Analysis of ACS using mCRL2

Bas Ploeger

Eindhoven University of Technology
Department of Mathematics and Computer Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
s.c.w.ploeger@tue.nl

1 Introduction

The Atacama Large Millimeter Array (ALMA) is a space observatory system that is currently being developed by astronomical organizations in Europe, North-America and Japan. It will consist of at least 50 antennas that operate in the millimeter and submillimeter wavelength range, and will be built in the Chilean Atacama desert at an altitude of approximately 5000 meters.

The ALMA Common Software (ACS) is a software infrastructure for ALMA that is based on CORBA. It is an essential framework within the ALMA software system, providing a collection of common software components and services that other parts of the system rely upon. Therefore, it is important for the ACS platform to be stable and reliable.

The purpose of our study is to analyse a part of the ACS framework using formal modelling techniques. In that approach, a model of a software system is constructed, which is subsequently analysed for errors by an exhaustive exploration of all possible states and execution sequences. The aim is to cover a wider range of cases and scenarios than are typically covered by a collection of test cases. We introduce this approach and the modelling framework mCRL2 in more detail in Section 2. The model that we created for ACS is described in Section 3 and the results of the analysis are described in Section 4. The report is concluded in Section 5.

Acknowledgements. We are grateful to the ALMA staff at Garching, Germany, in particular to Heiko Sommer, Joe Schwarz, Gianni Raffi and Matej Sekoranja for their kind support and fruitful collaboration.

2 Modelling Concurrent Systems in mCRL2

A concurrent system is a system in which a number of processes run in parallel. A process can be fully sequential (meaning it performs actions in sequence only) or consist, again, of a number of concurrent processes. The processes interact or communicate with each other and their environment to accomplish complex tasks. Interaction between processes is accomplished by passing messages. This

can be done synchronously, *i.e.* messages are sent and received at the same time, or asynchronously, *i.e.* messages are received later than they are sent (and are temporarily stored in a buffer in the mean time). Many large software systems today are designed as concurrent systems, including the software in the ALMA system.

For describing the behaviour of concurrent systems various languages have been proposed, including statecharts [7], Petri nets [10, 11] and process algebras [1, 8, 9]. These languages are *specification languages*: they allow to specify the behaviour of a system at a higher, more conceptual level than programming languages. This facilitates studying systems at an abstract level without worrying about implementation details. In this way, it can be checked whether a system contains conceptual flaws or design errors, even before it has been fully implemented. The specification of a system's behaviour on this higher level is usually called a *model*, as it typically abstracts away from certain details.

For the ACS analysis we use the specification language mCRL2 [4, 5], which is based on the process algebra ACP (Algebra of Communicating Processes) [2, 3]. It is accompanied by a toolset that supports the analysis of mCRL2 models. More information about mCRL2 can be found at http://www.mcrl2.org.

We identified a subset of ACS functionality of which we constructed a model based on discussions with ALMA engineers, documentation and source code. This model is analysed using the mCRL2 tools, where we focus on checking for deadlocks. Any error found in the model is discussed with the engineers, as it may not indicate an error in the real system. Modelling errors are not uncommon, especially in the first couple of draft versions of a model. By improving a model iteratively, a deeper understanding of the system is gained step by step until the model adequately captures the essence of the real system's behaviour.

3 ACS Model

We focus on the Manager and the startup sequence of components and containers. More specifically, we investigate whether the simultaneous handling of getComponent calls by different Manager threads can lead to problems, due to synchronization between the threads. We briefly describe the model at a high level here. The complete model is included in Appendix A. That model contains two Manager threads (which corresponds with a thread pool of size 2), one container and two components. Note that more instances of these processes can be added easily.

3.1 Manager

The Manager stores data that are shared among its threads, like the set of components that are currently active. The synchronization of accesses to these resources (by locking and freeing) is important and should be modelled correctly. We model every resource as a separate process in mCRL2. In particular, the following processes exist:

- ManagerActSync: The activation synchronization lock.
- ManagerComponents: The set of components.
- ManagerContainers: The set of containers.

