Engineering Human Flows in Smart Environments using Formal Techniques

FULL VERSION

*

Michael Harrison[1], Mieke Massink[2], and Diego Latella[2]

[1] School of Computing Science, Newcastle University
Newcastle upon Tyne, UK
`Michael.Harrison@ncl.ac.uk`

[2] Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR
Area della Ricerca, Via G. Moruzzi 1, Pisa, Italy
{`mieke.massink,diego.latella`}`@isti.cnr.it`

Abstract. While the cost of deploying a smart environment is likely to be high, the benefits of such systems are hard to quantify and predict. The potential for expensive failure is therefore considerable. This paper is concerned with how formal models of smart environments might be used to explore the consequences of the technology on users of the environment before deployment. Alternative models of interaction within smart environments are explored. The first focuses on the individual within the environment and the second provides an exploration of the impact of the designed environment on collective behaviour. It applies a recent approach that provides a quantitative analysis of systems with a very large number of entities with similar behaviour. The paper argues that there are particular properties relating to implicit interaction in immersive systems and to issues of crowd system interaction and discusses how these properties should be analysed. The relationship between these models is discussed and an agenda is established for tool supported methodology. This paper is concerned with an engineering approach to the analysis of human aspects of smart environments.

Key words: Formal Methods, Model-based usability analysis, Performance Evaluation Process Algebra, Ordinary Differential Equations,
Dynamic Signage Systems

1 Introduction

Smart environments can improve the experience and collective behaviour of users within physical spaces. They achieve improvement by using information about people in the space, by tracking them using technologies such as Bluetooth or RFID, by presenting timely and relevant information either using public displays or personal devices. They can infer action from the individual's context and they can influence collective behaviour when there are many people in the environment. A practical consequence of an effective smart environment design is to improve an individual's experience of the environment by providing more effective information and by responding intelligently to collective demands for resources.

To engineer a design efficiently that works for users both individually and collectively, techniques are required early in the design to model and analyse the effects of design decisions. The

* This work has been partially funded by the Italian MIUR/FIRB project tocai.it, by the EU project Resist/Faerus (IST-2006-026764), by the EU project Sensoria (IST-2005-016004) and by the Italian CNR/RSTL project XXL.

aim of early analysis is to predict the impact that the system will have before deployment and to provide information that can be used formatively in later refinements or versions of the design. While smart environments share many features with more conventional systems they also present novel features. This paper is concerned to explore and discuss these novel features using formal techniques. Firstly user interaction within smart environments is often an implicit background activity. The system will respond to changes in the user's context, for example as the user moves around the space the environment provides relevant information proactively. The system may also predict what the user's next action should be and do it, or at least partially complete it. For this action to be effective, information should be provided when and where the user needs it. The value and appropriateness of the information may be crucial to the user's experience of the environment (consider the frustration of sitting in a stationery train without information about why).

Secondly, a user's behaviour in a smart environment can be affected by other people that are also present within the environment. Waiting for the screening of carry-on baggage in an airport, for example, is affected by the queue and the behaviour of others in the queue, whether people in the queue are aware of the fact that they need to remove their laptops, or coats, or shoes or mobile phones, for example. Inappropriate action by others will affect the individual's experience. A system that can modify collective behaviour will affect the experience of the individual user by, for example, encouraging preparation for the screening process, smoothing the use of different scanning machines and so on.

Both these novel features are about human interaction in the smart environment and bring a new perspective to usability engineering. Neither fits well into existing approaches to human computer interaction. In this report several formal approaches are explored that could be used to engineer human aspects of smart environment design. The results that different modelling approaches can produce, both quantitative and qualitative, are illustrated.

A fundamental problem with formal modelling in relation to analyses of collective behaviours is how to deal with the state explosion that arises through attempts to apply model-checking techniques to models composed of multiple instances of processes required to define the collective behaviour. A recently proposed scalable model-based technique, Fluid Flow Analysis [14] is explored to analyse collective behaviour. This technique supports the analysis of many replicated entities with autonomous behaviour that collaborate by means of forms of synchronisation. It builds upon a process-algebraic approach and adds techniques for quantitative analyses to those for behavioural analysis. The technique has been successfully applied in areas such as large-scale Web Services [10] and Grid applications [4], but also in Systems Biology [6].

The technique consists in deriving automatically a set of Ordinary Differential Equations (ODEs) from a specification defined using Performance Evaluation Process Algebra (PEPA) [14]. The solution of the set of ODEs, by means of standard numerical techniques, gives insight into the dynamic change over time of aggregations of components that are in particular states. The approach abstracts away from the identity of the individual components. The automatic derivation of systems of ODEs from PEPA specifications, the algorithms to solve ODE and the generation of the numerical results are supported by the PEPA workbench [19].

The exploration of the various formal models is performed using a representative example introduced in Section 2. Section 3 presents a model addressing the conjoint behaviour of the smart environment and few individual users. Section 4 presents variants of the previous model addressing quantitative aspects of the collective behaviour of many users in a smart environment. In Section 5 an approach to the automatic synthesis of specifications of the model is described which uses high level domain specific information such as physical layout of the smart environment, routing information, size of groups of visitors and their start and final destinations. An example shows that the approach can be used to analyse models with a considerable (and realistic) number of spaces and visitors. Section 8 explores the role that these models could play in the exploration and early evaluation of smart environments. Section 7 analyses properties of one of the PEPA models with a reduced number of processes using stochastic model checking. Section 9 draws conclusions and gives an outline of future research.

2 The GAUDI system and some of its variants

A smart environment designed to guide people to a location within a building, perhaps a stadium or concert hall, or to direct evacuation in an emergency, is a relatively uncomplicated example of the kind of system that is of interest. Destination is inferred using an individual's "electronic ticket" in their mobile phone or, perhaps more relevantly in the case of evacuation, an active staff or visitor pass. An example of such a system has been prototyped and analysed at Lancaster University [17]. This system, named GAUDI, guides visitors unfamiliar with an office building to a particular office in the building. It is a simple example of a calm environment [20]. The system involves a set of situated displays in a building attached to walls and doors. The details of the actual space are not important from the point of view of the analysis (though scale issues will be discussed). It displays directions on situated displays (see Figure 2 for an example of the display) for a fixed period of time after a visitor has selected a required destination (using the display board in reception, see Figure 1). The GAUDI system supports infrequent visitors who are able to reach their destination before a time-out period has expired. Designs are required that are more responsive to visitors and can deal with larger volumes. Hence displays will be updated so that the directions are relevant only to those who occupy the space as people move in and out of the spaces.

The only *deliberate* action required by a visitor is to purchase the electronic ticket some time before the event or to obtain the visitor pass at reception. Otherwise all interactions with the system occur implicitly as visitors change their context, for example by moving from room to room or corridor to corridor.

This sort of system has broader significance in the context of large complex spaces such as museums, stadiums, concert halls not only to reach a required seat or exhibit for example, but also as an effective mechanism for guiding people to exits.

The properties that might be relevant to such a system are as follows:

1. How visible are the display directions and do people in the system interpret them correctly to take appropriate action to ensure that they reach their destination?
2. When there is an episode of the journey in which the direction is not visible will the person continue to proceed accurately in the appropriate direction?
3. Will the user obtain from the system a sense of progress towards the destination?
4. Will the user be able to recall the path in the future to make finding the office more efficient in the future?
5. Will the user get a sense of the whole building that will enable the reaching of other destinations more effectively? (These notions of efficient and effective would require further clarification if these properties were to be analysed more precisely).
6. Will the user eventually reach their destination regardless of how many other people are in the environment?
7. What is the probability that they will see a display relevant to their destination immediately on entering a space?
8. What is the probability that they will see a display relevant to them on the next display change? Or within some specified time limit?
9. What is the probability that they will reach their required destination within some specified time?
10. What will be the effect on the overall system when a certain percentage of users interpret the given directions wrongly?
11. What is the level of satisfaction, or anxiety among the user in the crowd?

These properties vary considerably in their content. While the questions at the top of the above list are more familiar in the context of usability, the questions towards the end of the list are more novel in content. These later properties may be summarised by the following generic characterisation:

- providing relevant information when and where it is needed

– providing a sense of progress towards the destination
– using routes and guidance to reduce the likelihood of bottlenecks

Different models are required to explore these different facets of the problem. Desirable design characteristics are likely to be derived by exploring the motivation for requirements. For example, the system may be required to allay anxiety about whether a person is likely to miss their flight while at the same time actively guiding passengers so that queues are kept to a minimum and resources are fully utilised. The anxiety might be reduced, as could be deduced from a process of user elicitation, by providing an accurate prediction of how long the passenger will have to wait in the queue. If a passenger is likely to miss departure then making him or her aware that the system has automatically notified relevant people of the delay could be helpful in this respect. The design could be configured in response to these requirements to give each person a sense of progress towards their destination or an estimate of how many others are heading towards the same destination.

To explore these different properties a number of formal models of several variants of the system and its users were developed in more detail. This variety of models were used deliberately to help understand which techniques would be effective and how the link between human system interaction within an immersive environment and crowd system interaction could be understood coherently.

All the models reflect the individual's active interaction with the environment occurring when information is obtained from the displays that are distributed around the system. The information that is read in each interaction is used to update the individual's position. When a user moves, or attempts to move, into a location there are a number of possible situations.

1. The space is full. A finite capacity is assumed for each location. A user attempts to get sight of the display but it is not always possible. In this situation a number of assumptions may be made about the user. The first possibility is that the user will not move unless it is clear that the receiving space has sufficient capacity, the second possibility is that a random move is made.
2. Someone heading for that destination is in the space already and the display is already showing the information. The user can then read the direction and move directly.
3. No one is currently in the space heading for the destination and there is a spare slot that is not currently in use. The effect of the arrival is to update the display with the required destination and direction.
4. All the display slots are showing direction information other than the required direction. The user must wait with other similar users until someone leaves the space so that the display can be updated.

This implicit interaction may be analysed either by focusing on an individual or a small group of individuals within the system or by focusing on collective behaviour while abstracting from individual behaviour. In this way it is possible to assess the relevance of information where and when it is required or to explore collective behaviour. Both approaches will be outlined in the rest of the paper. It is expected that these models should have a formative value in the design. They should provide an assessment of the impact of a particular design choice indicating how the design might be modified. An issue that is important and will be explored in more detail in the context of the different modelling approaches is how this analysis may be combined with and used to inform a human factors analysis.

The purpose in modelling this system is to explore the impact of this design on the people within the environment.

3 A qualitative model of the system

The first model of the design of the smart environment captures enough information to provide a basis for implementation. The aim is to focus on making the concepts of the design, insofar as

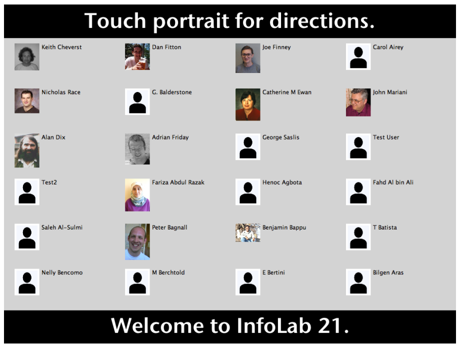

Fig. 1. The touch screen in reception

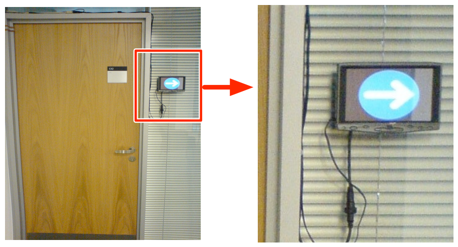

Fig. 2. The situated display

they relate to the people within the system, as precise and unambiguous as possible. The model is specified using the modelling language Promela which is the input language of the analysis tool SPIN [16]. SPIN provides both model-checking facilities as well as simulation in the sense of a stepwise exploration of the specified behaviour. Though Promela derives many of its notational conventions from the C programming language, it includes only primitives that are useful for the *modelling* of the coordination and synchronisation aspects of distributed systems and does not address computational aspects. This is what makes it modelling language, suitable for automatic analysis and for modelling aspects of the environment of a system or other interacting entities, rather than a programming language.

This model turns out to be too complex to be used for model-checking. Instead it is possible to perform a stepwise exploration of a system that includes a variety of collections of visitors. This provides the designer with an impression of how the system works. The tool can be used to visualise scenarios possibly with the support of human factors or domain experts.

Two further models are developed. One is obtained by making use of state encoding to reduce the number of processes to make it possible to carry out verification using the model-checking facility (second model).

A third model abstracts the identity of identical processes, such as those representing visitors heading for the same destination starting from the same place. This model (and some further variants) is specified in the PEPA process algebra which is amenable to the analysis of quantitative aspects of the behaviour. In this case, the model is used to study the impact of the design on the collective behaviour of people within the system. This model and its analysis will be discussed in Section 4.

3.1 A first model of the system

The model described in this section provides a rather detailed description of the main concepts to be adopted in the smart environment design. The focus here is on how the individual interacts with the environment. Two processes are central to the design of this system, the visitor and the distributor. The distributor pushes direction signs relevant to a visitor's destination and the current location. The distributor remains unchanged between variations of the Promela model. The visitor uses this information to decide where to go next. The key feature of the design from a usability point of view is the "wedge" between the distributor that publishes information, relevant to destination and current information, and the visitor. The model presented in this section is based on a building with rooms located at two floors. The floor layout is shown in Figure 3. The

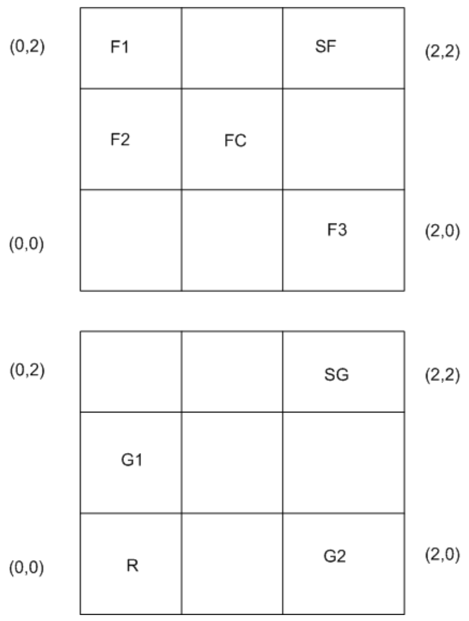

Fig. 3. The floor layout

intermediate processes that are involved in the interaction with the visitor provide an interface between the system and the human and this interface is to be the subject of analysis.

The distributor process receives a request from a display process. The `dreq` channel is used for the request and the display process sends both the token `tok` that indicates which location it is in and the destination `dest` for which the direction is required. The model contains a table of directions `route` which is used to supply the appropriate direction.

```
proctype distributor()
{
 byte tok;
 mtype dest;
 byte route[paths];

...
 route[54] = east;  /*F2*/
 route[55] = south;
 route[56] = north;
 route[57] = up;
 route[58] = west;
 route[59] = south;
 route[60] = down;
 route[61] = north;
 route[62] = west;
 ...
 do
 :: dreq?tok,dest ->  dbroad!route[(9*dest+tok)]
 od
}
```

The position of a visitor in the building is represented by three coordinates. The first coordinate is a Boolean and indicates the floor level. The second and third coordinate are the x-coordinate and the y-coordinate at the particular floor, see Figure 3. The visitor initially starts at reception (`upstairs==false`, `xcord==0` and `ycord==0`). It is assumed that the visitor selects a final destination (this is not modelled in the version of the system described here) when at the reception. The visitor's function is (1) to gain the capability to access the local display if there is one; (2) to sit in a vacant space; (3) to read the relevant information from the display if it can; (4) to stand; (5) to update its position. Individual visitors are identified by their visitor number (`vno`) in the model.

```
proctype visitor(byte vno; byte destination;
                 bool stairsdest; byte xdest; byte ydest)
{
mtype direction = blank;
chan returnchan = [0] of {mtype};
byte xcd;
byte ycd;
bool ustairs;
byte token;
bool arrived = false;
short dist=7;
short oldist=7;
mtype completed;
byte tn;
bool chk;

xcord[vno] = 0;
ycord[vno] = 0;
upstrs[vno] = false;

do
:: requestt!upstrs[vno],xcord[vno],ycord[vno] ->
{  replyt?token;
   if
   :: token == NS -> skip
   :: else ->
                { sit[token]!destination, returnchan;
                  do
                  :: { vrequest[token]!destination;
                       vreply[token]?direction;
                       if
                       :: direction==blank -> skip
                       :: else -> break
                       fi
                     }
                  od;
                  returnchan?completed}
    fi;
   updposition(vno);
...
 }
od
}
```

The visitor process interacts with three other processes: capability giver, display and place. It is these processes that are to be analysed in terms of the way they interact with visitors from a usability point of view.

1. The visitor gets its position by sending its location (coordinates and floor level) to a process called `capability giver` via channel `requestt`. This process returns a token via channel `replyt` if there is a display in the space corresponding to the location or a value `NS` to indicate that there is no display in this space. This process can be thought of as providing the capability to access the space.
2. The visitor uses the token (if it corresponds to a space for which there is a display) to request a place (`sit[token]`) thus regulating the capacity of the space. The sit request includes the current place, the visitor's destination and a link to the channel that can be used by the visitor to release the place when it has completed its transaction in this space.
3. Once the visitor has successfully found a place within the space it can access the display. The display process is accessed through a pair of channels defined by `token`. The display request uses the destination to match the required destination.
4. If the display is not able to provide the information it sends a blank direction and the visitor continues to try requesting until it succeeds.
5. Once the direction is obtained the place is released and the direction and current coordinates are updated using `updposition()`. If the visitor is in a location that is not associated with a display, i.e. `token == NS`, then the position is updated using the direction that was read last time the visitor was in a space with a display.

The `place` process services requests from visitors, sending the channel that can be used to release it back to the visitor.

```
proctype place(byte plce; chan sitok)
{ byte destination;
  chan stand = [0] of {bool};
   do
   :: sitok?destination, stand ->
          {counter[plce].spacecnt[destination]++;
           stand!false;
           counter[plce].spacecnt[destination]--;
          }
    od
 }
```

The place process increments and decrements counters that keep a tally of who is in the space waiting to read the display to get the direction associated with a particular destination. These processes control the capacity of the space.

