | x = P OR x = C},
    t=t3p2 OR t=t4p2 OR t=t1p0 OR
    t=t1p1 OR t=t2p0 OR t=t2p1 OR t=t5p2 -> singleton(P),
    t=tP2Rp0 OR t=tP2Rp1 OR t=tP2Rp2     -> {x:C | x = P OR x = R},
    t=tR2Ep0 OR t=tR2Ep1 OR t=tR2Ep2     -> {x:C | x = P OR x = E},
    ELSE -> emptyset
  ENDCOND,
p  := LAMBDA (n:N):
  COND
    n=tR2Pp0 OR n=tE2Rp1 -> singleton(tE2Rp0),
    n=t0p0               -> singleton(tR2Pp0),
    n=t1p0               -> singleton(t0p0),
    n=t2p0               -> singleton(t1p0),
    n=tP2Rp0             -> singleton(t2p0),
    n=tR2Ep0             -> singleton(tP2Rp0),
    n=tR2Pp1 OR n=tE2Rp2 -> singleton(tE2Rp1),
    n=t0p1               -> singleton(tR2Pp1),
    n=t1p1               -> singleton(t0p1),
    n=t2p1               -> singleton(t1p1),
    n=tP2Rp1             -> singleton(t2p1),
    n=tR2Ep1             -> singleton(tP2Rp1),
    n=tR2Pp2             -> singleton(tE2Rp2),
    n=g0p2               -> singleton(tR2Pp2),
    n=t4p2               -> singleton(t3p2),
    n=t5p2               -> singleton(g0p2),
    n=tP2Rp2             -> singleton(t5p2),
    n=tR2Ep2             -> singleton(tP2Rp2),
    ELSE                 -> emptyset
  ENDCOND,
cb := LAMBDA (t:N),(c:C):
  COND
    t=tE2Rp0 AND c=E -> singleton(p0),
    t=tE2Rp0 AND c=R -> emptyset,
    t=tE2Rp1 AND c=E -> singleton(p1),
    t=tE2Rp1 AND c=R -> emptyset,
    t=tE2Rp2 AND c=E -> singleton(p2),
    t=tE2Rp2 AND c=R -> emptyset,
    t=tR2Pp0 AND c=R -> singleton(p0),
    t=tR2Pp0 AND c=P -> emptyset,
    t=tR2Pp1 AND c=R -> singleton(p1),
    t=tR2Pp1 AND c=P -> emptyset,
    t=tR2Pp2 AND c=R -> singleton(p2),
    t=tR2Pp2 AND c=P -> emptyset,
    t=t0p0 AND c=P   -> singleton(p0),
    t=t0p0 AND c=C   -> emptyset,
    t=t0p1 AND c=P   -> singleton(p1),
    t=t0p1 AND c=C   -> emptyset,
    t=t0p2 AND c=P   -> singleton(p2),
    t=t0p2 AND c=C   -> emptyset,
    t=t3p2 AND c=P   -> singleton(p2),
    t=t4p2 AND c=P   -> singleton(p2),
    t=t1p0 AND c=P   -> singleton(p0),
    t=t1p1 AND c=P   -> singleton(p1),
```

```
    t=t2p0 AND c=P   -> singleton(p0),
    t=t2p1 AND c=P   -> singleton(p1),
    t=t5p2 AND c=P   -> singleton(p2),
    t=tP2Rp0 AND c=P -> singleton(p0),
    t=tP2Rp0 AND c=R -> emptyset,
    t=tP2Rp1 AND c=P -> singleton(p1),
    t=tP2Rp1 AND c=R -> emptyset,
    t=tP2Rp2 AND c=P -> singleton(p2),
    t=tP2Rp2 AND c=R -> emptyset,
    t=tR2Ep0 AND c=R -> singleton(p0),
    t=tR2Ep0 AND c=E -> emptyset,
    t=tR2Ep1 AND c=R -> singleton(p1),
    t=tR2Ep1 AND c=E -> emptyset,
    t=tR2Ep2 AND c=R -> singleton(p2),
    t=tR2Ep2 AND c=E -> emptyset,
    ELSE             -> emptyset
  ENDCOND,
ce := LAMBDA (t:N),(c:C):
  COND
    t=tE2Rp0 AND c=E -> emptyset,
    t=tE2Rp0 AND c=R -> singleton(p0),
    t=tE2Rp1 AND c=E -> emptyset,
    t=tE2Rp1 AND c=R -> singleton(p1),
    t=tE2Rp2 AND c=E -> emptyset,
    t=tE2Rp2 AND c=R -> singleton(p2),
    t=tR2Pp0 AND c=R -> emptyset,
    t=tR2Pp0 AND c=P -> singleton(p0),
    t=tR2Pp1 AND c=R -> emptyset,
    t=tR2Pp1 AND c=P -> singleton(p1),
    t=tR2Pp2 AND c=R -> emptyset,
    t=tR2Pp2 AND c=P -> singleton(p2),
    t=t0p0 AND c=P   -> singleton(p0),
    t=t0p0 AND c=C   -> singleton(p1),
    t=t0p1 AND c=P   -> emptyset,
    t=t0p1 AND c=C   -> emptyset,
