Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 315–332
www.elsevier.com/locate/entcs

Testing for the Conformance of Real-time Protocols Implemented by Operating Systems

Dieter Zöbel, David Polock, Andreas van Arkel [1]

Fachbereich Informatik
Universität Koblenz-Landau
Koblenz, Germany

Abstract

The priority inversion problem arises when prioritized processes concurrently attempt to enter critical sections. This phenomenon results in extremely pessimistic estimations of worst case response times for real-time processes. Various protocols against priority inversion have been proposed in the literature and are available at system call level of operating systems and run-time executives. They belong to two major families of protocols: the priority inheritance protocols (PIP) and to the priority ceiling protocols (PCP). These protocols have in common that they allow to derive more optimistic worst case response times.
In contrast to the importance of this predictability aspect in the context of time-critical applications a lot of PIP- and PCP-implementations are not correct and permit the violation of time bounds. This article presents an effective and flexible tool set applied here for the validation of the implementations of protocols of the PCP-family. Besides the manual setup and instrumentation major parts of the black-box validation process are executed automatically.

Keywords: priority ceiling protocols, time-critical applications, real-time operating systems, semi-automated protocol validation

1 Introduction

A large number of operating systems and run-time executives are contending for the market of real-time and/or embedded applications. In this context the major strategy of advertisement is to demonstrate the completeness in fulfilling the essential real-time features. One of these is the availability of one or more protocols against the phenomenon of priority inversion.

[1] Email: zoebel@uni-koblenz.de

doi:10.1016/j.entcs.2004.08.071

From the application programmer's viewpoint there exists a strong preference for operating systems and run-time executives implementing a defined priority inheritance protocol or priority ceiling protocol. These permit the application of well established disciplines of engineering to achieve predictability for a given real-time system. The necessary parameters to do this consist of:

- the worst case execution times of processes
- the durations of execution sequences inside critical sections
- the basic priorities of the processes
- (for the PCP only) the preliminary knowledge about critical sections potentially used by certain processes

Based on this input the worst case response times of processes can be derived which are necessary to verify the feasibility of the schedule (see [4], [6], [11] and [18]).

However, this engineering approach suffers from two major drawbacks. The first results from the informal description of the protocols in the original article [10]. Though the protocols are proved correct an immediate implementation would allow priority inversion [7] and also violate basic theorems [19]. At this point it is necessary to declare that the intent of our paper is not to call the merits of the inventors [10] of the PIP and PCP into question. Instead, it should be noticed that the problem referenced here is a general one: there exists an informally specified protocol, a formal model of the execution of real-time processes, a proof which is correct but a protocol which is not. This obvious contradiction is solved by the perception – which is not so obvious – that in reality not the protocol is proved correct but some other formal model.

The second drawback for the engineering approach results from the fact that the implementations are unreliable. In the majority of cases the pretentious advertisement with the availability of PIP- or PCP-protocols manifests in oversimplified excerpts, strange protocol interfaces and various forms of errors and defects. Several examples are given in [9], e.g. contradicting to the announcement the real-time operating system iRMX [16] implements a deferring mechanism for the disinheritance of accumulated priorities. So, the protocols once introduced to gain predictability turn out to be sources of uncertainty in the hands of the vendors of operating systems. In certain cases slight modifications of the feasibility calculation permit to even out these deficits (as it is with iRMX mentioned above), whereas in other cases the implementation faults are irreparable causing unexpected violations of time bounds. The following table underlines that the majority of operating systems and also several run-time systems provide an API for the avoidance of priority inversion:

	PIP-family	PCP-family	—
operating	Virtuoso	uC/OS II	OSE
systems	iRMX	RTLinux	pSOS
	LynxOS	OSEK/VDX	RTAI
	Solaris	Solaris	...
	:	:	:
run-time	:	Ada 95	:
systems	On Time RTOS32	Real-Time Java	RTOS-UH Pearl

The tool set presented in the sequel is a contribution to regain some degree of certainty, whether an operating system offers predictable PIP- or PCP-protocols or not. The roots of our approach are not new. There is a variety of papers intending to verify the protocol in its own description, that is to say not applicable to external implementations (see e.g. [8] or [5]). In contrast, our approach goes further in several ways. First, in that the formal description is usable for several purposes: as a formal basis of a proof system, as unequivocal instruction for implementation and as input for a tool set to validate protocol implementations (see figure 1). Second, the same formal description serves as an operational specification for the protocol. This property can be used for the assertion of invariant protocol properties which are dynamically testable, thereby establishing certifiable criteria and avoiding the diffusion of informal implementation proposals, e.g. as given in [3]. Furthermore, the tool set consists of a system of components interacting via well defined data interfaces which allow to validate process interaction in a very general fashion and, hence, not only applicable to the PIP- and PCP-protocols.