Each of these processes allows its resource to be locked and freed by Manager threads and grants access to the resource by the locking thread. The behaviour of a Manager thread is modelled by the following processes:

- ManagerThread: A thread that responds to getComponent and container login requests.
- MT_ActivateComponent: The procedure for activating a component (called by ManagerThread).

Every instance of ManagerThread has a unique identifier, which allows to distinguish actions performed by one instance from those performed by an other instance, *e.g.* when simulating the model.

3.2 Container

The Container process models a container. For the purpose of our model, this is a relatively simple process. A container can be in any of three states: *dead*, *loggingin* and *running*. In the latter state it can activate a component when requested by the Manager.

3.3 Component

Components are modelled by the Component process. They can be in any of two states: *dead* and *running*. The Component process is simple for our purposes.

3.4 Monitors

In Java, any object can be used for synchronization via its wait and notifyAll methods. The model includes a Monitor process that provides precisely this behaviour. It maintains a list of processes that are currently waiting for a notification, and can notify them when this notification has arrived. The latter is done via a call to the Monitor_NotifyAll process.

4 Results

The model of Appendix A is deadlock-free. Its state space consists of 3.484 states and 9.832 transitions and is visualized in Figure 1.

The deadlock-freedom depends strongly on the possibility to timeout on certain locks: if any of the timeout actions in the ManagerThread process are omitted, deadlocks are possible. These timeouts apply when acquiring an Activation Synchronization (ActSync) lock, which happens in two places:

1. When activating a component (right after a getComponent call).
2. When starting up a container (right after it has been determined that the container does not yet exist).

We now describe the deadlock situations that can occur when either of these timeouts would be omitted from the system, and discuss these situations afterwards.