The simplicity of the `display` process is complicated by the need to release a slot when no visitor is in the space requiring the direction for that destination.

```
proctype display(byte tkn; chan vrqst; chan vrply; chan acquireslot)
{ bool slotvis[spaces] = false;
  chan slotrequest[spaces] = [0] of {byte};
  chan slotreply[spaces] = [0] of {mtype};
  chan slotrls[spaces] = [0] of {bool};
  chan receiveslotinfo = [0] of {chan, chan, chan};
  byte usedslots=0;
  mtype direction;
  byte destination;
  do
  ::  slotvis[R] && (counter[tkn].spacecnt[R]==0) ->
                         {xrelease(R)}
  ::  slotvis[G1] && (counter[tkn].spacecnt[G1]==0) ->
                         {xrelease(G1)}
  ::  slotvis[G2] && (counter[tkn].spacecnt[G2]==0) ->
                         {xrelease(G2)}
  ::  slotvis[SG] && (counter[tkn].spacecnt[SG]==0) ->
                         {xrelease(SG)}
  ::  slotvis[SF] && (counter[tkn].spacecnt[SF]==0) ->
                         {xrelease(SF)}
  ::  slotvis[F1] && (counter[tkn].spacecnt[F1]==0) ->
                         {xrelease(F1)}
  ::  slotvis[F2] && (counter[tkn].spacecnt[F2]==0) ->
                         {xrelease(F2)}
  ::  slotvis[F3] && (counter[tkn].spacecnt[F3]==0) ->
                         {xrelease(F3)}
  ::  slotvis[FC] && (counter[tkn].spacecnt[FC]==0) ->
                         {xrelease(FC)}
  ::  vrqst?destination ->
                  if
                  :: slotvis[destination] ->  {xreply(destination)}
                  :: !slotvis[destination] && (usedslots<noslot)  ->
                          { acquireslot!destination,receiveslotinfo;
                            slotvis[destination]=true;
                            receiveslotinfo?slotrequest[destination],
                            slotreply[destination],slotrls[destination];
                            usedslots++;
                            xreply(destination)}
                    :: else -> vreply!blank
                  fi
  od
}
```

The display manages the slots that are relevant to the space. The array `slotvis` is used to record that the direction relating to a specific destination is visible in the slot. Whenever the direction is visible for a particular destination and there is no-one in the space waiting to read it then the slot is released. The release of unused slots is carried out in the initial part of the `do` statement within the process.

At the same time the display services requests for a direction associated with a destination. Three situations are relevant. In the first situation there is already a slot which shows the direction for the required destination and therefore the direction shown can be sent immediately back to the visitor. In the second situation there is no slot displaying the direction for the given direction but there is an available slot. In this case the slot is acquired and updated to show the direction for the required destination within the space. The direction is sent to the visitor and `slotvis` is updated to show that the display is visible. In the third case there is no available slot and therefore a `blank` is sent to the visitor who must then try again.

The remaining process in the system is the slot. A number of slots are defined for each space. When one of the slots receives a request — if there is more than one that is available then one of them is chosen nondeterministically — it also receives the channel that can be used to communicate back to the requesting process.

```
proctype slot(byte slno; byte token; chan sltcqr)
{ chan slotcap = [0] of {chan, chan, chan};
  chan sltrls = [0] of {byte};
  chan read = [0] of {byte};
  chan write = [0] of {mtype};
  byte dest;
  do
  :: sltcqr?dest, slotcap
           -> {dreq!token,dest;
               screens[token].tile[slno].dest = dest;
               dbroad?screens[token].tile[slno].dir;
               slotcap!read, write, sltrls;
               do
               :: read?dest -> write!screens[token].tile[slno].dir
               :: sltrls?dest -> break
               od
               }
  od
}
```

On first allocation the slot variables are updated with destination and direction from the distributor using `dreq` and `dbroad`. Thereafter the visitor uses the same channels to read this information.

There are a number of questions that will be explored using derivations of this model. The individual model is concerned with checking that the visitor is appropriately resourced. For the purposes of this analysis the question reduces to checking:

1. the direction will eventually be available to a visitor on arrival at a location
2. all visitors will reach their respective destinations
3. if the visitors follow the directions, and remember the direction when they are in a space without a display, then at each step they will get closer to their destinations.

3.2 The individual model

The second model, also using SPIN, was made more tractable by reducing the number of processes and encoding details of those waiting, the allocation of slots and other aspects of the design more efficiently. The second model continues to capture the key features of the implementation. The individual model captures the behaviour of the individual, the multi-slot display, the sensor that recognises the individual's presence in a space and the capacity of the space. The model describes the context and behaviour of the visitor, and describes the smart environment in terms of physical space. Physical space is described in terms of the physical coordinates of the people within the space. These coordinates are also used to describe where the sensors are and whether an visitor is detected in a space by the system. The model describes how and when visitors:

– move around physical space and are not always in range of a sensor
– are in range of a sensor and are able to see and read the direction that is relevant to them
– move in the direction indicated by the sign
– move into the space if there is an available place to see the display
– access displayed information once a place has been successfully found and use the direction displayed with the destination to move in the relevant direction
– make multiple attempts to get the relevant information when it is not immediately visible on the display
– relinquish their occupation of the space and update coordinates relevant to the direction that has been previously obtained.

As before, the model is based on a building organised on two floors (see floor layout Fig. 4), where R stands for reception, SG for staircase ground-floor, SF for staircase first floor, and others indicate specific rooms of the building. This version of the model reduces the number of states by collapsing processes, collecting together the display, place, and slot processes into a single process. The visitor engages in two significant interactions. The first is with the capability giver which, instead of returning a token to access the space as in the earlier model, returns two channels one (`ambs`) which can be used to request the direction and the other (`ambr`) to receive it. This process gives the visitor the capability to access the relevant display by providing the means to communicate with the displays. The token value is sent so that the capability giver can inform the display sensor, from which the visitor has departed, as well as the display sensor to which it has moved. The second interaction is to use the pair of channels to obtain the direction. In this case the visitor only receives a valid direction.

```
proctype visitor(byte vno; byte destination)
{mtype direction = blank;
chan ambs = [0] of {byte};
chan ambr = [0] of {mtype};
byte xcd;
byte ycd;
bool ustairs;
byte tok = ND;

xcord[vno] = 0;
ycord[vno] = 0;
upstrs[vno] = false;

do
:: requestt!upstrs[vno],xcord[vno],ycord[vno],
            tok, destination ->
            {if
             :: replyt?true,ambs,ambr,tok ->
                 {ambs!destination; ambr?direction}
             :: replyt?false,ambs,ambr, tok -> skip
             fi;
   updposition(vno)
 }
od
}
```

The distributor process is identical to the one described in the previous section. The capability giver calculates the token and then returns to the visitor either the channels that can be used to communicate with the display (indicating this using the first part of the message as `true`)

`replyt!true, public[token], publicv[token], token`

or sends false as the first parameter, indicating that the requested coordinates relate to an area where there is no public display. The token is used to inform the capability giver where the visitor has come from so that the numbers of visitors waiting in each space can be determined. The capability giver (as already mentioned) uses `departure[oldtok]!destination` and `arrival[token]!destination` to inform relevant displays of arrivals and departures.

```
proctype capabilitygiver()
{
byte floor[spaces];
byte gloor[spaces];
byte indx;
bool upstairs;
byte xcd;
byte ycd;
byte token;
byte oldtok;
byte destination;
gloor[0] = R;
gloor[1] = NS;
gloor[2] = G2;
gloor[3] = G1;
...
end:
do
:: requestt? upstairs, xcd, ycd, oldtok, destination  ->
       { indx = xcd+3*ycd;
         token = (upstairs -> floor[indx]:gloor[indx]);
         if
         :: (oldtok!=NS)&&(oldtok!=ND)&&(oldtok!=token) ->
                      departure[oldtok]!destination
         :: else -> skip
         fi;
         if
         :: (oldtok!=token) && (token!=NS) ->
                       arrival[token]!destination
         :: else -> skip
         fi;
         if
         :: token == NS ->
             replyt! false, publicd[R], publicv[R], token
         :: else ->
             replyt! true, publicd[token], publicv[token], token
         fi;
       }
od
}
```

There is one display sensor (`dispsen`) for each location. It receives messages from the capability giver (via `departure[token]`) and decrements the relevant counter indicating how many visitors are left within the space. The counters therefore replace the role of the `place` process. When it receives a departure relating to a particular space it decrements the total people waiting and the number of people waiting related to a particular destination. If the number waiting for a particular destination goes to zero then the relevant slot is "blanked" and made available for future use, and the number of slots used is decremented. If an arrival is received then the relevant values are incremented. In the context of this the process deals with the request for a visitor to see the direction of a particular destination. It does this by checking whether the required destination is already available (`visibleslot[destination]`). If it is available then stored direction information is sent immediately (`publicv[token]!screens[token].file[vd].dir`). If it is not and there is a free slot then the information is requested from the distributor and put into the relevant slot. If there is no relevant slot then either a blank is returned, if less than the capacity of the room, or alternatively a random move is made.

```
proctype dispsen(byte token)
{byte slots=0;
 byte destination = R;
 byte peoplewaiting[spaces];
 byte visibleslot[spaces]=noslot;
 byte vd = 0;

totalwaiting[token]=0;

   end:
   do
   :: departure[token]?destination ->
         { peoplewaiting[destination]--;
           totalwaiting[token]--;
           if
           :: ((peoplewaiting[destination]==0) &&
               (visibleslot[destination]!=noslot))
                    -> { vd = visibleslot[destination];
                         screens[token].tile[vd].dest=0;
                         screens[token].tile[vd].dir=blank;
                         visibleslot[destination]=noslot;
                         slots--
                        }
           :: else skip
           fi
         }
   :: arrival[token]?destination ->
         { peoplewaiting[destination]++;
           totalwaiting[token]++
         }
   :: publicd[token]?destination->
         { vd = visibleslot[destination];
           if
           :: vd!=noslot ->
                 publicv[token]!screens[token].tile[vd].dir
           :: vd==noslot ->
                if
                :: (slots<noslot) ->
                     { if
                       :: screens[token].tile[0].dest==0 ->
                             {visibleslot[destination]=0;
                              screens[token].tile[0].dest=destination}
                       :: screens[token].tile[1].dest==0 ->
                             {visibleslot[destination]=1;
                              screens[token].tile[1].dest=destination}
                       fi;
                       dreq!token,destination;
                       vd = visibleslot[destination];
                       dbroad?screens[token].tile[vd].dir;
                       publicv[token]!screens[token].tile[vd].dir;
                       slots++
                      }
                :: (slots==noslot) ->
                       if
                       :: totalwaiting[token] > roomcapacity ->
                             publicv[token]!north
                       :: totalwaiting[token] > roomcapacity ->
                             publicv[token]!south
                       :: totalwaiting[token] > roomcapacity ->
```

```
                              publicv[token]!east
                         :: totalwaiting[token] > roomcapacity ->
                              publicv[token]!west
                         :: totalwaiting[token]<=roomcapacity ->
                              publicv[token]!blank
                         fi
                   fi
                fi
             }
   od
}
```

3.3 Analysis of the individual model

The model captures the relationship between the visitor's physical position and location in the building. A visitor may "acquire" a position in the space to enable them to read the display. The problem is to decide what characteristics of the system support each visitor's current activity most effectively. In [8] this issue is discussed in more detail. In that paper a mobile device is used to control a process. The analysis indicated how the context and the display on the device provides information that supports the goals, plans and actions of the user as well as how the state of the system is represented in the various resources available to the user. There are a variety of ways of assessing the importance of these characteristics, for example by interviewing users of the existing system. The system can also be analyzed through properties of the system that can be explored by means of a model.

P1: When a visitor acquires a position in a location it is always possible to read the direction that is relevant to their destination on the display.
P2: A visitor will eventually reach the destination that is designated on his or her electronic ticket.
P3: At each step, if a visitor follows directions, this will result in getting closer to the required destination.
P4: Wherever the visitor is in the building, if it is possible to acquire a position that allows them to see the display then he or she can read information that gets them nearer to their destination.
P5: A visitor will eventually acquire a position in the space that they are attempting to enter.

P1 and P5 depend on the volume of other visitors within the system, and cannot be analyzed using a model that only includes a few instances of individuals. P4 is not satisfied by the proposed design both because some locations do not have public displays visible from them and because, when they are in view of a public display, it is not guaranteed that the relevant information is contained in the display. It is assumed that people will remember and continue in the direction they last read. The model can also be used to explore P2 and P3. For each design, as represented by a variety of models, P2 and P3 were analysed. In addition to the details of the interactions summarised above, 13 visitors were modelled together reaching all the possible destinations F1, F2, F3, G1 and G2.

P2 was checked by introducing an observer process that was notified whenever one of the visitors had reached their destination (signified by reaching the specified location and with one of the slots on the display showing a down arrow for the given destination). The observer process signifies satisfaction when all visitors in the model have reached their destination. P2 is checked by ensuring that there are no paths for which this is not possible. P3 is checked by including an assertion in the visitor process to check that the process never increases the distance between the visitor's current position and the destination location. Again the assertion was checked for all possible paths.

Two assertions are used to check these properties. The first assertion (checking P2) is attached to each visitor process to check that the visitor has arrived at the required destination. This is satisfied when the visitor reaches a space other than the stairwell in which the destination coincides with the space and the direction is `down` in of the slots available to that space. After this assertion is checked in this situation the visitor completes and notifies a monitoring process `idle` that has the role of checking that all the visitors in the system eventually terminate.

```
inline destcheck(p, t, v, ud, xd, yd)
 { arrived = arrived ||
((xcord[v]==xd)&&(ycord[v]==yd)&&
    (upstrs[v]==ud)&&(screens[p].tile[t].dir==down)&&
    (screens[p].tile[t].dest==p))
...
destcheck(destination,0,xno,stairsdest,xdest, ydest);
destcheck(destination,1,xno,stairsdest,xdest, ydest);
destcheck(destination,2,xno,stairsdest,xdest, ydest);
destcheck(destination,3,xno,stairsdest,xdest, ydest);
                              assert(arrived);
  end:                     do
                           :: finished!xno -> skip
                           od
```

The idle process simply keeps a tally of all the terminating processes:

```
proctype idle()
{ bool visited[vinos];
  byte fin;
  do
  :: finished?fin ->
         { terminated = (visited[fin]-> terminated: terminated+1);
           visited[fin]=true;
           if
           :: terminated==vinos ->  break
           :: else -> skip
           fi
         }
  od
}
```

The second property (P3) that is checked is that in each step the visitor approaches closer to the destination.

```
 if
   :: ((xcord[vno]==2)&&(ycord[vno]==2)) -> skip
   :: ((tok<NS)&&(totalwaiting[tok]>roomcapacity)) -> skip
   :: else -> assert(upstrs[vno]->(oldist<=dist):(dist<=oldist))
   fi;
```

The value of checking these properties is when they fail. These failures raise questions: why does a visitor fail to reach a destination? Why does a direction take the visitor further away? The model checker provides a trace, a sequence of steps in which the property is broken, for example the visitor moves away from destination. Domain and HCI specialists can be called upon to translate the behaviour of the system, including the users of the system, into a narrative, a story describing the sequence in terms of how the proposed smart environment design works. This narrative can then be used to explore the human factors that are considered to cause the problems. In summary, the approach provides useful insights when addressing issues that involve only a limited number of visitors in the system. However, the model-checking techniques involved do not support the analysis of models with a large number of visitors. This is because there are limitations on the size of the state-space that can be searched and stored effectively.

4 The collective model

The previous section indicates how models were used to check properties insofar as they related to an individual visitor in the context of an arbitrarily defined small set of other visitor processes. In practice, a class of important characteristics of the design relate to how the system functions in a broader context. When there are much larger numbers of visitors, can there be unexpected bottlenecks? Can the design be used to influence the individual behaviours and improve performance? Different designs may affect these flows of visitors through the system and may improve user experience by reducing wait times or reducing the number of display refreshes. This can be achieved and predicted through appropriate modelling and analysis. Aspects of collective behaviour can be addressed through the modelling of the display's content (in this case, how many display slots would guarantee a satisfactory throughput), the size and legibility of the display (thereby managing the number of people who can see the display at the same time, the capacity of the space and corridors connecting the spaces) and support the devising of algorithms for scheduling (though this is not addressed here). These are all questions that can be partially addressed using certain forms of performance modelling along with a representation of the design. The aim is to use these techniques to explore and optimise the design of the smart environment providing information that will improve user experience and reduce bottlenecks.

The models that are presented in this section and used for Fluid Flow analysis and stochastic simulation are similar to those developed in the previous section. There are a number of differences though. The first difference is that they capture stochastic time aspects. This means that they can be used to model, additionally features such as the average time visitors spend in various spaces when moving through the environment and interacting with it. A further difference is that the Fluid Flow technique keeps track only of how many processes of the same kind are in a particular state at any time and represents this number as a continuous rather than a discrete value. This means that the identity of the processes are abstracted. For example, making reference to the layout of the building presented in the previous section, the models can be used to visualise how many visitors (on average) of a group entering at reception and heading for room F3 are in the stairwell SG at a certain point in time.

In this section a single notation, PEPA [15], is used which is briefly introduced in Section 4.1. This notation allows a variety of analysis techniques to be applied to the model. In the sections following the PEPA description two different stochastic models of the GAUDI system are presented. The first model represents a smart environment in which each location has a number of individual displays (see Fig. 2), i.e. displays with a single slot, that visitors may consult. This implies that the information shown at different displays in a location to single users may be the same. The individual displays could in principle also be interpreted as the visitors' mobile phone displays or as any other kind of handheld display handed out to the visitor upon entering the building. In the models this will be reflected by instantiating a sufficient number of one-slot displays in each location so that each visitor that is present has access to such a display.

The second model extends the first one replacing the individual displays by a single shared display per location with a number of display slots. Each slot may show routing information in answer to requests made by the visitors present in the room. This routing information indicates the location to which a visitor should go to reach the final destination. The model of the display makes use of an arbiter that ensures that there are never slots displaying the same information.

Both models are amenable to stochastic model-checking when the number of processes, in this case those modelling visitors and displays, are severly limited in terms of space and performance. They can also be used for stochastic simulation. Often, the stochastic simulation results can be approximated by solving a set of Ordinary Differential Equations that can be automatically derived from the model. Such an approximation has the advantage that it can be generated in much less time then would be needed when using stochastic simulation. However, the results obtained by this Fluid Flow approach do not always correspond to the simulation results. Why this is so is still topic of theoretical study, see for example [13]. A practical way to deal with this issue is to perform a limited simulation of the model in addition to the ODE analysis to check sufficient correspondence with the ODE results. The limits of the simulation relate to the number of independent replications. These require relatively little time.

4.1 PEPA

In PEPA, systems can be described as interactions of components that may engage in activities in much the same way as in other process algebras. Components reflect the behaviour of relevant parts of the system, while activities capture the actions that the components perform. A component may itself be composed of components. The specification of a PEPA activity consists of a pair *(action type, rate)* in which *action type* denotes the type of the action, while *rate* characterises the negative exponential distribution of the activity duration. A positive real-valued random variable X is exponentially distributed with rate r if the probability of X being at most t, i.e. $Prob\{X \leq t\}$, is $1 - e^{r \cdot t}$ if $t \geq 0$ and is 0 otherwise, where t is a real number. The expected value of X is $1/r$. Exponentially distributed random variables are more tractable than general distributed ones because they enjoy the memoryless property, i.e. $Prob\{X > t + t' | X > t'\} = Prob\{X > t\}$ for $t, t' \geq 0$. For this reason, they are widely used in the modelling of the dependability and performance of real systems where they form the basis for *Continuous Time Markov Chains* (CTMC), see e.g. [12].