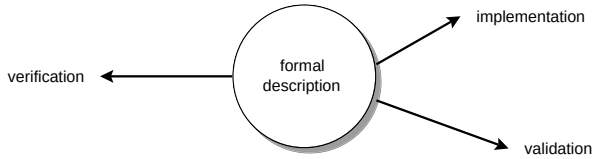

Fig. 1. The formal description as a basis for verification, implementation and validation

The paper has its emphasis on the formal protocol description, the components of the tool set operating on this description and the validation of protocol implementations, here exemplary explained with the PCP. Section 2 presents the formal description which has a substantial orientation towards a modular tool set for validation purposes. Additionally in section 2 it is mentioned how the formal description is used for protocol verification. Coarsely the process to validate protocol implementations is subdivided into two major steps, the generation of an execution space for parallel processes mapping to test suites in section 3 and the execution of instrumented real-time processes and their evaluation in section 4. The final section 5 assesses the results achieved so far

and argues that this approach may be applied beyond the range of protocols against priority inversion.

2 Interchangeable protocol format

The design of the formal description of protocols is a compromise between the versatility of the notation and the integration into various operative processes. The major operative process in our context is the test on conformance for certain operating systems. So, here the compromise consists of protocol specifications based on UML and the Z-specification notation [2], expressed in interchangeable XML-formats. More precisely from UML the class diagrams and the statecharts are used here. The meta-language which is behind their graphical representations is coded into document type definitions (DTD's) such that both class diagrams and statecharts are represented in XML. The architectural concept behind is borrowed from a proposal for the description of safety-critical systems [17] which is build upon three views: a structural, a reactive and a functional view.

In this context class diagrams represent the structural view. For the specification of protocols class diagrams describe the sets of relevant objects as well as their relations. Particularly for the protocols against priority inversion the relevant objects are the processes contending for critical sections which are administrated by a single scheduler (see figure 2). Any class diagram has a unique XML-representation for its attributes, methods and relations, e.g. for the scheduler's method `sigEnterCS` which is called when a process `p` tries to enter critical section `c` [2]:

```
     :
<class name="Scheduler">
     :
  <operation name="sigEnterCS"
             rtype=""
             rname=""
             observable="TRUE"
    <parameter name="p" type="Process" />
    <parameter name="c" type="CriticalSection" />
  </operation>
     :
</class>
     :
```

The reactive aspect of the protocol is specified by statecharts. Interaction in the case of these protocols happens between the active objects which are the processes and the scheduler. Based on a DTD statecharts are described in a unique notation, representing their nested structure, the states and the transitions from one state to another. E.g. for the method `sigEnterCS` from the

[2] The method `sigEnterCS` serves as an ongoing example throughout the paper.

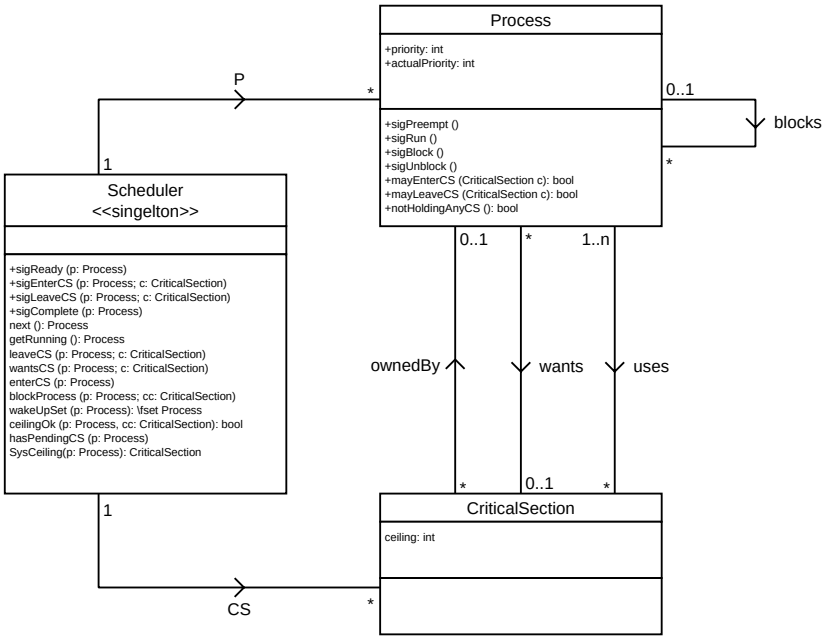

Fig. 2. Classes and associations

class diagram above there exists a corresponding transition in the statechart of the scheduler.

```
<statemachine name="Scheduler">
     :
  <transition name="T12"
            source="idle"
            target="wantsCS"
    <parameter name="p" type="Process" />
    <callevent call=sigEnterCS(p,c) />
  </transition>
     :
</statemachine>
```

With respect to transitions it has to be distinguished between receive events of the general form *["["guard"]"] op"("param")"* and send events of the form *["["guard"]"] "/" [[target"."] op"("param")"]*. The triggering of sequences of events happens when running processes enter or leave critical sections or run to completion. For instance the triggering event described by (see also figure 3)

```
[self.mayEnterCS(j)] / sched.sigEnterCS(self,j)
```

is guarded to verify, whether the process instance `p` is allowed to acquire the critical section `j`. If this is true the process sends a message to the instance `sched` of the scheduler class which has a corresponding receive event `sigEnterCS` in the statechart for the scheduler (figure 4). The sequence of

events terminates when a possibly new process continues execution in the status of running process.