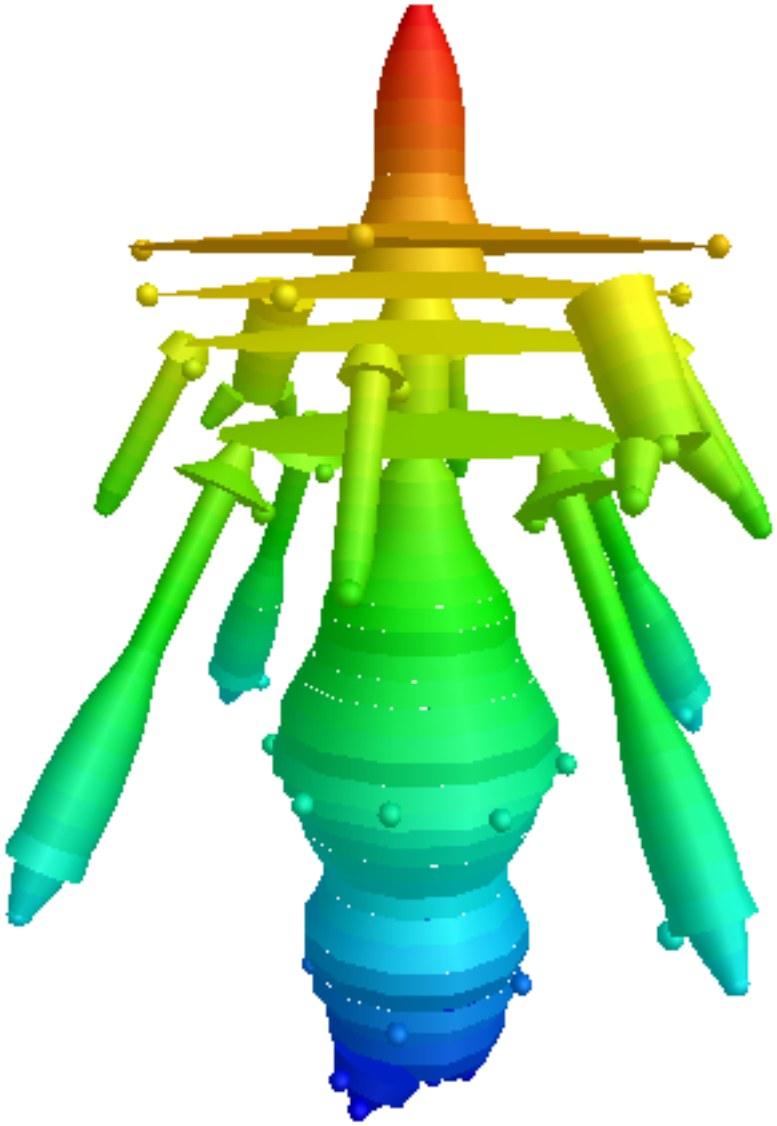

Figure 1: Visualization of the state space of the ACS model of Appendix A. The behaviour starts at the top and proceeds downwards in phases. At some points it may branch off into a separate part of the behaviour. Though not visualized here, it is possible to escape from these branches or move back up into an earlier phase. More information on the visualization technique can be found in [6, 12].

Omitting timeout 1. If the first timeout would be omitted, the following scenario is possible. Both Manager threads receive a getComponent request for the same component, say Comp1. The container on which this component is configured to run, Cont1, has not yet been started and has to be autostarted by the ACS framework. One of the threads acquires the ActSync lock for Comp1, determines that Cont1 has to be started, acquires the ActSync lock for Cont1 as well, sends a startup message to Cont1 and finally waits until Cont1 reports that it has been logged in. Meanwhile, the other thread blocks as it cannot acquire the ActSync lock for Comp1. Now, both Manager threads are waiting and no more threads are available to handle the login procedure of Cont1. Hence, the system has reached a deadlock.

Omitting timeout 2. If the second timeout would be omitted, the following scenario is possible. The Manager threads receive a getComponent request for components Comp1 and Comp2, respectively. Both components belong to the same container, Cont1. The scenario is then almost identical to the one described above, except that the second thread blocks on obtaining the ActSync lock for Cont1, instead of Comp1.

Discussion. As mentioned before, the deadlocks are not present in the current system because the timeouts ensure that a thread is freed after a certain amount of time. This amount of time is deliberately chosen to be not too tight in order to allow the system to proceed normally. The disadvantage is that the performance or responsiveness of the system may deteriorate if timeouts occur too frequently. Therefore, timeouts are used as a "last resort" in the current system, and they are generally not intended to be relied upon heavily.

As a possible other way to prevent the deadlocks described above, one might consider increasing the size of the thread pool. In the real system, the thread pool size is on the order of 100 threads. Theoretically, however, this does not solve the problem: given a thread pool of any finite size N, the deadlocks can be reached if N requests for the same component (or for components on the same container) arrive at the Manager. In case of an unlimited thread pool, the deadlocks cannot be reached under the assumption that only finitely many such requests will ever be handled by the Manager at any given time.

Another possible approach would be to make the getComponent requests asynchronous, so that a pending request for a certain component no longer occupies a thread in the Manager unnecessarily. This may adequately resolve the issue, even in the theoretical case. As it constitutes a rather fundamental change in the system, care must be taken when implementing this approach. Ideally the existing model is adapted first to verify that deadlocks are prevented indeed.

5 Conclusions

We have analysed a part of the ACS system by applying formal modelling techniques. We focused on the ACS Manager and the startup of components, along with their associated containers. The model was constructed based on discussions with engineers, the documentation and source code, and subsequently

analysed for deadlocks. No deadlocks were found. We found timeouts to be essential for deadlock-freedom of the system and have reflected on this issue.

As a model is an abstraction from reality, a caveat is in place: correctness of the model does not necessarily imply correctness of the real system. Apart from this, it is the responsibility of the analyst to ensure that the model faithfully describes the system's behaviour. Once this is taken care of adequately, formal modelling and verification techniques provide a powerful means to analyse a system by considering all possible states and execution sequences, including the ones that are not typically covered by test cases. As such, it is an important addition to the collection of software analysis techniques.

ALMA engineers noted that the process of constructing the model was already valuable by itself. Because the model has to be as accurate as possible, a thorough understanding of the system is necessary. For this, detailed questions about the system were asked to the engineers, which triggered them to reflect on certain design decisions or aspects of the implementation. This type of feedback was greatly appreciated.