    t=t0p2 AND c=P   -> singleton(p2),
    t=t0p2 AND c=C   -> emptyset,
    t=t3p2 AND c=P   -> singleton(p2),
    t=t4p2 AND c=P   -> singleton(p2),
    t=t1p0 AND c=P   -> singleton(p0),
    t=t1p1 AND c=P   -> singleton(p1),
    t=t2p0 AND c=P   -> singleton(p0),
    t=t2p1 AND c=P   -> singleton(p1),
    t=t5p2 AND c=P   -> singleton(p2),
    t=tP2Rp0 AND c=P -> emptyset,
    t=tP2Rp0 AND c=R -> singleton(p0),
    t=tP2Rp1 AND c=P -> emptyset,
    t=tP2Rp1 AND c=R -> singleton(p1),
    t=tP2Rp2 AND c=P -> emptyset,
    t=tP2Rp2 AND c=R -> singleton(p2),
    t=tR2Ep0 AND c=R -> emptyset,
    t=tR2Ep0 AND c=E -> singleton(p0),
    t=tR2Ep1 AND c=R -> emptyset,
    t=tR2Ep1 AND c=E -> singleton(p1),
```

```
      t=tR2Ep2 AND c=R -> emptyset,
      t=tR2Ep2 AND c=E -> singleton(p2),
      ELSE             -> emptyset
    ENDCOND,
  ms := LAMBDA (r:R):
    COND
      r=E0 -> {x:M | TRUE},
      r=R0 -> emptyset,
      r=P0 -> emptyset,
      r=P1 -> emptyset,
      r=C0 -> emptyset,
      ELSE -> emptyset
    ENDCOND
#)
```

B.3 Conjectures

- ancestor
  1. anc(tE2Rp2) = $\emptyset$
  2. anc(tR2Ep2) = $\emptyset$
  3. anc(tP2Rp1) = $\emptyset$
  4. anc(c0p2) = {g0p2}
  5. anc(c0p2) = {g0p2}
  6. anc(g0p2) = $\emptyset$
  7. anc(t5p2) = $\emptyset$
  8. anc(t0p2) = {g0p2}
  9. anc(t3p2) = {c0p2, g0p2}
  10. anc(t4p2) = {c0p2, g0p2}
  11. anc(tP2Rp2) = $\emptyset$
  12. anc(t3p2) = {c0p2, g0p2}
- allSucc
  1. allsucc(tR2Ep2) = $\emptyset$
  2. allsucc(tP2Rp2) = {tR2Ep2}
  3. allsucc(t5p2) = {tP2Rp2, tR2Ep2}
  4. allsucc(g0p2) = {t5p2, tP2Rp2, tR2Ep2}
  5. allsucc(c0p2) = {t5p2, tP2Rp2, tR2Ep2}
  6. allsucc(t0p2) = {t5p2, tP2Rp2, tR2Ep2}
  7. allsucc(t4p2) = {t5p2, tP2Rp2, tR2Ep2}
  8. allsucc(t3p2) = {t4p2, t5p2, tP2Rp2, tR2Ep2}
  9. allsucc(tR2Pp2) = {g0p2, t5p2, tP2Rp2, tR2Ep2}
- mat
  1. mat(tR2Ep2) = {p2}
  2. mat(t0p1) = {p1}

3. mat(t3p2) = {p2}
4. mat(c0p2) = {p2}
5. mat(g0p2) = {p2}

- setUnion

1. setUnion(I(t0p1),t0p1,Ce) = ∅
2. setUnion(I(tP2Rp1),tP2Rp1,Ce) = {p1}
3. setUnion(I(t0p0),t0p0,Ce) = {p0, p1}

- successful

tp = n:N | n ∈ T ∧ (n=tR2Ep2 ∨ n=t3p2)

1. successful?(tR2Ep2, tp)
2. ¬ successful?(t5p2, tp)
3. ¬ successful?(c0p2, tp)
4. ¬ successful?(g0p2, tp)

- successor

1. successor(tR2Ep2) = ∅
2. successor(tP2Rp2) = {tR2Ep2}
3. successor(t3p2) = {t5p2, t4p2}
4. successor(g0p2) = {t5p2}

- initiated

1. initiated(tp) = ∅
2. initiated(tp) = {g0p2}
3. initiated(tp) = {c0p2, g0p2}

- unsafe

1. unsafe = {R0}

- succ

1. succ(tR2Ep2,P) = ∅
2. succ(tP2Rp2,P) = {tR2Ep2}
3. succ(t5p2,P) = {tP2Rp2}
4. succ(tE2Rp0,P) = {tE2Rp1, tR2Pp0}
5. succ(tE2Rp1,P) = {tE2Rp2, tR2Pp1}
6. succ(g0p2,P) = {t5p2}
7. succ(c0p2,P) = {t5p2}
8. succ(t3p2,P) = {t4p2}
9. succ(t4p2,P) = {t5p2}
10. succ(t0p2,P) = {t5p2}

- max

1. max(Ga(g0p2)) = 1

2. max({2,4,3}) = 4
3. max({3}) = 3
4. max({2,3}) = 3

- bypassed

1. ¬ bypassed(tE2Rp2, tp)
   tp = n:N | n ∈ T ∧ (n=tR2Ep2 ∨ n=t3p2)
2. ¬ bypassed(tE2Rp2, tp)
   tp: n:N | n ∈ T ∧ (n=tE2Rp2 ∨ n=tR2Ep2 ∨ n=t3p2)
3. bypassed(t0p2, tp)
   tp: n:N | n ∈ T ∧ (n=t5p2 ∨ n=tE2Rp2 ∨ n=tR2Ep2 ∨ n=t3p2)
4. NOT bypassed(c0p2, tp)
   tp: n:N | n ∈ T ∧ (n=tE2Rp2 ∨ n=tR2Ep2
   note: argument c0p2 not possible by definition of *bypassed*
5. bypassed(t0p2, tp)
   tp: n:N | n ∈ T AND (n=tE2Rp2 ∨ n=tR2Ep2 ∨ n=t3p2