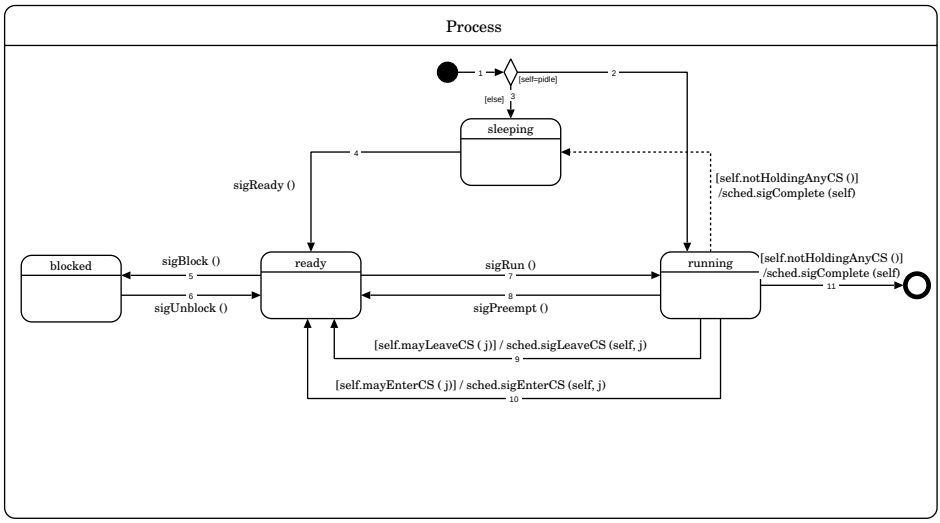

Fig. 3. Statechart for the processes

Another triggering event is real-time in a time triggered environment. This is modelled by the scheduler's transition `T10` attributed by the guard `[isSleeping(p)]` which may be fired as long as there are processes not set ready, yet.

Up to this point the class diagrams and statecharts specify the syntax of interactions between the processes and the scheduler. The semantics, particularly the prevention of forbidden interleavings, has to be expressed by a third means, here denoted by Z-specifications. Z was chosen for two major reasons, first the ease to specify the semantic behavior of the protocols in a mathematical fashion which also can be used for formal proofs. The second reason comes from the fact that there are several interpreters available to execute Z-specified protocols. So, a Z-interpreter is a valuable tool for the validation of protocol implementations.

Conforming to the structural and reactive models the Z-specification introduces the functional basis, here denoted *PCPSystem* for the PCP. The whole description is called schema and is subdivided in a syntactical part, defining the sorts and the kinds of mappings between them. For instance here is indicated that P is a finite set of processes and that *priority* is a mapping giving any process a unique priority. The following part is of semantic nature defining relations between values, for instance that the actual priority of a process is greater or equal to its basic priority:

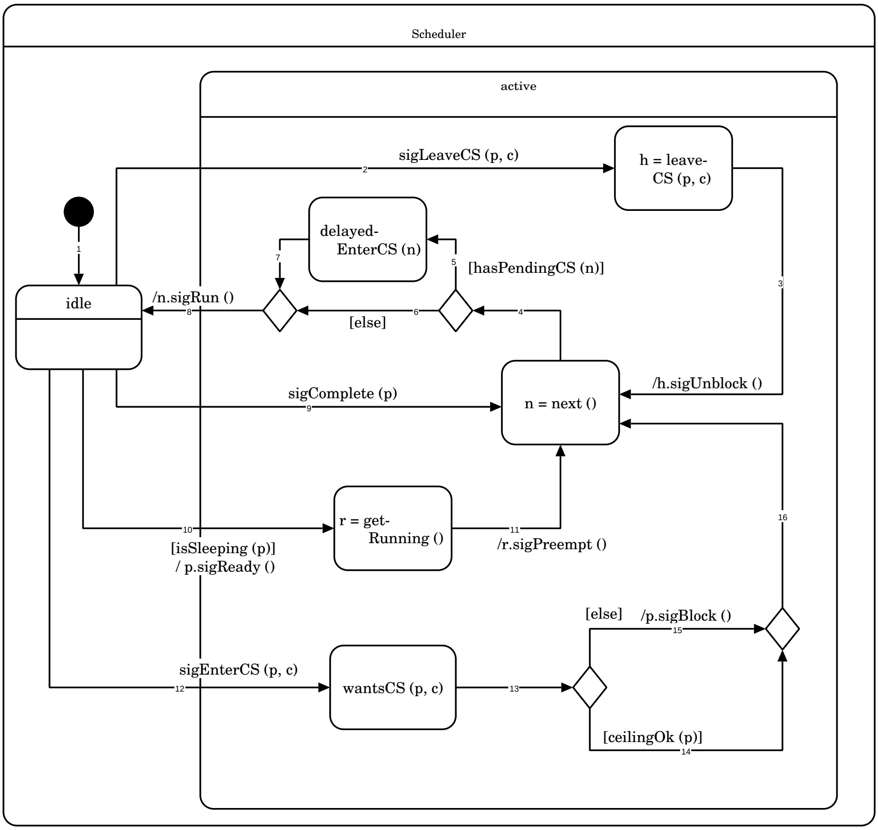

Fig. 4. Statechart for the scheduler

PCPSystem

$P : \mathbb{F}\, Process$
$CS : \mathbb{F}\, CriticalSection$
$priority : Process \rightarrowtail\!\!\!\!\!\!+\; \mathbb{N}$
$actualPriority : Process \nrightarrow \mathbb{N}$
$ceiling : CriticalSection \nrightarrow \mathbb{N}$
$ownedBy : CriticalSection \nrightarrow Process$
$wants : Process \nrightarrow CriticalSection$
$uses : Process \leftrightarrow CriticalSection$
$\vdots$

$\vdots$
$\forall\, p : P \bullet actualPriority(p) \geq priority(p)$
$ceiling = \{c : CS \bullet c \mapsto$
$\quad max(\{p : \mathrm{dom}(uses \rhd \{c\}) \bullet priority(p)\})\}$
$\vdots$

The PCP is highly reactive whereas the schema *PCPSystem* represents its static behavior only. So, it is necessary to define its dynamic behavior which corresponds to the transitions between states as depicted in the statecharts for the scheduler and the processes. Any transition, any guard and any state blob may modify the state space of *PCPSystem* and henceforth is represented by a dedicated Z-schema:

- Each transition of any statechart has a corresponding Z-schema, to denote the change of the state of a process or the state of the scheduler. For example by sending the `sigEnterCS`-message a process changes from the `running`-state to the `ready`-state. The respective Z-schema is `TRProcessT10`:

$$
\begin{array}{|l}
\hline
TRProcessT10 \\
\Delta PCPSystem \\
p? : Process \\
\hline
P = P' \\
p? \in P \\
CS = CS' \\
priority = priority' \\
stateProcess = stateProcess' \\
ownedby = ownedby' \\
wants = wants' \\
uses = uses' \\
stateProcess(p?) = running \\
stateProcess' = stateProcess \oplus \{p? \mapsto ready\} \\
\hline
\end{array}
$$

The notation $\Delta PCPSystem$ indicates the transition of the actual state *PCPSystem* to the new state $PCPSystem'$. The decisive change by $T10$ is reflected by the last two lines of the schema where in relation *stateProcess* the argument process $p?$ switches from the running to the ready state.

- Each guard has to be represented as a Z-schema returning a boolean value. For example the guard `self.notHoldingAnyCS()` has to be true when a process runs to completion.

$$
\begin{array}{|l}
\hline
notHoldingAnyCS \\
\Xi PCPSystem \\
p? : Process \\
\hline
p? \in P \\
p? \neq pidle \\
ownedBy \rhd \{p?\} = \{\} \\
\hline
\end{array}
$$

The notation $\Xi PCPSystem$ indicates that the state remains unchanged. The value *true* is returned if process $p?$ is outside of any critical section.

- Each state of any statechart potentially changes the overall system status. Particularly, this is true for the scheduler which is in charge to enforce the protocol semantics. As an example the `next`-schema has to determine and output which process $n!$ runs next:

next

$\Xi PCPSystem$

$n! : Process$

$stateProcess \rhd \{running\} = \{\}$

$n! \in P$

$stateProcess(n!) = ready$

$actualPriority(n!) = max(\{q : P \mid stateProcess(q) = ready \bullet$
$actualPriority(q)\})$

Clearly $n!$ is output to be the ready process with maximal actual priority.

- For technical reasons it is necessary that the free parameters of operations are instantiated, here with candidate processes or candidate critical sections. Corresponding Z-schemata are named according to the convention `GEN`*`ClassTransaction`* and invoked. The transaction name there leads to a certain method, whose parameters are indicated in the class diagram. A call to the `GEN`-schema is answered by the sequence of instantiating parameters. For example the call for `GENSchedulerT10` is invoked to determine the candidate processes which may become ready:

*GENSchedulerT*10

$p? : Process$

$p! : Process$

$isSleeping[p!/p?]$

Furthermore the characterizing invariant for the particular protocol should be expressed. This invariant simplifies the protocol verification, helps to detect errors in the formal description and monitors the validation process. In the case of the PCP a major predicate of the invariant states: When process $p1$ owns a critical section (relation *ownedBy*) , processes $p1$ and $p2$ both might use critical section $c2$ (relation *uses*), then the status of $c2$ is limited to be free or owned by $p1$.