Continuing on this, we found that the best way to gain deeper understanding of the system was to study the source code and ask questions to engineers. The documents were useful for gaining a high-level overview of the system, but were found to be not detailed or accurate enough for our purposes. Ideally, the documentation contains more precise descriptions or specifications of important parts of the system. By including models in the documentation (in whatever formalism or language) the system's behaviour can be described more precisely and unambiguously, which eases the communication between current engineers and towards novices.

Due to a limited amount of time, we have only modelled a small part of the Manager's behaviour. Therefore, extension of the model to include more aspects of its behaviour is an interesting direction. A natural extension would be to include the release of components and shutdown of containers. Also, more involved properties than deadlock-freedom can be considered for checking on the model. Examples of such properties are that the Manager never attempts to activate an activated component, and that a component is never released before it has been activated. Though it may seem trivial for such properties to hold, it is the author's personal experience that checking a number of them on a model can lead to suprising insights and results.

References

[1] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[2] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proc. 11th International Colloquium on Automata, Languages and Programming (ICALP 1984)*, volume 172 of *LNCS*, pages 82–94. Springer, 1984.

[3] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[4] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2007.

[5] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, pages 99–128. CRC Press, 2008.

[6] F. van Ham, H. van de Wetering, and J. J. van Wijk. Interactive visualization of state transition systems. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):319–329, 2002.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[10] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[11] C. A. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4), 2008.

[12] B. Ploeger and C. Tankink. Improving an interactive visualization of transition systems. In *Proc. 4th ACM Symposium on Software Visualization 2008 (SoftVis 2008)*, pages 115–124. ACM, 2008.