Furthermore, proper compositions of exponential distributions can be used for the approximation of any non-negative distribution. The PEPA expressions used in this article have the following syntax [3]:

$$\boxed{P ::= (\alpha, r).P \;\Big|\; P \;+\; P \;\Big|\; P \bowtie_L P \;\Big|\; A}$$

Behavioural expressions are constructed through prefixing. Component $(a, r).P$ carries out activity (a, r), with action type a and duration Δt which is an exponentially distributed random variable with rate r. After performing the activity, the component behaves as P. Component $P + Q$ models a system that may behave either as P or as Q, representing a race condition between the two components. The cooperation operator $P \bowtie_L Q$ defines the set of action types L on which components P and Q must synchronise (or cooperate); both components proceed independently with any activity not occurring in L. The expected duration of a cooperation of activities a belonging to L is a function of the expected durations of the corresponding activities in the components. Typically, it corresponds to the longest one (see [15] for definition of PEPA). An important special case is the situation where one component is *passive* (a rate $\top$ indicates this) in relation to another component. Here the total rate is determined by that of the active component only. The behaviour of process variable A is that of P, provided that a defining equation $A = P$ is available for A. Two shorthand notations are introduced. If the set L is empty $P \bowtie_L Q$ is written as $P|Q$. If there are n copies of P in parallel cooperating with m parallel copies of Q this is written as: $P[n] \bowtie_L Q[m]$.

One of the advantages of a formal, high-level specification language with a fully formal semantics is that it lends itself to the application of different analysis and evaluation techniques while preserving its semantics. For example, PEPA specifications can be analyzed by means of a stochastic model checker, such as PRISM [18], and it can also be used for stochastic simulation. As already mentioned PEPA specifications can be translated into sets of Ordinary Differential Equations (ODEs). A very brief summary of the approach follows; more details can be found in [14]. Suppose a PEPA model $S1[n1] \bowtie_{L1} S2[n2] \bowtie_{L2} \ldots \bowtie_{Lk-1} Sk[nk]$ is given, which is composed of $n1 + n2 + \ldots + nk$ sequential components. Each component Sj is defined by means of a PEPA defining equation $Sj = \ldots Sjr \ldots Sjv \ldots Sjw$, where $Sj, Sjr, Sjv \ldots Sjw$ are the relevant states of Sj; all such states are themselves characterised by means of proper defining equations. The solution of the set of ODEs associated with the PEPA model is a set of continuous functions. In particular, there is one function $\mathcal{S}(t)$ for each state S occurring in the original specification and,

[3] There are some restrictions on the nesting of parallel processes in the dialect of PEPA suitable for the translation to ODEs. For the sake of simplicity this issue is not discussed here, for more details see [14].

for each time instant t, $\mathcal{S}(t)$ yields a continuous approximation of the total number of components which are in state S at time t, given the initial conditions $\mathcal{S}1(0) = n1, \mathcal{S}2(0) = n2, \ldots, \mathcal{S}k(0) = nk$. Notice that the fact that the values $n1, n2, \ldots, nk$ of the number of components in the system (at the initial configuration) can be very high, e.g. also in the order of millions, makes the approach intrinsically scalable. In the experiments described in Section 4.2, results are compared with those obtained via stochastic simulation.

4.2 A stochastic model of GAUDI with multiple individual displays

The collective model takes the same kind of processes as the individual model as its starting point. Processes model the behaviour of a visitor, the individual displays and the room capacity.

Some aspects of the case-study are deliberately modelled at a higher level of abstraction than in Section 3 or somewhat simplified. An apparent lack of other work attempting to model smart environments with many users or the use of the Fluid Flow approach made the development of relatively small and modular models appealing. An important goal was that these models could easily be extended (e.g. with larger buildings, more and different groups of visitors, etc.) to explore the potential of the technique. Some preliminary results are described in Section 4.2 and 4.3. The aim was to be as general as possible leaving elements of the model open to different interpretations. For example, the model describes 'locations' a, b and so on, but such locations could be in principle rooms or corridors, only characterised by their capacity, the presence of displays and their direct connections to other parts of the smart environment. Visitor processes have been kept simple with the aim that in future models more complexity will be investigated, for example addressing error behaviour or other kinds of relevant behavioural patterns.

The main abstractions and generalisations may be summarised as:

1. The models are based on the concept of general 'spaces', called locations, rather than on a coordinate system to keep track of the position of visitors in the environment.
2. The routing information is distributed over the displays in the locations in the sense that each slot of a display can provide the name of the next location for any possible final destination. The routing information is static, i.e. currently in the stochastic models routing information is not dynamically changed while visitors are moving through the building. For this reason there is no distributor process in these models.
3. Most of the analysis is shown for a limited model based on five locations. This is only for the purpose of presentation. Larger models can be generated automatically, for example one consisting of 26 locations and several hundred visitors was also explored.

In other respects the model has the same structure and character as the individual model. Each location has a number of places and a number of individual one-slot displays that can be consulted by visitors that are present in the location. A display acts as sensor accepting requests for information. The PEPA model consists of three kinds of processes modelling the behaviour of a visitor, the one-slot display and place. Displays and places are instantiated for each particular location.

In this section a model of the GAUDI system is presented which has five locations, two of which are final destinations. The locations are called a, b, c, d and e. The locations are connected in the following way: (a,b), (b,c),(c,d) and (b,e) as shown in Fig. 4.

The following gives an informal description followed by a formal description in PEPA of each of the processes. The scenario focusses on two groups of visitors that are entering the building from location a. One group of 15 visitors heading for destination e and one group of 45 visitors heading for destination d. A visitor heading for destination e first tries to get a place in location a (first action of the process *VisEtoLa* in the PEPA specification called $(lasd, s)$). This is modelled by action $(lasd, s)$ in the PEPA specification which is encoding that the visitor is looking for a place to sit down in location a ($lasd$ reads as '*l*ocation a *s*it *d*own'). If there is a place available ($lasd$ needs to synchronise with the same action in one of the place-processes in location a) then the visitor sits down and requests information to be displayed for the required destination e (state *VisEesda*). The parameter s stands for the rate of this action as described in the section introducing PEPA.

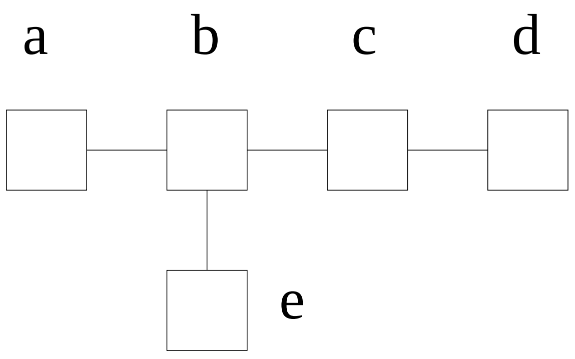

Fig. 4. Layout of the building

Action *laee* is an event that captures the idea that 'a visitor currently seated in location a and is heading for final destination or exit e and requests information on which location to go next'. The request is engaged as soon as there is an individual one-slot display (also called slot in the sequel) in location a that can display such information. This is again modelled through synchronisation of this action with its counterpart in a process modelling a slot in location a. When the information is displayed, the visitor *stands up* (action $lasu$) and receives the information (i.e. any of the matching destinations in the process *VisEeRec* below) from the slot. For example, if the visitor receives the action *eealb* this means that the visitor heading for e, currently at a needs to proceed to location b. It does so by following the behaviour described by process $VisEetoLb$ (i.e. visitor with final destination e now first needs to proceed to location b). The arrival at the final destination is modelled by the process that remains in the state $VisEeArrived$ forever.

The full specification of the visitor process in PEPA follows:

$$
\begin{aligned}
VisEetoLa &= (lasd, s).VisEesda\\
VisEesda &= (laee, a).VisEesua\\
VisEesua &= (lasu, s).VisEeReca\\
VisEetoLb &= (lbsd, s).VisEesdb\\
VisEesdb &= (lbee, a).VisEesub\\
VisEesub &= (lbsu, s).VisEeRecb\\
VisEetoLc &= (lcsd, s).VisEesdc\\
VisEesdc &= (lcee, a).VisEesuc\\
VisEesuc &= (lcsu, s).VisEeRecc\\
VisEetoLd &= (ldsd, s).VisEesdd\\
VisEesdd &= (ldee, a).VisEesud\\
VisEesud &= (ldsu, s).VisEeRecd\\
VisEetoLe &= (lesd, s).VisEesde\\
VisEesde &= (leee, a).VisEesue\\
VisEesue &= (lesu, s).VisEeArrived
\end{aligned}
$$

$$\begin{aligned}
\mathit{VisEeReca} &= (\mathit{eeale}, r).\,\mathit{VisEetoLe} + \\
&\quad (\mathit{eeala}, r).\,\mathit{VisEetoLa} + \\
&\quad (\mathit{eealb}, r).\,\mathit{VisEetoLb} + \\
&\quad (\mathit{eealc}, r).\,\mathit{VisEetoLc} + \\
&\quad (\mathit{eeald}, r).\,\mathit{VisEetoLd} \\
\mathit{VisEeRecb} &= (\mathit{eeble}, r).\,\mathit{VisEetoLe} + \\
&\quad (\mathit{eebla}, r).\,\mathit{VisEetoLa} + \\
&\quad (\mathit{eeblb}, r).\,\mathit{VisEetoLb} + \\
&\quad (\mathit{eeblc}, r).\,\mathit{VisEetoLc} + \\
&\quad (\mathit{eebld}, r).\,\mathit{VisEetoLd} \\
\mathit{VisEeRecc} &= (\mathit{eecle}, r).\,\mathit{VisEetoLe} + \\
&\quad (\mathit{eecla}, r).\,\mathit{VisEetoLa} + \\
&\quad (\mathit{eeclb}, r).\,\mathit{VisEetoLb} + \\
&\quad (\mathit{eeclc}, r).\,\mathit{VisEetoLc} + \\
&\quad (\mathit{eecld}, r).\,\mathit{VisEetoLd} \\
\mathit{VisEeRecd} &= (\mathit{eedle}, r).\,\mathit{VisEetoLe} + \\
&\quad (\mathit{eedla}, r).\,\mathit{VisEetoLa} + \\
&\quad (\mathit{eedlb}, r).\,\mathit{VisEetoLb} + \\
&\quad (\mathit{eedlc}, r).\,\mathit{VisEetoLc} + \\
&\quad (\mathit{eedld}, r).\,\mathit{VisEetoLd} \\
\mathit{VisEeArrived} &= (\mathit{nop}, a).\,\mathit{VisEeArrived}
\end{aligned}$$

The specification of a visitor also includes potential behaviour which is not used in the particular example used to illustrate the model (that is, visitors going from a to e via b). As can be seen in Sect 5, this more general specification schema allows for models to be easily generated automatically from architectural specifications of buildings. The potential extra behaviour in the specification does not affect the actual model as far as correctness or state-space size is concerned because of the way the activity synchronisations have been designed.

The model of a visitor has three rate parameters. Each models the average time needed to perform the related activity. The average duration of activities is defined by their rates and is assumed to be measured in minutes. So, for instance, letting $s = 10$ implies that the average time a visitor needs for sitting down or standing up is 6 sec. (i.e. 0.1 min.). Rate $a = 2$ models the average time a visitor needs to make a request equal to 30 seconds. The rate $r = 1$ models the average time to receive the requested information equal to 1 minute.

It is further assumed that visitors are arriving over a certain period of time and heading for different destinations. This is modelled in the following way:

$$\mathit{Vis} = (\mathit{nop}, \mathit{v0}).\,\mathit{VisEetoLa} + (\mathit{nop}, \mathit{v1}).\,\mathit{VisEdtoLa}$$

The action *nop* stands for a dummy no-operation action. The rates $v0$ and $v1$ can be used to tune the rate of generation of the two types of visitors (i.e. those heading to e and those heading to d) and their relative number. For example, in the case of 60 visitors in total, the number of visitors heading for location e, i.e. the number of $VisEetoLa$ processes, will amount to $v0/(v0 + v1) \times 60$ while the number of $VisEdtoLa$ processes amounts to $v1/(v0 + v1) \times 60$.

Note that this models an arrival pattern of visitors in which most visitors arrive ‘early’ and, while time passes, there are a diminishing number of visitors arriving. This is because the visitors are generated with a rate of $N(t) \times (v0 + v1)$, where $N(t)$ is the number of processes in state Vis at time t. By manipulating $v0$ and $v1$, the relative number of each kind of visitors can be defined and the total of $v0 + v1$ models the spreading over time of the arrival of visitors.

This is not the only way in which the arrival of visitors can be modelled. An alternative would be to keep the generation rates $v0$ and $v1$ independent from the size of the visitor group. For example:

$$\begin{aligned}
\mathit{VisAE} &= (\mathit{nop}, \mathit{v0}).\,\mathit{VisEetoLa} \\
\mathit{VisAD} &= (\mathit{nop}, \mathit{v1}).\,\mathit{VisEdtoLa}
\end{aligned}$$

Examples of the use of this variant will be illustrated in Sections 5 and 6. Further alternatives could be considered to reasonably approximate the particular characteristics of the arrival pattern of visitors in the case at hand.

The individual one-slot displays (called *slots* in the specification) for each location can receive a request from a visitor heading for a particular final destination and then generates the proper information in answer to the request.

$$\begin{aligned}
\mathit{SlotLa} &= (\mathit{laee}, a).\mathit{SlotLaRespEe} + (\mathit{laed}, a).\mathit{SlotLaRespEd} \\
\mathit{SlotLaRespEe} &= (\mathit{eealb}, r).\mathit{SlotLa} \\
\mathit{SlotLaRespEd} &= (\mathit{edalb}, r).\mathit{SlotLa}
\end{aligned}$$

The model of slots in other locations and for requests for other destinations are similar.

Finally, the model of a place in a location is very simple. It can either be free and accept a visitor sitting down and then become full, or it is full and the visitor decides to stand up and releases the place that becomes free.

$$\begin{aligned}
\mathit{PlaceFreeLa} &= (\mathit{lasd}, s).\mathit{PlaceFullLa} \\
\mathit{PlaceFullLa} &= (\mathit{lasu}, s).\mathit{PlaceFreeLa}
\end{aligned}$$

The models of places in other locations are similar.

The complete system is composed of the visitor model, with 60 visitors in total, the slots and places all synchronising over shared actions. The number of slots and places that have been instantiated in each location (i.e. 60) has been made sufficiently large because a reliable ODE approximation requires that none of the involved quantities (and therefore its rates) over time reduces to zero. A reduction to zero may lead to a high probability of error in the calculations involved in solving the set of differential equations. The ODE approximation will be used as a result to get an impression of the resources needed to let visitors move through the building following a particular route encoded in the slot-processes.

$$\begin{aligned}
&\mathit{Vis}[60] \\
&\bowtie_L \\
&(\mathit{SlotLa}[60] \parallel \mathit{SlotLb}[60] \parallel \mathit{SlotLc}[60] \parallel \mathit{SlotLd}[60] \parallel \mathit{SlotLe}[60] \parallel \\
&\mathit{PlaceFreeLa}[60] \parallel \mathit{PlaceFreeLb}[60] \parallel \mathit{PlaceFreeLc}[60] \parallel \mathit{PlaceFreeLd}[60] \parallel \mathit{PlaceFreeLe}[60])
\end{aligned}$$

where:

$$\begin{aligned}
L &= (\mathit{Requests} \cup \mathit{Responses} \cup \mathit{Sitdown} \cup \mathit{Standup}) \\
\mathit{Requests} &= \{lXeY | X \in \{a, b, c, d, e\}, Y \in \{e, d\}\} \\
\mathit{Responses} &= \{eYXlZ | X, Z \in \{a, b, c, d, e\}, Y \in \{e, d\}, y \neq x\} \\
\mathit{Sitdown} &= \{lXsd | X \in \{a, b, c, d, e\}\} \\
\mathit{Standup} &= \{lXsu | X \in \{a, b, c, d, e\}\}
\end{aligned}$$

Analysis of the model with individual displays In the following a number of plots show the average quantities of processes of each type over time, under the above assumptions. The figures show both the results for the ODE approximation and the result for stochastic simulation. For the ODE analysis the adaptive step-size 5th order Dormand Prince ODE solver is used as provided in the PEPA workbench [19, 11]. For the stochastic simulation Gillespie's stochastic algorithm [9] is used which is also provided in the workbench. For the ODE analysis time ranged from 0 to 20 minutes, with 100 data points, a step size of 1.0E-3, and relative and absolute error equal to 1.0E-4. For the simulations the same time range and number of data points were used on 10 independent replications with a confidence interval of 0.05.

Even though the number of independent replications in the simulation has been relatively small, the results of both analyses techniques show good correspondence. The advantage of the ODE approximation is however that in general it produces the results much faster than simulation.

A closer look at the various results gives a flavour of the kind of information a Fluid Flow analysis may provide. Fig. 5 shows visitors starting at location a and arriving at their respective

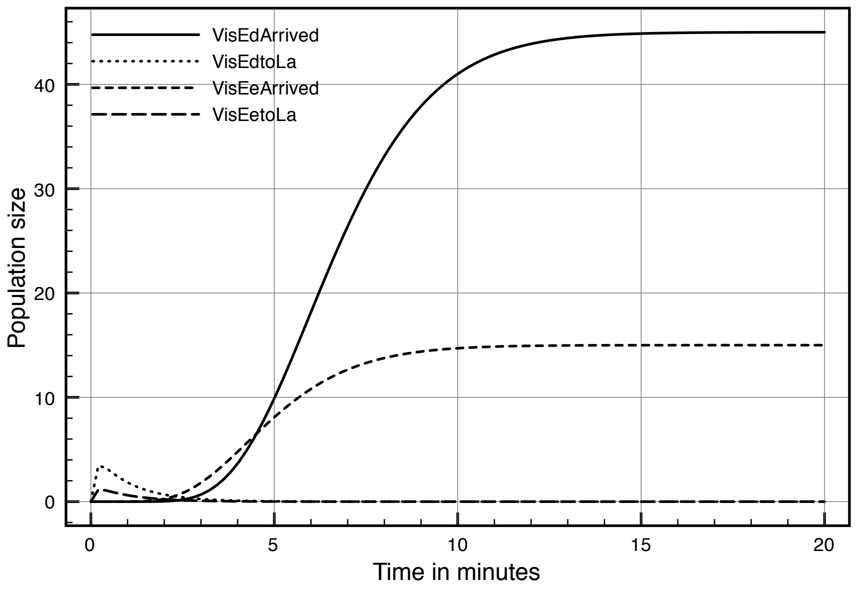

Fig. 5. Visitors arriving at destinations (ODE approximation)

destinations. The values chosen for $v0$ and $v1$ in the specification were 0.25 and 0.75. One quarter of the 'generated' visitor processes are those with destination e and three quarters those with destination d. The total number of visitors are 60, hence $0.25 * 60 = 15$ visitors would be expected to arrive at destination e and the remaining 45 at destination d. This is indeed the case, as shown in Fig. 5. It can also be seen that those for destination e arrive relatively early, and indeed those visitors need to visit location b to arrive at d. Those with destination d need to visit locations b and c and therefore arrive a little later. The number of visitors for each destination that are kept from entering the building (curves labelled $VisEdtoLa$ and $VisEetoLa$) are very low, meaning that, on average, there are sufficient places available in location a given this profile.