PCPSystemInv

:

:

$stateScheduler = idle$

$\forall\, p1, p2 : P;\ c1, c2 : CS \mid (c1, p1) \in ownedBy \wedge$
$(p1, c2) \in uses \wedge (p2, c2) \in uses \bullet$
$\text{ran}\ (\{c2\} \lhd ownedBy) = \{\} \vee \text{ran}\ (\{c2\} \lhd ownedBy) = \{p1\}$

The invariant *PCPSystemInv* is strong enough to prove the fundamental properties of the system under the priority ceiling protocol, e.g.: A process $p1$ owning a critical section $c1$ will not be blocked under the PCP (corresponding to Lemma 9 in [10]). This property can be derived immediately from *PCPSystemInv*, because those processes (like $p2$) which are potentially able to block $p1$ (expressed by relation *uses*) are not allowed to own any conflicting critical section (as $c2$ would be). So, protocol verification on one hand has to derive the decisive properties which finally show the validity of timing properties. On the other hand the validity of the invariant *PCPSystemInv* has to established by the application of the formal description of the protocol. In detail this has to be verified in by the following steps:

(i) Initially the scheduler is in the state *idle* and *PCPSystemInv* is satisfied.

(ii) Any sequence of transitions leading from the *idle* state of the scheduler back to the *idle* state preserve the *PCPSystemInv*.

(iii) Any possible sequence of transitions is finite and triggered when the scheduler in state *idle*.

In contrast to the original approach the new one proves the protocols in their own terms.

3 The derivation of test suites

Let us imagine the possible benefits of a tool to assess the implemented protocols against priority inversion with respect to their real-time features. An ideal outcome of a test tool would be to give the application programmer the assertion that the invariant *PCPSystemInv* corresponding to the PCP holds for the real-time operating system under test (further designated as implementation under test IUT) or that some other, probably weaker assertion holds. But this imagination is still far from being realized in an effective and flexible tool set.

On the other hand there is so much literature available on conformance

testing and on the automatic generation of test suites. However, almost entirely in the focus of this literature are communication protocols which primarily lead to the derivation of test suites for two communicating sequential systems. Only a few scientific work has been done for conformance testing (e.g. [14] and [1]) and specification based testing (e.g. [13] and [12]) in the scope of real-time systems.

In contrast to these approaches the information which can be derived from the IUT is very tight. Status information about the execution of the protocol has to be provided by the actually running process which is executed in the context of a certain set of processes contending for a certain set of critical sections. In this context a test suite is a set of particular execution sequences which is run as test cases on the IUT.

A tool set based on test suites is as powerful as the assumption holds that a test suite represents the behavior of an equivalence class of computations. However, it may be that a piece of software is error prone to a certain input value n and works well for values $< n$ and for values $> n$. Errors of this kind cannot be caught by the way test suites are used here. Instead we look for the lowest value n where some representative course of execution is possible, e.g. to demonstrate the indirect inheritance of priority it suffices to consider 3 processes. By the observation that a test suite worked according to the specification for 3 processes some kind of confidence is given that the system under test also works well for any higher number $> n$. Hence, the proposed test suites only guarantee for correct implementations under the hypothesis that correctness is independent of running parameters.

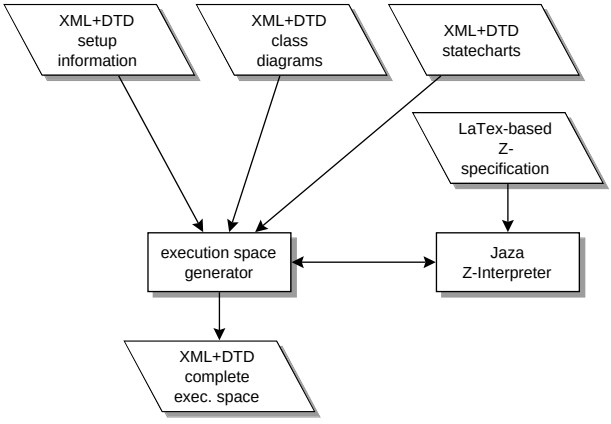

Fig. 5. The architecture of the execution space generator

As a preliminary step to the generation of test suites the execution space of the parallel processes has to be determined and analyzed. In this context

the execution space consists of a set of finite execution sequences of processes contending for critical sections. Two major components are used for this step: a syntax directed execution space generator and for the semantics an interpreter for Z-specifications. For the latter a Jaza-interpreter (see [15]) operates on the Z-schemata for a designated protocol version. The execution space generator constitutes a front end animating the Jaza-interpreter interactively or in batch mode. Setup information describes the test scenario consisting of the processes and their priorities, the critical sections and the protocol-specific information on class diagrams, statecharts and Z-schemata (see figure 5). E.g. for the event `sigEnterCS` which triggers transition *TRProcessT10* a sequence of corresponding Z-schemata is evaluated directed by the respective statecharts (track the transitions in figure 3 and 4):

$$
\begin{aligned}
&(mayEnterCS \wedge TRProcessT10)\\
&(TRSchedulerT12)\\
&(wantsCS)\\
(TRSchedulerT13)&(\neg ceilingOK \wedge TRSchedulerT15)(TRSchedulerT16)\\
&(next)\\
(TRSchedulerT4)&(hasPendingCS \wedge TRSchedulerT5)\\
&(delayedEnterCS)\\
(TRSchedulerT7)&(TRSchedulerT8)\\
&(TRProcessT7)
\end{aligned}
$$

Thereby a sequence of intermediate states has to be generated ending in method `sigRun` corresponding to transition *TRProcessT7*. For the derivation of test suites only the starting state and the final state are relevant to document which process wanted to enter a critical section and that possibly some other process has been chosen for execution due to the rules of the protocol.