A mCRL2 Model

```
% Specification of the ACS Manager along with Containers and Components.
% Author: Bas Ploeger

% Data sort for the state of a container or component
sort State = struct dead | loggingin | running;

% Data sort for identifying the various processes in the system
% (Manager threads, components, etc.)
sort ID = struct none | Comp1 | Comp2 | Cont1 | MT1 | MT2 | ContLogin;

% Functions for checking if an ID indicates a component/container
map  is_component: ID -> Bool;
     is_container: ID -> Bool;
var  x: ID;
eqn  is_component(x) = (x in {Comp1,Comp2});
     is_container(x) = (x in {Cont1});

% Function returning the container of a component
map  getCont: ID -> ID;
eqn  getCont(Comp1) = Cont1;
```

```
     getCont(Comp2) = Cont1;

% Function for adding elements to a list of IDs. It is ensured that the
% list contains no duplicates and remains sorted.
map  add_sorted: ID # List(ID) -> List(ID);
var  c,d:ID;
     L:List(ID);
eqn  add_sorted(c,[]) = [c];
     add_sorted(c,d |> L) =  if( c < d , c |> d |> L ,
           d |> if(c==d , L , add_sorted(c,L)) );

%%% Action declarations %%%

act

% For container life cycle
rcv_startup_container, snd_startup_container, startup_container,
rcv_login, snd_login, login, rcv_authenticate, snd_authenticate,
authenticate, rcv_auth_ok, snd_auth_ok, auth_ok, rcv_auth_not_ok,
snd_auth_not_ok, auth_not_ok: ID # ID;

% For component life cycle
rcv_activate_component, snd_activate_component, activate_component: ID #
ID # ID;
rcv_create, snd_create, create, rcv_initialize, snd_initialize,
initialize, rcv_execute, snd_execute, execute: ID # ID;

% For accessing/manipulating data stored in the Manager
rcv_has_component_man, snd_has_component_man, has_component_man,
rcv_no_component_man, snd_no_component_man, no_component_man,
rcv_add_component_man, snd_add_component_man, add_component_man,
rcv_has_container_man, snd_has_container_man, has_container_man,
rcv_no_container_man, snd_no_container_man, no_container_man,
rcv_add_container_man, snd_add_container_man, add_container_man,
rcv_acquire_actsync_lock, snd_acquire_actsync_lock,
acquire_actsync_lock, rcv_release_actsync_lock,
snd_release_actsync_lock, release_actsync_lock: ID # ID;

% For synchronization of access to data stored in the Manager
rcv_free_container, snd_free_container, free_container,
rcv_lock_container, snd_lock_container, lock_container,
rcv_free_component, snd_free_component, free_component,
rcv_lock_component, snd_lock_component, lock_component: ID;

% For synchronization on monitor objects
rcv_wake_up, snd_wake_up, wake_up, rcv_notify_all, snd_notify_all,
notify_all, rcv_wait, snd_wait, wait: ID # ID;
timeout: ID;

% External triggers
get_component: ID # ID;


%%% Process definitions %%%
```

```
% A monitor object
proc Monitor(m:ID,waitlist:List(ID)) =
     sum n:ID .
     ( rcv_wait(n,m) . Monitor(m,add_sorted(n,waitlist))
       +
       rcv_notify_all(n,m) .
       ( (waitlist == []) -> Monitor(m,waitlist)
                          <> Monitor_NotifyAll(m,waitlist) ) );

proc Monitor_NotifyAll(m:ID,waitlist:List(ID)) =
     snd_wake_up(head(waitlist),m) .
     ( (tail(waitlist) != []) -> Monitor_NotifyAll(m,tail(waitlist))
                              <> Monitor(m,[]) );

% Manager

% The activation synchronization locks
proc ManagerActSync(actsync:Set(ID)) =
     sum m,n:ID .
     (n in actsync) -> rcv_release_actsync_lock(m,n)
                       . ManagerActSync(actsync - {n})
                    <> rcv_acquire_actsync_lock(m,n)
                       . ManagerActSync(actsync + {n});

% The component store
proc ManagerComponents(comps:Set(ID),lock:ID) =
     (lock == none) ->
     sum m:ID . snd_lock_component(m) . ManagerComponents(comps,m)
     +
     rcv_free_component(lock) . ManagerComponents(comps,none)
     +
     sum c:ID . is_component(c) ->
     ((c in comps) -> snd_has_component_man(lock,c)
                      . ManagerComponents(comps,lock)
                   <> snd_no_component_man(lock,c)
                      . ManagerComponents(comps,lock)
       +
       rcv_add_component_man(lock,c)
       . ManagerComponents(comps + {c},lock) );

% The container store
proc ManagerContainers(conts:Set(ID),lock:ID) =
     (lock == none) ->
     sum m:ID . snd_lock_container(m) . ManagerContainers(conts,m)
     +
     rcv_free_container(lock) . ManagerContainers(conts,none)
     +
     sum c:ID . is_container(c) ->
     ( (c in conts) -> snd_has_container_man(lock,c)
                       . ManagerContainers(conts,lock)
                    <> snd_no_container_man(lock,c)
                       . ManagerContainers(conts,lock)
       +
```

```
       rcv_add_container_man(lock,c)
       . ManagerContainers(conts + {c},lock) );

% Manager thread