The results obtained via stochastic simulation, presented in Fig. 6, shows a similar tendency. The plots in that figure are less smooth because the simulation has been restricted to a small number of independent replicas.

Fig. 7 shows the average number of places occupied over time. At the beginning all places are free, but when visitors start to arrive, a number of places are getting occupied in the various locations. It is easy to observe that the maximum of the average number of occupied places varies per location. These numbers would provide a good indication of the required number of places needed in each location. In fact, the specified number of resources, 60 places in each location, clearly represents an over-estimation. The curves also show when which locations are most congested and when they are underused. This can be helpful information when planning alternative routing strategies. In this case, as before, the ODE approximation and the stochastic simulation results (Fig. 8) are largely in agreement.

Fig. 9 shows the number of free slots over time and gives an indication on the number of individual displays that would be needed in each location not to make visitors wait for information indicating their next destination. There is a clear correspondence in this model between the number of occupied places in a location and the number of slots in use. This is as expected since each visitor that occupies a place also makes a request for information and gets the response eventually presented on their individual display. Again, simulation results shown in Fig. 10 confirm the tendencies shown in the Fluid Flow results in Fig. 9.

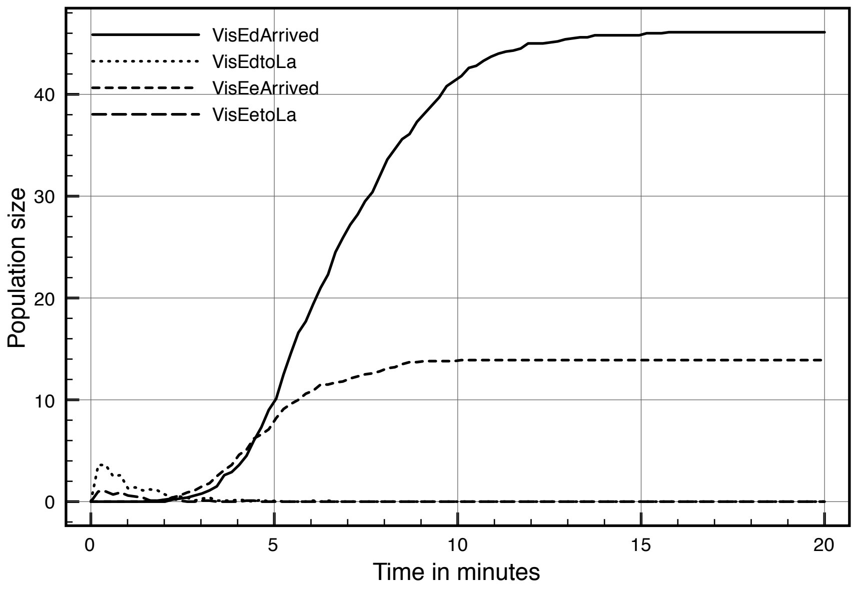

Fig. 6. Visitors arriving at destinations (stochastic simulation)

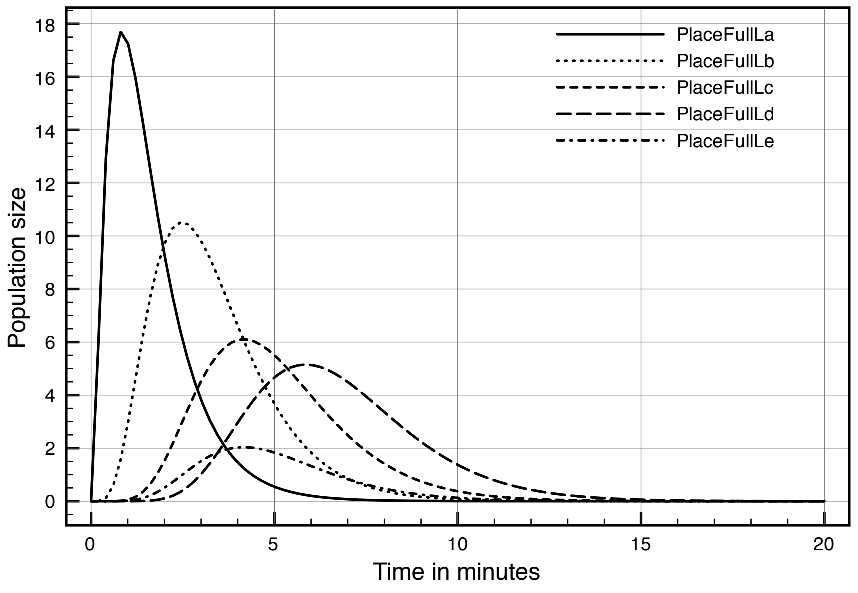

Fig. 7. Number of places occupied over time (ODE approx.)

The total rate of generation of visitors is $0.25 + 0.75 = 1.0$. This indicates that each visitor arrives after one minute on average. Since there are 60 visitors at the start, and the group of visitors is limited, it can be seen that most of them arrive within a relatively short time span. It is easy to model situations where visitors arrive over a much longer time span by simply reducing the respective rates proportionally. An example of the effect of reducing the rates to 0.025 and 0.075

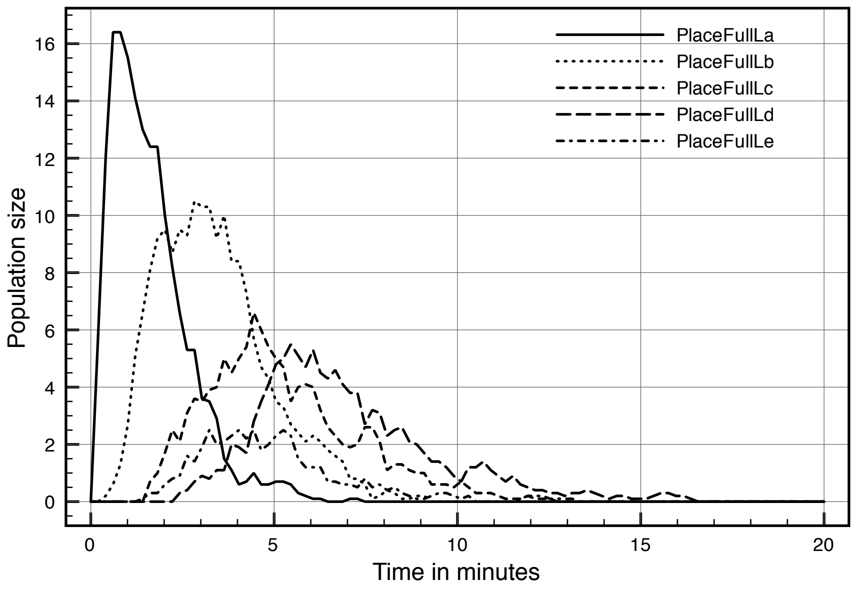

Fig. 8. Number of places occupied over time (stochastic simulation)

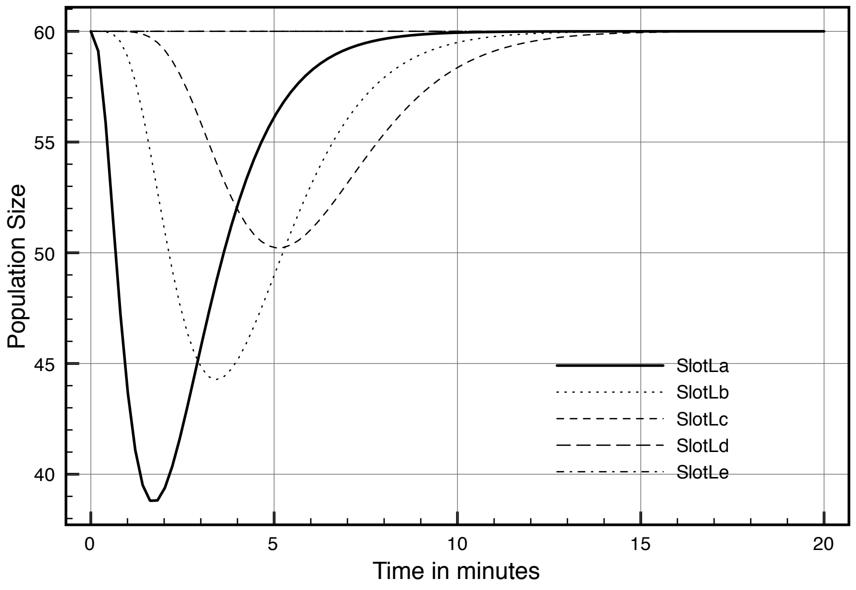

Fig. 9. Number of slots free over time (ODE approx.)

respectively is shown in Fig. 11. Note the different time-scale in comparison with the preceding figures.

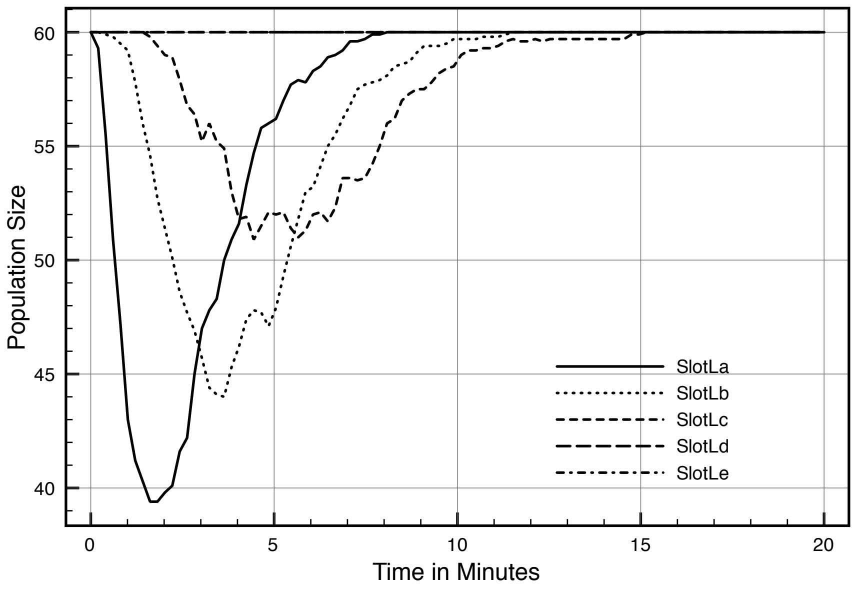

Fig. 10. Number of slots free over time (stochastic simulation)

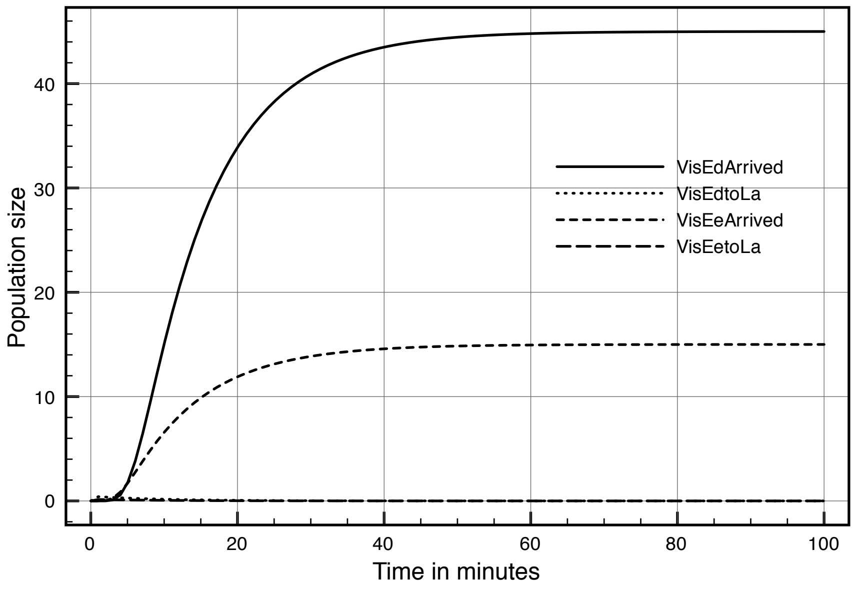

Fig. 11. Number of visitors arriving (ODE approx.)

4.3 A stochastic model with shared displays

Although a smart environment where every user has their own individual display is an interesting example to study, there are also many situations in which people share a common display on which information is made public. In this section a variant of the previous model is developed replacing individual displays by a single display per location that is shared by the visitors present in that location. The display is assumed to have several slots so that routing information for different destinations can be displayed at the same time.

The model of the GAUDI system with shared displays consists of some additional processes modelling a slot arbiter which takes care of the fact that a shared display does not show two or more slots with the same information. As before, each type of process other than the visitor is instantiated for each specific location. In particular, to have a general specification able to deal with different flows of visitors, a distinct arbiter is used for each location/destination pair. The focus in this analysis in on location a and destination e. The process *ArbLaEe* is described which deals with requests from those visitors present in location a having final destination e.

Process *ArbLaEe* starts with an empty display and waits for a request for information from a visitor. If such a request arrives (*laee*), it sets up a slot to display the required information by sending a request to the $SlotManager$ process (*slaee*). Further requests for the same information are then no longer handled by the arbiter but directly by the relevant slot. Requests for routing information for different destinations are still handled by the arbiter which will start up new slots via the $SlotManager$ as long as there are empty slots available.

Slots also become free after a while because the requested information is shown only for a limited amount of time. When the slot becomes free, the arbiter is informed via the $SlotManager$ and adjusts its state that records the number and kind of slots currently in use.

The full PEPA specification of the arbiter process for location a and destination e is described below. All activities internal to arbiters and slots are considered to be relatively fast, thus a rate of 100 has been chosen for them:

$$\begin{aligned}
\mathit{ArbLaEe} &= (\mathit{laee}, a).\mathit{ArbLaEeGetSlot} \\
\mathit{ArbLaEeGetSlot} &= (\mathit{slaee}, 100.0).\mathit{ArbLaEeFreeSlot} \\
\mathit{ArbLaEeFreeSlot} &= (\mathit{sfreelae}, 100.0).\mathit{ArbLaEe}
\end{aligned}$$

The arbiter processes for the other location/destinations are similar. If the destination is only a final destination, the arbiter does not need to set up a slot for visitors as they arrive. So, for location e, assuming that visitors entering e have reached their destination, the arbiter is simply:

$$\mathit{ArbLeEe} = (\mathit{leee}, a).\mathit{ArbLeEe}$$

The initial state of the $SlotManager$ for location a is state *SmanLae*. In this state, the $SlotManager$ can receive a request (*slaee*) from the local arbiter to set-up a slot that displays routing information for that location (a) to a particular destination (e). The request is followed by an initialisation message from the *SlotManager* to a free slot (*sslaee*) and a transition of the *SlotManager* to state *ReleaseSlotLaee*. The reverse occurs when a slot is getting free. This event ($freelae$) is notified to the *SlotManager*, in state *ReleaseSlotLaee*, and then forwarded to the local arbiter that keeps track of free slots and the information on display.

$$\begin{aligned}
&\mathit{SmanLae} &&= (\mathit{slaee}, 100.0).\mathit{StartSlotLaee} \\
&\mathit{StartSlotLaee} &&= (\mathit{sslaee}, 100.0).\mathit{ReleaseSlotLaee} \\
&\mathit{ReleaseSlotLaee} &&= (\mathit{freelae}, f).\mathit{InformArbLae} \\
&\mathit{InformArbLae} &&= (\mathit{sfreelae}, 100.0).\mathit{SmanLae}
\end{aligned}$$

The $SlotManager$ processes for the other location/destinations are similar. A slot of a display at location a can be set up to display routing information for a particular destination after receiving a message from the $SlotManager$. After it has been set-up, the slot responds to visitors that are currently seated in location a. The process *SlotLaRespEe* handles requests of visitors with destination e when the slot has been set-up for showing routing information for that destination (process *SlotLaRespEd* handles requests of visitors with destination d when the slot has been set-up for destination d, and so on). The local routing information is included in the local *Slot* processes. *SlotLaRespEe* first displays location b (*eealb*, with rate r), and then waits for other possible requests from visitors (*laee*) in race condition with the timeout for releasing the slot ($freelae$). If a new request arrives before the timeout expires, location b is displayed, with rate rr (assumed to be larger than r). Action $freelae$ synchronises with the local $SlotManager$ that in turn takes care of communicating the release of the slot to the local arbiter.

$$\begin{aligned}
\mathit{SlotLa} &= (\mathit{sslaed}, 100.0).\mathit{SlotLaRespEd} + (\mathit{sslaee}, 100.0).\mathit{SlotLaRespEe} + \\
&\quad (\mathit{sslaec}, 100.0).\mathit{SlotLaRespEc} \\
\mathit{SlotLaRespEd} &= (\mathit{edalb}, r).\mathit{SlotLadispEd} \\
\mathit{SlotLadispEd} &= (\mathit{laed}, a).\mathit{SlotLashowEd} + (\mathit{freelad}, f).\mathit{SlotLa} \\
\mathit{SlotLashowEd} &= (\mathit{edalb}, \mathit{rr}).\mathit{SlotLadispEd} \\
\mathit{SlotLaRespEe} &= (\mathit{eealb}, r).\mathit{SlotLadispEe} \\
\mathit{SlotLadispEe} &= (\mathit{laee}, a).\mathit{SlotLashowEe} + (\mathit{freelae}, f).\mathit{SlotLa} \\
\mathit{SlotLashowEe} &= (\mathit{eealb}, \mathit{rr}).\mathit{SlotLadispEe} \\
\mathit{SlotLaRespEc} &= (\mathit{ecalb}, r).\mathit{SlotLadispEc} \\
\mathit{SlotLadispEc} &= (\mathit{laec}, a).\mathit{SlotLashowEc} + (\mathit{freelac}, f).\mathit{SlotLa} \\
\mathit{SlotLashowec} &= (\mathit{ecalb}, \mathit{rr}).\mathit{SlotLadispEc}
\end{aligned}$$

The places available in each location that may be occupied by visitors are modelled by the following processes where e.g. $PlaceFreeLa$ models a free place in location a and when a visitor sits down (action $lasd$) it gets occupied, which is modelled by state $PlaceFullLa$.