From the viewpoint of validating a certain IUT only the running process is able to reflect the status of the system. Such states where the scheduler is idle and a certain process is running are named observable states. They are of major importance, because here

- the invariant *PCPSystemInv* which is an assertion for the set of processes has to be satisfied and
- the running process can output its identity and its actual priority

So, the sequence of protocol states can be compressed to the observable states which are really relevant for the subsequent steps of validation.

Besides the compression to the observable states a further reduction of the execution space can be applied. No true contention for critical sections is produced when a process gets ready in the presence of ready processes which

are all of higher priority. However, this reduction cannot prevent that the execution space grows dramatically, for example shrinks by this rule from $n!$ to $\sum_{i=1}^{n-1}(n-i)!$ different sequences for processes to become ready (where n is the number of processes). For any sequence which is left any process has $\sum_{j=0}^{m}\binom{m}{j}j!$ possibilities to order the acquisition of m critical sections. These simple calculations reveal that the execution space can be generated for low numbers of processes and critical sections only, even in the presence of compression and reduction. Additional policies are needed for the definition of observation focuses. Such a focus may start from a certain state reachable from the start and evolving thenceforth. To achieve this effect feedback with the execution space generator is needed. As final outcome the test suite is derived which is the set of all viable paths (explained in the next section) in this reduced execution space (see figure 6).

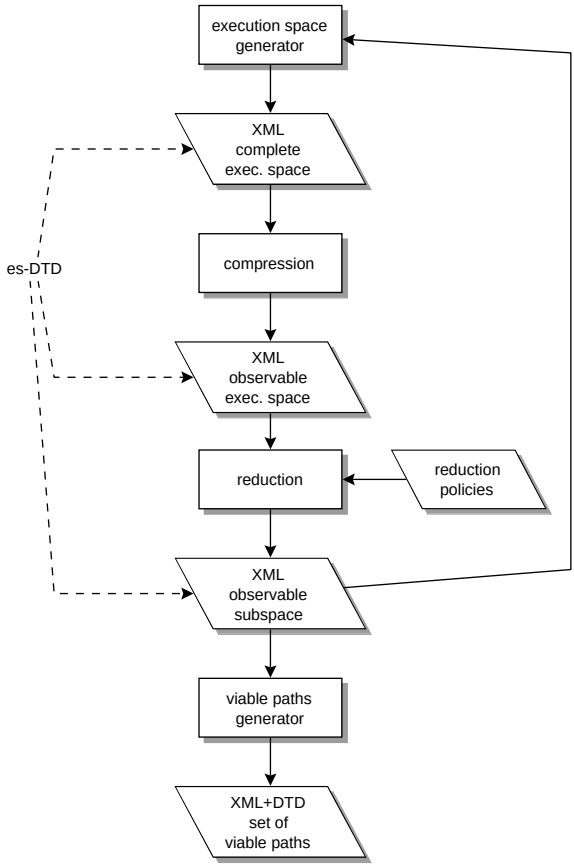

Fig. 6. The derivation of test suites

4 The validation process

The validation process is sequential, in testing one viable path of the test suite after the other. A viable path in this context consists of an execution sequence for each process augmented by the intent to interleave these processes in a certain way. So any process gets a particular basic priority and a ready time. Here e.g. the execution sequence for process P_2 with priority 12 and a relative ready time 4 eventually requesting critical section `b` [3]:

```
  :
<process name = "p2" priority="12">
<ready time="4"/>
<execute time="1"/>
<enter name="b"/>
<execute time="1"/>
<leave name="b"/>
<execute time="1"/>
<end/>
</process>
  :
```

All these operations of the execution sequence above are time triggered and scheduled to last one unit of time. Based on this assumption a viable path executed under a given protocol results in a unique interleaved execution sequence of all processes, henceforth called a test case. Both, viable path and test case are denoted in XML for purposes of interchangeability. To illustrate the nature of the test case it follows an excerpt beginning with the start of process P_2 at time 4 [4]:

```
  :
<exp time="4" process="2" priority="12"> <execute time="1"/> </exp>
<exp time="5" process="2" priority="12"> <enter name="b"/> </exp>
<exp time="6" process="1" priority="12"> <execute time="1"/> </exp>
<exp time="7" process="3" priority="14"> <execute time="1"/> </exp>
<exp time="8" process="3" priority="14"> <enter name="b"/> </exp>
  :
```

Notice that at time 6 process P_1 inherits priority 12. A respective graphical representation is more condensed and intuitive (see figure 7 [5]). To go a little bit deeper into the PCP, let us assume that P_1 is specified as a process operating on critical sections a and b. In the particular execution sequence

[3] The interchange format `es-DTD` describes the execution space for a certain set of processes and critical sections (see figure 6).

[4] The interchange format `tc-DTD` describes the the test cases consisting of viable paths (see figure 7).