proc ManagerThread(m:ID) =
     sum c:ID . is_component(c) ->
     ( get_component(m,c)
       .(timeout(m)
         +
         snd_acquire_actsync_lock(m,c)
         . rcv_lock_component(m)
         .(rcv_has_component_man(m,c)
           . snd_free_component(m)
           +
           rcv_no_component_man(m,c)
           . snd_free_component(m)
           . rcv_lock_container(m)
           .(rcv_has_container_man(m,getCont(c))
             . snd_free_container(m)
             . MT_ActivateComponent(m,c)
             +
             rcv_no_container_man(m,getCont(c))
             . snd_free_container(m)
             .(timeout(m)
               +
               snd_acquire_actsync_lock(m,getCont(c))
               . snd_startup_container(m,getCont(c))
               . snd_wait(m,ContLogin)
               . rcv_wake_up(m,ContLogin)
               . snd_release_actsync_lock(m,getCont(c))
               . MT_ActivateComponent(m,c)
              )
            )
          )
         . snd_release_actsync_lock(m,c)
        )
       . ManagerThread(m)
     )
     +
     sum c:ID . is_container(c) ->
     ( rcv_login(m,c)
       . snd_authenticate(m,c)
       .(snd_auth_not_ok(m,c)
         +
         snd_auth_ok(m,c)
         . rcv_lock_container(m)
         .(rcv_has_container_man(m,c)
           +
           rcv_no_container_man(m,c)
           . snd_add_container_man(m,c)
          )
         . snd_free_container(m)
         . snd_notify_all(m,ContLogin)
        )
```

```
        . ManagerThread(m)
      );

% Activate a component
proc MT_ActivateComponent(m:ID,c:ID) =
      snd_activate_component(m,getCont(c),c)
      . rcv_lock_component(m)
      . snd_add_component_man(m,c)
      . snd_free_component(m);

% Container

proc Container(c:ID, s:State, comps:Set(ID)) =
      sum m:ID . (
      (s == dead) -> rcv_startup_container(m,c)
                     . Container(c,loggingin,comps)
      +
      (s == loggingin) ->
        snd_login(m,c)
        . rcv_authenticate(m,c)
        .( rcv_auth_ok(m,c) . Container(c,running,comps)
           +
           rcv_auth_not_ok(m,c) . Container(c,dead,comps) )
      +
      (s == running) ->
        sum d:ID . is_component(d) ->
        rcv_activate_component(m,c,d)
        . snd_create(c,d)
        . snd_initialize(c,d)
        . snd_execute(c,d)
        . Container(c,s,comps + {d}) );

% Component

proc Component(c:ID,s:State) =
      (s == dead) ->
        sum d:ID .
        rcv_create(d,c)
        . rcv_initialize(d,c)
        . rcv_execute(d,c)
        . Component(c,running);

%%% Initial process specification %%%

init

% Allow only these actions, block all others
allow(
{ login, authenticate, auth_ok, auth_not_ok, startup_container,
  activate_component, create, initialize, execute, get_component,
  has_component_man, no_component_man, add_component_man,
  has_container_man, no_container_man, add_container_man,
  acquire_actsync_lock, release_actsync_lock, free_container,
  lock_container, free_component, lock_component, notify_all, wake_up,
```

```
  wait, timeout },

% Synchronously communicating actions
comm(
{ rcv_login|snd_login -> login,
  rcv_authenticate|snd_authenticate -> authenticate,
  rcv_auth_ok|snd_auth_ok -> auth_ok,
  rcv_auth_not_ok|snd_auth_not_ok -> auth_not_ok,
  rcv_startup_container|snd_startup_container -> startup_container,
  rcv_activate_component|snd_activate_component -> activate_component,
  rcv_create|snd_create -> create,
  rcv_initialize|snd_initialize -> initialize,
  rcv_execute|snd_execute -> execute,
  rcv_has_component_man|snd_has_component_man -> has_component_man,
  rcv_no_component_man|snd_no_component_man -> no_component_man,
  rcv_add_component_man|snd_add_component_man -> add_component_man,
  rcv_has_container_man|snd_has_container_man -> has_container_man,
  rcv_no_container_man|snd_no_container_man -> no_container_man,
  rcv_add_container_man|snd_add_container_man -> add_container_man,
  rcv_acquire_actsync_lock|snd_acquire_actsync_lock -> acquire_actsync_lock,
  rcv_release_actsync_lock|snd_release_actsync_lock -> release_actsync_lock,
  rcv_free_container|snd_free_container -> free_container,
  rcv_lock_container|snd_lock_container -> lock_container,
  rcv_free_component|snd_free_component -> free_component,
  rcv_lock_component|snd_lock_component -> lock_component,
  rcv_wake_up|snd_wake_up -> wake_up,
  rcv_notify_all|snd_notify_all -> notify_all,
  rcv_wait|snd_wait -> wait },

% The parallel processes that constitute the system, along with their
% initial states

ManagerActSync({}) || ManagerComponents({},none)
|| ManagerContainers({},none)
|| ManagerThread(MT1) || ManagerThread(MT2)
|| Container(Cont1,dead,{})
|| Component(Comp1,dead)
|| Component(Comp2,dead)
|| Monitor(ContLogin,[])

));
```