$$\begin{aligned}
\mathit{PlaceFreeLa} &= (\mathit{lasd}, s).\mathit{PlaceFullLa} \\
\mathit{PlaceFullLa} &= (\mathit{lasu}, s).\mathit{PlaceFreeLa}
\end{aligned}$$

The other places are defined in a similar way. The model of the visitors is similar to that presented in Section 4.2. A full presentation of this specification is omitted. The generation of the various visitor processes and their relative rates only are shown. In this case there are four groups of visitors. One group starting in a heading for d, one group starting from c heading to e, one group starting from a heading to c and the last group starting from d and heading to a. There are 400 visitors in total, so with the specified rates for $v0$ to $v3$ the groups are composed of respectively 25, 75, 100 and 200.

$$\begin{aligned}
\mathit{v0} &= 0.0625 \\
\mathit{v1} &= 0.1875 \\
\mathit{v2} &= 0.25 \\
\mathit{v3} &= 0.5 \\
\mathit{Vis} &= (\mathit{nop}, \mathit{v0}).\mathit{VisEdtoLa} + (\mathit{nop}, \mathit{v1}).\mathit{VisEetoLc} + \\
&\quad (\mathit{nop}, \mathit{v2}).\mathit{VisEctoLa} + (\mathit{nop}, \mathit{v3}).\mathit{VisEatoLd}
\end{aligned}$$

Finally the overall composition of the system is given by the following PEPA composition expression. It composes in parallel 400 visitors, divided into four different groups, each with a different destination, and 100 places and 2 slots per location. The processes synchronise over all the actions listed with the cooperation operator. There is one $SlotManager$, modelling the local display, and one $arbiter$ per location for each possible destination. Each display has 2 slots.

$$\begin{aligned}
&(\mathit{SmanLad}[1] \parallel \mathit{SmanLae}[1] \parallel \mathit{SmanLac}[1] \parallel \\
&\mathit{SmanLbd}[1] \parallel \mathit{SmanLbe}[1] \parallel \mathit{SmanLbc}[1] \parallel \mathit{SmanLba}[1] \parallel \\
&\mathit{SmanLcd}[1] \parallel \mathit{SmanLce}[1] \parallel \mathit{SmanLca}[1] \parallel \\
&\mathit{SmanLde}[1] \parallel \mathit{SmanLdc}[1] \parallel \mathit{SmanLda}[1] \parallel \\
&\mathit{SmanLed}[1] \parallel \mathit{SmanLec}[1] \parallel \mathit{SmanLea}[1] \parallel \mathit{Vis}[400]) \\
&\bowtie L \\
&(\mathit{SlotLa}[2] \parallel \mathit{SlotLb}[2] \parallel \mathit{SlotLc}[2] \parallel \mathit{SlotLd}[2] \parallel \mathit{SlotLe}[2] \parallel \\
&\mathit{ArbLaEd}[1] \parallel \mathit{ArbLaEe}[1] \parallel \mathit{ArbLaEc}[1] \parallel \mathit{ArbLaEa}[1] \parallel \\
&\mathit{ArbLbEd}[1] \parallel \mathit{ArbLbEe}[1] \parallel \mathit{ArbLbEc}[1] \parallel \mathit{ArbLbEa}[1] \parallel \\
&\mathit{ArbLcEd}[1] \parallel \mathit{ArbLcEe}[1] \parallel \mathit{ArbLcEc}[1] \parallel \mathit{ArbLcEa}[1] \parallel \\
&\mathit{ArbLdEd}[1] \parallel \mathit{ArbLdEe}[1] \parallel \mathit{ArbLdEc}[1] \parallel \mathit{ArbLdEa}[1] \parallel \\
&\mathit{ArbLeEd}[1] \parallel \mathit{ArbLeEe}[1] \parallel \mathit{ArbLeEc}[1] \parallel \mathit{ArbLeEa}[1] \parallel \\
&\mathit{PlaceFreeLa}[100] \parallel \mathit{PlaceFreeLb}[100] \parallel \mathit{PlaceFreeLc}[100] \parallel \mathit{PlaceFreeLd}[100]) \parallel \mathit{PlaceFreeLe}[100])
\end{aligned}$$

where:

$$\begin{aligned}
L &= (\mathit{Requests} \cup \mathit{Responses} \cup \mathit{ArbReqSlot} \cup \mathit{StartSlot} \cup \\
&\quad \mathit{ArbFreeSlot} \cup \mathit{ReleaseSlot} \cup \mathit{Sitdown} \cup \mathit{Standup}) \\
\mathit{Requests} &= \{lXeY | X \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}\} \\
\mathit{Responses} &= \{eYXlZ | X, Z \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}, y \neq x\} \\
\mathit{ArbReqSlot} &= \{slXeY | X \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}\} \\
\mathit{ArbFreeSlot} &= \{sfreelXY | X \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}\} \\
\mathit{StartSlot} &= \{sslXeY | X \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}\} \\
\mathit{ReleaseSlot} &= \{freelXY | X \in \{a,b,c,d,e\}, Y \in \{a,c,e,d\}\} \\
\mathit{Sitdown} &= \{lXsd | X \in \{a,b,c,d,e\}\} \\
\mathit{Standup} &= \{lXsu | X \in \{a,b,c,d,e\}\}
\end{aligned}$$

4.4 Stochastic simulation of the model with shared displays

The model described in Section 4.3 is analysed using stochastic simulation. The values for the parameters of the model are defined in Table 1. Time units are in minutes.

Rate	Meaning
$s = 10$	On average a visitor takes 6 seconds to sit down on a free seat
$a = 1$	On average a visitor needs 1 minute to request information
$r = 2$	On average a slot takes 30 seconds to display the required info
$rr = 10$	On average a visitor takes 6 seconds to read indications already show
$f = 0.2$	On average information remains 5 minutes on display

Table 1. Definition of parameter values

The first result based on stochastic simulation explores resources (Fig. 12) and in particular how the number of occupied places in each location changes over time. This provides insight in potential congestion. It does not explain why the congestion occurs but it provides data that makes it possible to make comparison between alternative routes for groups of visitors through the building. The results are shown for a time interval ranging from 0 to 300 minutes showing average values calculated over 10 independent replications (100 data points, confidence interval 0.05).

The simulation indicates that, as expected, in this example the places in locations a and d, where many visitors start from, are fully occupied for some time. Location d is occupied for almost the whole period until all visitors have arrived at their final destination. Furthermore, the simulation can be used to show the number of visitors arriving at their respective destinations e, a, d and c (Fig. 13) as well as the distribution of free slots over the various locations (Fig. 14). Although there are 4 different destinations that visitors are heading for, the figure correctly shows that at most two slots are occupied at any time. In location d at most one slot is used. In fact, only visitors coming in at d and heading for location a (via c) request information in location d.

4.5 Fluid flow analysis of the model with shared displays

In this section the results of a Fluid Flow (ODE) analysis of the model with shared displays introduced in Section 4.3 is given. Figure 15 shows the number of places occupied in the various locations by the visitors while they move from the locations from which they enter the building to their respective destinations. The first observation is that in location d the maximum number of places is being occupied for a certain amount of time (to be precise, for about 100 minutes approximately as shown by the related curve). In the other locations there are almost always enough free places available for visitors. The correspondence between the simulation results shown

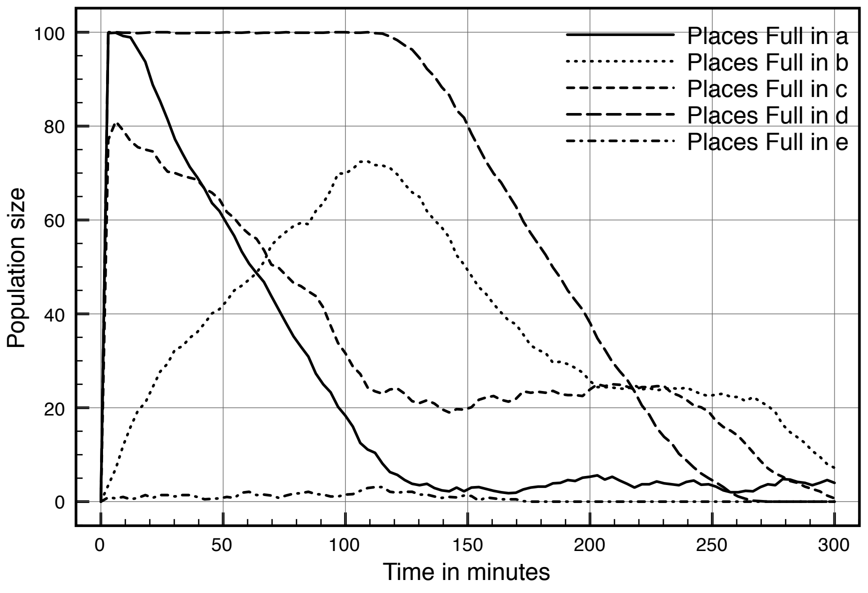

Fig. 12. Number of full places/seats over time

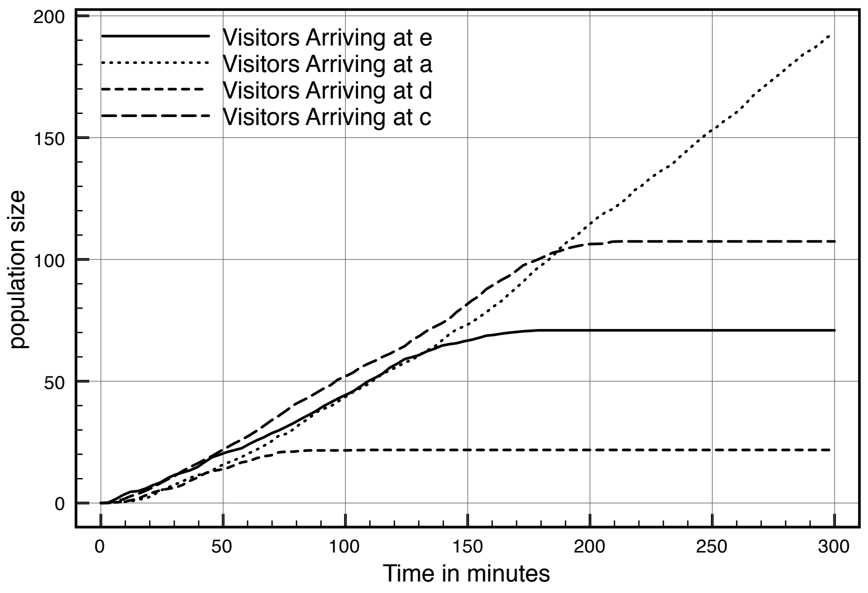

Fig. 13. Number of visitors arriving at *a*, *c*, *d*, and *e*

in Figure 12 and the results obtained via ODE (Fig. 15) suggests that for larger numbers the ODE results are predictive.

Analysis of the specification can also provide information about how long it takes before all visitors have arrived at their destination. This is shown in Figure 16. It can be seen that already after 50 minutes all visitors with final destination *d* have arrived. The figure shows also that

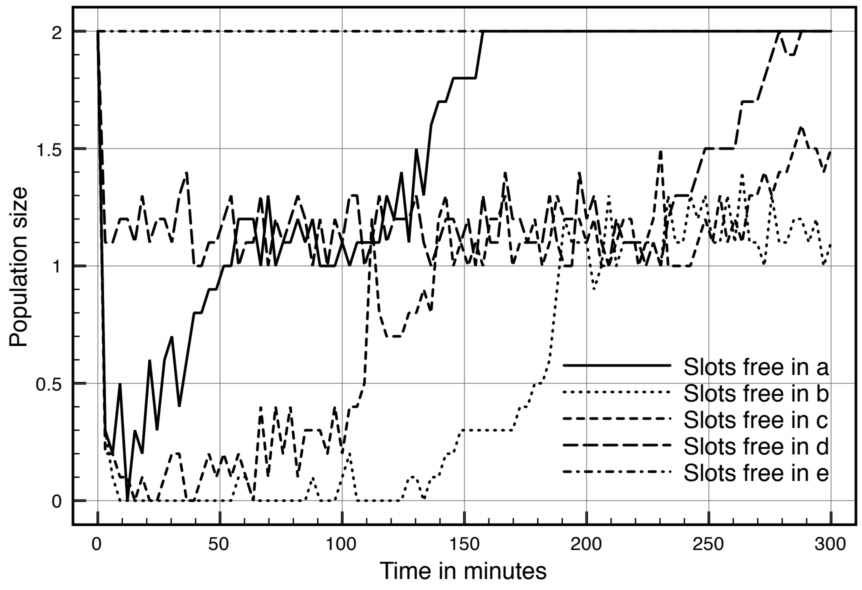

Fig. 14. Number of free slots over time

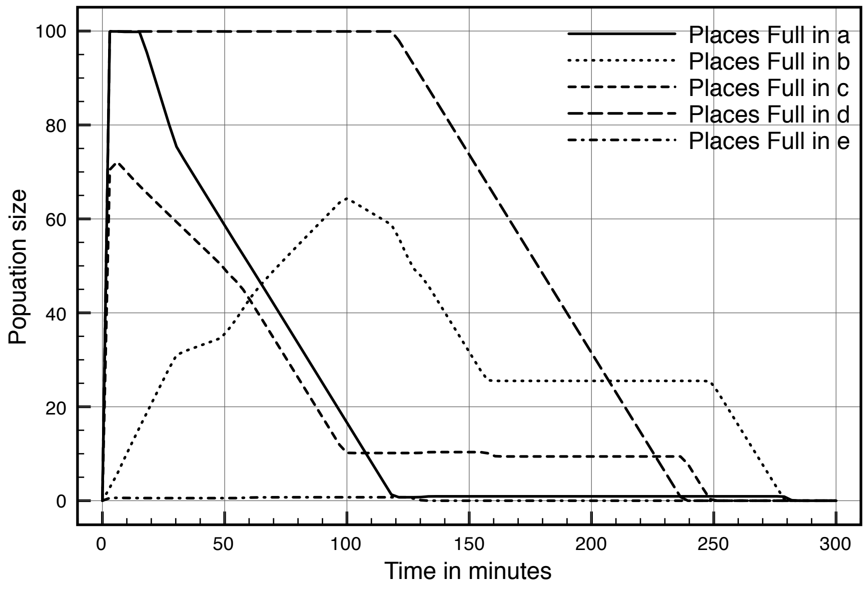

Fig. 15. Occupied places over time [ODE]

the total number of visitors for destination d is 25, that for e is 75, that for c is 100 and that for destination a is 200, corresponding to what has been specified. Also in this case the graph corresponds to the results of the simulation.

Figure 17 shows the average number of empty slots on the displays in the various locations. Initially all slots are empty, but they rapidly get occupied when visitors arrive at the locations.

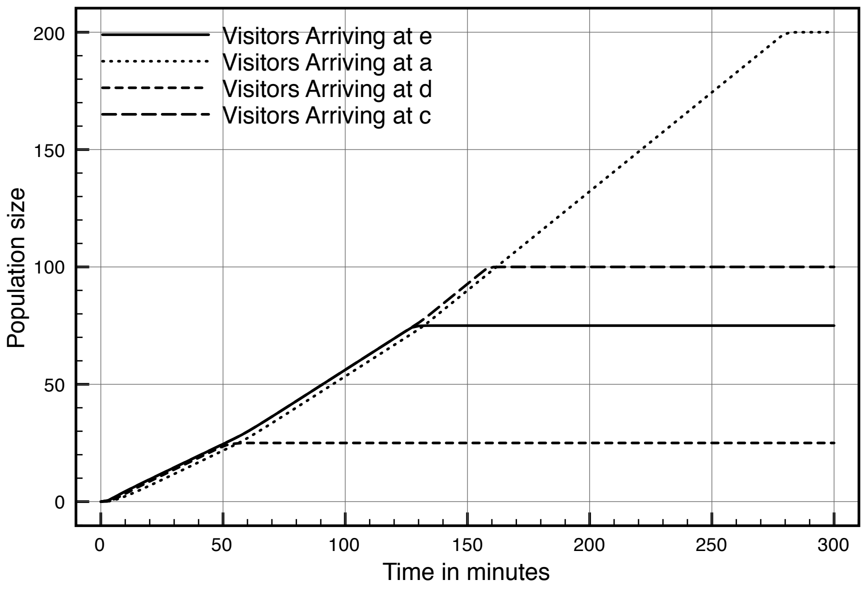

Fig. 16. Visitors arriving at their final destinations

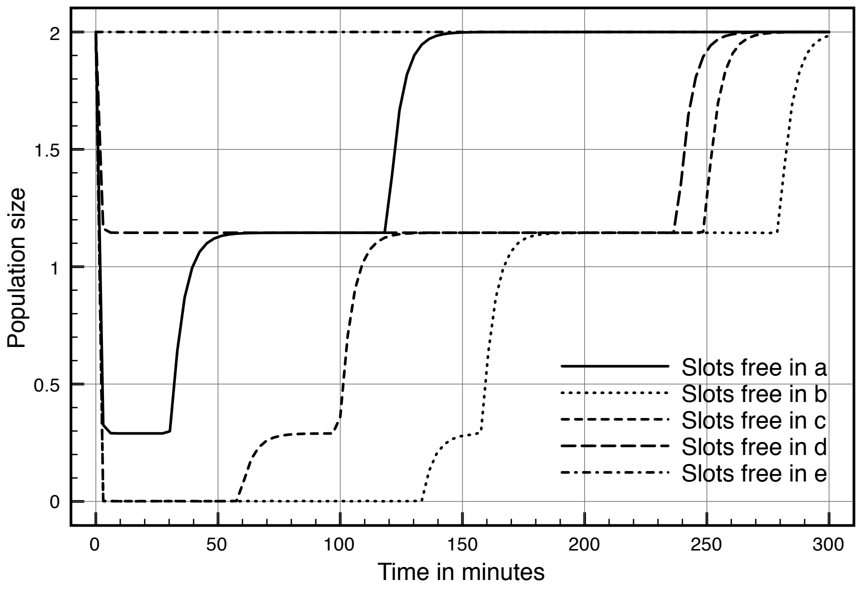

Fig. 17. Free slots in display in each location

When all visitors have arrived and the information on the displays has been removed, all slots return to their empty state. The displays have at most two slots occupied, as specified. In location d only at most one slot is occupied because only visitors heading for a single destination pass by. In location e the slots remain always free because it is only a final destination in the current model, so no visitors ever need to get information on where to go next.

The results in this section show that the ODE technique can indeed deal with a very high number of visitors. This is because it represents them, and their states, as continuous quantities instead of as a large set of discrete entities with interleaving behaviour. While simulation can always be applied to the PEPA specifications, they can be time-consuming if a high level of accuracy is required. In that case, a large number of independent replications would have to be generated and analysed.