[5] These diagrams are generated automatically. They indicate the actual priority of the running process. These priorities depend on inheritance, due to the blocking relations between processes. Blocking is introduced by the unsuccessful acquisition of critical sections. To make this evident the operations `enter` and `leave` are depicted by ascending and descending rectangles with the name of the critical section inside. An open, ascending rectangle indicates that the acquisition a critical section was not successful and led to a blocking state for the attempting process.

under test here P_1 does not enter b. However, conform to the PCP process P_2 has to be blocked when trying to enter critical section b. Later it can be seen that an existing implementation of the PCP produces a different test case thereby disclosing its non-conformity.

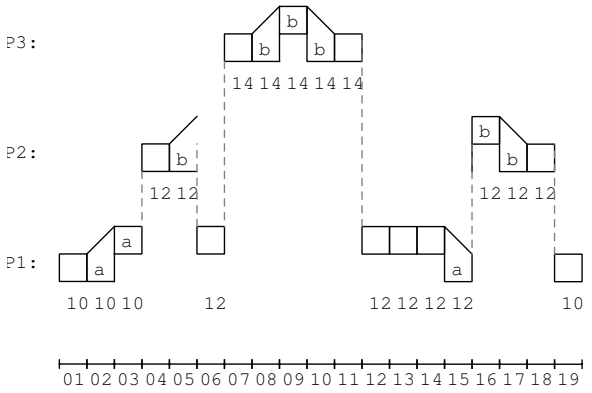

Fig. 7. The execution of the test case as specified by the PCP

Now time is ripe to focus on the IUT. Typically the operating system or run-time executive do not dispose of the standard operations `enter` and `leave` as called from the test case. An emulation of these protocol operations in terms of the IUT has to be coded manually. The critical sections as well as the processes have to be created. E.g. for Solaris a critical section to be requested under the PCP has to be created by attributing its ceiling value by the system call

```
pthread_mutexattr_setprioceiling()
```

and after initialization can be used by `ptread_mutex_lock` and `ptread_mutex_unlock` in place of `enter` and `leave`.

Furthermore all operations like `enter`, `leave` and `execute` have to be executed on the IUT in a time triggered fashion lasting one unit of time. The duration of this time unit has to be tuned at least to a value that the running process is able to reflect the actual state of the IUT. Here the decisive information consists of the process identity and its inherited priority. These activities are called instrumentation (see figure 8).

Sometimes a IUT does not provide system calls for the direct output of the actual (inherited) priority. For this case a further instrumentation with indicator processes is necessary prolonging the validation process.

Based on the viable path descriptions the processes are executed by the IUT and produce the same XML output format for their real behavior as for the conform behavior generated by he Jaza-interpreter. This makes it easy to

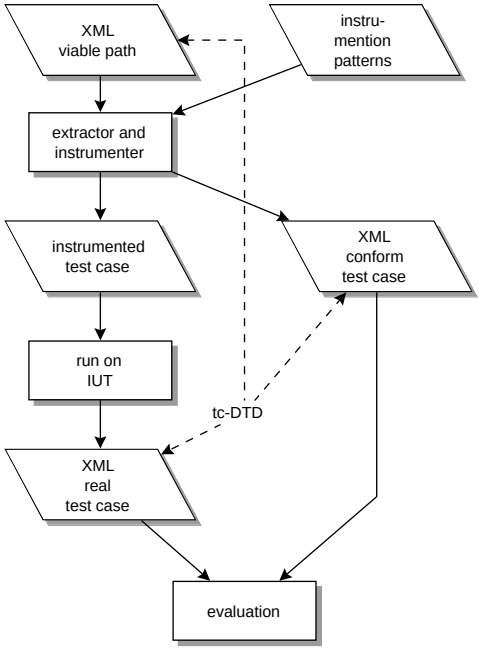

Fig. 8. The validation process

check for violations. Applied to the operating system Solaris which pretends to implement the PCP (as well as the PIP) the evaluation shows deviations. It can be seen (figures 7 and 9) that process P_2 is blocked by Solaris at a very early instant of time. The restriction to the interleaving of processes due to that protocol is stronger than requested by the *PCPSystemInv*-invariant. This observation leads to the supposition that Solaris implements the so called *highest locker protocol*. This protocol belongs to the PCP-family which is a simplification of the original PCP in limiting the degree of interleaving. The timeliness calculations of this version have to take into account that processes with a medium priority are delayed unnecessarily.