For certain kinds of analysis it has been found that an ODE approximation is much faster and gives a good approximation of the limit behaviour of what could be obtained by simulation that considered a large number of independent replications. The reliability of the results obtained with ODE can be easily (and often quickly) checked by comparing the results with a 'limited' simulation where only a few independent executions are considered. If there is a reasonable correspondence, then the ODE results can be considered sufficiently reliable. Simulation will always provide less smooth curves when only a few executions are considered than curves obtained with the ODE analysis. If there is no reasonable correspondence, then, for that particular analysis, reference should be made only to results obtained by simulation.

5 Automatically generating PEPA/GAUDI-specifications

One of the obstacles to the use of the analysis technique illustrated here is the effort required to develop the stochastic models. This is particularly a problem if the aim is to compare results in relation to different routing schedules, different groups of visitors or even varying layouts of the buildings. The PEPA process algebra involves few basic operators and provides versatility in terms of the different automatic analysis techniques that can be applied to the same specification. Its very simplicity means that specifications can become lengthy. It is necessary to encode many aspects of the modelled system using processes and action names.

To reduce the effort of generating these model variants a program was written that synthesises PEPA models automatically, given domain specific input such as the layout of the building, routing information, grouping of visitors and their final destination. The PEPA specifications for the guidance system were made sufficiently modular and each component as general as possible to enable the appropriate instantiation. The program takes three input files of triples:

– (location name, number of places, number of slots) for each location. Each triple represents the number of available places and the number of available slots for each location in the building.
– (entering location, final destination, number of visitors). Each triple represents a group of visitors with the location where they enter, their final destination and the number of members in the group.
– (location name, final destination, next location). This file is a routing table where for each location and destination (of the visitors) the next location is indicated.

This information is sufficient to generate a corresponding PEPA specification that is amenable to ODE-analysis. Although the synthesiser is currently a prototype, it shows that the approach is viable. An example is used to illustrate the approach which is a variant of that presented in Section 3.1. The layout of the example building with its rooms is presented in Fig. 18. A GAUDI model is considered with multiple individual displays as in Section 4.2. The only difference is the way visitors groups are generated. This will be explained later.

As before each location is assumed to have a sufficient number of places where visitors can sit down and a sufficient number of individual displays. The location R represents the reception where visitors arrive at the building. The location SG denotes the staircase at ground-level whereas GF denotes the same staircase at the first floor. The locations at ground-level are GA and GB, those at the first floor are FA, FB and FC.

A configuration was explored in which 120 visitors start at the reception with 20 of them heading for room FB at the first floor and 100 heading to room GB at the ground floor. The complete contents of the three files for the example at hand can be found in Annex A.

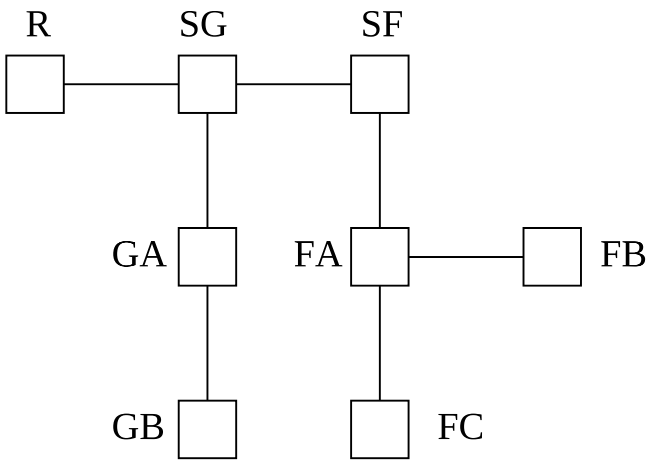

Fig. 18. Map of a building similar to that of the example in Section 3.1

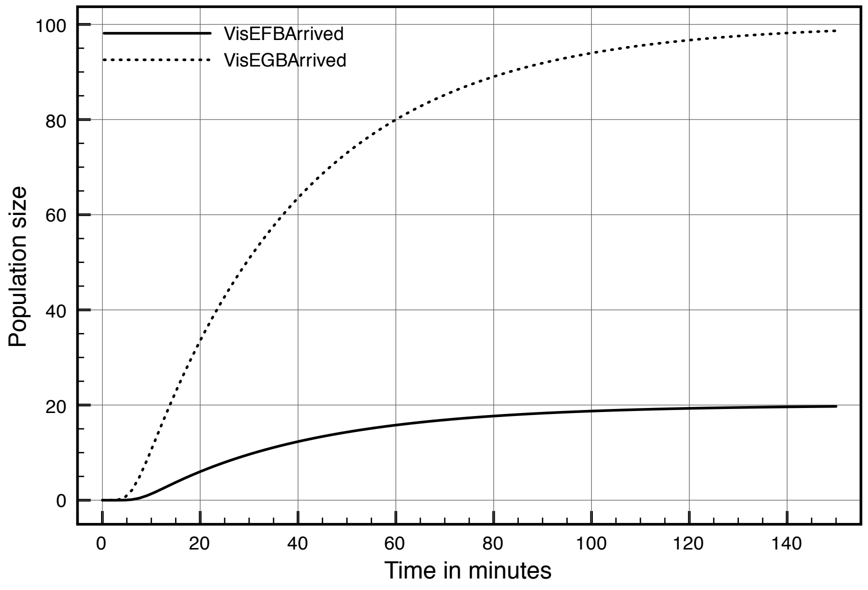

Fig. 19. Arrival of visitors

This simple information is sufficient, for a given configuration of a building and relevant information about visitors and their destinations, to generate automatically a corresponding PEPA specification. This specification is amenable to the same simulation and ODE analyses as illustrated in Section 4. For this specification a version was used that defines the arrival of visitors in the following way:

$$VisRFB = (nop, vFBR).VisEFBtoLR$$
$$VisRGB = (nop, vGBR).VisEGBtoLR$$

where $vFBR = vGBR = 0.03$.

In Figure 19 the number of arriving visitors is shown for the building of Figure 18 for 120 visitors starting at the reception with 20 of them heading for room FB at the first floor and 100 heading to room GB at the ground floor.

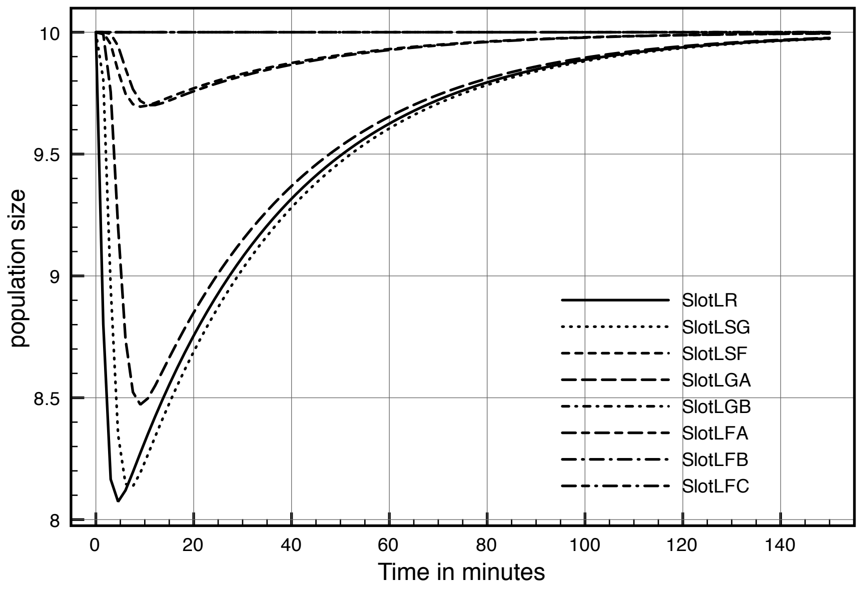

Fig. 20. Unused displays on a total of 10 per location

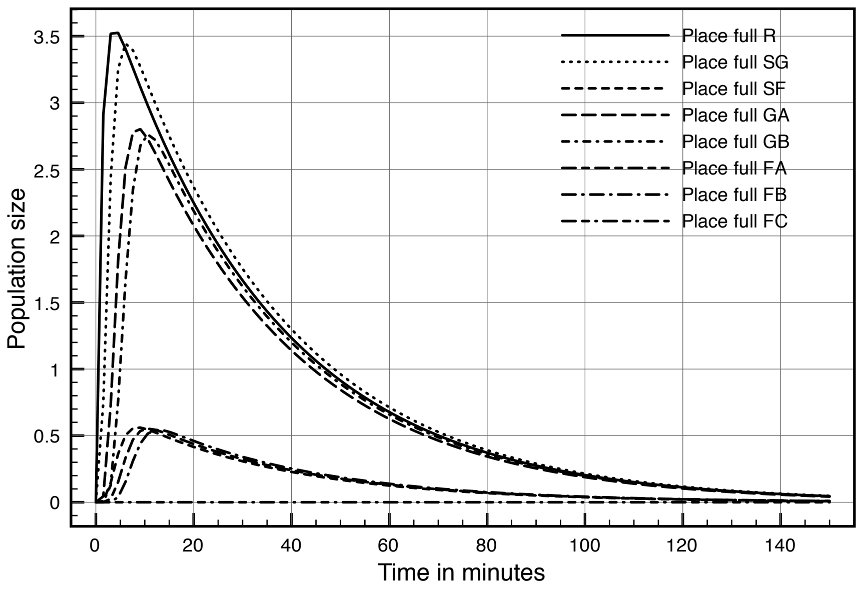

Fig. 21. Occupied places over time

Figures 20 and 21 show the use of slots and places over time for the same configuration.

The GAUDI synthesiser makes it possible to investigate configurations where visitors start in different locations (i.e. not necessarily at the reception) and evaluate the consequences of such a distribution of visitors. For example, Figure 22 shows again the number of places occupied of the same structure, but now with the 100 visitors heading for location *GB* starting from location

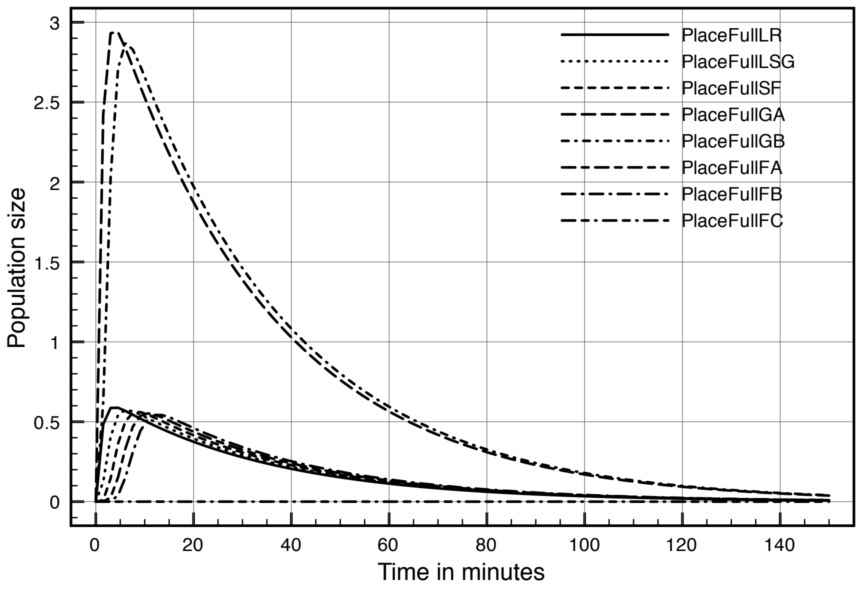

Fig. 22. Places occupied when visitors start from locations R and GA

GA. This scenario keeps both flows of visitors separate. Comparing Figure 21 with Figure 22 it can be seen that in locations R and SG the number of occupied places has diminished due to the different flow of the visitors while at the same time the number of occupied places in location GA has increased. This corresponds to what would be expected for this new scenario.

A prototype of the synthesiser has been implemented in the functional programming language Haskell [5]. This language allows for fast prototyping.

6 Scaling up

The GAUDI synthesiser also made it possible to explore larger examples. One of the concerns for being able to apply this approach to realistic situations is that it should be possible to handle configurations with a large number of people and locations that are connected in interesting ways. As a first larger experiment a configuration has been defined with 26 locations (labelled a to z). In each location a number of visitors enter that range from 10 to 120 with a total of 420. Each individual visitor is entering their location at a rate of 0.03. The values of the parameters a, r and s are 1, 2 and 10 respectively. This means that request rates, response rates and average time that visitors need to sit-down and stand-up is the same in each location that they visit. This assumption is only made for simplicity. There is no technical reason that prevents the definition of these rates per location or per visitor group as desired.

Furthermore, all visitors are assumed to move to the unique exit g. The configuration of the locations is as follows:

- Location g is the single exit location
- Locations a-b-c-g are connected in a line.
- Locations f-e-d-c-g are connected in a line.
- Locations z-y-x-g are connected in a line.
- Locations w-v-u-t- ... -i-h-g are connected in a line.

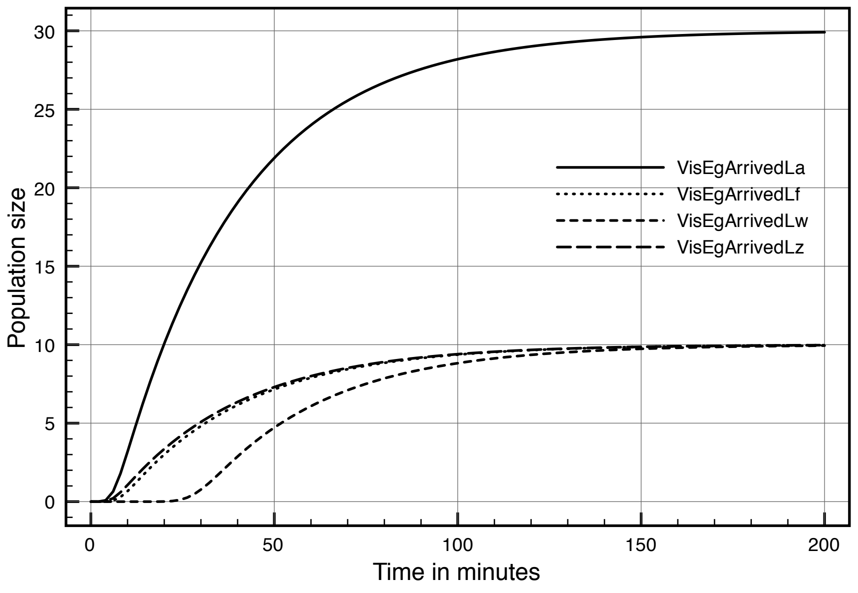

Fig. 23. Visitors arriving at g from locations a, f, w and z.

So, the configuration has a kind of star-architecture where three 'corridors of locations' each terminate in location g, except that visitors coming from locations f-e-d are also passing through c before reaching g.

The initial distribution of the visitors and the resources in each location is given in Table 2.

Table 2. Initial distribution of visitors, places and slots over the locations

process	a	b	c	d	e	f to z
visitors	30	20	20	120	20	10
places	30	20	20	120	20	20
slots	10	10	10	10	10	10

The configuration generated a PEPA model with multiple individual one-slot displays. The result of the ODE analysis for visitors arriving at location g from locations a, f, w and z is shown in Figure 23. The visitors from location w have to pass through many other locations before arriving at the exit location g. This is also reflected in the figure where the curve labelled $VisEgArrivedLw$ shows that visitors from w start to arrive only after 25 time units, whereas for example visitors from location a start to arrive already after 4 time units.

Figure 24 shows the number of free slots in the locations c, h and x obtained using ODE analysis, and Figure 25 shows the simulation results for 10 independent replications of free slots for the same locations. The number of slots occupied in place c is clearly higher than in the other two locations shown. This is due to the particular configuration of the locations and the routing which makes it necessary for visitors from many locations to pass through location c causing an accumulation of visitors in that area.

This is an example of the kind of analysis that can be performed with this approach. The ODE analysis can show, in a much faster way, potential effects of concentration of visitors in certain areas of a building depending on the particular routes that they are following. ODE analysis for

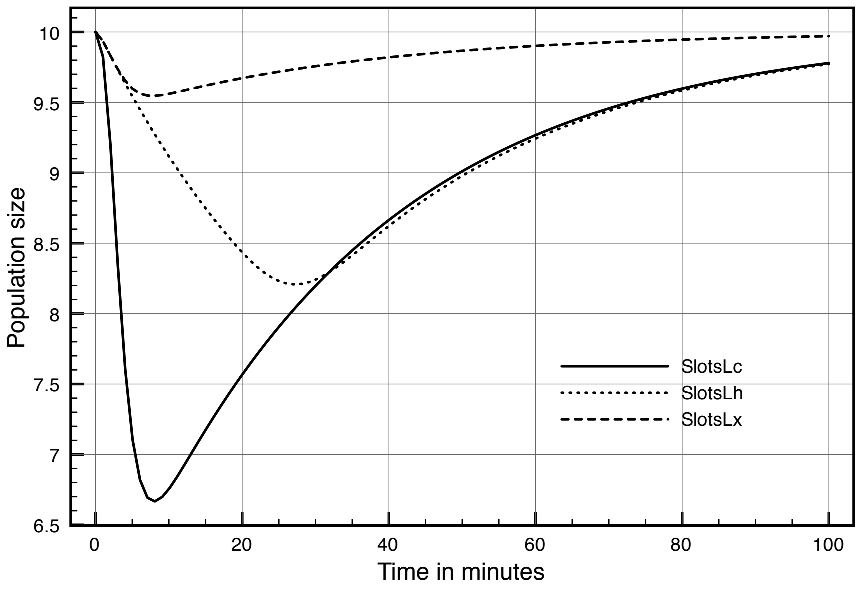

Fig. 24. Free slots in locations c, h and x, ODE-analysis.

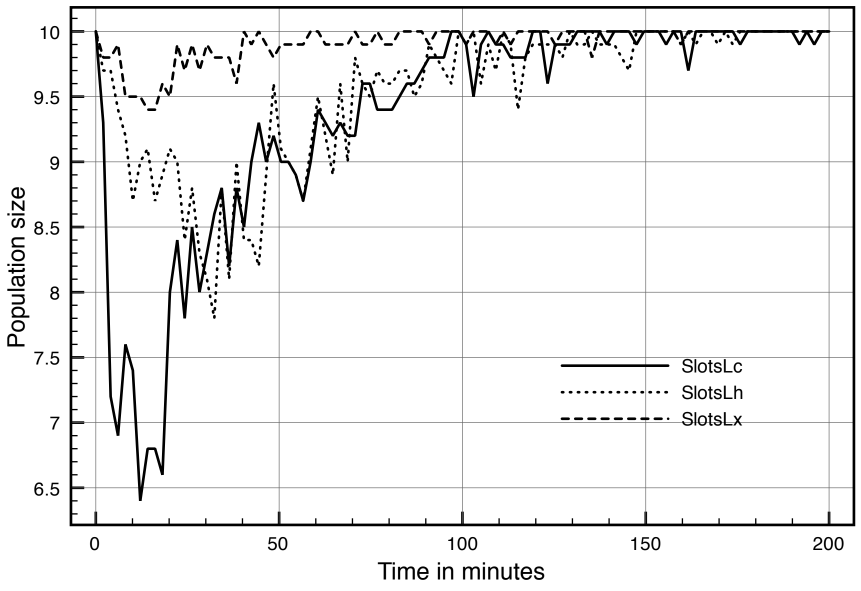

Fig. 25. Free slots in locations c, h and x, simulation results.

this configuration takes just a couple of minutes on a Dual Core 2.3 GHz PowerPC G5 with 1.5 GB memory running a Mac OS X Version 10.4.11 operating system. In comparison, the stochastic simulation performed with Gillespie's algorithm for only 10 independent replications takes almost 30 minutes.

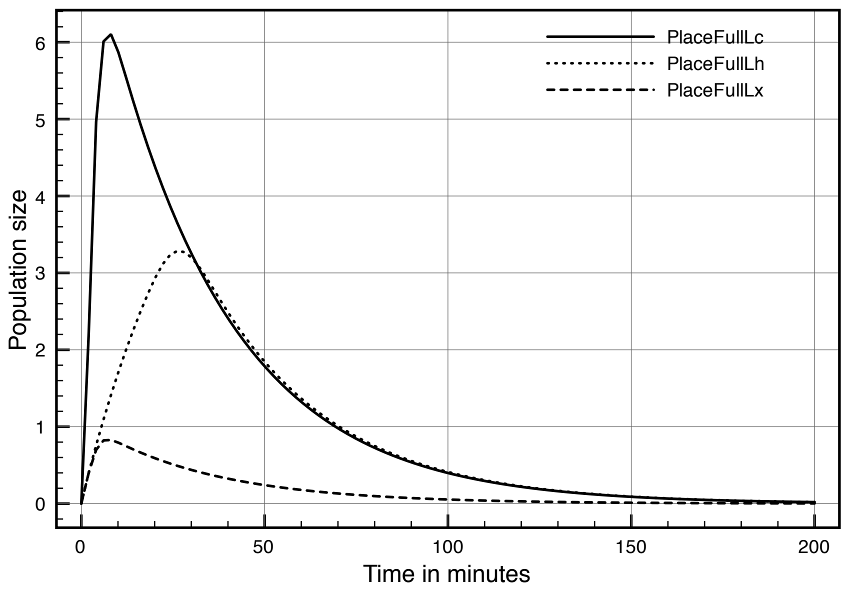

Fig. 26. Occupied places in locations c, h and x, simulation results.

Figure 26 shows the number of places occupied in the same locations using ODE analysis. Also in this case a clear increase in the occupancy of places can be noted for location c.

7 Analysis with PRISM

PEPA specifications can be automatically translated into the specification language of the stochastic model checker PRISM [18]. Although this analysis technique cannot handle specifications with the large number of visitors that would be of interested, it can be used to verify properties of specifications in which the number of visitors is limited to just a few. This way properties of the specification can be checked that help to understand whether the specification models what was intended.

The properties can be expressed in the stochastic temporal logic CSL. Continuous Stochastic Logic (CSL) [2, 3] is a stochastic variant of the well-known Computational Tree Logic (CTL) [7]. CTL allows for stating properties over *states* as well as over *paths*.

CSL extends CTL using two probabilistic operators that refer to the steady-state and transient behaviour of the system being studied. The steady-state operator refers to the probability of residing in a particular state or set of *states* (specified by a state formula) in the long run. The transient operator allows refers to the probability mass of the set of *paths* in the CTMC that satisfies a given (path) property. In order to express the time span of a certain path, the path operators until ($\mathcal{U}$) and next (X) are extended with a parameter that specifies a time interval. Let I be an interval on the real line, p a probability value and $\bowtie$ a comparison operator, *i.e.* $\bowtie \in \{ <, \leq, \geq, > \}$. The syntax of CSL is:

State formulae

$$\Phi ::= a \;\Big|\; \neg\Phi \;\Big|\; \Phi \vee \Phi \;\Big|\; \mathcal{S}_{\bowtie p}(\Phi) \;\Big|\; \mathcal{P}_{\bowtie p}(\varphi)$$

$\mathcal{S}_{\bowtie p}(\Phi)$: probability that Φ holds in steady state $\bowtie p$
$\mathcal{P}_{\bowtie p}(\varphi)$: probability that a path fulfills $\varphi \bowtie p$

Path formulae

$$\varphi ::= X^I\,\Phi \;\Big|\; \Phi\,\mathcal{U}^I\,\Phi$$

$X^I\,\Phi$: next state is reached at time $t \in I$ and fulfills Φ
$\Phi\,\mathcal{U}^I\,\Psi$: Φ holds along path until Ψ holds at $t \in I$

The meaning of atomic propositions (a), negation ($\neg$) and disjunction ($\vee$) is standard. Using these operators, other Boolean operators like conjunction ($\wedge$), implication ($\Rightarrow$), true (TRUE) and false (FALSE), and so forth, can be defined in the usual way. The state formula $\mathcal{S}_{\bowtie p}(\Phi)$ asserts that the steady-state probability for the set of states satisfying Φ, the Φ-states, meets the bound $\bowtie p$. $\mathcal{P}_{\bowtie p}(\varphi)$ asserts that the probability measure of the set of paths satisfying φ meets the bound $\bowtie p$. The operator $\mathcal{P}_{\bowtie p}(.)$ replaces the usual CTL path quantifiers $\exists$ and $\forall$. In CSL, the formula $\mathcal{P}_{\geq 1}(\varphi)$ holds if *almost all* paths satisfy φ. Moreover, clearly $\exists\,\varphi$ holds whenever $\mathcal{P}_{>0}(\varphi)$ holds.

The formula $\Phi\,\mathcal{U}^I\,\Psi$ is satisfied by a path if Ψ holds at time $t \in I$ and at every preceding state on the path, if any, Φ holds. In CSL, temporal operators like $\diamond$, $\square$ and their real-time variants $\diamond^I$ or $\square^I$ can be derived, *e.g.*, $\mathcal{P}_{\bowtie p}(\diamond^I\,\Phi) = \mathcal{P}_{\bowtie p}(\text{TRUE}\,\mathcal{U}^I\,\Phi)$ and $\mathcal{P}_{\geq p}(\square^I\,\Phi) = \mathcal{P}_{<1-p}(\diamond^I\,\neg\Phi)$. The untimed next and until operators are obtained by $X\,\Phi = X^I\,\Phi$ and $\Phi_1\,\mathcal{U}\,\Phi_2 = \Phi_1\,\mathcal{U}^I\,\Phi_2$ for $I = [0,\infty)$. In a variant of CSL the probability p can be replaced by a question mark, denoting that one is looking for the value of the probability rather than verifying whether the obtained probability respects certain bounds.

The stochastic model checker PRISM [18] is a prototype tool that supports, among others, the verification of CSL properties over CTMCs. It checks the validity of CSL properties for given states in the model and provides feedback on the calculated probabilities of such states where appropriate. It uses symbolic data structures (*i.e.* variants of Binary Decision Diagrams). PRISM accepts system descriptions in different specification languages, among which the process algebra PEPA and the PRISM language—a simple state-based language based on the Reactive Modules formalism of Alur and Henzinger [1].

PRISM is used to verify a number of simple qualitative and quantitative CSL properties that are useful to get feedback on whether the specification is indeed behaving as expected to a certain extent. As an example, the PEPA specification of Section 4.2 is restricted to 4 visitors. The translated version of this specification in the PRISM language can be found in the Annex. The translation has been performed with the PEPA to PRISM compiler that was kindly made available to us by Stephen Gilmore of the University of Edinburgh. The specification results in a CTMC composed of 249344 states and 1572000 transitions.

First some simple properties where checked to verify the behaviour of the specification. From the parameters for $v0$ and $v1$ it would be expected that 25% of the visitors are those heading for final destination e and 75% those to final destination d. The following results show that this is indeed the case:

```
// What is the probability that a generic visitor process evolves into a visitor heading
// to exit e?
P=? [ true U (Vis_STATE=VisEetoLa)]
// Result: 0.25

// What is the probability that a generic visitor process evolves into a visitor heading
// for exit d?
P=? [true U (Vis_STATE=VisEdtoLa)]
// Result: 0.75
```

Another simple property is to verify whether visitors arrive at their final destination. As an example the probability that a generic visitor process ends up in a state indicating that it arrived at final destination e is evaluated. The resulting probability is 0.25, which corresponds to what was expected.

```
// What is the probability that a generic visitor process arrives at destination e?
P=? [ true U (Vis_STATE=VisEeArrived)]
// Result: 0.25
```

A limited quantitative verification can be performed. For example, the probability that two different instances of the visitor both eventually arrive at their respective destination has been calculated. Likewise it is possible to evaluate the probability that they both arrive within 5 time units.

```
// What is the probability that two instances of a visitor arrive at destinations e and d?
P=? [true U (Vis_2_STATE=VisEeArrived & Vis_3_STATE=VisEdArrived) ]
// Result: 0.1875

// What is the probability that this happens within 5 time units?
P=? [ true U<=5 (Vis_2_STATE=VisEeArrived & Vis_3_STATE=VisEdArrived)]
// Result: 0.0026523
```

Further verification can show whether the place at a particular location gets eventually occupied which is indicating that a visitor passed by at some point. The same question can be asked about occupancy of a slot and whether it eventually responds to request of a certain type.

```
// What is the probability that the place in location c gets eventually occupied?
P=? [ true U (PlaceFreeLc_STATE=PlaceFullLc)]
// Result: 0.996

// What is the probability that the slot in location c responds eventually to a
// request for exit d?
P=? [true U (SlotLc_STATE=SlotLcRespEd)]
// Result: 0.996

// What is the probability that eventually the slot in c responds to a request for
// exit e?
P=? [true U (SlotLc_STATE=SlotLcRespEe)]
//Result: 0.0
```

The following property is more interesting. It can be used to verify that visitors are always coming closer to their final destination. For example, what is the probability that a visitor heading for destination d, once it arrived at location c could eventually go back to location b? This probability is zero, indicating that such a situation does not occur.

```
// What is the probability that eventually a visitor in location c heading for d
// passes by location b?
P=? [ true U (Vis_STATE=VisEdtoLb) {Vis_STATE=VisEdtoLc}]
// Result: 0.0
```

The following property shows that visitors cannot 'skip' a location on the way to their destination. Starting in location a and heading for e a visitor must visit location b, given the defined configuration. The following property evaluates the probability that the slot in location b remains always free until a visitor heading to e has arrived at its final destination. This probability is zero, as expected.

```
// What is the probability that slot b remains free until a visitor heading for e has
// arrived at its destination?
P=? [ (SlotLb_STATE=SlotLb) U (Vis_STATE=VisEeArrived)]
// Result: 0.0
```

The above are only a few examples of the kind of properties that can be verified this way. Model checking with the stochastic model checker PRISM in this case plays a similar role as that of SPIN model checking in the earlier sections. In fact, the PEPA model used for stochastic model checking is a kind of 'individual model' of the GAUDI system in the sense that it is a model with a very limited number of processes in which their identity is preserved. What is important to note though is that when using PEPA the *same* specification can be analysed both applying model-checking techniques as well as applying techniques that are suitable for verification of the collective behaviour.

8 Discussion and future work

This report presents the results of a first exploration of the Fluid Flow approach and more traditional model checking techniques for the modelling and analysis of smart environments. As a representative, but not too complex, example several variants of a dynamic signage system that guides visitors through a building have been considered.

The first models of the system have been specified in Promela, the specification language of the SPIN model-checker. In those models, the position of visitors is relative to a coordinate space and situations are considered in which visitors are not always able to see the displays with routing information. The routing information to the displays is distributed by a central capability giver to the relevant local displays and can be reconfigured dynamically. The detailed model can be explored only by means of stepwise traversal of the specification. The state-space is too large to allow for an analysis by means of model-checking. A reduced version does allow for model-checking a number of properties for a very limited number of visitors. Although stepwise simulation can be very useful for an exploration of a specification by designers, it is clear that model-checking can be used only for very limited models, due to the state-space explosion problems that appear when many visitors are modelled as separate, autonomous processes evolving in parallel.

Further models have been specified in the process algebra PEPA. In these models, the position of the visitors is not modelled as coordinates in a coordinate space, but simply as a state that indicates in which room the visitor resides. The recently developed Fluid Flow analysis, that translates a PEPA specification into a set of Ordinary Differential Equations allows for a number of interesting abstractions to render also models with many visitors and locations amenable to quantitative analysis. The first abstraction is that only the *number* of visitors in a particular state are considered, thus abstracting from the identity of the many processes representing the behaviour of the visitors. A second abstraction is the observation that when visitors move from room to room. In reality the order of arrival of visitors is relevant to the order in which the visitors get their response to their requests. At first sight, this would require a model based on queues. However, for the purpose of quantitative analysis, the order in which visitors are served turns out to be irrelevant since they are all perfectly interchangeable. In fact, changing the order of visitors in a queue does not change the time it would take to move all of them from one room to another.

Using these abstractions, PEPA models have been analysed with many visitors, an interesting number of rooms with a high capacity can be analysed in a rather short time. Moreover, relatively simple synthesising programs can easily be developed to assist designers of smart environments to generate the PEPA models from domain specific information only, such as the architecture of the building, the groups of visitors and the capacity of the various rooms.

These promising results provide an interesting way in which formal methods may be used to explore a whole variety of different designs of smart environments before their deployment with relatively little effort.

The results in this report, though, are still preliminary. There are a number of important and interesting aspects that need to be addressed. First of all, there is the issue of validation of the models. Perhaps, some known systems with many users could be modelled and their results be compared with real observations.

Also from the modelling point of view there are a number of extensions and variants that would be interesting to explore further:

– Modelling of different arrival patterns of visitors, differentiating for example the characteristics of different groups of visitors
– Modelling more complex behaviour of visitors, for example including probabilistic error-behaviour or alternative behaviour in case waiting times exceed certain limits
– Different routing strategies, for example sending visitors with the same destination via different routes with certain probabilities
– Modelling a continuous flow of arriving and leaving visitors
– Investigating different ways to make the analysis easier to use for practitioners in a time and cost efficient way using more sophisticated model-synthesisers.

9 Conclusions

This paper indicates two directions that are proposed for the future development of engineering techniques for smart environments that have a particular focus on the people within the system. The first direction is towards a broader understanding of what usability means in such environments. While more conventional usability analysis is required to assess interaction between device and user, further techniques are required to deal with the implicit interactions that take place in these systems as well as the particular requirements and concerns of crowd-system interaction. The second direction is to address the problem that it is not possible to explore a system in its target environment and sometimes it is difficult to envisage what the impact of a proposed design would be in such an environment. At the same time stakeholders will require strong justification for the introduction of a new system in a commercially or safety sensitive environment (and airport or accident and emergency for example) before such a system can be deployed. This report investigated several model-based analysis techniques to explore the contribution they could provide in analysing usability aspects of smart environments where many people are present before fielding the system. Traditional model-checking techniques such as the SPIN model-checker may help to investigate the interaction taking place between individuals and the environment when moving through the smart environment. However, the presence of many users moving at the same time through the same space and interacting with it in an implicit way, and under certain time-constraints, poses special challenges. An approach based on Ordinary Differential Equations, derived from process algebraic specifications modelling both large groups of users and the environment, may provide complementary useful analysis of the collective behaviour and the more quantitative aspects of usability of smart environments. The paper raises several issues that must be addressed and that are related to each other. One question is how these relatively complex and sophisticated techniques can be made easier to use for usability engineers such that the effort of using them is low with respect to the predictive value that such models may have. Automatically synthesising models starting from domain oriented information, such as the architecture of a building, routing information and visitor characteristics, should be further investigated. A related question is which processes, patterns of properties, and generic smart environment models can be envisaged that will make the task of automatically synthesise these models possible and convenient for developers and which makes it easier to evaluate their predictive potential. These are some of the topics for our future work.

Acknowledgements We would like to thank Stephen Gilmore for kindly providing a stand-alone version of his PEPA to PRISM compiler and his help in overcoming some problems with ODE-analysis.

References

1. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
2. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

3. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.
4. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Enhancing the effective utilisation of grid clusters by exploiting on-line performability analysis. In *Proceedings of CCGRID'05*, pages 317–324. IEEE Computer Society Press, 2005.
5. R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Press, 1998.
6. M. Calder, S. Gilmore, and J. Hillston. Automatically deriving ODEs from process algebra models of signalling pathways. In *Proceedings of CMSB'05*, pages 204–215, 2005.
7. E. M. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.
8. G. J. Doherty, J. C. Campos, and M. D. Harrison. Resources for situated actions. In *XVth International Workshop on the Design, Verification and Speci fication of Interactive Systems (DSV-IS 2008)*, number 5136 in Lecture Notes in Computer Science, pages 194–207. Springer-Verlag, July 2008.
9. D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
10. S. Gilmore and M. Tribastone. Evaluating the scalability of a web service-based distributed e-learning and course management system. In M. Bravetti, M. Nunes, and G. Zavattaro, editors, *Proceedings of WS-FM'06*, number 4184 in Springer Lecture Notes in Computer Science, pages 214–226. Springer-Verlag, 2006.
11. Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *In Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, number 794 in Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1994.
12. B. Haverkort. Markovian models for performance and dependability evaluation. In E. Brinksma, H. Hermanns, and J. P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, number 2090 in Springer Lecture Notes in Computer Science. Springer-Verlag, 2001.
13. R. A. Hayden and J. T. Bradley. ODE-based general moment approximations for pepa. In *Proceedings of the 7th Workshop on Process Algebra and Stochastically Timed Activities.* http://pastaworkshop.org/proceedings, 2008.
14. J. Hillston. Fluid flow approximation of pepa models. In *Proceedings of QEST'05*, pages 33–43. IEEE Computer Society, 2005.
15. J.A. Hillston. *A compositional approach to performance modelling.* Cambridge University Press, 1996.
16. G.J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual.* Addison Wesley, 2003.
17. Christian Kray, Gerd Kortuem, and Antonio Krüger. Adaptive navigation support with public displays. In Robert St. Amant, John Riedl, and Anthony Jameson, editors, *Proceedings of IUI 2005*, pages 326–328, New York, 2005. ACM Press.
18. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P.G. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools Proceedings 12th International Conference, TOOLS 2002*, number 2324 in Springer Lecture Notes in Computer Science, page 200. Springer-Verlag, 2002.
19. M. Tribastone. The PEPA plug-in project. In M. Harchol-Balter, M. Kwiatowska, and M. Telek, editors, *Proceedings of QEST'07*, pages 53–54. IEEE Computer Society Press, 2007.
20. Mark Weiser, Rich Gold, and John Seely Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal*, 38(4):693–696, 1999.

A Input files to generate PEPA specification

In this section an example is given of the input files needed to compile the PEPA model for the configuration introduced in section 5.