5 Conclusion

This paper describes a complex tool set for conformance testing applied to the PCP. Particularly the components for compression and reduction as well as for the evaluation have a prototypical character so far. Nevertheless several striking results are already available. The one above with Solaris as IUT is more or less representative for the general strategy to use the PCP as an attractive label and to implement some weaker protocol, here the highest locker protocol. The pitfall is that in general application programmers are

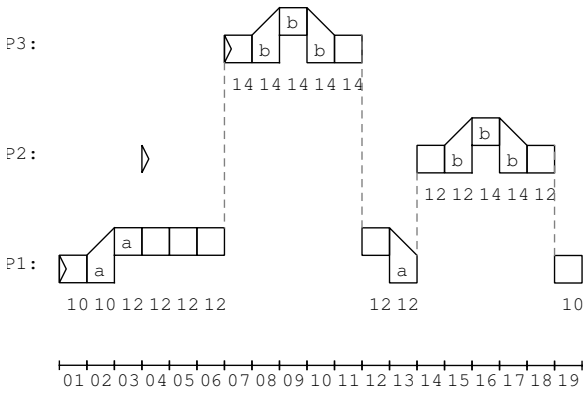

Fig. 9. The real test case produced by Solaris as IUT

not aware of this fact and its implications on timeliness calculations.

Even more striking are the results with respect to the PIP. On one hand none of the operating systems tested so far is conforming to the PIP-variant which satisfies all theorems associated with the PIP (corresponding to the original article [10]). On the other hand undetected flaws of the original protocol have been disclosed when designing the formal description for the PIP [19]. In this sense our tool set may be considered as one step to more trusted implementations in an application area where confidence is required. Due to the semi-automated validation process the experience is that an entire setup, instrumentation, test suite generation and evaluation can be completed in up to two days for a given operating system or run-time executive. This indicates: validation is not necessarily expensive.

As explained in this article the tool set has a sophisticated structure and a notable degree of automation. Several intermediate files are necessary. However, only two DTD-defined data formats (denoted `es-DTD` and `tc-DTD`) are used. This underlines the open source philosophy behind and should be understood as an animation for other scientists to supply improved components and to instrument this tool set for other purposes than the validation of protocols against priority inversion.

References

[1] Béatrice Bérard and Laurent Fribourg. Automated verification of a parametric real-time program: the ABR conformance protocol. In *International Conference on Computer Aided Verification (CAV'99)*, pages 96–107, Trento, Italy, July 1999. Springer Verlag.

[2] J. P. Bowen, A. Fett, and M. G. Hinchey. ZUM'98: The Z-formal specification notation. In *Lecture Notes in Computer Science (LNCS) 1493*, Berlin, Germany, September 1998. Springer.

[3] P. A. Buhr, A. S. Harij amd P. L. Lim, and J. Chen. Object-oriented real-time concurrency. *ACM SIG-PLAN Notices*, 35(10):29–46, October 2000.

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling, Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[5] Bruno Dutertre. Formal analysis of the priority ceiling protocol. In *IEEE Real-Time Systems Symposium (RTSS'2000)*, pages 151–160, Orlando, 2000.

[6] C.M. Krishna and Kang G. Shin. *Real-Time Systems*. McGraw-Hill Companies, New York, 1997.

[7] P. J. Moylan, R.E. Betz, and R.H. Middleton. The priority disinheritance problem. Technical Report EE9345, University of Newcastle, 1993.

[8] Michael Pilling, Alan Burns, and Kerry Raymond. Formal specifications of inheritance protocols for real-time scheduling. *Software Engineering Journal*, 5(5):263–279, 1990.

[9] David Polock and Dieter Zöbel. Conformance testing of priority inheritence protocols. In Danielle C. Young, editor, *Proceedings of the seventh International Conference on Real-Time Computing Systems and Applications (RTCSA'2000)*, pages 404–408, Cheju Island, South Korea, December, 12th -14th 2000. IEEE Computer Society.

[10] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[11] Alan C. Shaw. *Real-Time Systems and Software*. John Wiley and Sons, New York, 2001.

[12] Hendrik Thane, Anders Petterson, and Hans Hansson. Integration testing of fixed priority scheduled real-time systems. In Steve Liu Iain Bate, editor, *IEEE/IEE Real-Time Embedded Systems Workshop*, London, December 2001.

[13] Andreas Ulrich and Hartmut König. Specification-based testing of concurrent systems. In *IFIP Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'97)*, Osaka, Japan, November 1997.

[14] M. Ümit Uyar and alii. Generation of realizable conformance tests under timing constraints. In *MILCOM'98*, Boston, October 1998.

[15] U. Utting. *Jaza User Manual and Tutorial*. Online Manual Version V0.86, Univeristy of Waikato, 2000.

[16] Ch. Vickery. *Real-Time and Systems Programming for PCs*. McGraw-Hill, Blue Ridge Summit, PA, 1993.

[17] Matthias Weber. Combining statecharts and Z for the design of safety-critical control systems. In Marie-Claude Gaudel and James Woodcock, editors, *Industrial Benefits and Aavances in Formal Methods (FME'96)*, volume 1051 of *LNCS*, pages 307–326. Springer-Verlag, March 1996.

[18] Dieter Zöbel and Wolfgang Albrecht. *Echtzeitsysteme - Grundlagen und Techniken*. Lehrbuch. International Thomson Publishing Company, Bonn, Albany, 1995.

[19] Dieter Zöbel and David Polock. Priority inversion revisited. In Joël Goossens, editor, *12th International Conference on Real-Time Systems (RTS'2004)*, pages 190–203, Paris Expo, March 2004. Birp, 11, rue du Perche, 75003 Paris.