The definition of the rate variables of the model is default and the same variable is used for each location and visitor group. This is just for simplifying the presentation and explore some configurations. There is no problem to extend the compiler allowing different rates and to provide these as an additional input file.

```
r = 1 ;
s = 10;
a = 2;
```

```
%
vFBR = 0.03;
vGBR = 0.03;
```

Besides the rates a configuration-file is required with triples of (location, nr. of places, nr. of slots) providing information on available places (seats) and slots on the display in each location.

```
loc places slots
R 120 120
SG 120 120
SF 120 120
GA 120 120
GB 120 120
FA 120 120
FB 120 120
FC 120 120
```

The second argument consists of a file with triples of (location, destination, nr. of visitors) providing information on the number and the destinations of various groups of visitors.

```
loc dest visitors
R FB 20
R GB 100
```

The third argument provides the distributed routing information. It consists of a file with triples providing current location, final destination and next destination:

```
loc dest next
R FB SG
R GB SG
SG FB SF
SG GB GA
SF FB FA
SF GB SG
GA FB SG
GA GB GB
GB FB GA
GB GB GB
FA FB FB
FA GB SF
FB FB FB
FB GB FA
FC FB FA
FC GB FA
```

Finally, in the fourth argument is the name of the file in which the resulting PEPA-model will be saved.

B PRISM Specification of a GAUDI model with five locations

In this section a PRISM specification is presented of the configuration with 5 locations (a, b, c, d and e) as in Section 4.2, each with one seat and one display and with four visitors that enter at location a. One visitor has as destination location e, the others location d. The configuration is as in Figure 4. The PRISM specification is obtained automatically from the PEPA specification by slightly modifying the PEPA specification to remove the typical ODE-PEPA constructs that allow to specify the number of copies of processes involved in the composition.

```
// Output from the PEPA-to-PRISM compiler
// Version 0.04 "Vennel Gate"
// Released: 14-02-2006
//
// Model file: TESTS/5loc_prism

// All PEPA models define CTMCs so mark this as a stochastic model
stochastic

// The rates used in the model
rate r = 1;
rate s = 10;
rate a = 2;
rate v0 = 0.025;
rate v1 = 0.075;

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const Vis = 0;
const VisEdArrived = 1;
const VisEdReca = 2;
const VisEdRecb = 3;
const VisEdRecc = 4;
const VisEdRece = 5;
const VisEdsda = 6;
const VisEdsdb = 7;
const VisEdsdc = 8;
const VisEdsdd = 9;
const VisEdsde = 10;
const VisEdsua = 11;
const VisEdsub = 12;
const VisEdsuc = 13;
const VisEdsud = 14;
const VisEdsue = 15;
const VisEdtoLa = 16;
const VisEdtoLb = 17;
const VisEdtoLc = 18;
const VisEdtoLd = 19;
const VisEdtoLe = 20;
const VisEeArrived = 21;
const VisEeReca = 22;
const VisEeRecb = 23;
const VisEeRecc = 24;
const VisEeRecd = 25;
const VisEesda = 26;
const VisEesdb = 27;
const VisEesdc = 28;
const VisEesdd = 29;
const VisEesde = 30;
const VisEesua = 31;
const VisEesub = 32;
const VisEesuc = 33;
const VisEesud = 34;
const VisEesue = 35;
const VisEetoLa = 36;
const VisEetoLb = 37;
const VisEetoLc = 38;
const VisEetoLd = 39;
const VisEetoLe = 40;
module Vis

        Vis_STATE : [0..40] init Vis;

        [nop] (Vis_STATE = Vis) -> v0 : (Vis_STATE' = VisEetoLa);
        [nop] (Vis_STATE = Vis) -> v1 : (Vis_STATE' = VisEdtoLa);
        [nop] (Vis_STATE = VisEdArrived) -> a : (Vis_STATE' = VisEdArrived);
        [edald] (Vis_STATE = VisEdReca) -> r : (Vis_STATE' = VisEdtoLd);
        [edala] (Vis_STATE = VisEdReca) -> r : (Vis_STATE' = VisEdtoLa);
        [edalb] (Vis_STATE = VisEdReca) -> r : (Vis_STATE' = VisEdtoLb);
        [edalc] (Vis_STATE = VisEdReca) -> r : (Vis_STATE' = VisEdtoLc);
        [edale] (Vis_STATE = VisEdReca) -> r : (Vis_STATE' = VisEdtoLe);
        [edbld] (Vis_STATE = VisEdRecb) -> r : (Vis_STATE' = VisEdtoLd);
        [edbla] (Vis_STATE = VisEdRecb) -> r : (Vis_STATE' = VisEdtoLa);
        [edblb] (Vis_STATE = VisEdRecb) -> r : (Vis_STATE' = VisEdtoLb);
        [edblc] (Vis_STATE = VisEdRecb) -> r : (Vis_STATE' = VisEdtoLc);
        [edble] (Vis_STATE = VisEdRecb) -> r : (Vis_STATE' = VisEdtoLe);
        [edcld] (Vis_STATE = VisEdRecc) -> r : (Vis_STATE' = VisEdtoLd);
        [edcla] (Vis_STATE = VisEdRecc) -> r : (Vis_STATE' = VisEdtoLa);
        [edclb] (Vis_STATE = VisEdRecc) -> r : (Vis_STATE' = VisEdtoLb);
        [edclc] (Vis_STATE = VisEdRecc) -> r : (Vis_STATE' = VisEdtoLc);
        [edcle] (Vis_STATE = VisEdRecc) -> r : (Vis_STATE' = VisEdtoLe);
        [edeld] (Vis_STATE = VisEdRece) -> r : (Vis_STATE' = VisEdtoLd);
        [edela] (Vis_STATE = VisEdRece) -> r : (Vis_STATE' = VisEdtoLa);
        [edelb] (Vis_STATE = VisEdRece) -> r : (Vis_STATE' = VisEdtoLb);
        [edelc] (Vis_STATE = VisEdRece) -> r : (Vis_STATE' = VisEdtoLc);
        [edele] (Vis_STATE = VisEdRece) -> r : (Vis_STATE' = VisEdtoLe);
        [laed] (Vis_STATE = VisEdsda) -> a : (Vis_STATE' = VisEdsua);
        [lbed] (Vis_STATE = VisEdsdb) -> a : (Vis_STATE' = VisEdsub);
        [lced] (Vis_STATE = VisEdsdc) -> a : (Vis_STATE' = VisEdsuc);
        [lded] (Vis_STATE = VisEdsdd) -> a : (Vis_STATE' = VisEdsud);
        [leed] (Vis_STATE = VisEdsde) -> a : (Vis_STATE' = VisEdsue);
        [lasu] (Vis_STATE = VisEdsua) -> s : (Vis_STATE' = VisEdReca);
        [lbsu] (Vis_STATE = VisEdsub) -> s : (Vis_STATE' = VisEdRecb);
        [lcsu] (Vis_STATE = VisEdsuc) -> s : (Vis_STATE' = VisEdRecc);
        [ldsu] (Vis_STATE = VisEdsud) -> s : (Vis_STATE' = VisEdArrived);
        [lesu] (Vis_STATE = VisEdsue) -> s : (Vis_STATE' = VisEdRece);
        [lasd] (Vis_STATE = VisEdtoLa) -> s : (Vis_STATE' = VisEdsda);
        [lbsd] (Vis_STATE = VisEdtoLb) -> s : (Vis_STATE' = VisEdsdb);
        [lcsd] (Vis_STATE = VisEdtoLc) -> s : (Vis_STATE' = VisEdsdc);
        [ldsd] (Vis_STATE = VisEdtoLd) -> s : (Vis_STATE' = VisEdsdd);
        [lesd] (Vis_STATE = VisEdtoLe) -> s : (Vis_STATE' = VisEdsde);
        [nop] (Vis_STATE = VisEeArrived) -> a : (Vis_STATE' = VisEeArrived);
        [eeale] (Vis_STATE = VisEeReca) -> r : (Vis_STATE' = VisEetoLe);
        [eeala] (Vis_STATE = VisEeReca) -> r : (Vis_STATE' = VisEetoLa);
        [eealb] (Vis_STATE = VisEeReca) -> r : (Vis_STATE' = VisEetoLb);
        [eealc] (Vis_STATE = VisEeReca) -> r : (Vis_STATE' = VisEetoLc);
        [eeald] (Vis_STATE = VisEeReca) -> r : (Vis_STATE' = VisEetoLd);
        [eeble] (Vis_STATE = VisEeRecb) -> r : (Vis_STATE' = VisEetoLe);
        [eebla] (Vis_STATE = VisEeRecb) -> r : (Vis_STATE' = VisEetoLa);
        [eeblb] (Vis_STATE = VisEeRecb) -> r : (Vis_STATE' = VisEetoLb);
        [eeblc] (Vis_STATE = VisEeRecb) -> r : (Vis_STATE' = VisEetoLc);
        [eebld] (Vis_STATE = VisEeRecb) -> r : (Vis_STATE' = VisEetoLd);
        [eecle] (Vis_STATE = VisEeRecc) -> r : (Vis_STATE' = VisEetoLe);
        [eecla] (Vis_STATE = VisEeRecc) -> r : (Vis_STATE' = VisEetoLa);
        [eeclb] (Vis_STATE = VisEeRecc) -> r : (Vis_STATE' = VisEetoLb);
        [eeclc] (Vis_STATE = VisEeRecc) -> r : (Vis_STATE' = VisEetoLc);
        [eecld] (Vis_STATE = VisEeRecc) -> r : (Vis_STATE' = VisEetoLd);
        [eedle] (Vis_STATE = VisEeRecd) -> r : (Vis_STATE' = VisEetoLe);
        [eedla] (Vis_STATE = VisEeRecd) -> r : (Vis_STATE' = VisEetoLa);
        [eedlb] (Vis_STATE = VisEeRecd) -> r : (Vis_STATE' = VisEetoLb);
        [eedlc] (Vis_STATE = VisEeRecd) -> r : (Vis_STATE' = VisEetoLc);
        [eedld] (Vis_STATE = VisEeRecd) -> r : (Vis_STATE' = VisEetoLd);
        [laee] (Vis_STATE = VisEesda) -> a : (Vis_STATE' = VisEesua);
        [lbee] (Vis_STATE = VisEesdb) -> a : (Vis_STATE' = VisEesub);
        [lcee] (Vis_STATE = VisEesdc) -> a : (Vis_STATE' = VisEesuc);
        [ldee] (Vis_STATE = VisEesdd) -> a : (Vis_STATE' = VisEesud);
        [leee] (Vis_STATE = VisEesde) -> a : (Vis_STATE' = VisEesue);
        [lasu] (Vis_STATE = VisEesua) -> s : (Vis_STATE' = VisEeReca);
        [lbsu] (Vis_STATE = VisEesub) -> s : (Vis_STATE' = VisEeRecb);
        [lcsu] (Vis_STATE = VisEesuc) -> s : (Vis_STATE' = VisEeRecc);
        [ldsu] (Vis_STATE = VisEesud) -> s : (Vis_STATE' = VisEeRecd);
        [lesu] (Vis_STATE = VisEesue) -> s : (Vis_STATE' = VisEeArrived);
        [lasd] (Vis_STATE = VisEetoLa) -> s : (Vis_STATE' = VisEesda);
        [lbsd] (Vis_STATE = VisEetoLb) -> s : (Vis_STATE' = VisEesdb);
        [lcsd] (Vis_STATE = VisEetoLc) -> s : (Vis_STATE' = VisEesdc);
        [ldsd] (Vis_STATE = VisEetoLd) -> s : (Vis_STATE' = VisEesdd);
        [lesd] (Vis_STATE = VisEetoLe) -> s : (Vis_STATE' = VisEesde);

endmodule

// We make another copy of module Vis
module Vis_2 = Vis[Vis_STATE=Vis_2_STATE]
endmodule

// We make another copy of module Vis
module Vis_3 = Vis[Vis_STATE=Vis_3_STATE]
endmodule

// We make another copy of module Vis
module Vis_4 = Vis[Vis_STATE=Vis_4_STATE]
endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const SlotLa = 0;
const SlotLaRespEd = 1;
const SlotLaRespEe = 2;
module SlotLa

        SlotLa_STATE : [0..2] init SlotLa;

        [laee] (SlotLa_STATE = SlotLa) -> a : (SlotLa_STATE' = SlotLaRespEe);
        [laed] (SlotLa_STATE = SlotLa) -> a : (SlotLa_STATE' = SlotLaRespEd);
        [edalb] (SlotLa_STATE = SlotLaRespEd) -> r : (SlotLa_STATE' = SlotLa);
        [eealb] (SlotLa_STATE = SlotLaRespEe) -> r : (SlotLa_STATE' = SlotLa);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const SlotLb = 0;
const SlotLbRespEd = 1;
const SlotLbRespEe = 2;
module SlotLb
```

```
        SlotLb_STATE : [0..2] init SlotLb;

        [lbee] (SlotLb_STATE = SlotLb) -> a : (SlotLb_STATE' = SlotLbRespEe);
        [lbed] (SlotLb_STATE = SlotLb) -> a : (SlotLb_STATE' = SlotLbRespEd);
        [edblc] (SlotLb_STATE = SlotLbRespEd) -> r : (SlotLb_STATE' = SlotLb);
        [eeble] (SlotLb_STATE = SlotLbRespEe) -> r : (SlotLb_STATE' = SlotLb);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const SlotLc = 0;
const SlotLcRespEd = 1;
const SlotLcRespEe = 2;
module SlotLc

        SlotLc_STATE : [0..2] init SlotLc;

        [lcee] (SlotLc_STATE = SlotLc) -> a : (SlotLc_STATE' = SlotLcRespEe);
        [lced] (SlotLc_STATE = SlotLc) -> a : (SlotLc_STATE' = SlotLcRespEd);
        [edcld] (SlotLc_STATE = SlotLcRespEd) -> r : (SlotLc_STATE' = SlotLc);
        [eeclb] (SlotLc_STATE = SlotLcRespEe) -> r : (SlotLc_STATE' = SlotLc);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const SlotLd = 0;
const SlotLdRespEe = 1;
module SlotLd

        SlotLd_STATE : [0..1] init SlotLd;

        [ldee] (SlotLd_STATE = SlotLd) -> a : (SlotLd_STATE' = SlotLdRespEe);
        [lded] (SlotLd_STATE = SlotLd) -> a : (SlotLd_STATE' = SlotLd);
        [eedlc] (SlotLd_STATE = SlotLdRespEe) -> r : (SlotLd_STATE' = SlotLd);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const SlotLe = 0;
const SlotLeRespEd = 1;
module SlotLe

        SlotLe_STATE : [0..1] init SlotLe;

        [leee] (SlotLe_STATE = SlotLe) -> a : (SlotLe_STATE' = SlotLe);
        [leed] (SlotLe_STATE = SlotLe) -> a : (SlotLe_STATE' = SlotLeRespEd);
        [edelb] (SlotLe_STATE = SlotLeRespEd) -> r : (SlotLe_STATE' = SlotLe);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const PlaceFreeLa = 0;
const PlaceFullLa = 1;
```

```
module PlaceFreeLa

        PlaceFreeLa_STATE : [0..1] init PlaceFreeLa;

        [lasd]
            (PlaceFreeLa_STATE = PlaceFreeLa)
            -> s :
            (PlaceFreeLa_STATE' = PlaceFullLa);
        [lasu]
            (PlaceFreeLa_STATE = PlaceFullLa)
            -> s :
            (PlaceFreeLa_STATE' = PlaceFreeLa);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const PlaceFreeLb = 0;
const PlaceFullLb = 1;
module PlaceFreeLb

        PlaceFreeLb_STATE : [0..1] init PlaceFreeLb;

        [lbsd]
            (PlaceFreeLb_STATE = PlaceFreeLb)
            -> s :
            (PlaceFreeLb_STATE' = PlaceFullLb);
        [lbsu]
            (PlaceFreeLb_STATE = PlaceFullLb)
            -> s :
            (PlaceFreeLb_STATE' = PlaceFreeLb);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const PlaceFreeLc = 0;
const PlaceFullLc = 1;
module PlaceFreeLc

        PlaceFreeLc_STATE : [0..1] init PlaceFreeLc;

        [lcsd]
            (PlaceFreeLc_STATE = PlaceFreeLc)
            -> s :
            (PlaceFreeLc_STATE' = PlaceFullLc);
        [lcsu]
            (PlaceFreeLc_STATE = PlaceFullLc)
            -> s :
            (PlaceFreeLc_STATE' = PlaceFreeLc);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const PlaceFreeLd = 0;
const PlaceFullLd = 1;
module PlaceFreeLd
```

```
        PlaceFreeLd_STATE : [0..1] init PlaceFreeLd;

        [ldsd]
            (PlaceFreeLd_STATE = PlaceFreeLd)
            -> s :
            (PlaceFreeLd_STATE' = PlaceFullLd);
        [ldsu]
            (PlaceFreeLd_STATE = PlaceFullLd)
            -> s :
            (PlaceFreeLd_STATE' = PlaceFreeLd);

endmodule

// Descriptive names for the local states of this module, taken
// from the PEPA input model.  Duplications will be omitted.
const PlaceFreeLe = 0;
const PlaceFullLe = 1;
module PlaceFreeLe

        PlaceFreeLe_STATE : [0..1] init PlaceFreeLe;

        [lesd]
            (PlaceFreeLe_STATE = PlaceFreeLe)
            -> s :
            (PlaceFreeLe_STATE' = PlaceFullLe);
        [lesu]
            (PlaceFreeLe_STATE = PlaceFullLe)
            -> s :
            (PlaceFreeLe_STATE' = PlaceFreeLe);

endmodule


// The system equation
system
        ((Vis  |||  (Vis_2 |||  (Vis_3  |||  Vis_4)))
            |[laee, lbee, lcee, ldee, leee, laed, lbed, lced, lded, leed,
            eeala, eealb, eealc, eeald, eeale, eebla, eeblb, eeblc, eebld, eeble,
            eecla, eeclb, eeclc, eecld, eecle, eedla, eedlb, eedlc, eedld, eedle,
            edala,  edalb, edalc, edald, edale, edbla, edblb, edblc, edbld, edble,
            edcla, edclb, edclc, edcld, edcle, edela, edelb, edelc, edeld, edele,
            lasd, lasu, lbsd, lbsu, lcsd, lcsu, ldsd, ldsu, lesd, lesu]|
            (SlotLa ||| (SlotLb ||| (SlotLc ||| (SlotLd  ||| (SlotLe  |||
            (PlaceFreeLa ||| (PlaceFreeLb  ||| (PlaceFreeLc  ||| (PlaceFreeLd  |||  PlaceFreeLe))))))))))
endsystem

// End of output from the PEPA-to-PRISM compiler
```

The above specification results in a CTMC composed of 249344 states and 1572000 transitions.